Vérification automatique pour l'exécution sécurisée de composants Java

Pierre Parrend, Stéphane Frénot

Vérification automatique pour l'exécution sécurisée de composants Java

Pierre Parrend, Stéphane Frénot

INRIA ARES / CITI, INSA-Lyon, F-69621, France
tel. +334 72 43 71 29 - fax. +334 72 43 62 27

RÉSUMÉ. *Les plates-formes dynamiques de services permettent d'exécuter simultanément plusieurs composants fournis par des tiers. Ceci apporte une grande flexibilité dans leur utilisation, aussi bien en environnements à ressources limitées que dans le cas de serveurs d'applications. Toutefois, les implications pour la sécurité du système sont encore mal connues: quels sont les risques posés par l'exécution de composants tiers pour la plate-forme d'execution ? pour les autres composants ? Comment y remédier ? A partir d'expérimentations réalisées sur la plate-forme Java/OSGi, nous proposons une classification des vulnérabilités des plates-formes dynamiques de services. Deux cas sont considérés: les vulnérabilités de la plate-forme elle-même, et les vulnérabilités des composants. Plusieurs solutions sont proposées pour résoudre ces vulnérabilités. Premièrement, le Contrôle d'accès basé Composants (CBAC, pour Component-based Access Control) permet de limiter l'accès à des méthodes dangereuses de la plate-forme ou des composants. La validation est effectuée par analyse statique de code. La configuration est entièrement déclarative, ce qui rend cette approche extensible, et adaptée pour la protection de méthodes fournies par des composants tiers. Deuxièmement, l'Analyse de Composants faibles (WCA, pour Weak Component Analysis) permet d'identifier les vulnérabilités des composants, par analyse statique de code également. CBAC et WCA exploitent la phase d'installation des composants pour réaliser les vérifications nécessaires. Seuls les composants valides sont installés. WCA peut également être utilisé lors du dévelopement pour améliorer la qualité du code.*

ABSTRACT. *Service-oriented Programming (SOP) platforms allow to simultaneously execute several components provided by third parties. This introduces flexibility in applications for embedded environments and for application servers. However, the security implications are not well understood so far. Which are the risks of executing third party components for the platform ? For other components ? How to remedy to them ? Based on experiments performed with the Java/OSGi platform, we propose a classification of SOP platform vulnerabilities. Two cases are to be considered: platform vulnerabilities and component vulnerabilities. Several solutions are proposed to solve these vulnerabilities. First, Component-based Access Control (CBAC)*

L'objet. Volume 8 – n°2/2008, pages 1 à 15

enables to restrict the acccess to sensitive methods in the platform or components. Validation is performed through code static analysis. Configuration is fully declarative, which makes the approach extensible and suitable for handling third party components. Secondly, Weak Component Analysis (WCA) enables to identify vulnerabilities in the component Bytecode with a similar static analysis technique. CBAC and WCA exploit the installation phase of componentts to perform required validations. Only valid components can then be installed. WCA can also be used as a development tool to enhance code quality.

MOTS-CLÉS : *Analyse statique, composants, Service-oriented Programming, Java, OSGi.*

KEYWORDS: *Static analysis, components, Service-oriented Programming, Java, OSGi.*

1. Introduction

Les plates-formes dynamiques de services sont utilisées de plus en plus fréquement pour le développement d'applications complexes. Elles sont caractérisées par la mise en œuvre de techniques de virtualisation, de modularité et de services accessibles localement selon le principe de la programmation orientée service (SOP, Service-oriented programming) (Bieber *et al.,* 2001). Elles sont employées dans des contextes très variés, pour des applications de haut niveau écrites en Java ou pour la plate-forme .Net, en environnement connectés : systèmes embarqués, serveurs d'application. Nous nous intéressons ici aux plates-formes Java.

Dans un tel contexte, le niveau de sécurité fourni par l'environnement d'exécution est primordial, à la fois pour l'acceptation de ces technologies et pour l'exploitation de leurs capacités. Un exemple marquant est la possibilité d'étendre les plates-formes lors de leur exécution avec des composants découverts dynamiquement dans l'environnement. Ces composants peuvent être signés ou non par des fournisseurs tiers, mais aucun outil n'existe actuellement pour évaluer leur nocivité.

Nous proposons par conséquent une étude complète des propriétés de sécurité des plates-formes à composants. Afin de faciliter leur intégration, ces composants interagissent par le biais de services accessibles localement et définis sous forme d'une interface de programmation. Pour nos expérimentations, nous utilisons comme référence la plate-forme OSGi (OSGi Alliance, 2007), qui est une plate-forme dynamique de services typique. Les vulnérabilités exploitables par les composants qui y sont installés, que ce soit sur la plate-forme elle même ou dans les composants, sont classifiées. Deux solutions sont proposées, CBAC et WCA, qui permettent respectivement de résoudre les vulnérabilités de la plate-forme et des composants. Elles utilisent l'analyse statique de Bytecode, ce qui permet de les utiliser aussi bien en tant qu'outil de développement qu'en tant que protection dans les plates-formes en production. Le contrôle d'accès basé composants (CBAC, Component-based Access Control) identifie les appels dangereux effectués par un composant. L'analyse de composants vulnérables (WCA, Weak Component Analysis) identifie les vulnérabilités dans le code que les composants exposent aux autres sous forme de classes exportées ou de services accessibles localement.

Le document est organisé comme suit. La section 2 présente les propriétés de sécurité des plates-formes dynamiques de services. La section 3 présente les vulnérabilités de ces plates-formes. La section 4 présente les outils de validation de composants, CBAC et WCA. La section 5 présente les expérimentations. La section 6 conclut ce travail.

2. Plates-formes dynamiques de services : les enjeux de sécurité

Les plates-formes dynamiques de services sont des plates-formes à composants qui supportent le paradigme de programmation orientée service. Elles sont composées de trois éléments principaux : une machine virtuelle, un mécanisme de support de la

modularité et un mécanisme de support de la programmation orientée service. Chacun de ces trois éléments introduit des mécanismes de sécurité qui lui sont propres et qui se combinent selon le principe de sécurité en profondeur (Saltzer *et al.,* 1973). Chacun introduit également des vulnérabilités spécifiques.

2.1. *Des environements favorables à la sécurité*

La machine virtuelle (MV) joue un rôle fondamental dans la sécurité des plates-formes dynamiques de services en isolant les applications du système : celles-ci ne peuvent accéder, sans autorisation explicite, au système d'exploitation sous-jacent. Seule l'interface de programmation de la MV est accessible. Comme elle est bien plus réduite qu'un système d'exploitation, elle est également bien plus facile à sécuriser. La machine virtuelle introduit également la portabilité des applications et des mécanismes de sécurité. La portabilité des applications évite la mutiplication des configurations, difficile à gérer et donc à sécuriser. Les administrateurs peuvent se concentrer sur la réalisation d'une version stable d'application et la réutiliser ensuite en conservant ses propriétés notamment de sécurité. La portabilité des mécanismes de sécurité permet également de conserver les configurations de sécurité. Enfin, la machine virtuelle garantit l'utilisation d'un langage de programmation dont les propriétés de sécurité sont établies, et contrôlées lors de l'exécution. Les propriétés de sécurité spécifique à la MV et au langage Java sont les suivantes (Gong *et al.,* 2003) :

– Sûreté de type : chaque élément du langage est associé à un type donné et ne peut exécuter que des opérations définies par ce type. En particulier, un langage à sûreté de type ne contient pas de pointeurs, qui permettraient la manipulation de données non typées et d'adresses mémoires.

– Gestion automatique de la mémoire : la gestion de la mémoire est masquée du langage de programmation, et n'est donc pas à charge du développeur. Ceci permet d'éviter une source d'erreurs fréquentes, d'augmenter la productivité du développeur, et dans la majorité des cas d'optimiser les performances du sytème en effectuant la désallocation mémoire dès que possible.

– Validation de Bytecode : le format du Bytecode qui est exécuté par la machine virtuelle est validé avant son chargement en mémoire afin de garantir sa compatibilité avec la spécification du langage. En particulier, ceci permet d'éviter l'exécution de code forgé dans un but malicieux qui n'aurait pas été produit par un compilateur valide.

– Modularité : Java permet d'isoler les espaces de nommage des composants par l'utilisation de chargeurs de classes. Chaque chargeur de classes peut utiliser ses propres classes et les classes de ses parents, dont le chargeur de classes 'bootstrap' qui contient les classes de l'API standard.

La modularité consiste à organiser les applications sous forme de composants qui mettent des classes à disposition des autres composants du système. Telle qu'elle est supportée par les plates-formes à composants, elle ne peut être considérée comme un mécanisme de sécurité. Cependant, elle est une bonne pratique qui permet d'améliorer

la testabilité d'un système et l'isolation de ses donnés (Miller *et al.,* 2004). Elle permet donc d'améliorer la qualité et la stabilité des applications, et donc réduit le nombre d'erreurs qui sont susceptibles d'être exploitées à des fins malicieuses.

Le paradigme de programmation orientée service consiste à permettre aux composants de communiquer par le biais de services accessibles localement, c'est-à-dire d'objets qui peuvent être partagés. En lui-même, il n'apporte pas de bénéfice pour la sécurité. Il ajoute de la flexilibité, et donc de la mouvance dans les interactions entre composants. C'est donc un facteur de risque supplémentaire.

2.2. *Vulnerabilités des applications à composants Java*

Bien que le langage Java ai été conçu pour être sécurisé, il comporte un certain nombre de vulnérabilités recensées dans la litérature (The Last Stage of Delirium. Research Group., 2002, Long, 2005, Lai, 2008). Les applications Java doivent donc être développées de manière à ce que ces vulnérabilités ne soient pas exploitables.

En particulier, un certain nombre de mécanismes standards du langage Java et de la JVM peuvent être exploités pour subvertir le système (Long, 2005) :

– Attributs publics : les données dont l'accès n'est pas restreint peuvent être modifiées par les autres classes.
– Classes internes : l'absence de support pour les classes internes au niveau Bytecode force les compilateurs à générer les classes internes comme des classes simples présente dans le même package. Pour permettre l'accès aux champs privés, les attributs et méthodes voient leur visibilité augmentée au niveau 'package'.
– Sûreté de type : l'usage du même nom pour des objets de types différents peut conduire dans certains cas à de la confusion. Ce peut être un exploit visant les classes internes.
– Sérialisation : les données sérialisées sans être encryptées peuvent être lues par tous.
– JVM-Tool Interface (JVM-TI) : un outil de surveillance et de gestion de la JVM, exécuté à l'extérieur de celle-ci. En particulier, l'état des threads de la JVM peut être modifié. L'usage de JVM-TI ne demande pas de permission particulière.
– Gestion JMX : l'API Java Management Extension permet d'accéder et de modifier l'état du système et des applications. Elle ne peut être utilisée que si un serveur JMX existe dans l'application.

Les éléments qui composent les plates-formes dynamiques de services introduisent eux aussi des risques de sécurité. La MV est un facteur d'extensibilité : elle permet l'exécution de code moins sûr fourni par des tierces parties. Les vulnérabilités présentes dans l'API Java peuvent donc être facilement exploitables. De plus, il est plus facile d'instancier une JVM que de créer un environnement d'exécution physique à part entière. Le nombre de JVM est donc bien plus important, et plus difficile à identifier, ce qui peut poser des problèmes en particulier dans des réseaux d'entreprises soumis à des politiques de sécurité clairement définies. Les chargeurs de

classes ne fournissent qu'une isolation limitée. En particulier, les classes systèmes sont partagées. Certaines données, en particulier les attributs statiques, telle la variable static `out` de la classes `System`, peuvent provoquer des interférences entre les composants. De plus, aucun mécanisme d'isolation de ressource n'existe par défaut. La programmation orientée service permet de publier des services sous forme d'objets. Si ces objets sont des singletons, ils sont partagés entre plusieurs clients qui n'ont pas forcément établi une confiance réciproque. Les risques sont donc plus grand que si des classes sont partagées sans que leurs instances le soient.

Au niveau de la plate-forme à composants, chaque couche est susceptible d'introduire des vulnérabilités. Dans le cas d'OSGi, il s'agit des couches suivantes :

– Couche module : elle effectue la résolution de dépendances au niveau package entre les bundles OSGi.
– Couche cycle de vie : elle supporte les opérations de gestion du cycle de vie des composant : installation, démarrage, arrêt, désinstallation, mise à jour.
– Couche service : elle met à disposition un annuaire de services, dans lequel des services peuvent être publiés et découverts. Ces services sont accessibles localement et identifié par l'interface Java qu'ils implémentent.

3. Une classification des vulnérabilités Java

Les composants à service sont destinés à être exécutés dans les plates-formes dynamiques de services. La définition de mécanismes de sécurité adaptés doit être fondée sur une connaissance précise des risques qu'ils posent pour la sécurité des applications. L'exécution de composants à service présente un risque à la fois pour la plate-forme d'exécution et pour les autres composants du système. Nous définissons donc deux taxonomies complémentaires : celle concernant les implantations des vulnérabilités de la plate-forme Java/OSGi elle-même, et celle concernant les implantations des vulnérabilités des composants à services Java. Des taxonomies complémentaires décrivent les propriétés des composants malicieux. Elles facilitent l'identification des vulnerabilités concernées, et permettent de mieux appréhender leurs conséquences.

Chacune de ces vulnérabilités fait l'objet d'une implantation de référence, afin de démontrer son exploitation, et d'une description précise. Les données concernant les vulnérabilités sont rassemblées dans deux catalogues, *Bundles malicieux* (Parrend *et al.,* 2007) exploitant la plate-forme, et *Bundles vulnérables* (Parrend *et al.,* 2008c) susceptibles d'être exploités par d'autres.

3.1. *Composants malicieux*

Les attaques réalisées par des composants malicieux peuvent être caractérisées par un ensemble de propriétés : la localisation du code malicieux dans le composant, l'implantation de l'attaque, sa cible, ses conséquences, ainsi que le moment d'introduction et d'exploitation du code d'attaque. Les taxonomies représentant l'implantation des

vulnérabilités sont présentées dans les sections 3.2 et 3.3. Les taxonomies relatives à la localisation du code malicieux et aux conséquences des attaques sont présentées ci-dessous.

Le code malicieux peux se trouver dans différents éléments d'un composant. Cette classification correspond naturellement à la structure d'un composant tel qu'il est défini par exemple dans le cadre d'OSGi.

– Archive : une vulnérabilité est exploitée au travers du format de l'archive du composant, par exemple sa taille, sa structure.
– Fichier `Manifest.MF` : une vulnérabilité est exploitée par le biais de valeurs spécifiques des méta-données contenues dans le fichier Manifest.
– Activateur : une vulnérabilité est exploitée dans la classes 'BundleActivator' qui permet de démarrer, configurer et arrêter un composant.
– Code applicatif : une vulnérabilité est exploitée par le code du composant.

- Code natif : une vulnérabilité est exploitée par le biais d'un appel à du code natif.
- Code Java : une vulnérabilité est exploitée par le code Java lui-même.
- API Java : une vulnérabilité est exploitée par un appel à une méthode de l'API standard Java.
- API OSGi : une vulnérabilité est exploitée par un appel à une méthode de l'API OSGi.

– Bundle 'Fragment' : une vulnérabilité est exploitée par le biais du mécanisme `Fragment`, qui permet de partager l'ensemble des classes entre deux composants différents.

Les conséquences d'une attaque menée par un composant malicieux peuvent être :

– Indisponibilité : l'entité cible de l'attaque, à l'intérieur de la plate-forme, n'est plus disponible. Elle est soit arrêtée soit désinstallée. S'il s'agit d'une plate-forme, toutes les entités qui en dépendent, par exemple les composants, sont également indisponibles.
– Chute de performance : l'entité cible de l'attaque souffre d'une perte de performance. S'il s'agit de la plate-forme, toutes les entités qui en dépendent s'exécutent avec moins de ressources.
– Accès indu, dans le cadre des appels de service : le composant accède aux données et/ou au code de l'entité cible de l'attaque. Dans la mesure où dans un langage de programmation, tel Java, aucune différence n'existe entre l'accès en lecture et en écriture, les données peuvent être lues ET modifiées.

Cette classification des conséquences est différente des propriétés utilisées habituellement en sécurité informatique, qui sont la disponibilité, l'intégrité et la confidentialité (Lindqvist *et al.,* 1997). La propriété de disponibilité correspond aux attaques à déni de service. Elle est raffinée ici en deux catégories, 'indisponibilité' et 'chute de performance', qui correspondent pour les plates-formes OSGi en particulier à deux ensembles de vulnérabilités distincts. Les propriétés d'intégrité et de confidentialité sont groupées sous le terme 'accès indu', dans la mesure où les langages

de programmation de haut niveau, contrairement aux protocoles réseaux et aux systèmes d'exploitations, ne permettent pas de contrôler le type d'accès, en lecture ou en écriture.

3.2. *Vulnérabilités des plates-formes*

Les vulnérabilités des plates-formes dynamiques de services sont référencées dans le catalogue *Bundles malicieux* (Parrend *et al.,* 2007). Elles permettent à un composant installé d'exploiter deux types de faiblesses de leur environnement d'exécution : des erreurs d'implantation, mais également des fonctionnalités qui s'avèrent être dangereuses quand elles-sont exécutées par du code mal intentionné.

Les vulnérabilités sont identifiées et validées de la façon suivante. Une recherche systématique des vulnérabilités potentielles a été effectuée dans la littérature, mais également auprès de développeurs expérimentés. Pour chacune d'entre elle, un composant malicieux a été développé pour démontrer leur facilité d'exploitation.

Le tableau 1 présente la taxonomie concernant les implantations des vulnérabilités de la plate-forme Java/OSGi.

Le tableau contient les informations suivantes : l'entité concernée, c'est-à-dire la machine virtuelle ou la plate-forme à composants, la couche à l'intérieur de l'entité, la propriété spécifiquement concernée. Ces propriétés peuvent être des fonctions dont l'utilisation peut être dangereuse, ou bien des erreurs qui doivent être corrigées. Le nombre d'occurrences correspond pour chaque propriété au nombre de vulnérabilités identifiées.

Les vulnérabilités sont liées presque à part égale à la JVM (dix-huit occurrences) et à la plate-forme OSGi (dix-sept occurrences). On pourra également remarquer qu'aucune couche n'est exempte de vulnérabilité. Du coté Java, le 'Runtime' (Environnement d'exécution), l'API et le langage lui-même sont en cause. En ce qui concerne OSGi, les couches 'Cycle de vie', 'Module' et 'Service' contiennent chacune un certain nombre de faiblesses.

La présence de vulnérabilités dans la JVM s'explique par le modèle d'exécution nouveau qui est permis par les plates-formes dynamiques de services. Des composants fournis par des tiers peuvent être découverts durant l'exécution, installés et exécutés. Dans la plupart des applications Java existant actuellement, les composants sont sélectionnés lors du développement, et peuvent donc être controlés. Cette évolution du modèle économique d'utilisation n'est pas accompagnée par une remise en question du modèle de sécurité, qui s'avère être insuffisant.

La présence de vulnérabilités dans la plate-forme OSGi s'explique par la relative nouveauté de cet environnement, et par sa diffusion plus limitée. De plus, la plupart des applications d'OSGi sont soit open source, par exemple dans la plate-forme Eclipse, soit des systèmes embarqués fermés, comme dans la plate-forme d'inforécréation des séries 5 BMW. Dans le premier cas, la communauté considère que

Entité	Couche	Propriété	Erreur/ Fonction	Occurrences
Operating System			Fonction	2
JVM	Runtime	Méthodes d'arrêt du Runtime	Fonction	2
		Gestion des Threads	Fonction	3
	API	ClassLoader	Fonction	3
		Reflexion	Fonction	3
		Gestion de Fichiers	Fonction	1
		Execution de Code natif	Fonction	2
	Langage	Pas de sureté des algorithmes	Erreur	4
OSGi	Cycle de Vie	Désinstallation propre	Erreur	1
		Gestion des Bundles	Fonction	2
		Activateur invalide	Erreur	2
		Archive invalide	Erreur	3
	Module	Fragments	Fonction	3
		Metadonnées invalides	Erreur	3
	Service	Pas de contrôle sur les services publiés	Erreur	2
		Workflow invalide	Erreur	1

Tableau 1. *Taxonomie : implémentations des vulnérabilités de la plate-forme Java/OSGi*

les applications et plug-ins existants sont bienveillants. A notre connaissance, aucun cas de bundle OSGi malicieux n'a été jusqu'à présent identifié. Dans le second cas, les possibilités de mise à jour du système sont restreintes au vendeur, qui peut donc contrôler le code exécuté. Ce choix est nécessaire pour des environnement qui peuvent être sensibles, comme des outils télématiques pour l'automobile, mais ne permet pas d'exploiter pleinement les possibilités offertes par les plates-formes dynamique à composants.

3.3. *Vulnérabilités des composants*

Les vulnérabilités des composants à service sont référencées dans le catalogue *Bundles vulnérables* (Parrend *et al.,* 2008c, Parrend *et al.,* 2008a). Elles sont de trois

catégories, selon le type d'application qui est effectivement concerné. Les vulnérabilités peuvent être exploitables dans les applications autonomes, dans les composants de type 'partage de classes' comme dans les packages exportés dans la plate-forme OSGi, ou bien dans les composants de type 'partage d'objets' comme dans les services de cette même plate-forme. Chaque catégorie englobe les vulnérabilités de la précédente : un composant à partage de classes sera aussi vulnérable vis-à-vis des vulnérabilités exploitables dans les applications autonomes. Les composants à services sont donc concernés par l'ensemble des vulnérabilités identifiées.

De même que pour les vulnérabilités des plates-formes, celles-ci sont issues de la littérature et de l'expérience. Elles sont validées par le développement de couples bundle OSGi malicieux/bundle OSGi vulnérable afin de montrer qu'elles sont exploitables directement par tout composant installé dans la plate-forme.

Le tableau 2 présente la taxonomie concernant les implantations des vulnérabilités des composants dans un environment Java/OSGi.

Les vulnérabilités sont caractérisées par le vecteur d'attaque qui rend leur exploitation possible - ici, l'interaction entre composants - et leur implantation, c'est-à-dire l'exposition du code vulnérable permettant leur exploitation, la propriété concernée, et le type de propriété. L'exposition d'une classe peut être de niveau classe, par exemple avec les packages exportés en OSGi, de niveau objet ou SOP, par exemple avec les services publiés par des bundles OSGi, ou interne aux composants. Cette dernière catégorie concerne également les applications autonomes.

Les applications autonomes sont concernées par un type de vulnérabilité : la sérialisation. En effet, des données sérialisées peuvent être lues et modifiées par n'importe quel entité disposant d'un accès vers le support où elles sont stockées : disque, réseau.

Les composants à partage de classes peuvent être exploités par le biais de l'exposition de leur représentation interne, ou par l'évitement de vérifications de sécurité qui devraient être obligatoires. La représentation interne d'une classe peut être ses variables statiques, ou la structure de son code. Les vérifications de sécurité sont typiquement des appels au gestionnaire de sécurité, c'est-à-dire la classe `SecurityManager`. Ils sont insérés dans le code lui-même. S'ils sont présents dans un ou des constructeurs de la classe, ils peuvent être évités par les mécanismes d'instanciation qui n'utilisent pas le constructeur, comme le clonage ou la désérialisation.

Les composants à partage d'objets sont vulnérables à l'exposition de leur représentation interne, au défaut de validation des paramètres de leurs méthodes, et à l'invalidité du flux (Workflow) de services qui est généré dynamiquement. L'exposition de la représentation interne peut être réalisée par le renvoi d'un élément mutable tel un `Set` ou une `Collection` qui peut être utilisé a posteriori pour modifier les données de la classe. Elle peut également être réalisée par l'insuffisance du contrôle d'accès aux données, par exemple si la visibilité d'une variable est excessive. Le défaut de validation des paramètres peut être dû à l'absence totale de validation, à la réalisation de validation sans copie des données, ce qui permet leur modification entre leur validation et leur utilisation, et à une validation incomplète.

Vecteur d'Attaque	**Implementation**			**Occurrences**
Interactions entre composants	App. Autonome	Serialization		1
	Partage de Classes	Représentation interne exposée	Elément mutable dans une variable statique	2
			Reflexion	3
			Fragments	2
			Control insuffisant	2
		Appels évitables au gestionnaire de sécurité Java	A l'instanciation	4
			Dans un appel de méthode	5
	Partage de Classes ou SOP	Synchronisation		2
	SOP	Représentation interne exposée	Renvoi d'une référence à un élement mutable	2
			Contrôle insuffisant	4
		Validation erronée de paramètres	Paramètre non validé	3
			Paramètre validé non copié	1
			Paramètre validé et copié	4
			Paramètre non final	2

Tableau 2. *Taxonomie : implémentations des vulnérabilités des composants dans un environment Java/OSGi*

Certaines vulnérabilités peuvent concerner à la fois les composants à partage de classes et les composants à partage d'objets. Il s'agit en particulier de la synchronisation, qui permet de geler l'exécution d'un appel à une méthode et des appels ultérieurs à cette même méthode. Cette vulnérabilité ne peut être exploitée que si une des dépendances de la méthode synchronisée est amenée à bloquer. Elle concerne les composants à partage de classes si l'appel à la méthode synchronisée se fait par le biais d'une méthode statique, et uniquement les composants à partage d'objet sinon.

3.4. *Niveau de sécurité des plates-formes Java/OSGi*

Ces deux taxonomies relatives aux vulnérabilités des plates-formes dynamiques de services et à celles des composants à services constituent une vision globale des enjeux de sécurité introduits par de tels environnements d'exécution. Afin de connaître les risques réellement encourus par les systèmes basés sur ces technologies, nous évaluons le niveau de sécurité des principales implantations de la plate-forme OSGi.

L'évaluation du niveau de sécurité des plates-formes est basée sur une métrique que nous introduisons, le *Taux de Protection*, définie dans le tableau 3.

La quantification du niveau de sécurité nécessite l'utilisation d'outils adéquats. Un tel outil est le *Taux de Protection* (PR, Protection Rate). Son objectif est double : il doit permettre d'évaluer la protection apportée par un mecanisme de sécurité isolé, et il doit permettre de comparer des environnements d'exécution complets tels Java/OSGi. Le *Taux de Protection* est défini comme le complément de la surface d'attaque vulnérable et de la surface d'attaque du système de référence. Il est exprimé comme suit :

$$PR = \left(1 - \frac{Surface\ d'attaque\ du\ systeme\ evalue}{Surface\ d'attaque\ du\ systeme\ de\ reference}\right) * 100 \qquad [1]$$

Il peut également être exprimé comme le quotient de la surface d'attaque (Howard *et al.,* 2005) qui est protégée par le ou les mécanismes de protection considérés et de la surface d'attaque du système de référence, sans aucune protection.
Cette métrique est nécessairement exprimée par rapport au nombre de vulnérabilités connues. Elle doit donc être mise à jour au fur et à mesure de l'évolution des connaissances concernant le système. Cette connaissance partielle permet cependant d'évaluer deux propriétés importantes : le bénéfice apporté par un mécanisme de sécurité donné, et le niveau de sécurité relatif entre deux implémentations d'une même plate-forme. Elle ne prétend pas représenter un niveau de sécurité absolu du système étudié.

Tableau 3. *Définition du taux de protection*

Le tableau 4 présente le taux de protection pour les principales implantations de la plate-forme OSGi, ainsi que l'apport du principal mécanisme de sécurité standard, les permissions Java, qui sont mises en œuvre par le gestionnaire de sécurité. Le taux de

protection est ici le quotient du nombre de vulnérabilités protégées par une implantation donnée de la plate-forme OSGi, le nombre d'erreurs protégées, et du nombre total de vulnérabilités identifiées, le nombre d'erreurs connues. La référence ici est le catalogue *Bundles malicieux* exploitant les vulnérabilités de la plate-forme, puisqu'il s'agit ici d'évaluer les plates-formes et non les composants qu'elles exécutent.

Plate-forme	**# d'erreurs protégées**	**# d'erreurs connues**	**Taux de Protection**
Concierge	0	28	0 %
Felix	1	32	3,1 %
Knopflerfish	1	31	3,2 %
Equinox	4	31	13 %
Java Permissions	13	32	41 %
Concierge with perms	10	28	36 %
Felix with perms	14	32	44 %
Knopflerfish with perms	14	31	45 %
Equinox with perms	17	31	55 %

Tableau 4. *Taux de protection pour les principales plates-formes OSGi*

Nous observons que les implantations de la plate-forme OSGi sont par défaut vulnérables à la quasi totalité des vulnérabilités identifiées. Certaines, à l'image d'Equinox, sont un peu plus robustes, mais cette robustesse reste marginale.

Les permissions Java apportent une nette amélioration dans le niveau de sécurité global du système. Toutefois, elles sont loin d'être suffisantes pour protéger la plate-forme et les composants contre l'exécution de code malicieux.

4. Des outils de validation pour les composants à service Java

Afin de protéger les plates-formes dynamiques de services et les composants qu'elles exécutent de la présence de code malicieux, nous proposons de valider les composants au moment de leur installation sur la plate-forme par analyse statique de code. Cette stratégie est adaptée au caractère dynamique des environnements cibles.

Deux mécanismes de protection sont proposés. CBAC (Component-based Access Control) vise à identifier les appels dangereux et à s'assurer que les composants qui les effectuent disposent des droits suffisants. WCA (Weak Component Analysis) vise à garantir que les composants installés sont dépourvus des vulnérabilités courantes. Dans les deux cas, un composant qui ne correspond pas à la politique définie n'est pas installé.

4.1. *Le patron de sécurité : 'Vérification des composants à l'installation'*

CBAC et WCA sont des implantations du patron de sécurité 'Vérification des composants à l'installation', qui consiste à effectuer des contrôle de sécurité immédiatement avant l'installation du composant sur la plate-forme. Ce patron permet d'intégrer dans le système des composants proposés par des fournisseurs tiers et découverts dynamiquement, s'ils remplissent les critères de vérification. Une implantation connue de ce patron est le code porteur de preuve (PCC, Proof Carrying Code) (Necula, 1997).

Le problème auquel ce patron apporte une solution est l'installation de code de fournisseurs non connus, ou vis-à-vis desquels la confiance n'est pas suffisante. L'objectif est d'identifier un certain nombre de propriétés de sécurité des composants à partir de leur code. Il s'agit également de réaliser un maximum de validation lors de l'installation plutôt que lors de l'exécution. Les validations à l'exécution ont plusieurs défauts. Elles sont coûteuses en temps, et impactent donc les performances des applications concernées. Elles conduisent à des erreurs lors de l'utilisation du système, ce qui n'est pas souhaitable du point de vue de l'utilisateur, et risque de conduire à de nouvelles vulnérabilités si ces erreurs ne sont pas gérées attentivement. Dans la mesure où les validations à l'installation font partie d'un ensemble d'opérations coûteuses comme le téléchargement ou la validation de signature numérique, elles sont susceptibles d'avoir un impact faible voire non observable sur la performance du système.

La solution consiste à effectuer la validation du code du composant dès le moment où celui-ci est disponible. Afin de préserver les attaques TOCTOU (Time of Check to Time of Use), c'est-à-dire les attaques par modification d'une ressource après sa modification par un mécanisme de sécurité, il est nécessaire que le composant ne puisse pas être modifié après sa validation. Ce patron est composé des entités suivantes : la plate-forme, qui initie la validation, le validateur, qui analyse le code, et la base de données de politiques, qui définit les propriétés qui doivent être vérifiées. Tous les composants qui ne sont pas valides selon les politiques de sécurité sont rejetés. Les autres sont installés. Les limitations de ce patron sont celles de l'analyse statique. Celle-ci génére nécessairement des faux positifs, c'est-à-dire que des composants sont rejetés alors qu'ils n'effectuent pas d'opérations interdites.

4.2. *CBAC : identification des appels dangereux*

L'objectif du Contrôle d'accès basé Composants (CBAC, Component-based Access Control) (Parrend *et al.,* 2008b) est de prévenir l'installation de composants contenant des appels dangereux qu'ils ne sont pas autorisés à exécuter. Il s'agit de valider lors de l'installation des politiques de sécurité similaires à celles mises en œuvre par le gestionnaire de sécurité Java lors de l'exécution. Celui-ci souffre en effet de plusieurs limitations qui préviennent son utilisation systématique dans les systèmes de production. Les contrôles sont effectués dans le code, et ne peuvent donc pas être étendus pour du code existant ; les performances d'un système sécurisé sont 20 à 30 % in-

férieures à celles d'un système non sécurisé ; les contrôles de sécurité à l'exécution aboutissent en cas d'échec à un comportement non stable du système, à moins qu'un effort très important de gestion de ces erreurs ait été mis en œuvre.

Son principe est le suivant. Les appels directs, exécutés par le composant lui-même, et les appels indirects, exécutés par le biais de composants dont il dépend, sont considérés. Ces informations sont extraites par analyse statique de code pour chaque composant, et stockées afin d'optimiser la prise en compte des dépendances. Elles sont dans l'implantation actuelle représentées par une liste des appels exécutés par le composant. Les politiques de sécurité sont composées de deux parties. Premièrement, les appels de méthodes considérés comme dangereux sont listés. Ceci permet d'enrichir la liste ultérieurement, par exemple si une méthode déjà utilisée s'avère contenir une vulnérabilité. Deuxièmement, les appels de méthodes autorisés pour chaque fournisseur de composant sont listés. Le fournisseur est identifié par la signature numérique qu'il appose dans le composant, ce qui garantit son identité.

La validation de sécurité est réalisée en deux étapes. Premièrement, les appels réalisés dans le code du composant sont validés vis-à-vis de la politique de sécurité. Ensuite, les appels exécutés dans les dépendances du composant sont validés. Un appel est autorisé soit s'il est inoffensif, soit s'il est à la fois dangereux et autorisé pour toute la pile d'appel.

Les paramètres suivants sont définis pour décrire les entités d'une plate-forme à composants, dans le modèle CBAC.

pf **:** la plate-forme à composants,
b_i **:** l'identifiant du bundle analysé,
$\{b\}_i$ **:** la liste des bundles dont le bundle i dépend,
$C_{S_{pf,b_i}}$ **:** les appels dangereux exécutés par le bundle i directement vers la plate-forme ou directement vers les bundles dont il dépend,
p_i **:** le fournisseur du bundle b_i,
A_{p_i} **:** l'ensemble d'appels dangereux autorisés pour le fournisseur p du bundle i,
PSC_{b_i} **:** l'ensemble des appels dangereux exécutés (Performed Sensitive Calls, PCSs) par le bundle i, directement ou par le biais d'appels de méthodes vers d'autres bundles,
$PSC_{\{b\}_i}$ **:** l'ensemble des appels dangereux exécutés par les bundles dont i dépend. $PSC_{\{b\}_i} = \sum_{b_j \ in \ \{b\}_i} PSC_{b_j}$.

Un composant N est valide si :

$$b_i, p_i, pf \vdash A_{p_i} \wedge PSC_{\{b\}_i}, \neg PSC_{\{b\}_i} \quad [2]$$

$$avec \ PSC_{b_i} = C_{S_{pf,b_i}} \vee PSC_{\{b\}_j} \quad [3]$$

La formule 2 signifie qu'un bundle peut être installé quand tous les appels qu'il effectue directement vers la plate-forme ou vers les autres bundles et tous les appels

qu'il effectue par le biais d'autres bundles sont soit dangereux et autorisés par les politiques de sécurité, soit inoffensifs. La formule 3 signifie que l'ensembles des appels dangereux effectués par un bundle comprend les appels effectués par le bundle lui-même et ceux effectués par ses dépendances. La démonstration est disponible en tant qu'annexe de (Parrend *et al.,* 2008b)[1]

Pour des raisons de performances, CBAC est implémenté avec la bibliothèque de manipulation de Bytecode ASM (Bruneton *et al.,* 2002). Les politiques de sécurité sont exprimées dans un langage proche des permissions Java, afin de limiter autant que possible le temps d'apprentissage pour les développeurs. CBAC est disponible sous deux formes : une bibliothèque Java, contenant une méthode `fr.inria.ares.cbac.Checker.check()`, et des outils pour la ligne de commande tels `cbac_validity.sh` qui permet de vérifier la validité d'un composant et `cbac_requirements.sh` qui permet d'extraire les autorisations nécessaires. La bibliothèque est utilisée entre autre avec l'implantation Apache Felix[2] d'OSGi. Un patch a été réalisé pour Felix 1.0.0. Sa taille est de 66,2 Ko.

Les politiques de sécurité CBAC sont définies comme suit : les méthodes sensibles, ainsi que les meta-données sensibles, sont identifiées. Ensuite, les autorisations d'exécution sont spécifiées pour chaque fournisseur de composant. Cette approche déclarative permet de faire évoluer les politiques au fil de la vie du système.

Le tableau 5 montre un exemple de fichier de politique CBAC.

Les méthodes définies comme sensibles (`sensitiveMethods`) sont ici les méthodes de manipulation de flux de données, ainsi que les opérations liées à la sécurité. Les méta-données sensibles (`sensitiveManifestAttributes`) sont constituées des fragments OSGi, qui permettent d'exécuter un composant dans le même espace de nommage qu'un autre déjà installé. Le fournisseur de composant Bob a l'autorisation d'utiliser les méthodes de manipulation de flux de données et certaines méthodes liées à la sécurité.

La figure 1 montre les performances de CBAC, associé à la validation de signature numérique, pour les bundles développés dans le cadre du projet Felix.

La durée d'une validation CBAC est comprise entre 20 ms, pour les petits composants qui sont de loin les plus nombreux dans la distribution Felix, et 120 ms pour les composants de grande taille. Dans les deux cas, cette durée est très inférieure au temps nécessaire pour valider la signature numérique. L'impact de CBAC sur les performances du système est donc très réduite, de l'ordre de 5 % du temps de l'installation.

La limitation principale de l'implantation actuelle de CBAC est la granularité importante de la validation réalisée par le biais de listes d'appels contenus dans les composants. Ceci risque d'imposer un nombre important de faux positifs, c'est-à-dire des appels présents dans le code mais non effectués. En particulier, les appels présents

1. http ://www.rzo.free.fr/parrend08cbac.php
2. http ://felix.apache.org/

```
sensitiveMethods {
 java.io.ObjectInputStream.defaultReadObject;
 java.io.ObjectInputStream.writeInt;
 java.security.*;
 java.security.KeyStore.*;
 java.io.FileOutputStream.<init>;
};

sensitiveManifestAttributes {
  Fragment-Host;
}

grant Signer:bob {
  Fragment-Host;
  java.io.ObjectInputStream.defaultReadObject;
  java.io.ObjectInputStream.writeInt;
  java.io.FileOutputStream.<init>;
  java.security.Security.addProvider;
  java.security.NoSuchAlgorithmException.<init>;
  java.security.KeyStore.getInstance;
};
```

Tableau 5. *Exemple d'un fichier de politique CBAC*

dans les dépendances sont actuellement estimés de manière peu précise. La solution sera d'utiliser des techniques d'analyse de code plus avancées comme les graphes de flux de contrôle ou les graphes de flux de données. Les performances risquent d'en être affectées.

L'apport de CBAC est de proposer un mécanisme de sécurité qui permet la définition de politiques de sécurité extensibles avec un coût en performance raisonnable.

4.3. *WCA : identification des composants vulnérables*

L'objectif de WCA (Weak Component Analysis) est de prévenir l'installation de composants vulnérables et donc susceptibles d'être exploités par du code malicieux. Les vulnérabilités visées sont celles identifiées dans la taxonomie 'implantation des vulnérabilités des composants'. Il s'agit de vérifier l'absence de ces vulnérabilités dans le code d'accès, que le composant met à disposition des autres : les classes contenues dans des packages exportés, ainsi que les services publiés localement.

Son principe est le suivant. Pour chaque classe du code d'accès, la structure de la classe et le code des méthodes sont analysée pour identifier la présence de propriétés

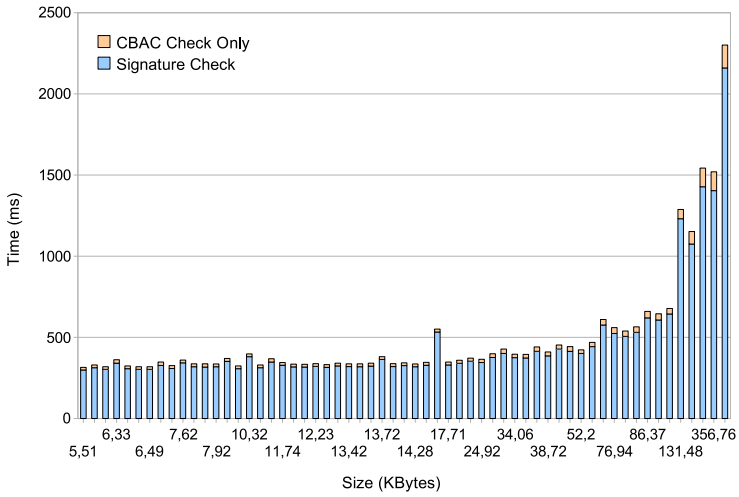

Figure 1. *Performances de l'analyse statique de composants : signature numérique et validation CBAC pour des composants malicieux*

susceptibles de créer une vulnérabilité. Chaque vulnérabilité est définie par un patron de vulnérabilité qui définit les combinaisons de propriétés qui sont exploitables. Un fichier de politique permet de configurer la réaction de WCA lors de l'identification d'une vulnérabilité : le stockage de l'information en vue de la génération d'un rapport, ou l'échec de la validation.

Le tableau 6 présente un exemple de patron de vulnérabilité utilisé avec l'outil WCA : l'‘appel de méthode synchronisé'.

Dans ce patron, chaque vulnérabilité est définie par un ensemble de propriétés dont la présence simultanée permet l'exploitation du code par un composant tiers. Elle est associée à un message d'erreur permettant aux développeurs ou aux administrateurs de connaitre l'origine de l'erreur.

Le tableau 7 présente une classe vulnérable, selon le patron précédent.

Une telle méthode synchronisée sera considérée comme une vulnérabilité par WCA.

Comme CBAC, WCA est disponible à la fois sous forme de bibliothèque Java, par exemple pour l'intégration avec une implantation d'OSGi, et sous forme d'outils en ligne de commande. Trois versions existent. Une version XML permet de modifier les définitions de vulnérabilités et de sélectionner celles qui doivent être identifiées. Un fichier de politique, également au format XML, définit la réaction de WCA lors de la détection d'une vulnérabilité. La taille de la bibliothèque XML est de 60,6 Ko. Une version ‘hardcoded' contient une définition en Java des patrons de vulnéra-

```
  <vs:vulnerability>
    <vs:vulnerabilityRef>
      <vs:catalog_id>wb</vs:catalog_id>
      <vs:src_ref>java</vs:src_ref>
      <vs:id>44</vs:id>
    </vs:vulnerabilityRef>
    <vs:message>Synchronized method call. If the method call is
blocked for any reason (infinite loop during execution, or delay
 due to an unavailable remote resource), all subsequent clients
that call this method are freezed.
    </vs:message>
    <vs:weaknessScope>publicClasses</vs:weaknessScope>
    <vs:method>
      <vs:access>synchronized</vs:access>
    </vs:method>
  </vs:vulnerability>
```

Tableau 6. *Exemple de patron de vulnérabilité : l''appel de méthode synchronisé'*

```
public class FileManagementImpl
{
  public synchronized File getFileFromNetwork(String name,
                                              String network)
  {
    //method code
  }
}
```

Tableau 7. *Exemple d'une vulnérabilité : méthode* `synchronized`

bilité, afin d'augmenter les performances. Le fichier de politique est conservé afin de définir la réaction du système selon le type de vulnérabilité identifié. Une version 'hardcoded.defaultreject' est configurée pour rejeter systématiquement les composants vulnérables. Son objectif est de limiter le coût en performance au maximum quand la validation est effectuée à l'installation du composant dans une plate-forme de services.

La figure 2 montre les performances de WCA pour un ensemble de composants vulnérables tests.

Le surcoût est compris entre 1 et 5 ms pour la validation par WCA 'hardcoded.defaultreject', entre 10 et 20 ms pour la validation par WCA 'hardcoded' et

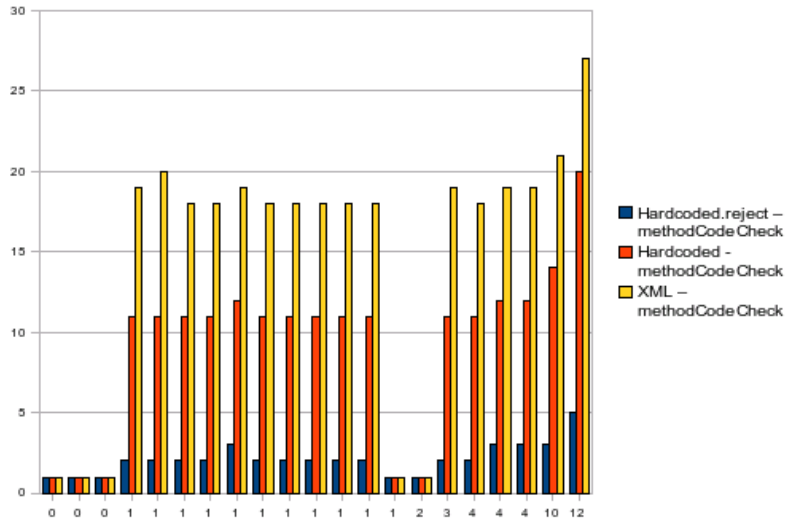

Figure 2. *Performances de l'analyse statique de composants : validation WCA pour des composants vulnérables*

entre 18 et 27 ms pour la validation par WCA 'XML'. Le coût est négligeable, de l'ordre de 1 ms, jusqu'à 4 ms pour les très gros composants non représentés ici, quand aucune classe d'accès n'est définie dans le composant. Les classes d'accès comprennent les packages exportés et les services accessibles localement. Lorsque la validation a lieu, elle est impactée par le coût de lecture des politiques de sécurité : 10 ms par fichier, et donc 20 ms dans le cas de WCA 'XML'. Ce coût n'est présent que lors de la première validation. Dans le cas d'une validation dans une plate-forme dynamique de services, il sera donc réduit d'autant à partir du deuxième composant installé. La faiblesse de ce surcoût se confirme pour des ensembles de composants plus importants comme ceux existant dans le projet Felix.

La principale limitation de WCA est la simplicité du format du patron de vulnérabilité qui ne permet pas d'exprimer des propriétés plus complexes. D'autres techniques devront être envisagées : l'utilisation de machines à état pour représenter l'exécution des méthodes de la JVM, de graphes de contrôles de flux et de contrôle de données (Hovemeyer *et al.,* 2004).

L'apport de WCA est de filtrer les composants lors de leur installation afin de n'installer que ceux qui sont libres de vulnérabilités communes. Ceci permet d'augmenter la robustesse du système, et donc son niveau de sécurité. Il peut être utilisé à la fois comme outil de développement ou comme protection lors de l'installation des composants.

Mécanisme de sécurité	# d'erreurs protégées	# d'erreurs connues	Taux de Protection
CBAC (plate-forme)	16	32	50 %
WBA (composants)	12	33	36 %
CBAC + WBA (composants)	14	33	42 %
CBAC + WBA (tout)	30	65	46 %

Tableau 8. *Taux de protection pour les mécanismes CBAC et WBA*
[Ces résultats concernent un sous-ensemble des mécanismes de sécurité possibles. CBAC et WBA doivent être utilisés en combinaison avec d'autres mécanismes afin de garantir un bon niveau de sécurité]

4.4. *Niveau de securité de CBAC et WCA*

Une fois que des mécanismes de sécurité adaptés aux plates-formes dynamiques sont définis, il est possible d'évaluer le niveau de sécurité qu'ils apportent pour le système. Ces mécanismes ont comme objectif de résoudre un certain nombre de vulnérabilités non protégées dans le système cible. Il s'agit de lever des verrous technologiques, et non de proposer des solutions à l'ensemble des risques connus. Par ailleurs, comme il est communément admis dans la communauté travaillant dans la sécurité, il n'y a pas de 'Silver Bullet' (Brooks, 1987), c'est-à-dire de mécanisme omnipotent. Un haut niveau de sécurité ne peut être obtenu que par la combinaison de plusieurs de ces mécanismes, comme nous le montrons dans la section 5.

Le tableau 8 donne les valeurs du taux de protection pour les mécanismes CBAC et WCA.

CBAC permet de protéger 50 % des vulnérabilités identifiées dans le catalogue 'Bundles malicieux' exploitant la plate-forme. Il permet également de protéger 2 vulnérabilités du catalogue 'Bundles vulnérables'. En ce qui concerne les vulnérabilités de la plate-forme, il s'agit d'une amélioration notable qui peut être renforcée par d'autres mécanismes de sécurité. WCA permet de protéger 36 % des vulnérabilités des composants. Dans la mesure où aucun des mécanismes connus d'analyse de code tels FindBugs (Hovemeyer *et al.,* 2004) ou PMD (Copeland, 2005) ne permet de les identifier, il s'agit d'un progrès important, même s'il ne peut pas être considéré comme suffisant.

CBAC et WCA représentent donc des avancées importantes pour la sécurisation des plates-formes dynamiques de services, et ceci à un coût minime en terme de performances. Chacun de ces outils a une marge importante d'amélioration : CBAC comme WCA peuvent être étendus avec des méthodes d'analyse de code de granularité plus fine comme les graphes de contrôle de flux, qui permettent de représenter la structure logique du programme, ou de contrôle de données, qui permettent de représenter la propagation des données.

Cependant, il n'est possible d'obtenir un haut niveau de sécurité pour un environnement d'exécution que par des techniques d'ingénierie de code. Il est nécessaire d'adapter la plate-forme sur laquelle les composants sont exécutés afin de la rendre plus robuste et de protéger les vulnérabilités qui ne peuvent pas être aisément détectées au niveau du code.

5. Expérimentation : réalisation d'un environment d'exécution sécurisé pour composants à service

L'analyse statique de code s'avère être un outil puissant pour améliorer le niveau de sécurité des plates-formes dynamiques à composants. Cependant, certaines vulnérabilités ne peuvent pas être identifiées et donc protégées de cette manière. Il s'agit en particulier des vulnérabilités de la plate-forme elle-même et de celles liées à la consommation de ressources. Les premières doivent être corrigées. Les secondes ne peuvent être prévenue que lors de l'exécution des composants.

5.1. *Architecture*

Nous proposons d'exécuter les composants à service sur une plate-forme Java/OSGi sécurisée. L'implantation OSGi est remplacée par Hardened OSGi, que nous définissons dans (Parrend *et al.,* 2007). La machine virtuelle est remplacée par la JnJVM (Thomas *et al.,* 2008), qui est une machine virtuelle Java qui intégre un modèle d'isolation dédié pour OSGi (Geoffray *et al.,* 2008) : chaque chargeur de classes, et donc chaque bundle, est exécuté dans un 'Isolate' spécifique.

La figure 3 montre l'architecture d'une implantation Java/OSGi sécurisée : CBAC et WCA sont executés avec une plate-forme robuste composée de Hardened OSGi et de la JnJVM.

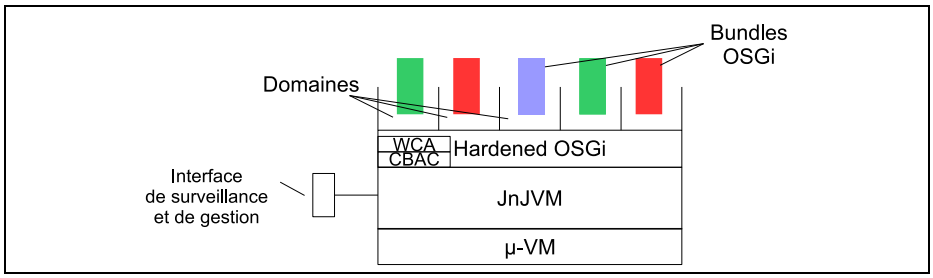

Figure 3. *CBAC et WCA executé avec Hardened OSGi/JnJVM*

Hardened OSGi consiste en l'implantation d'un ensemble de recommendations dans une plate-forme OSGi. Son objectif est de résoudre les vulnérabilités liées à ce que nous considérons comme des erreurs dans la plate-forme, c'est-à-dire des mécanismes qui peuvent être corrigés et ne nécessitent pas la mise en place de contrôle

d'accès. Ces recommendations visent à améliorer l'implantation des plates-formes. Elles sont les suivantes : ne pas rejeter les imports dupliqués, vérifier la taille des bundles avant le téléchargement et avant l'installation, pour ne pas surcharger le disque dans le cas d'appareils à mémoire restreinte, vérifier la signature numérique selon la spécification OSGi, et non pas avec les mécanismes proposés par la JVM par défaut qui sont moins stricts, lancer l'activateur des bundles dans un thread spécifique pour les JVM supportant les threads, nettoyer l'ensemble des données liées à un bundle lors de la désinstallation pour éviter la présente de données 'zombie' non accessibles, et fixer un seuil au nombre de services que chaque bundle peut publier. Les trois couches d'OSGi, cycle de vie, module et service sont concernées. Notre implantation Hardened Felix est une extension de Apache Felix.

La JnJVM est, plus qu'une implantation de la JVM, un outil de génération de machines virtuelles. Ces machines virtuelles sont exécutés au dessus de la Micro-Virtual Machine. Elle permet de combiner différents modules d'exécution, par exemples des 'Garbage Collectors' ou des compilateurs 'Just In Time'. Elle dispose d'un mode spécifique, le mode `service`, qui met en œuvre le modèle d'isolation défini dans (Geoffray *et al.,* 2008). Les propriétés d'isolation sont les suivantes. Chaque bundle dispose d'une copie des variables statiques systèmes, qui sont partagées par défaut, afin d'éviter des éventuelles interférences. Il fait l'objet d'un décompte des ressources utilisées : nombre de threads, temps CPU et mémoire. Des maxima peuvent être fixés, ce qui permet de limiter les opérations des bundles trop gourmands ou de les désinstaller si besoin. Les isolats JnJVM sont donc plus flexibles que ceux définis par exemple par le JSR 121 de Sun (Bryce, 2004) : ils supportent la communication par appel de méthode entre bundles en enregistrant les services dans une mémoire partagée, ce qui garantit des performances bien supérieures aux IPCs dans la proposition de Sun.

5.2. *Quantification du niveau de sécurité*

Une nouvelle évaluation du niveau de sécurité obtenu avec une plate-forme dynamique de services robuste est effectuée. Il s'agit de valider l'architecture proposée, composée de Hardened OSGi et de la JVM, avec les outils CBAC et WCA.

Le tableau 9 donne les valeurs du taux de protection pour un environnement Java/OSGi sécurisé, avec CBAC, WCA, Hardened OSGi et JnJVM.

Hardened OSGi protège de 25 % des vulnérabilités du catalogue 'Bundles malicieux'. Il s'agit des vulnérabilités spécifiques à OSGi qui sont des erreurs, et non des fonctions dangereuses. La JnJVM protège de 44 % des vulnérabilités de ce même catalogue. Il s'agit des vulnérabilités liées à la JVM et en particulier de celles liées à l'usage excessif de ressources du système. La combination JnJVM + Hardened OSGi + CBAC protège de 94 % des vulnérabilités de la plate-forme. Les deux vulnérabilités non protégées sont liées pour l'une à la présence de méta-données non valides dans le fichier Manifest, qui conduit à un déni de service potentiel du bundle contre lui-même, et l'absence de validation du flux (Workflow) des services, qui sont découverts

Mécanisme de sécurité	**# d'erreurs protégées**	**# d'erreurs connues**	**Taux de Protection**
Hardened OSGi (HO)	8	32	25 %
JnJVM	14	32	44 %
HO + JnJVM	18	32	56 %
HO + CBAC + JnJVM (plate-forme)	30	32	94 %
HO + CBAC + JnJVM + WBA (tout)	44	65	68 %
HO + CBAC + JnJVM + WBA + Revue de code manuelle	65*	65	100* %
*Le niveau de sécurité obtenu par une revue de code manuelle ne peut pas être considéré comme une garantie en soi : un auditeur humain est capable d'identifier tous les types de vulnérabilités qu'il connait, mais n'identifiera très probablement pas toutes leurs occurrences dans une application.			

Tableau 9. *Taux de protection pour un environnement Java/OSGi sécurisé, avec CBAC, WBA, Hardened OSGi et JnJVM*

dynamiquement et donc susceptibles d'être instables. La première peut ne pas être considérée comme un vulnérabilité, dans la mesure où elle ne permet pas l'attaque par d'autres composants. La deuxième illustre la nécessité d'étudier les propriétés de sécurité specifiques aux workflows de services, qui sont pour le moment méconnues.

En ce qui concerne l'ensemble des vulnérabilités identifiées sur la plate-forme dynamique de services Java/OSGi, la combinaison JnJVM + Hardened OSGi + CBAC + WCA fournit un taux de protection de 68 %. Les vulnérabilités présentes dans les composants étant pour un certain nombre d'entre elles complexes, leur identification nécessite le recours à une revue de code manuelle. Cette approche permet théoriquement de trouver toutes les vulnérabilités connues des auditeurs, mais souffre des défauts de l'intervention humaine. Elle n'apporte pas de garantie quant à l'absence effective de vulnérabilités. Par ailleurs, ceci implique que le code source des composants soit disponible, et induit donc une forte restriction dans le modèle de déploiement des composants OSGi en particulier. L'implémentation de WCA doit donc être rafinée afin d'automatiser le processus et de supporter la validation des composants lors de leur installation.

6. Conclusions et Perspectives

Nous proposons d'améliorer le niveau de sécurité des plates-formes dynamique de services par la méthodologie suivante. Tout d'abord, les vulnérabilités des plates-formes et des composants qu'elles exécutent sont identifiées et implémentées pour prouver leur facilité exploitation. Des taxonomies sont définies pour les décrire. En-

suite, des modèles d'analyse statique de code sont définis pour identifier et prévenir ces vulnérabilités à l'installation des composants. Il s'agit de modèles relativement simples, dans la mesure où nous considérons que l'effort principal pour sécuriser un système est de connaître ses faiblesses. Ils sont amenés à évoluer pour gagner en expressivité. Leur principe et le gain en performance apporté par rapport aux mécanismes de sécurité à l'exécution sont validés par les expérimentations proposées. Afin d'obtenir un système sécurisé, ces techniques d'ingénierie de code sont combinées avec des techniques liées à la machine virtuelle afin de résoudre un certain nombre de vulnérabilités non protégées : les faiblesses d'implantation de la plate-forme et le contrôle de ressources en particulier. L'évaluation du niveau de sécurité pour les plates-formes montre qu'un taux de protection de 94 % est obtenu en combinant ces différentes approches. En ce qui concerne les vulnérabilités des composants, nos outils sont les seuls qui existent. Ils permettent un taux de protection de 42 %. Seule une revue manuelle de code permet d'obtenir de meilleurs résultats.

Les perspectives de ce travail sont nombreuses. Premièrement, les catalogues de vulnérabilités sont amenés à être enrichis selon l'expérience de la communauté. Nous ne prétendons pas à l'exhaustivité, même si l'objectif est de rassembler le plus grand nombre possible de vulnérabilités identifiées dans la littérature et par notre propre expérience. Ensuite, les outils d'analyse de code développés peuvent être enrichis. L'utilisation de machines à état, de graphes de contrôle de flux et de données permettra d'augmenter le nombre de vulnérabilités identifiées. Enfin, des analyses similaires pourront être effectuées pour d'autres types de systèmes : d'autres plates-formes Java, par exemple en se focalisant sur les propriétés de sécurité des flux (Workflows) de services accessibles localement, qui s'avèrent introduire des risques par eux-même, ou bien des plates-formes d'autres langages, comme .Net, ou des langages dont le développement est plus récent comme Python ou Ruby.

7. Bibliographie

Bieber G., Carpenter J., « Introduction to Service-Oriented Programming (Rev 2.1) », OpenWings Whitepaper, April, 2001.

Brooks F. P., « No Silver Bullet : Essence and Accidents of Software Engineering », *Computer*, vol. 20, n° 4, p. 10-19, April, 1987.

Bruneton E., Lenglet R., Coupaye T., « ASM : a code manipulation tool to implement adaptable systems », *Adaptable and Extensible Component Systems Conference*, Grenoble, France, November, 2002.

Bryce C., « Isolates : A New Approach to Multi-Programming in Java Platforms », LogOn Technology Transfer OT Land Whitepaper, http ://www.bitser.net/isolate-interest/papers/bryce-05.04.pdf, May, 2004.

Copeland T., *PMD Applied*, Centennial Books, November, 2005. ISBN : 0-9762214-1-1.

Geoffray N., Thomas G., Clément C., Folliot B., « Towards a new Isolation Abstraction for OSGi », *First Workshop on Isolation and Integration in Embedded Systems (IIES 2008)*, p. 41-45, April, 2008.

Gong L., Ellison G., Dadgeforde M., *Inside Java 2 Platform Security - Architecture, API Design, and Implementation, Second Edition*, Addison-Wesley, 2003. ISBN-13 : 978-0201787917.

Hovemeyer D., Pugh W., « Finding bugs is easy », *ACM SIGPLAN Notices*, vol. 39, p. 92 - 106, 2004. COLUMN : OOPSLA onward.

Howard M., Pincus J., , Wing J. M., *Computer Security in the 21st Century*, Springer, chapter Measuring Relative Attack Surfaces, p. 109-137, March, 2005.

Lai C., « Java Insecurity : Accounting for Subtleties That Can Compromise Code », *IEEE Software*, vol. 25, n° 1, p. 13-19, 2008.

Lindqvist U., Jonsson E., « How to Systematically Classify Computer Security Intrusions », *IEEE Symposium on Security and Privacy*, p. 154-163, May, 1997.

Long F., Software Vulnerabilities in Java, Technical Report n° CMU/SEI-2005-TN-044, Carnegie Mellon University, October, 2005.

Miller M. S., Tulloh B., , Shapiro J., « The Structure of Authority - Why security is not a separable concern », An invited talk given at Second International Mozart/Oz Conference, http ://www.cetic.be/moz2004/, October, 2004.

Necula G. C., « Proof-Carrying Code », *Conference Record of POPL '97 : The 24th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, Paris, France, p. 106-119, jan, 1997.

OSGi Alliance, « OSGi Service Platform, Core Specification Release 4.1 », Draft, May, 2007.

Parrend P., Frénot S., Java Components Vulnerabilities - An Experimental Classification Targeted at the OSGi Platform, Research Report n° RR-6231, INRIA, 06, 2007.

Parrend P., Frénot S., « Classification of Component Vulnerabilities in Java Service Oriented Programming (SOP) Platforms », *Conference on Component-based Software Engineering (CBSE'2008)*, vol. 5282/2008 of *LNCS*, Springer Berlin / Heidelberg, October, 2008a.

Parrend P., Frenot S., « Component-based Access Control : Secure Software Composition through Static Analysis », *in* , E. Tanter, , C. Pautasso (eds), *Software Composition*, vol. 4954/2008 of *LNCS*, Springer Berlin / Heidelberg, Budapest, p. 68-83, March, 2008b.

Parrend P., Frénot S., More Vulnerabilities in the Java/OSGi Platform : A Focus on Bundle Interactions, Technical Report n° RR-6649, INRIA, September, 2008c.

Saltzer J. H., Schroeder M. D., « The Protection of Information in Computer Systems », *Fourth ACM Symposium on Operating System Principles*, October, 1973.

The Last Stage of Delirium. Research Group., « Java and Java Virtual Machine. Security vulnerabilities and their exploitation techniques. », *Black Hat Briefings*, 2002.

Thomas G., Geoffray N., Clement C., Folliot B., « Designing highly flexible virtual machines : the JnJVM experience », *Software : Practice & Experience*, John Wiley & Sons, Ltd., 2008.

Article reçu le 31/08/2008.
Version révisée le ? ?/ ? ?/2008.
Rédacteur responsable : NAME, SURNAME

SERVICE ÉDITORIAL – HERMES-LAVOISIER
14 rue de Provigny, F-94236 Cachan cedex
Tél : 01-47-40-67-67
E-mail : revues@lavoisier.fr
Serveur web : http://www.revuesonline.com

ANNEXE POUR LE SERVICE FABRICATION

A FOURNIR PAR LES AUTEURS AVEC UN EXEMPLAIRE PAPIER
DE LEUR ARTICLE ET LE COPYRIGHT SIGNE PAR COURRIER
LE FICHIER PDF CORRESPONDANT SERA ENVOYE PAR E-MAIL

1. ARTICLE POUR LA REVUE :
   L'objet. Volume 8 – n°2/2008

2. AUTEURS :
   Pierre Parrend, Stéphane Frénot

3. TITRE DE L'ARTICLE :
   Vérification automatique pour l'exécution sécurisée de composants Java

4. TITRE ABRÉGÉ POUR LE HAUT DE PAGE MOINS DE 40 SIGNES :
   Vérification de composants Java

5. DATE DE CETTE VERSION :
   25 novembre 2008

6. COORDONNÉES DES AUTEURS :

   – adresse postale :

   INRIA ARES / CITI, INSA-Lyon, F-69621, France
   tel. +334 72 43 71 29 - fax. +334 72 43 62 27

   – téléphone : 04 72 43 71 29

   – télécopie : 00 00 00 00 00

   – e-mail : pierre.parrend@insa-lyon.fr

7. LOGICIEL UTILISÉ POUR LA PRÉPARATION DE CET ARTICLE :
   LaTeX, avec le fichier de style `article-hermes.cls`,
   version 1.2 du 03/03/2005.

8. FORMULAIRE DE COPYRIGHT :
   Retourner le formulaire de copyright signé par les auteurs, téléchargé sur :
   `http://www.revuesonline.com`

SERVICE ÉDITORIAL – HERMES-LAVOISIER
14 rue de Provigny, F-94236 Cachan cedex
Tél : 01-47-40-67-67
E-mail : revues@lavoisier.fr
Serveur web : http://www.revuesonline.com