N° d'ordre : 2008 ISAL 0117 Année 2008

Thèse

Software Security Models for Service-Oriented Programming (SOP) Platforms

À présenter devant

L'Institut National des Sciences Appliquées de Lyon
École Doctorale Informatique et Information pour la Société

pour l'obtention du

Grade de Docteur

spécialité informatique

par

Pierre Parrend

À soutenir le 9 Décembre 2008

Devant le jury composé de

Président :	Prenom Nom,	Titre,	QQpart
Directeurs :	Stéphane Ubéda,	Professeur,	INSA de Lyon
	Stéphane Frénot,	Maître de conférences,	INSA de Lyon
Rapporteurs :	Didier Donsez,	Professeur,	Université Grenoble I
	Ralf Reussner,	Professeur,	Technische Universität Karlsruhe
Examinateurs :	Ciaran Bryce,	Maître d'enseignement et de recherche,	Université de Genève
	Pierre-Etienne Moreau,	Chargé de Recherche, habilité à diriger des recherches,	INRIA Lorraine

Thèse effectuée au sein du Centre d'Innovation en Télécommunications et Intégration de Services (CITI) de l'INSA de Lyon et de l'équipe Architecture Réseaux et Systèmes (Ares) de l'INRIA Rhône-Alpes

Résumé

Les plates-formes dynamiques de services (SOP, pour 'service-oriented programming') sont des environnements d'exécution génériques qui définissent un modèle d'architecture logicielle structuré: les composants communiquent par le biais d'interfaces explicites, ce qui facilite la configuration et l'évolution de ces systèmes. Les plates-formes SOP utilisent leur environnement réseau pour réaliser des tâches fonctionnelles, mais également pour améliorer leur capacité de gestion et d'évolution. Elles sont exploitées dans des contextes variés, des serveurs d'application aux systèmes embarqués médicaux ou automobiles. La flexibilité apportée par les plates-formes SOP permet l'intégration de composants de plusieurs sources aussi bien lors de la conception qu'à l'exécution.

Cette tendance induit cependant un risque important. Peu d'outils existent pour évaluer la qualité des systèmes résultants, et aucun ne garantit que les composants sélectionnés ne sont pas malicieux. Dans des contextes applicatifs tels que les systèmes e-Business ou les systèmes embarqués sensibles, l'existence d'attaques n'est pas à exclure.

L'assurance de sécurité logicielle (Software Security Assurance) définit des méthodes pour le développement d'applications sûres, mais se concentre sur les systèmes monolithiques. Son principe est le suivant: les vulnérabilités doivent être identifiées et résolues tôt dans le cycle de vie pour éviter les attaques lors de l'exécution et limiter les coûts de réparation. Cependant, cette approche ne peut s'appliquer directement aux applications à composants, où le développement n'est pas nécessairement contrôlé par l'intégrateur, et où l'intégration peut avoir lieu à l'exécution de manière automatisée.

Nous proposons par conséquent de réaliser une analyse de sécurité pour une plate-forme SOP de référence, la plate-forme OSGi, et de fournir des mécanismes de protection adaptés aux besoins ainsi identifiés.

L'analyse de sécurité de la plate-forme OSGi est réalisée par une méthode spécifique, $\mathcal{SPIP}$, le *Processus Spirale de Prévention d'Intrusion* (*Spiral Process for Intrusion Prevention*). Elle permet l'évaluation des vulnérabilités du système cible et de la protection apportée par les mécanismes de sécurité associés. Le résultat de l'analyse est : les vulnérabilités de la plate-forme Java/OSGi, et les vulnérabilités des composants SOP Java.

Plusieurs mécanismes de protection sont développés pour prévenir l'exploitation des vulnérabilités identifiées. Ils sont implémentés dans la plate-forme elle-même et au niveau des composants. *OSGi Robuste* (*Hardened OSGi*) est un ensemble de recommandations pour la mise en œuvre de plates-formes OSGi résistantes. *CBAC*, le contrôle d'accès basé composants (Component-based Access Control) est un mécanisme de contrôle d'accès qui vérifie lors de l'installation qu'un composant n'exécute que les appels explicitement autorisés. Son objectif est d'être plus flexible que le gestion de sécurité Java, de garantir que seuls les composants valides soient installés et de réduire autant que possible le coût de vérification en terme de performance. *WCA*, l'analyse de composants faibles (Weak Component Analysis), est un outil pour identifier les vulnérabilités exploitables dans les composants SOP selon l'exposition des classes: les objets partagés tels les services SOP, les classes partagées et les classes internes

des composants ne sont pas concernés par les mêmes vulnérabilités.

Nos propositions sont validées par leur intégration avec une JVM sécurisée dédiée aux applications OSGi, la JnJVM. Les propriétés de sécurité de l'environnement ainsi réalisé sont encourageantes.

Une présentation de cette thèse en français est disponible dans [PF08d].

Summary

Service-oriented programming (SOP) platforms are generic execution environments enforcing a proper architectural model for applications: software components communicate through well-defined interfaces, which eases the configuration and evolution of applications. These platforms take advantage of their networked environment to perform distributed functional tasks, but also to enhance their management and evolution capacity. They are involved in numerous contexts, from applications servers to embedded health-care and automotive systems. The increased flexibility brought in by SOP platforms enables to integrate components provided by different issuers during the design phase and even at runtime.

This trend has nonetheless a serious drawback. Few tools exist to assess the actual quality of the resulting systems, and none is available to guarantee that the selected components do not perform malicious actions. In applications such as e-Business systems or sensitive embedded systems, the intervention of attackers can not be excluded.

Software Security Assurance provides methods for the development of secure applications, but focuses on monolithic systems. Its principle is the following one: vulnerabilities should be identified and solved as early as possible in the life-cycle to avoid runtime abuses and to reduce patching costs. However, this approach is not well-suited for component applications: the development process is not controlled by the integrator. When the integration is performed at runtime, no human intervention is possible to evaluate the quality of the components.

We therefore propose to perform a security analysis of one prototypical SOP platform, the OSGi platform, and to provide protection mechanisms tailored to the identified requirements.

The security analysis of the OSGi platform is performed with a dedicated method we define for security benchmarking, $\mathcal{SPIP}$, the *Spiral Process for Intrusion Prevention*. It supports the assessment of vulnerabilities of the target system and of the protective power of associated security mechanisms. The output of the analysis is: the vulnerabilities of the Java/OSGi platform, and the vulnerabilities of Java SOP components.

Several protections mechanisms are developed to prevent the exploitation of identified vulnerabilities. They are implemented in the platform itself and at the component level. *Hardened OSGi* is a set of recommendations for building more robust implementations of the OSGi platform. *CBAC*, Component-based Access Control, is an access control mechanism that verifies at install time that a component only performs calls it is authorized to. It intends to be more flexible than the Java security manager, to ensure that policy-compliant components only are installed and to reduce as much as possible the verification performance overhead. *WCA*, Weak Component Analysis, is a tool for identifying exploitable vulnerabilities in SOP components, according to the exposition of classes: shared objects, *i.e.* SOP services, shared classes, and component internal classes are not plagued by the same type of vulnerabilities.

Our propositions are validated through their integration with a secure JVM dedicated to OSGi applications, the JnJVM. The resulting environment proves to have very encouraging security benchmarking results.

Remerciements

Je remercie tout particulièrement Stéphane Frénot, mon directeur de thèse, qui a permis la réalisation de cette thèse dans l'équipe INRIA-Ares, puis INRIA-Amazones, du laboratoire CITI de l'INSA de Lyon. Il a su à la fois me fournir un environnement de recherche et une vision technique forts, et me donner l'autonomie de travailler sur des problématiques complémentaires mais différentes de celles qui existaient dans l'équipe. Merci également à Stéphane Ubéda, en tant que co-directeur de thèse et directeur de laboratoire, qui a fait preuve d'une efficacité constante dans ses deux fonctions, et qui a permit à cette thèse, comme à tant d'autres, de se dérouler dans de bonnes conditions.

Je remercie Didier Donsez, Professeur à l'Université de Grenoble, et Ralf Reussner, Professeur à l'Université de Karslruhe, Allemagne, d'avoir accepté d'être les rapporteurs de ce document. Et en particulier d'avoir permis sa rédaction en anglais.

Je remercie également Ciarán Bryce, de l'Université de Genève, et Pierre-Etienne Moreaux, de l'INRIA Lorraine, de s'être intéressés à mon travail et d'avoir accepté de faire partie du jury de thèse.

Plusieurs personnes ont également contribué au contenu de cette thèse. Je citerai Gaël Thomas pour ses retours concernant la sécurisation de la machine virtuelle Java, Peter Kriens pour ses remarques concernant l'intégration de mes propositions dans les plates-formes OSGi, et Capers Jones pour les précieuses informations qu'il m'a fournies sur les processus de qualité logiciels.

Je remercie l'équipe Middleware du CITI pour leur présence au long de ces trois années, les discussions de recherche et le soutien logistique que tous ont pu m'apporter pour la réalisation et l'organisation de la soutenance de thèse: Yvan Royon, Frédéric Le Mouël, Amira Ben Amida, Hajer Chamekh, Noha Ibrahim.

Je remercie également tous ceux qui m'ont fait découvrir le monde de la recherche : Bertrand David, Professeur à l'Ecole Centrale de Lyon, qui m'a encadré pendant le Master recherche, ainsi que Isabelle Augé Blum, du CITI, avec qui j'ai réalisé mon projet de fin d'étude d'Ingénieur INSA, sans qui je n'aurais sans doute pas décidé de m'orienter dans cette voie.

Un grand merci à tous les amis que j'ai rencontré pendant cette thèse, et à ceux que j'ai retrouvé après ces années de travail parfois un peu trop intensif.

Je remercie enfin mes parents, qui m'ont permis de réaliser un Master recherche après la fin de mes études. Je remercie mes beaux parents d'avoir supporté mes humeurs de thésard ne sachant pas toujours ce qu'il cherche. Et je remercie naturellement Pauline, ma chère et tendre épouse, qui a sacrifié ses soirées et ses week-ends au moins autant que moi pendant cette période, et qui m'a apporté un soutien bien indispensable à l'aboutissement de cette thèse.

Contents

List of Figures

List of Tables

Listings

Terms are used in a manner that aims at being coherent, and at providing a comprehensive understanding of the subject of study. When no consensus arises in the community, they can be used with slightly different meanings than in the work of other authors.

Terms that are defined in Appendix A are marked with a star (*) at their first occurrence in each chapter.

1 Introduction

1.1 The Argument of this Thesis

Dynamic applications are applications that can evolve at runtime, *i.e.* can be reconfigured and extended without the need for rebooting their execution environments*. They can thus gain new functionalities and adapt themselves to their unstable environment. Dynamic applications exploit the component-based software engineering (CBSE) [SGM02] and service-oriented programming* (SOP) [BC01] paradigms to enhance development agility and runtime flexibility. They are often built on top of virtual machines and take advantage of the portability they provide and of the robustness properties of high level languages. These paradigms are integrated in service-oriented programming platforms* which provide component* binding through local services*[1]. Through advanced features such as the discovery of components in the environment, they pave the way for groundbreaking applications in the domains of connected homes, pervasive environments, or application servers.

Dynamic applications can be built according to three trust models:

- Legacy applications, where all components are developed by the same provider.
- Design time integration of applications, where components are developed by various providers and integrated by the application architect. In this case, the architects trust the component providers and can test the code to challenge this trust.
- Automated runtime integration in 'open dynamic applications'. In this case, the components are discovered dynamically in the environment by providers on which little to no control exists. The only guarantee that is usually set is the presence of a digital signature* which certifies that the code is provided by a known entity*. Consequently, no functional or quality evaluation is performed.

The last configuration would enable to build truly dynamic applications which evolve according to their environments. However, this vision would require that guarantees can be set on the component behavior related to component functionalities, quality, or benevolence. Only specific techniques such as proof-carrying-code [Nec97] for algorithmic safety exist so far. They only enable to express specific properties such as the absence of infinite loops or of memory leaks. No comprehensive method exists to manage the security* issues generated by open dynamic applications.

A recent trend in application security* consists in creating software that is provably built-in secure, *i.e.* on which assertions related to the code property can be expressed. It is what

[1] Service-oriented Programming (SOP) provides therefore a perspective on software components. It is not to be confused with Service-oriented architectures (SOA), which enable to interact remotely, although both can be used complementarily

is called 'Software Security'* [McG06]. Its main objectives is to avoid the presence of vulnerabilities* in the architecture and in the code of applications as early as possible in the life-cycle. We consider that the application of such an approach can help increase the security status of dynamic applications, *i.e.* their ability to withstand attacks*, provided their specific properties are taken into account.

Consequently, the work of this thesis is motivated by the following methodological assumption:

> Secure SOP platforms are required to make the vision of dynamic applications a reality. This challenge is made realistic thanks to the intrinsic properties of available SOP platforms but require to enforce additional, new security paradigms such as 'software security'.

The experiments that are reported in this thesis are conducted on the Java/OSGi platform. A common claim in the OSGi community is that this execution environment is a very secure one since it supports a scheme for component isolation that is more complete than many others. It is even sometimes considered as the *universal local middleware*, because it provides facilities for component management that are exploited by several other SOP platforms or Web servers. It is actually designed with promising properties: component isolation through class loaders which enables to load each component is a specific naming space, important set of execution permissions and specific security mechanisms that are pervasive to the specifications. However, very few use cases are reported that would confirm that the OSGi platform is suited to withstand adversarial environments. Moreover, very few tools are broadly available to support the secure execution of OSGi systems*.

The choice of the target system is therefore motivated by the following technical assumption:

> The Java/OSGi platform is a promising but currently immature SOP platform for building secure systems for dynamic applications.

The implementation of protection mechanisms for the OSGi platform must take its specific properties into account:

- Components can be discovered dynamically from the environment, from providers that are not necessarily trusted.
- Security checks can thus only be mandatorily performed when the components are available, *i.e.* when they are downloaded on the platform. They can rely on component Bytecode only.

1.2 Context

1.2.1 The Software Crisis

Middleware development and software engineering are structured around the response they provide to the successive crises of software development. The historical crisis is on the way to be resolved by manageable software such as SOP platforms. As the number of networked

economically sensitive applications increase dramatically, a second crisis is gaining attention: the software security crisis.

The 'Ball of Mud' and the emergence of manageable software The historical 'software crisis' is due to the important complexity of software programs related to the available development methodologies. The term dates back to the 'Software Engineering' NATO Conference of 1968 [Ran96], where the requirements for a proper management of software development processes have first been explicited. The dramatic increase of hardware speed made it hard for developers to build programs that could exploit the available resources while controlling the actual duration of the software projects. The term has been used until the 1990's, where the web forced the number of programmers to explode.

The principles of the requirements for managing the crisis are known since Dijskra's Turing Award lecture, *The Humble Programmer* [Dij72]: systems have to be intellectually manageable; they should be organized hierarchically; it should be possible to extract proof of correctness. These requirements represent two kinds of constraints: they can be either embedded in the programming language, or may require external theorem provers.

Over the year, languages and systems have evolved to avoid the creation of 'Big Balls of Mud' [FY97], *i.e.* monolithic software with deeply intertwined code. Techniques have been developed to enforce better programming models: Object-oriented programming, component-based development. SOP platforms are currently the most complete solution. They rely on efficient execution platforms and enforce programming best practices. They enable to achieve both better productivity and better code quality.

The software security crisis and software security assurance *

As the management of software projects seems to be a concern of less intensity, another crisis is gaining attention: the software security crisis. Since more and more applications are developed for connected devices, the weaknesses that once where simple bugs turn out to be vulnerabilities that enable attacker to abuse the systems.

The response of the software engineering community to the software security crisis is called *software security assurance*. It consists in integrating security measures throughout the development life-cycle to prevent the presence of as much vulnerabilities as possible in the code. Of course, the objective is to limit the cost caused by security failures. Figure 1.1 [Jon99] shows the average of defects found throughout the life-cycle of software and the cost to repair them[2].

One error that costs 1 US dollar ($) to repair during the design phase costs 25 $ during coding, between 100 and 150 $ during tests and up to 16 000 $ if it is corrected post release. This estimation does not take into account the by-products of such errors such as reputation loss.

Strong efforts are therefore advocated to enforce software security early in the life-cycle.

1.2.2 SOP Platforms

SOP platforms are characterized by the programming paradigms they rely on: Virtualization*, modularity* and service-oriented programming. The target system of our analysis, the OSGi platform, is an example of a widespread SOP platform.

[2]The graph is reproduced with the kind authorization of the author

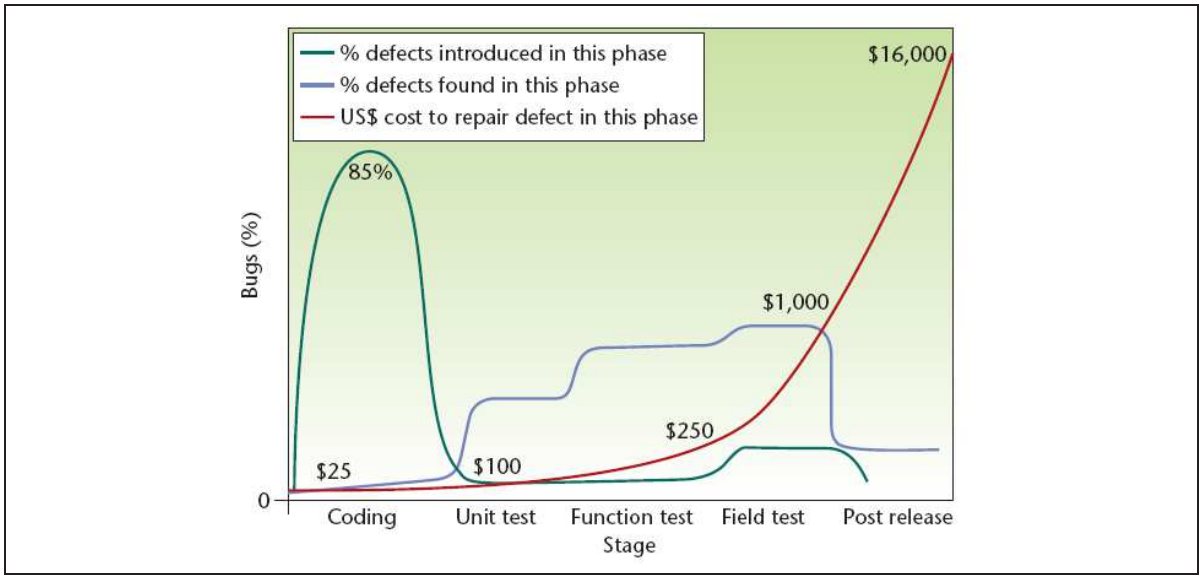


Figure 1.1: Average of defects found throughout the life-cycle of software and the cost to repair them

Virtualization and Modularity Virtualization is a broad term that refers to the abstraction of computer resources. At the application level, it enables to control the execution of full programs while being independent of both the underlying hardware and operating system. It is suited for resource constraint environments: it does not emulate a full computer but only provides required support for executing general purpose applications.

Modularity is a software design technique that increases the extent to which software is composed from separate parts called modules or components. It aims in particular at making these modules re-usable [McI69]. Components can serve different goals: execution unit [Hal01], service unit or deployment unit. These goals can co-exist in a single component model. The type of components that we are concerned about is Off-the-Shelf components* (OTS). OTS is a generic term that appoints readily available components, as opposed to components that are built according to pre-defined constraints. OTS can be commercial OTS (COTS) [CL00, Voa98], open source components, legacy software or components that are discovered at runtime in the environment through repositories such as the OSGi Open Bundle Repository, OBR [AH06].

Service-oriented programming Service-oriented programming [BC01] consists in letting software components installed on the same execution platform communicate through local services. It is presented in Chapter 5. It is built on Object-Oriented-Programming (OOP) and component models, and emphasizes the possibility of improved software re-use [SVS02] and the evolution of applications at runtime.

The OSGi platform One of the platforms of choice for executing dynamic Java applications is the OSGi platform [All07a]. It has originally been designed for lightweight and connected devices, for instance automotive entertainment systems or health care systems. Its clean deployment and programming model fosters its adoption in numerous other contexts such as

application servers. Components can be discovered, installed and managed throughout their life-cycle and can be un-installed to let place to other ones. Moreover the memory footprint of the platform itself is quite reduced.

These mechanisms prove to be useful for a wide range of Java-based systems. More and more application servers such as JBoss, Jonas, Weblogic, and Integrated Development Environments (IDEs) such as Eclipse use it to manage components and their dependencies.

However, and even though the standard Java security model has been tailored for OSGi, few work related to the new security challenges brought in by this platform have been released.

1.2.3 SOP Platforms and Security

SOP platforms exploit the security features of their building blocks, such as the virtual machine and the modular support. Java SOP platforms, of course, heavily rely on the Java security model. However, the highly connected and extensible applications they enable to build introduce new security challenges. The main attack vectors* are presented, as well as the limitations of mainstream software security assurance approaches.

Virtualization and modularity Virtualization provides a layer that isolates the applications from the underlying system. The benefits are manyfold. First, isolation from the underlying operating system enhances the security of the system. Code can no longer access the OS. Only the VM which is smaller and contains well-defined interfaces must be secured. Next, high level languages are often safer as native languages because they support advanced security models and are designed with built-in properties such as type-safety. Moreover, virtualization enables the reuse of secure architectures and thus prevents the multiplication of ad-hoc less tested solutions.

Modularity can be considered in itself as another good practice for building secure systems [MTS04]. Building an application out of small components with strict boundaries between them enhances the isolation between its different parts and makes the system far more testable as a monolithic structure. This increases the stability of the system and makes it less exposed to denial-of-service attacks or ill-defined behavior. Moreover, component models often integrate additional patterns that enhance the security level* of systems that are built on top of them [MTS04] even though they are not explicitly meant to be protection mechanisms. Such patterns are information hiding, abstraction, principle of least authority, absence of global namespace, use of patterns of safe cooperation, to only cite a few.

The Java security model The security in Java SOP applications in based on the Java security model. It is composed by two complementary elements: the default features of the Java language and virtual machine, and the Java security manager.

The default features of the Java language and VM are designed to avoid abuses of the system by malicious programs (See Chapter 4). They are type safety, Bytecode validation at class loading, automated memory management through garbage collection, and modularity. Modularity is enforced by the loading of classes from various origins in different class loaders to prevent undue access and name conflicts.

The Java security manager enforces access control in Java applications. It is based on the notions of 'principal'*, *i.e.* the owner of a code component, and a 'domain', *i.e.* the set of rights a principal has for executing sensitive code. Its principle is the following: in dangerous

methods, calls to the security manager enable to check the availability of sufficient execution rights for all principals of the current call stack.

The Java security model is designed to enable the safe execution of untrusted Applets and in particular to prevent Bytecode forgery. Its support for secure modularity is limited to a comprehensive but not flexible access control mechanism. One of the goals of our work is to determine to what extend it is suitable for dynamic applications, and whether it complies with their specific requirements.

Attack vectors Introducing flexibility in applications also means introducing new ways for malicious actors to exploit the system. For typical applications of SOP platforms such as web applications, multimedia automotive or health systems this is clearly not acceptable and prevents vendors from exploiting the full potential of the platform.

We strongly believe that the dynamism of SOP platforms can only be leveraged if its security implications are well understood and risks of malicious exploitation reduced. At the moment we began this work this was clearly not the case, in particular in the context of the OSGi platform. Our objective is therefore to identify the actual threats to SOP platforms and potential solutions that would enable users to take the best out of this technology.

Figure 1.2 shows an overview of the threats on a SOP system and the subject of this thesis: exploiting weaknesses of SOP platforms.

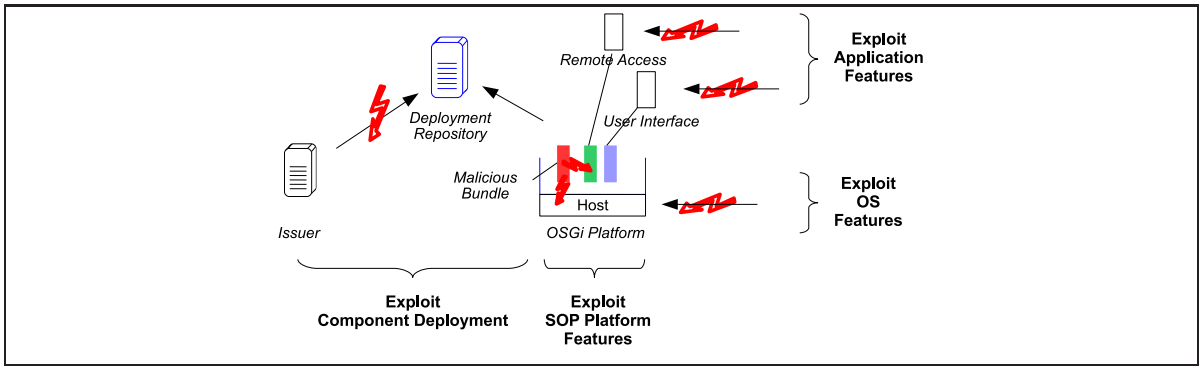


Figure 1.2: The weaknesses in a SOP dynamic application

The following attack vectors can be exploited.

- Bundle* deployment: the code can be intercepted, read and/or modified.
- Application interfaces: Graphical User Interfaces (GUI) and remote interfaces such as RMI or Web Services which are often used in conjunction with the platform.
- Attack against the JVM from the host system.
- Execution of malicious bundles inside the platform.

We choose to focus to the attack vectors specific to SOP platforms. The deployment essentially induce development effort. Execution of malicious bundles needs to be analyzed thoroughly to identify exploitable vulnerabilities it introduces and to propose new protection mechanisms

Limitations of software security assurance approach The 'software security assurance' approach is relatively straightforward for monolithic systems, where the code is developed

by a single organization. For dynamic applications, where the code originates from third party providers, it highlights a paradox: code should be 'built-in' free of vulnerability, but is not necessarily available. Building secure dynamic applications therefore imply to tackle this paradox.

1.3 Document Outline

Defining a secure SOP execution environment involve two complementary challenges: a methodological challenge, and a technical challenge. The methodological challenge is the relevance of the *Software Security* approach to build secure component platforms*. The technical challenge is the suitability of the security level service-oriented programming platforms such as the OSGi platform can provide for its use cases. The study of these challenges is performed through three main steps: a state of the art survey on these topics, the security analysis of the target system and the proposition of a set of security mechanisms that intend to solve the identified vulnerabilities.

Preliminary development A pre-requisite for building secure Java SOP platforms is to ensure that the deployment process is itself secure, *i.e.* that components can not be modified between the time they are released and the time they are installed on a client* platform as presented in Chapter 2. No open source tools exist for the OSGi platform. We therefore developed tools to support verifying the digital signature of OSGi bundles which is more strict than the signature for standard Java Archives and a tool for signing and publishing bundles. The standard security mechanism for secure execution in Java, the enforcement of *Permissions* through a security manager, also need to be supported. This enables the evaluation of the security level that can be provided by an OSGi platform and provides a reference implementation for comparison with our own propositions.

Part I presents a survey of the state of the art of research and technology for software security in Chapter 3, the security mechanisms in the Java world in Chapter 4 and the target system of our study, SOP platforms, in Chapter 5.

The properties of the Software Security approach and related techniques are given in Chapter 3 page 21. This research and technical domain is defined. It concerns specific security aspects that are bound with the properties of the code of the application, rather than network-level, system-level security or security issues that are tackled after the development of applications. The concept of *Software Security Assurance*, *i.e.* the process of ensuring that software is free of vulnerabilities with a certain level of confidence, is introduced. Techniques that can enforce security at the software level are of three main types independent of the considered programming language. The first approach relies on software engineering, *i.e.* modification of the development life-cycle and the second one on code engineering, *i.e.* constraints and modifications on the code.

The security features of Java platforms are presented in Chapter 4 page 39. Several vulnerabilities are well-known and others are disclosed in very recent publications. The default security model for Java platforms is based on `Permissions`. It is also named 'Stack-based Access Control' since it implies the verification of the whole call stack when security checks are performed. This ensures that a method can be executed if all the callers are allowed to

access it. Various extensions to this security model are proposed. They imply either the modification of the platform, the addition of constraints on the code to be executed or behavior injection through code transformation.

SOP platforms are discussed in Chapter 5 page 57. The SOP paradigm is defined. The structure of significant SOP platforms is identified and examples are given. The specific security requirements of this execution environment are identified. Lastly, the OSGi platform and its support for service-oriented programming are presented. Security aspects are discussed.

Part I introduces state of the art methodologies for security analysis, and presents existing protection principles and mechanisms for Java environments. It defines SOP platforms, the target system of our analysis.

Part II presents the security analysis of the target system. A dedicated methodology, $\mathcal{SPIP}$, is defined in Chapter 6. The results of its application on Java SOP platforms, at the example of the OSGi platforms, are given in Chapter 7.

The methodology of this thesis is presented in Chapter 6 page 75. The scientific and technical motivation for this work lies in the absence of efficient tools that enable the realization of secure systems that are based on Java SOP platforms but also on the lack of information related to the vulnerabilities that appear in such execution environments. A specific process for security analysis of complex systems, in particular execution environments, is defined. The $\mathcal{SPIP}$ process is a ***S**piral **P**rocess for **I**ntrusion **P**revention*. It consists in a series of iterations of system security* evaluation and enhancement. Each iteration is built up by a *Vulnerability Assessment** phase where weaknesses are analyzed and a *Protection Assessment** phase where protection mechanisms are evaluated. The application of $\mathcal{SPIP}$ to Java SOP platforms builds the remaining of the thesis. The results of vulnerability assessment for the Java/OSGi SOP platform are presented in Chapter 7 page 87. The first step is to refine the tools for security analysis to make them fine-tuned for our target system: taxonomies and a *descriptive Vulnerability Pattern** for security benchmarking* of component platforms are defined. A more generic metric, the *Protection Rate*, is introduced to quantify the protection level that is brought by each potential security mechanism. It can also be used to compare various implementations of the Java/OSGi SOP platform. The second step is to perform the actual vulnerability assessment. This is performed for the vulnerabilities that are induced by the Java/OSGi SOP platform itself and for the vulnerabilities that originate in the component code.

Part III presents a set of solutions to the security requirements that are identified in previous part. The individual mechanisms we propose are presented in Chapter 8. An integrated secure Java/OSGi execution environment is presented in Chapter 9, and a global security assessment is performed.

The mechanisms for secure execution in the Java/OSGi SOP platform that build the core of our proposition are presented and assessed in Chapter 8 page 115. An example scenario is given to highlight the requirements and benefits of the solutions. The first of these security solutions aims at solving vulnerabilities that originate in the implementation of the OSGi framework itself. It consists in a set of implementation recommendations. The next propositions are implementations of the *Install Time Firewall* Security Pattern*. It consists in performing security verification on the component Bytecode immediately before its installation. This approach has both advantages not to require the co-operation of the code

issuer which may not be benevolent and to avoid runtime overhead. It is already used for approaches such as Proof-Carrying-Code. The first example of the *Install Time Firewall* is the ***C**omponent-**B**ased **A**ccess **C**ontrol*, CBAC. Its objective is to replace Java Permissions which prove to have several serious drawbacks with execution permissions that are computed through static analysis. The second example is the ***W**eak **C**omponent **A**nalysis*, WCA. Its objective is to parse the code of bundles to ensure that the classes that are shared with others are free from vulnerabilities. It is also based on static analysis. Vulnerabilities are defined through a *formal Vulnerability Pattern* for Java classes which is distinct from the descriptive Vulnerability Pattern that is introduced for documentation purpose. Each of these mechanisms is discussed and the protection it provides is quantified. The result of the integration of these propositions with a secure JVM which enforces resource isolation between the class loaders and thus the OSGi bundles is presented in Chapter 9 page 143. The secure JVM that is used is the JnJVM which takes advantages of OSGi class loaders to create Isolates and transparently performs resource and access control. The constraints that need to be guaranteed on Java/OSGi bundles to enforce the maximal security are summarized. The security benchmarking of the integrated system validates the proposition. It highlights requirements that still need to be addressed.

Part IV, Conclusions The results of this work are discussed: security benchmarking processes and tools, taxonomies and security mechanisms that are proposed greatly improve the security status of Java SOP platforms. They pave the way to further research efforts that are required to turn this set of complementary security mechanisms into a prototype that could be exploited in production environments. The state of development is also evoked. So as to ease the exploitation of this work the requirements identified in term of support for industrial use cases as well as in term of research effort are summarized.

As a conclusion, the assumption that lay the main motivation of this thesis, which is presented at the beginning of this first Section, is revisited to take the lessons of the present thesis into account.

Preliminary Development: secure Deployment for the OSGi Platform

2

The objective of this research work is to improve the state of the art of security* mechanisms for the Java SOP platforms* at the example of the OSGi platform. This requires that standard tools are available so as to identify their limitations, possible extensions, as well as relevant security requirements.

In the case of the OSGi platform the publication of security tools is quite limited. This is first due to the fact that industry developments are often kept as added value closed source projects. This is also due to the fact that open source OSGi implementations project mostly focus on functional rather than non functional features.

The development of a secure OSGi ecosystem involves to tackle the two aspects of security for extensible Java SOP platforms: secure deployment and secure execution. Preliminary development for secure deployment involves an OSGi-compliant library for verification of the digital signature* as well as a graphical tool for signing and publishing components*. Preliminary development for secure execution entails the support of bundle*-specific Java permissions.

2.1 Bundle Digital Signature

Digital Signature ensures that the integrity of the bundles is preserved during the deployment process and that the issuers that sign them are authenticated and trusted. We released an open source implementation in the INRIA Gforge, under the name SFelix[1]. This project is an extension of the Apache Felix implementation of the OSGi platform with the subset of the OSGi R.4 Security Layer in charge of the digital signature verification [All05a]. It is presented in [FD06], [PF06a] and [PF07b]. Implementation details are given in [PF06b]. The structure of a signed bundle and the verification algorithm are first presented. Next the criteria of validity of a digital signature are discussed for various tools and environments: the Sun `jarsigner` [Sun03], Java programs with a security manager [GED03], the Felix open source OSGi implementation and the SFelix extension of Felix.

Structure of a signed Bundle Being a Java Archive, an OSGi bundle contains its payload, classes and resources and meta-data in the `META-INF/` directory. Integrity is guaranteed by storing the value of the hash digest value for each file of the archive in the `META-INF/MANIFEST. MF` file. Since the Java 1.5 version all meta-data except the manifest itself are included. So as to enable the co-existence of several signers an intermediate file, the `Signature` file, is created and stored in the `META-INF/` directory. It contains the hash

[1] http://sfelix.gforge.inria.fr/

values of the sections of the `MANIFEST.MF` file: the 'Main-Attributes' and each other manifest entries such as the hash value for each archive file. The digital signature in the cryptographic sense [Sch96] is extracted from this `Signature` file and stored in the `Signature Block` file together with the public key certificate(s) of the signer. The format of this `Signature Block` file is PKCS #7 [BK98], or CMS (Cryptography Message Syntax) [Hou04]. Digital signature uses pairs of hash and public key cryptography algorithms such as SHA-1 and DSA.

Validation Criteria Whereas signing tools are available, tools for verifying the validity of digital signature are not compliant with OSGi R4 specification. This is due to the fact that OSGi requirements for digital signature are stronger than those of standard Java Archives [Sun03], where classes can be added and removed without invalidating the signature. The objective of this restriction is to prevent runtime security exceptions when classes that are not signed would be called as well as the additional overhead that is implied by related verifications at class loading.

Table 2.1 shows the criteria of validity of the digital signature for various environments.

Error Type	**Sun Jarsigner**	**Java with Security Manager**	**Felix**	**SFelix**
Unsigned Archive	W	A	R	R
Unknown Signer	A	A	R	R
Addition of Resource	A	A	A	R
Removal of Resource	A	A	A	R
Modification of Resource	R	R	W	R
Unvalid Order of Resources	A	A	A	R
Signature of Embedded Archive Unvalid	R	R	W	R
Time Of Check	Test	Exec	Exec	Install

A: Accept; R: Reject; W: Warning.

Table 2.1: Behavior of several tools and frameworks in the presence of unvalid archives

Sun `jarsigner` tool considers the modification of archive resources and invalid signature of embedded archive as errors. Unsigned archive cause a warning to be emitted. This behavior has direct implications on the execution of Java application with a security manager: errors cause the interruption of the execution but warning are not considered. For instance an unsigned archive is executed as an archive with a valid signature. Removal of signature information is thus sufficient to by-pass the default verification mechanism. Open source

OSGi implementations does not fully solve this limitation. For instance in Felix the validity of certificates is checked but the validity of digital signature itself is handled over to the security manager. Consequently addition, removal or dis-ordering of resources are not considered as errors, which violates OSGi specification.

Our SFelix implementation is developed to comply with these specifications and reject the bundles at install time whenever the digital signature is not compliant with OSGi R4 specification. Install time validation has the advantage not to impair the runtime performance of applications.

Performances Figure 2.1 shows the performances of the bundle digital signature validation process, according to the class number. The tests are performed with a Sun JVM 1.5.

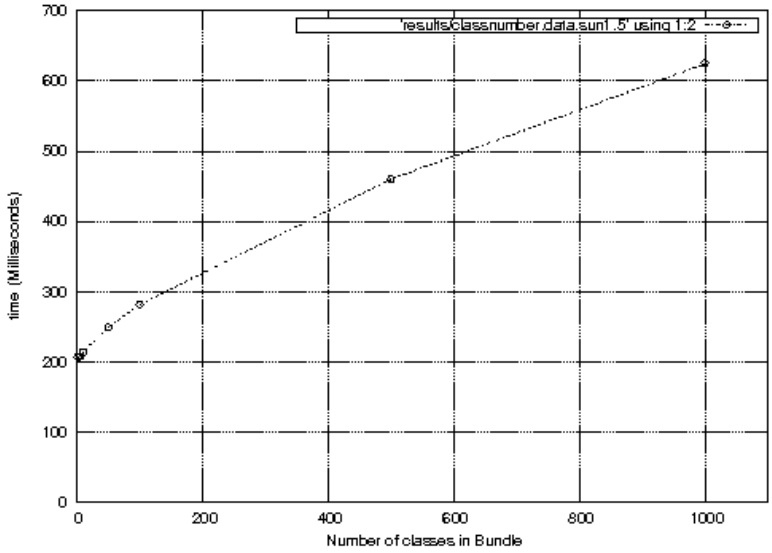


Figure 2.1: Performances of bundle digital signature validation, according to the class number

Figure 2.2 shows the performances of the bundle digital signature validation process, according to the class number when it is repeated. The tests are performed with a Sun JVM 1.5.

This mechanism will be re-used for further verifications such as Component Based Access Control (CBAC) (see Section 8.3) and Weak Component Analysis (WCA) (see Section 8.4) to provide the identity of the component provider.

2.2 Bundle Publication

Principles The SF-Jarsigner tool[2] supports the secure publication of OSGi bundles. It provides features to sign and publish bundles on a public repository: the Open Bundle Repository (OBR) [AH06]. OSGi client* platforms can then discover which bundles are available for installation, download them and let the SFelix verification take place. The SF-Jarsigner tool is presented in [PF07b]. Implementation details are given in [PF06b]. The functionalities of

[2] http://sf-jarsigner.gforge.inria.fr/

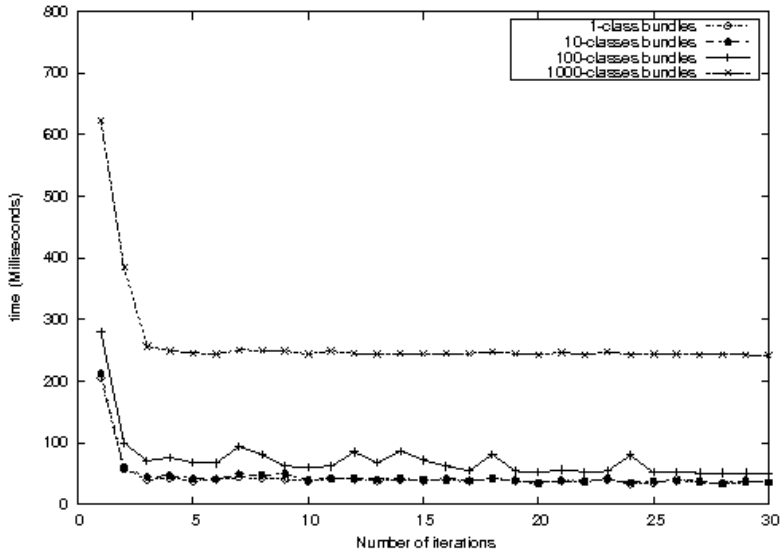


Figure 2.2: Performances of bundle digital signature validation, according to the class number, with repetitions

the SF-Jarsigner tool are first presented. Next, an extension of the OSGi bundle life-cycle is defined to enable remote management for instance through the MOSGi console[3] [RF07].

The SF-JarSigner Tool contains four panels: the key management, signature and verification, OBR management and publication panels.

The key management panel enables to load a pair of public/private keys pair for the digital signature from a Java `keystore` (`JKS` format).

The signature and verification panel enables to select set of bundles to be signed and to specify the local directory where they are stored before being published. Alternatively it can be used to verify the validity of existing digital signature with regard to the `keystore` loaded in previous step.

The OBR management panel enables to store information related to a set of remote file servers: access protocol, login information, archive directory, meta-data file reference. These meta-data contained in the OBR file provide data related to the bundles such as size, dependencies and various properties to enable the OSGi platform to perform dependency resolution and to load bundles only if they are required and if all dependencies are available for proper execution. New file servers can be added, removed and stored for further use. Only the `FTP` protocol is supported so far.

The publication panel enables to upload signed bundles onto a file server. Bundles can be selected individually or sent all together.

Once uploaded these bundles can be identified through the OBR file by all client platforms that maintain a reference to it. Since the bundle reference is given as a URL they do not need to be stored on the same server. The `HTTP` protocol is usually used to retrieve both OBR file and bundles.

[3] http://mosgi.gforge.inria.fr/

Extension of the Bundle Life-Cycle The management of OSGi platforms requires to track the state of the bundles that are installed. For instance, the MOSGi platform gives information about the bundle state, whether they are installed, active, or resolved. These states are defined in the OSGi R4 specification. However, the case of bundle rejection because of invalid signature is not considered. In this case no information about invalid bundles are available. We therefore propose to introduce an additional state in the life-cycle of OSGi bundles: the `REJECTED` state to enable the observation of the aborted installation for bundles with invalid digital signature [RPF+07].

Figure 2.3 presents the life-cycle of OSGi bundles with security management support.

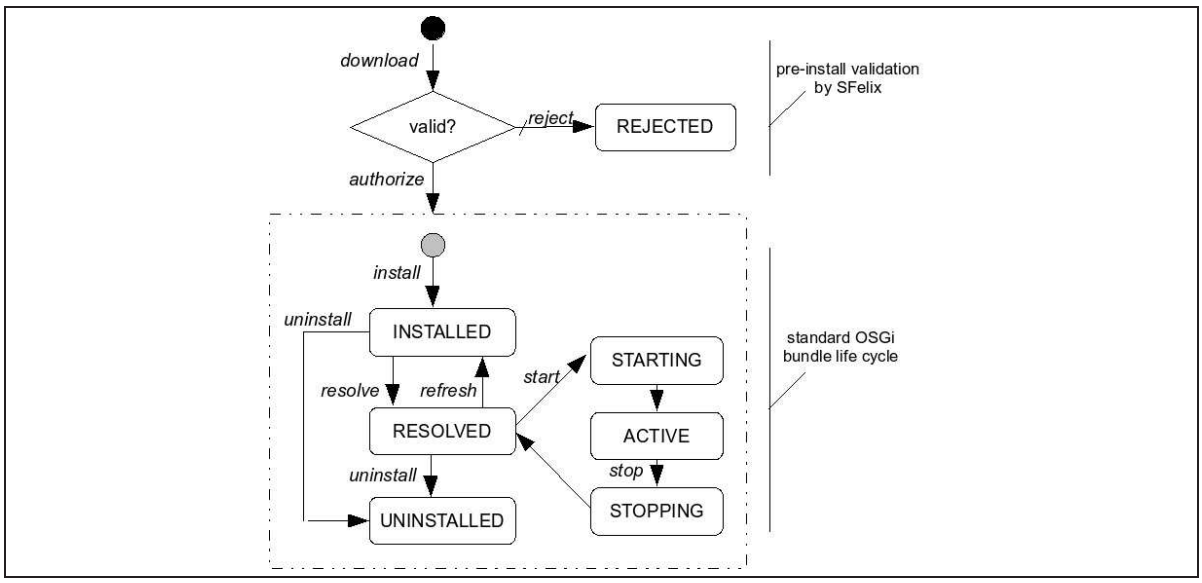


Figure 2.3: Life-cycle of OSGi bundles with security management support

Deployment and Identity-based Cryptography Together with Samuel Galice we worked on another improvement of the process of bundle publication, namely the cryptographic scheme for digital signature. Java Archives and OSGi bundles are usually signed using the SHA-1 and DSA algorithms. We propose to exploit the properties of Identity-Based Cryptography to ease the key management process [PGFU07].

Identity-based cryptography derives the public key of actors - here the bundle signers - from their identity and a seed from a *Public Key Generator*. Each client can thus verify the validity of any signature without previously having access to its public key certificate: the distribution of PKG parameters to clients is sufficient. This means that only the signers need to contact the PKG. In the case of bundle publication where few signers and numerous clients exists, this makes the key management process lighter. Moreover, through regular regeneration of the signers private key for instance on a daily basis no key revocation is necessary.

The main drawback of the Identity-based cryptographic scheme is that the PKG is a single point of failure in the system*: its compromission provides the attacker with sufficient information to forge false signatures. Even though some variants such as the CZK-IBS make possible to prove the forgery a posteriori this weakness may limit the exploitation of Identity-Based Cryptography to protected closed networks rather than exposing such an infrastructure

on the Internet.

2.3 OSGi Bundles as Java Protection Domains

Java Protection Domains * Access Control in Java platforms is enforced through Java Permissions. In the context of the OSGi platform code is provided by issuers that do not necessarily know each other and therefore may not trust each other. Neither does the OSGi platform itself trust bundles that are discovered from the environment. It is therefore essential to perform security checks so as to prevent the bundles from executing dangerous methods.

Java Permissions are associated with the bundles through Permission Domains. Each bundle is started in the Permission Domain of its signer. For each signer a set of Java permissions is defined in the `java.policy` configuration file. This mechanism enables the platform administrator to define the minimal set of permissions that are required to execute each bundle. If a bundle intends to perform a sensitive call that is not expressively allowed the call aborts.

Java Permissions are defined in the Java 1.2 specification so as to relieve the sandboxing model for Applets: code can be trusted to execute some actions, but not some others [GMPS97]. For instance a local file management application needs access to the file system but not to the network.

Since protection domains where not implemented in the OSGi platform we use for implementation, Apache Felix, it was necessary for us to code this feature.

Conclusion Figure 2.4 presents the preliminary development that were necessary before the beginning of the actual research work on security models for Java SOP platforms.

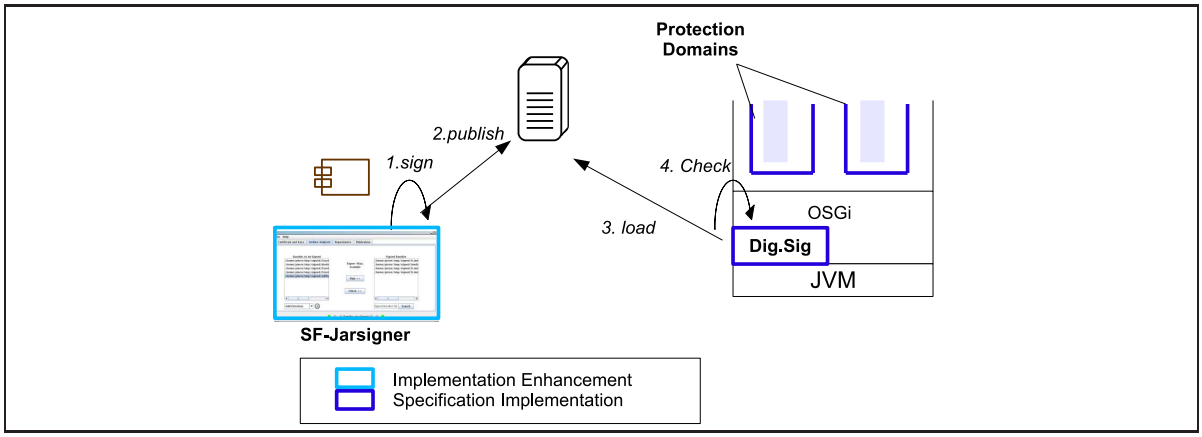


Figure 2.4: Overview of preliminary developments

The Apache Felix implementation of the OSGi platform is extended with specification-compliant features: validation of the digital signature of bundles which is part of the OSGi Security Layer and bundle-specific Permission Domains. We also developed a tool for signing and publishing bundles, the SF-Jarsigner which exploit the Open Bundle Repository (OBR) format from the OSGi Alliance. Experiments with Identity-based Cryptography show that this technology can be used with benefits to ease key management in large scale deployment of OSGi platforms. Related software remains proof-of-concept and is not mature enough to be released.

Part I

Background and Related Works

3 Software Security

The development of secure software is historically based on external protection mechanisms: code sandboxing, black-box testing, as well as on reactive patching. Whereas this approach improves the security* status of applications, it does not tackle the root of unsecurity: applications are mainly not built to be secure, because developers and security practitioners live in different worlds.

The dramatic increase of networked applications imply that their are built to withstand attacks* instead of relying on external mechanisms meant to hide their weaknesses. This requires that developers are trained, and that convenient tools are provided to help them create secure applications.

This trend lead to the inception of the *Software security* domain. Its objective is to avoid the 'patch hell' and to limit costly upgrades of deployed software.

3.1 What is Software Security ?

3.1.1 Definition

Software security is concerned with *Black Hat** activities such as developing exploits* and malwares and with *White Hat* activities which consists in building systems* that resist to attacks from the Black Hat world. Both should be considered as complementary parts [HM04]. Since the goal of this thesis is to define a secure execution platform*, we concentrate here on the White Hat approach. Software Security* is identified as a specific research field by Gary McGraw [McG04] and defined in [McG06]. Its core claim is that security should be *built-in* into systems[1].

[1] https://buildsecurityin.us-cert.gov/

Software Security is the technical and research field that is concerned with the realization of provably secure software systems. The objectives of Software Security are listed below:

- To enforce security as an intrinsic property of the software, rather than a feature to be added afterwards.
- To ensure security properties of all projects, not only highly critical systems such as nuclear plants or aviation and railway control systems; commercial software such as mobile devices or entertainment systems also need to be protected.
- To build secure and usable systems [FSH03].
- To make the security level* measurable.
- To exploit the full development life-cycle to enhance the security level of the software.

Trinity of trouble Security issues in software are due to three factors:

- Complexity.
- Extensibility.
- Connectivity.

These features are intrinsic properties of almost all software that is produced in our connected world. They are the core properties of software that provides both a rich user experience and a satisfactory manageability.

Pillars of Software Security The Software Security approach is characterized by four basis concepts, or 'pillars':

- Applied risk management [McG06]: software security expenses should be made if and only if they prevent losses that are predictably more important.
- Knowledge [McG06] (see Section 3.2): software security should be deduced from a deep understanding of the system and of its weaknesses.
- Pro-activity [HL02, VM01]: software security should be performed early in the life-cycle to avoid costly refactoring once the system is deployed.
- Software security techniques (see Section 3.3): secure design and coding should rely on well-defined and possibly automated techniques.

Topics of Software Security The activities around Software Security are the following [McG06] ones:

- Code-breaking techniques: identification of vulnerabilities* and related exploits.
- Building secure software, *e.g.* through risk management and security-aware life-cycle.
- Designing and using secure languages and platforms, *e.g.* type-safe languages and virtual machines.
- Designing secure software, *e.g.* through Security Patterns*.
- Making sure that software is secure, *e.g.* through tests, white-box audit and benchmarking.
- Educating software developers, architects and users about how to build security in, *e.g.* by providing security related data in a convenient format.

3.1.2 The Domains of Computer Security

Building secure computing systems is not a new topic in itself. However, security is often perceived as a set of features: cryptographic libraries, authentication and authorization libraries, or a property of the execution environment*: hardened OSes, virtual machine sandboxing. The security of the applications themselves often relies on a very intuitive and unorganized approach where developers enforce generic good development practice, but little resource is dedicated to security. This matter of fact is due to 1) the usual scarce resources in software projects which do not enable to perform tasks that do not directly aim at developing new functionalities and 2) the gap between the developer and software engineering community on the first hand and the security community on the second hand. Security development and research is historically performed in following areas: cryptography, network security*, operating system security* and application security*. The first and the last areas deal essentially with providing security libraries. The second and the third deal with with defining stable protocols and systems that are free from vulnerabilities.

Figure 3.1 shows the various *Security Domains* in the common applicative stack for execution environments.

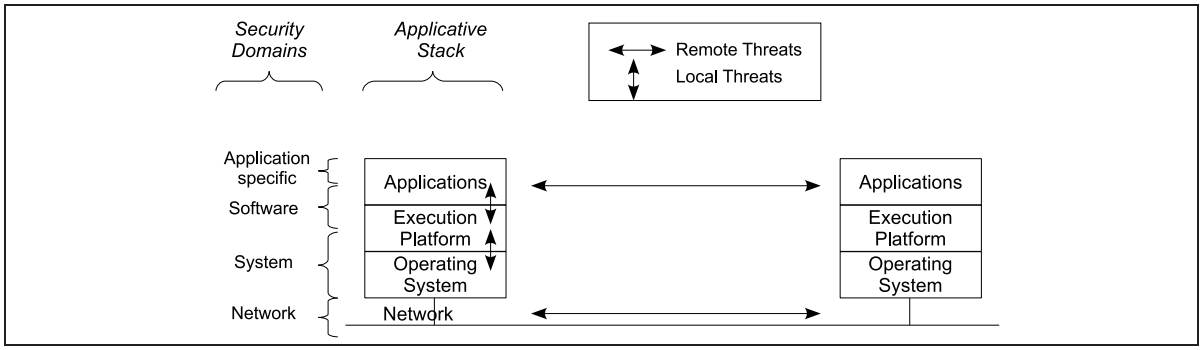


Figure 3.1: The security domains of computer security in the applicative stack

Well established approaches obviously do not address one fundamental requirement of today's software: as far as almost every piece of software is connected to the Internet and thus accessible to hackers, it should be built so as to withstand attacks.

Software Security and Network Security The objectives of Network Security [CBR03] is to define secure communication protocols and to control security in the network through firewalls and intrusion detection systems. Its main challenge is to provide protection mechanisms that do not contain vulnerability themselves. Because they introduce more software and are exposed to the Internet, systems like firewalls are a target of choice for hackers. Network-based mechanisms prove to be efficient against network attacks. Nonetheless, they fail to prevent the exploitation of weak code [McG06]. This is particularly true of high-level firewalls such as HTTP or applicative firewalls which often fail to provide adequate solutions because applicative protocols are more complex and evolutive, as well as less defined, than network protocols.

The challenge for Software Security in relationship with Network Security is to provide applications that are not vulnerable to network traffic that can not be protected through firewalls or IDS.

Software Security and Operating System Security Operating System Security is the domain of security that deals with the prevention and detection of attacks against an OS. The ultimate goal of such attacks is to 'own' the OS, *i.e.* to have full administration rights on it. The challenges of secure OSes are: to build systems that are sufficiently secure for common users, to build hardened OSes such as Security Enhanced (SE) Linux versions [Sri06] or Asbestos [Ste06] and to ensure the integrity of the OS itself through secure bootstrap mechanisms [AFS97]. Alternative to secure OSes are virtual machines or secure middlewares that enable more flexibility in the system design and thus support advanced security schemes (see for instance [VNC+06]). Because of their inherent complexity, securing OSes is performed in an empirical manner which questions the fact that OSes can actually be secure at all [THB06]. Attempts at modeling full systems to evaluate their security exist [SSLA06] but remain marginal. Their efficiency can be discussed: vulnerabilities are often very low level features that are abstracted away in most models.

The challenge for Software Security in relationship with Operating System Security is to take advantage of system level mechanisms such as isolation in an efficient and configurable manner to exploit them accordingly to the application requirements.

Software Security and Application Security Application Security is the domain of security that deals with the prevention and detection of attacks against applications through post-development analysis and protection mechanisms. Its main activities are [McG06] as follows;

- Sandboxing.
- Black-box testing.
- Protection against malicious code.
- Code obfuscation.
- Executable lock down.
- Runtime monitoring.
- Policies management and enforcement.

It considers software as an entity* that should be protected through external measures and focuses on the interactions between the application and the outside world: users, servers, clients*. The Application Security approach faces several challenges. First, it does not provide suitable tools to identify flaws* early in the development life-cycle. This can have important financial consequences because of the cost involved by late bug correction (See Figure 1.1 page 6). Secondly, it does not support the management of security issues in On-the-Shelf (OTS) applications: tools and methods are not available to evaluate efficiently the security status of software outside its development process. Application Security only provides an external view of the software system. Complementing it with Software Security can overcome its limitations by defining the intrinsic properties of the software that ensure the security of the resulting system.

3.1.3 Software Security Assurance

The objective of Software Security is to enhance and guarantee the actual security level of programs. It can be achieved through dedicated *Software Security Assurance** methodologies which typically enrich software engineering processes with security related activities [GWM+07]. The requirement for software security assurance is to define tools, techniques and metrics for building and managing securing software [Bla05, BKF06].

Several industrial initiatives aim at integrating software security assurance in the standard development process. Examples are Windows security pushes, back to 2002 [HL02] and the Software Assurance Forum for Excellence in Code (SafeCode) initiative [Sof08] which gathers Juniper, Microsoft, Nokia, SAP, Symantec.

Definition Software Security Assurance is 'the basis for gaining justifiable confidence that software will consistently exhibit all properties required to ensure that the software, in operation, will continue to operate dependably despite the presence of sponsored (intentional) faults'. In practical terms, this means that 'such software must be able to resist most attacks, tolerate as many as possible of those attacks it can not resist and contain the damage and recover to a normal level of operation as soon as possible after any attacks it is unable to resist or tolerate' [GWM+07].

The NASA [nas92] defines Software Security Assurance as all activities that ensures conformity to requirements, standards and procedures during the software development life-cycle: requirement and specifications, testing, validation and reporting.

Secure Development Life-Cycle The Life-Cycle phases of Software Security Assurance are shown in Figure 3.2.

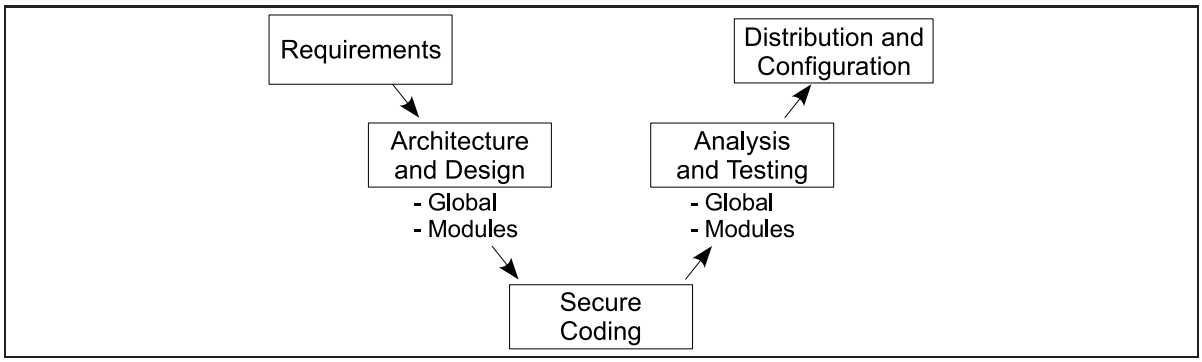


Figure 3.2: The life-cycle phases of software security assurance

They are the following ones [GWM+07]:

- *Requirements* for secure software directly affect the likelihood that the software will be insecure. They are directed toward reducing or eliminating vulnerabilities. They are tied to the software development plan and to project management direction. They are not to be confused with requirements for security functionalities that concern access control, identification, authentication and authorization, and cryptographic functions.
- *Architecture & Design* consist in performing a security analysis of the system architecture and of individual components*, as well as in integrating secure design principles and patterns. It should be performed for the whole system as well as for individual modules.
- *Secure Coding* implies a set of issues that should be considered such as language choice, compiler, library and execution environment choice, coding conventions and rules, comments, documentation of security sensitive code and constructs, integration of OTS software and filters and wrappers.

- *Analysis & Testing* represent the most widespread best practices for security assurance. They can be parted in 'white-box' techniques such as static analysis, property based testing, fault injection, fault propagation and dynamic analysis of source code, and 'black-box' techniques such as binary analysis, penetration testing, fuzz testing and automated vulnerability scanning. It should be performed for the whole system as well as for individual modules.
- *Distribution & Configuration* aim at minimizing the opportunities for malicious actors to gain access and to tamper the software during its transmittal from its issuer to its consumer.

Software Security Assurance thus includes activities that are specific to Software Security, but takes issues from the Application Security domain into account to build a coherent system.

Representative Secure Development Life-Cycles (SDLC) are the McGraw process for 'Building Security in', Microsoft SDLC and the GIAC framework. McGraw process for 'Building Security in' [McG06] is built around 7 touch-points: Code Review, Architectural Risk Analysis, Penetration Testing, Risk-based Security Testing, Abuse Cases, Security Requirements Elicitation, Security Operations. Microsoft SDLC [HL02, LH05] consists in enriching the standard development life-cycle with incremental security improvements and emphasizes strongly on developer training. The Global Information Assurance Certification (GIAC)[2] framework [McC02] which is based on following principles: integrate security as part of the design, assume a hostile environment, use open standards, minimize and protect system elements to be trusted, protect data at the source, limit access to need-to-known, authenticate, do not subvert in place security solutions, fail securely, log, monitor and audit, and maintain accurate system time and date.

Other software security assurance proposals exist, for instance to support component based development [Kim04] or to exploit certification schemes such as the Common Criteria [Llo05].

The AEGIS model [FSH03] is the mapping of security concerns to Boehm's spiral development model [Boe86].

A comparison of several approaches for software security assurance is proposed by the Software Engineering Institute of Carnegie Mellon University [Noo06]. It emphasizes on the Capability Maturity Model (CMM) which aims at certifying development processes rather than the software itself.

3.2 Security Requirement Elicitation

The first step in a software security assurance process is the elicitation of the security requirements. It implies to have a precise knowledge of the vulnerabilities of the system under study and to be able to perform precise benchmark.

3.2.1 Vulnerability Identification

The tools for vulnerability identification are taxonomies, Reference Vulnerability Information* (RVI) databases and Top N reminder lists. Attack patterns structure these informations.

[2] http://www.giac.org/

Vulnerability Taxonomies Taxonomies provide a fine grain description of the properties of each vulnerability.

Each taxonomy* should verify the properties of a valid taxonomy as defined by [Krs98] and [HL98]. These properties are the following: objectivity, determinism, repeatability, specificity (disjunction), observability.

The seminal works on vulnerability taxonomy have been performed by Abbott [ACD+75] and Bisbey [BH78]. The flaws are classified by type of error (such as incomplete parameter validation). This approach turns out not to support deterministic decisions since one flaw can often be classified in several categories according to the context. To solve this problem, Landwehr [LBMC94] defines three fundamental types of taxonomies for vulnerabilities: classification* by genesis of the vulnerability, by time of introduction and by location (or source).

Moreover, vulnerabilities should be considered according to specific constraints or assumptions since there existence depends most of the time on the properties of the environment [Krs98]. These assumptions make it necessary to rely on a well defined system model. For instance, such a model is proposed for generic computing systems by the Process/Object Model [BAT06]. Consequently it is difficult for generic purpose databases to rely on specific taxonomies: the Common Vulnerability Enumeration [BCHM99] project has given up the use of taxonomies.

Extensive discussions of vulnerability taxonomies can be found in [Krs98] and [SH05].The CWE (Common Weaknesses Enumeration) Project maintains a web page with additional references and a graphical representation of each taxonomy[3].

Reference Vulnerability Information (RVI) Databases Extensive databases are meant to maintain up to date references on known software vulnerabilities, to force the system vendor to patch the error before hackers can exploit it. They are also known as Reference Vulnerability Information (RVI), or Refined Vulnerability Information (RVI) sources. Two main types of RVI exists: the vulnerability mailing lists and the vulnerability databases.

The main mailing lists are the following:

- Bugtraq, 1993 onwards (see http://msgs.securepoint.com/bugtraq/),
- Vulnwatch, 2002 onwards (see http://www.vulnwatch.org/),
- Full Disclosure, 2002 onwards (see among others http://seclists.org/).

The reference vulnerability databases are the following. They are meant to publish and maintain reference lists of identified vulnerabilities.

- CERT (Computer Emergency Response Team) Database. It is based on the Common Language for Security Incidents [HL98][4].
- CVE (Common Vulnerabilities and Exposures) Database[5].
- CWE (Common Weaknesses Enumeration) Database. It is bounded with the CWE and aims at tracking weaknesses and flaws that have not yet turned out to be exploitable for attackers[6].

[3] http://cwe.mitre.org/about/sources.html
[4] http://www.cert.org/, Carnegie Mellon University
[5] http://cve.mitre.org/, US Department of Homeland Security and Mitre Corporation
[6] http://cwe.mitre.org/index.html, US Department of Homeland Security and Mitre Corporation

- CIAC (Computer Incident Advisory Capability) Database[7].
- OSVDB, Open Source Vulnerability Database[8]. It is centered at Open Source Products.

Complementary RVI Sources are the following organizations: SecuriTeam [9], Packet Storm Security [10], the French Security Incident Response Team [11], ISS X-Force [12], Secunia and SecurityFocus.

The limitations of the RVIs is that they follow no stable policy, which makes comparison between sources and between the item of a given sources difficult [Chr06].

Reminder Lists Since catalogs are not so easy to remember and therefore to put into practice, several 'Top N' lists have been defined. The motivation for such lists is the recurrent drawbacks of other approaches: vulnerability catalogs do not provide a useful overview of the identified vulnerabilities [Chr06].

One classification of computer security intrusions is given by Lindqvist [LJ97]. It contains external and hardware misuse, and several software misuse cases: bypassing intended control, active and passive misuse of resources, preparation for other misuse cases...

The Plover classification[13] is an example of rationalization of Vulnerability catalogs to support analysis. It is based on the MITRE CVE Database and contains 300 specific entries that reflect 1400 vulnerabilities identified in the CVE database. Its goal is to suppress redundancy from the original database so as to enable scientific analysis, *e.g.* using statistical approaches [Chr05].

The Nineteen Deadly Sins of software systems are defined by Michael Howard, from Microsoft [HLV05]. They describe the most common vulnerabilities that are found in enterprise information systems. They concern Web based systems, the architecture of the information systems and the technologies involved.

The Open Web Application Security Project (OWASP) maintains a TOP 10 of Web Applications vulnerabilities[14]. It concerns input validation, data storage, as well as configuration and error management. Another consortium for Web Application security enforcement, the WASC (Web Application Security Consortium), provides its own threat classification[15].

A convenient vulnerability list is provided by Gary McGraw, through the Seven Kingdoms of software vulnerabilities [McG06] [TCM05]. The number 7 is chosen to be easily remembered. Each entry is completed with phyla *i.e.* precise example of the broader categories that are defined by the kingdoms. The kingdoms are the following: input validation and representation, API abuse, security features, time and state, error handling, code quality, encapsulation + environment. This classification is targeted at enterprise information systems.

Attack and Vulnerability Patterns The descriptive spirit of Design Pattern [Ale77], [GHJV94], [MM97], is well suited for application in the security fields, where the question of organiza-

[7] http://www.ciac.org/ciac/index.html, US Department of Energy
[8] http://osvdb.org/, Open Source
[9] http://www.securiteam.com/, Beyond Security Company
[10] http://packetstormsecurity.nl/, Non-Profit Organization
[11] http://www.frsirt.com/, A.D.Consulting Company
[12] http://xforce.iss.net/xforce/alerts, IBM Company
[13] http://cve.mitre.org/docs/plover/
[14] http://www.owasp.org/index.php/OWASP_Top_Ten_Project
[15] http://www.webappsec.org/projects/threat/

tion and exploitation of the knowledge is central to the protection of systems. Two types of patterns are defined in the security domain: Attack Patterns and Vulnerability Patterns*.

Attack Patterns represent potential attacks against a system. They model the preconditions, process and postconditions of the attack. They can be combined with attack trees, so as to automate the identification of attacks that are actually build from simpler atomic attacks [MEL01]. An extensive presentation of the applications of attack patterns is given in the book by Markus Schumacher [Sch03]. The use of Attack Patterns together with software architecture description to identify vulnerabilities is described by Gegick [GW05]. The limitation of this approach is that the attacks as well as the system must be modelized. This makes the approach impractical and often not realistic based on the available knowledge.

The Vulnerability Patterns are used in catalogs of vulnerabilities. They often contain a limited number of informations that are meant to identify the vulnerability, but also to not make it easily reproduceable without a reasonable amount of effort to prevent lazy hackers from exploiting the vulnerability databases as a source of ready-to-exploit attack references.

We list here the most wide-spread Vulnerability Patterns, along with the attribute they contain.

- Rocky Heckman pattern[16]: Name, type, subtype, AKA, description, more information.
- CERT (Computer Emergency Response Team) pattern: name, date, source, systems affected, overview, description, qualitative impact, solution, references.
- CVE[17] (Common Vulnerability and Exposures) pattern: name, description, status, reference(s).
- CIAC[18] (US Department of Energy) pattern: identifier, name, problem description, platform, damage, solution, vulnerability assessment, references.

These Vulnerability Patterns are quite simple ones. They have an informative goal but do not intend to support the reproduction of the vulnerability with a minimum of effort, as other patterns do. This approach makes sense relative to their use context - making users and administrators aware of the existence of the flaws - but are not sufficient to support detailed analysis of the related vulnerabilities.

3.2.2 Software Security Benchmarking

The definition of quantified security requirements is based on security benchmarking*. It is based on the assumption that security is not to be considered as an absolute value but should be measurable in order to compare various flavours of a system or a given implementation against a reference implementation. Further references related to security benchmarking can be found on the Security Metrics web page[19].

Requirements for security benchmarking research are explicited by the NIST Samate (Software Assurance Metrics and Tool Evaluation) project[20] [Bla05]. It is dedicated to improving software assurance by developing methods to enable software tool evaluations, measuring

[16] http://www.rockyh.net/
[17] http://cve.mitre.org/
[18] http://www.ciac.org/ciac/index.html
[19] http://www.securitymetrics.org/
[20] http://samate.nist.gov

the effectiveness of tools and techniques and identifying gaps in tools and methods. It is complemented by the SRD (Standard Reference Dataset) project which aims at providing a knowledge base for assessing software security tools.

SRD requirements to support software security tools evaluation are:

- Identify classes of security flaws and vulnerabilities.
- Develop metrics to assess software.
- Identify classes of software security assessment* techniques.
- Document the state of the art in software security assessment tools.
- Develop measures to evaluate tools.
- Develop a collection of reference awed or vulnerable programs.

Samate goals are high level ones which express the need to propagate the technical information in industrial software projects.

- People: provide education and training.
- Process: define life cycle development process, best practices, standards.
- Technology: develop new tools.

Limitations Metrics and statistics build the basis of security assessment. However, they can not be considered as fully trustworthy data for following reasons [Chr06]:

- variation in editorial policy: quantity and type of cataloged vulnerabilities may vary over time, especially for databases where the administration team is evolving; this can lead to inconsistency in the available data.
- fractured vulnerability information: each RVI source collects its own data; catalogues may not be coherent. They often can not be translated from one to another.
- lack of complete cross referencing between RVI sources: competition between RVIs or simply resource limitation make it not possible to track the data from all (even public) RVIs, so no database can be considered to be comprehensive.
- unmeasurable research community bias: researchers vary in skill set, preferences for certain vulnerability types or target product; identified vulnerabilities therefore reflect more the effort spent on analyzing systems than their actual security level.
- unmeasurable disclosure bias: vendors and researchers vary in their disclosure models, so again the identified vulnerabilities reflect more the analysis and publication habit that the actual state of the systems.

These limitations directly impact the validity of vulnerability catalogs and of the security benchmarking that is based on them.

Counter Metrics The first step in security benchmarking is to count occurrences of security related properties. These metrics can be related to the development process or to the code itself [Che06]. Metrics of the first category include the number if identified threats, performed tests, measures of the security-related activities or measures related to the education of the development teams. They can also concern analysis coverage: How much of the code is thoroughly tested ? How much has been reviewed through automated tools ? Through manual review ? Code metrics can measure the number of found bugs, the number of different bugs,

the number of occurrences by vulnerability category, by severity, or the number of times a system is mentioned in an RVI.

These simple metrics are often used as evolution indicators rather that as absolute values. Moreover, they tend to express the thoroughness of the benchmarking process rather than the actual security status of the target software. For instance, a software that is known to contains an important number of bugs during early development phases is likely to be well tested.

Estimation of System Exposure The security level of a software system can be estimated through its exposure. Related metrics are the attack surface*, the approximation of the number of bugs per Line of Code (LoC) and the coverage brought in by protection mechanisms.

The attack surface $surf$ is defined as a part of Microsoft SDLC [HPW05]. It is impacted by five dimensions of software systems:

- $targets$: processes and data which corruption is the goal of attacks,
- $enablers$: processes and data which corruption enables further exploits,
- $channels$: means of communicating informations from a sender to a receiver,
- $protocols$: the rules for exchanging information associated with a channel,
- $access\_rights$, that limit the access to the different resource types such as processes, data and channels.

The attack surface is expressed as:

$$surf = f(targets, enablers, channels, protocols, access\_rights) \tag{3.1}$$

where f can be an additive function that takes the interactions between dimensions into account, or can be weighted to reflect their relative importance.

The approximation of N_{bug} the number of bugs in an application is deduced from experience by considering the number of bugs per Line of Code (LoC) in similar projects [HM04]. Typical values range from 5 bugs/KLoC (1000 LoC) for code that has undergone intensive testing to 50 bugs/KLoC for well-tested code. The approximation N_{bug} is given by:

$$Size * 5/1000 < N_{bug} < Size * 50/1000 \tag{3.2}$$

where $Size$ is the number of LoC (Lines of Code) in the application.

Coverage is a metric from the domain of fault tolerance [Arn73]. It expresses the ability of a fault tolerant system to withstand a certain number of faults. The evaluation of coverage is based on the comparison of the values of Mean Time to Failure (MTTF) for the default and the resilient system. To the best of our knowledge, no such metric exists to assess protection mechanisms in the security domain.

Estimation of Damage Potential and actual Risk The risk implied by attacks can be expressed as damage potential, or as actual risk [HM04]. The financial risk can then be deduced.

Damage potential and actual risk are expressed according to a certain number of factors. The authors suggest to rate the risks as high, medium or low unless more precise measures are available. The metrics depends on following variables:

- *Attack_Potency*, the potential to create damage. High-potency attacks are likely to cause problems that are noticeable by the users. Medium-potency attacks are likely to cause problems that are noticeable by the administrators. Low-potency attacks do not cause noticeable problems.
- *Target_Exposure*, the measure of how difficult it is to carry an attack. Low-exposure attacks are those that are blocked by a firewall. Medium-exposure attacks are blocked by suitable application configuration. High-exposure attacks can not be blocked through the system configuration. If target exposure can not be measured, it should be estimated to 100 %.
- *Impact*, the measure of the harm the attack causes to the system. For instance, a database that is vulnerable to request injection and fully exposed but contains no data has 0 % of risk to leak private information. If impact can not be measured, it should be estimated to 100 %.

The *Damage_Potential* metric is expressed as:

$$\begin{aligned} 1 < Attack\_Potency < 10 \\ 0 < Target\_Exposure < 1.0 \\ Damage\_Potential = Attack\_Potency * Target\_Exposure/10 \end{aligned} \tag{3.3}$$

The *Damage_Potential* has a range value from 0 to 1. It can also be expressed as a percentage.

The actual risk *Actual_Risk* brought in by a vulnerability is expressed as:

$$\begin{aligned} 0 < Impact < 1 \\ Actual\_Risk = Damage\_Potential * Impact \end{aligned} \tag{3.4}$$

The *Actual_Risk* has a range value from 0 to 1. It can also be expressed as a percentage.

The financial risk fin_risk bound with a given asset can then be measured:

$$fin\_risk = Actual\_Risk * Asset\_value \tag{3.5}$$

with *Asset_value* the financial value of the protected asset. Typically, protection mechanisms should be implemented when their cost is inferior to the financial risk.

Mean Time to Intrusion; Minimum Time to Intrusion More complex metrics exist to express the security properties of a software system such as mean time to intrusion (MTTI) and minimum time to intrusion (MinTTI) [VGMC96]. They are based on the estimation of the skills of potential attackers. They are no absolute measures but relative ones that help compare different versions of the same system.

3.3 Techniques for Software Security

The core technical activities for building secure software are architecture and design and secure coding. For critical systems, they can be complemented by certification of the development process or of specific properties of the software.

3.3.1 Secure Architecture and Design

Design Principles The core principles of secure software design are well understood since the publication of the Saltzer and Schröder Principles [SS73].

- *Economy of mechanism*: keep the design as simple and small as possible.
- *Fail-safe defaults*: base access decisions on permission rather than exclusion.
- *Complete mediation*: every access to every entity must be checked for authorization.
- *Open design*: the design should not be secret.
- *Separation of privileges*: where feasible, protection mechanisms that require two keys to unlock should be used instead of simpler mechanisms with one single key.
- *Least privilege*: every program and every user of the system should operate using the least set of privileges necessary to complete the job.
- *Least common mechanism*: minimize the amount of mechanism common to more that one user and depended on by all users.
- *Psychological acceptability*: its is essential that the human interface be designed for ease of use, so that the users routinely and automatically apply the protection mechanisms correctly.

Additional requirements have been elicited more recently, in particular to support the specific properties of component-based architectures [GWM+07]:

- *Security-aware error and exception handling*: in particular, attacks should not lead to software crash (denial-of-service) or to the release of system-related or applicative data.
- *Mutual suspicion*: components should not trust each other when they are not explicitly intended to.
- *Isolation and constraint of untrusted processes*: misbehavior of un- or less trusted components should not affect the system.
- *Isolation of trusted/high consequence processes*: integrity and availability of sensitive components should be protected by isolating them.

Security Patterns The objectives of security patterns is: capture expert knowledge, be domain independent *i.e.* usable in diverse contexts such as network, operating system, software and application security, and be reusable. They are a direct application of the design pattern concept to the domain of security. A comprehensive survey of security patterns is provided by [YWM08].

The most important collection of security patterns is provided by Kienzle and Elder [KETEH01]. Their report gathers 29 patterns dedicated to web applications and parted in two categories: structural patterns and procedural patterns. Only patterns that are not specific to web applications are presented here.

The proposed versatile structural patterns are: hidden implementation, minefield, partitioned application, secure assertion, server sandbox, trusted proxy, and validated transaction.

The proposed procedural patterns for the development process are: build the system from the ground up, choose the right stuff, log for audit.

Some patterns describe specific security features. These patterns should be seen more as informative as actual reference, but they provide the advantage of supporting explicit definitions of concepts that are not well documented elsewhere in the literature. Patterns for access control are provided by [RFMP06]: Authorization, RBAC, Multi-level security, file authorization, Virtual Address Space access control, reference monitor, session, Single Access Point,

Check Point. Patterns for secure development are given by [Rom01]: Authoritative Source of Data, Layered Security, Risk Assessment and Management, 3rd Party Communication, The Security Provider, White Hats* Hack Thyself, Fail Securely, Low Hanging Fruit.

Domain specific patterns are targeted at specific environments. Sun Core Security Patterns[21] initiative defined patterns for J2EE and enterprise applications [LC05, Bru06]. They are categorize in Web Tier patterns (Authentication enforcer, Intercepting validator...), Business Tier patterns (Container Managed Security, Dynamic Service* Management), Web Service Tier patterns (Message Interceptor, Secure Message Router), Identity Management and Service Provisioning (Single sign-on delegator, Password synchronizer). Sun maintains a list of technologies that implement the patterns. The other domains where numerous patterns are defined is privacy and anonymity [Sch02b].

3.3.2 Secure Coding

The objective of secure coding is to enforce language based security. The first principle is to train developers to avoid vulnerabilities and the second one is to embed protection mechanisms in the language through secure language type systems or at least in automated tools. Secure coding is the agile way of implementing software security: responsibility of the code quality is shared among developers. A discussion of the important business stakes related to this topic is to be found at [the08].

Activities of Secure Coding Security issues for the coding activities of the software implementation phase include [GWM+07]:

- language choice,
- compiler, library and execution environment choices,
- definition of coding conventions and rules,
- insertion of comments,
- documentation of security-sensitive code, constructs and implementation decisions,
- integration of non-developmental software,
- identification of filters and wrappers need,
- and, of course, writing secure code.

Writing secure code is eased by a set of methodologies that can be applied during development, in parallel or between successive development phases: code review, static analysis and security testing.

Code Review is performed manually to identify and solve security issues in the code. It must be performed by someone who is not the author of the code to make the code be analyzed as it performs, not as it is intended to perform. Some authors advise reviewers to have a black hat experience so as to better understand the actually threats the software is exposed to [HM04].

The OWASP suggests to perform reviews as coherent sessions organized around a meeting [OWA08]. The process is intended to be lightweight, *i.e.* the preparation should not exceed the duration of the meeting itself. Roles during a review process are the moderator, who selects the reviewers and conduct the meetings, the authors of the code, the inspectors who

[21] http://www.coresecuritypatterns.com/

perform the review, the reader who presents the code during the review meeting and the scribe who records the raised issues during the code review. Each meeting should consist in following steps:

- Initialization: the moderator plans the meeting and prepares the review package with the author. The review package contains the source code, the document, review checklists, coding rules as well as other useful material such as the output of static analysis tools.
- Preparation: after receiving the review package, the inspectors study the code to search for defects. Typically, the duration of this phase is similar to the duration of the following meeting.
- Meeting: the reader presents each code excerpt to the participants. Reviewers bring up issues they identified during the preparation. Ambiguities can also be identified through the (mis)interpretation of the code by the reader.
- Correction: the authors address the issues identified and the moderator validates their completeness.

Less formal review processes skip either the preparation or the meeting phase. The process can also be iterated again.

Another code review process is proposed by M. Howard from Microsoft [How06]. It focuses on the expertise of the reviewer and involves less team interactions. It can also serve as guidelines for the preparation phase of the previous process.

- Prioritize.
- Review the code.
  - Rerun all available tools.
  - Look for common vulnerability patterns.
  - Dig deep into risky code.

These generic processes must be implemented for each target language and system. A strong experience and documentation relative to their security issues is required. Resources can be found for instance for the C language in [HM04] and in the 'CERT C Secure Coding Standard'[22], for the C++ language with the 'CERT C++ Secure Coding Standard'[23], for Linux and Unix systems in [Whe03], for Windows-based systems in [HL02], for Java in [Sun07] and for Web application in [OWA08].

Static Analysis supports the automation of code review as well as the verification of complex code properties. It aims at handling the problem of code volume by improving the identification of known vulnerabilities by several orders of magnitude and at letting human reviewers focus on hard issues. Two approaches for static analysis can be identified: production tools and research efforts.

The key characteristics of tools for code analysis are the following [McG06].

- Be designed for security: although security analysis is a subset of software quality, the knowledge that underlies it is very specific.

[22] https://www.securecoding.cert.org/confluence/display/seccode/CERT+C+Secure+Coding+Standard
[23] https://www.securecoding.cert.org/confluence/pages/viewpage.action?pageId=637

- Support multiple tiers: as applications are often written in several languages and for several platforms, each of these technologies must be supported.
- Be extensible: security problems evolve, grow and mutate and tools should adapt accordingly.
- Be useful for security analysts and developers alike: analysis output should provide sufficient data for developers to enable them to fix the identified issues; this property makes tools an excellent way of training security-unaware developers.
- Support existing development processes and Integrated Development Environments (IDEs).
- Make sense to multiple stakeholders: metrics should be provided that support not only secure development but also the business decisions related to the software to be built; release managers, development managers and executives should get useful insight from analysis tools.

A wide variety of tools exist. The historical tool for finding bugs is Lint for the C language which was presented by the Bell Labs in 1978 [Joh78]. The better known recent tools are the Rough Auditing Tool for Security[24] (RATS) for C, C++, Perl, Php, Python, FlawFinder[25] for C and C++, ITS4[26] [VBKM00] for C and C++, Splint[27] for C. MOPS [CW02] applies an original approach of model checking to identify source code defects. Example of commercial tools are Fortify 360[28] which supports analysis for C/C++, .NET, Java, JSP, ASP.NET, ColdFusion, ASP, PHP, VB6, VBScript, JavaScript, PL/SQL, T-SQL and COBOL and configuration files, or Coverity Prevent [29].

Two approaches exist for static code analysis: source code analysis and binary analysis. Source code analysis tools support the identification of more complex vulnerability patterns, especially for languages that are compiled into low level binaries such as C. They thus provider a deeper analysis. Binary analysis tools are very useful to assess the quality of code which source code is not available, for instance for COTS.

Research efforts reflect the diversity of potential solutions for enhancing static code analysis. Some tools exploit assertions to enrich the set of properties that can be verified such as ESC/Java [30] which exploits the Java Markup Language (JML). Static analysis can be extended to support taint code analysis, *i.e.* to control the propagation of sensitive data [KZZ06]. It can also be used to validate weak typed languages such as scripting languages [XA06]. Advanced solutions consists in combining static analysis with code injection to optimize runtime security controls [Sch00]: this approach is called 'security passing style' (SPS). Experiments have been performed to reduce the overhead of Java security manager. However, proposed implementations do not prove to be faster than the default runtime analysis by Sun. The actual benefit of SPS is the flexibility it introduces in the security models. A powerful approach is to integrate security checks in the language definition itself and to delegates verification to language constructs. An example is provided by the secure type system for Java defined in [Boy04].

[24] http://www.fortify.com/security-resources/rats.jsp
[25] http://www.dwheeler.com/flawfinder/
[26] http://www.cigital.com/its4/
[27] http://www.splint.org/
[28] http://www.fortify.com/
[29] http://www.coverity.com/
[30] http://kind.ucd.ie/products/opensource/ESCJava2/

The collateral consequence of static analysis is that it enables a seamless training of the developers, since identified vulnerabilities are documented so as to ease the correction process.

Testing is the activity that validates the security properties of a given software system. A comprehensive framework is defined by the OWASP Testing Guide [OWA07].

Security testing should be based on following principles:

- There is no Silver Bullet: no single tool can make software secure,
- Think strategically, not tactically: tests should be performed during the initial development of software as much as possible; the cost overhead is by far reduced when compared to the classical wait-and-patch approach to security,
- The SDLC is king: developers are comfortable with classical development life-cycles; security should be integrated to limit organizational overhead,
- Test early and test often,
- Understand the scope of security: according to the considered project,
- Mindset: security testers should think 'outside the box', not only in term of foreseen functionalities,
- Understanding the subject: the application should be well documented and tests performed according to it,
- Use the right tools: in order to increase efficiency,
- The devil is in the details: every section of code should be reviewed,
- Use source code when available,
- Develop metrics: to monitor the evolution of the project.

Testing processes can be either integrated in a V life-cycle [OWA07] or performed in an agile manner using abuse cases [McG06] as reference.

Conclusion Software Security provides a set of methods and tools that can be integrated in the classical software development life-cycle to enhance and control the security level of built systems. By preventing vulnerabilities rather than patching them, it promises to complement in a powerful way network security, because firewalls have a hard time in preventing the access to flawed applications, operating system security, since security responsibility is transfered to a certain extent from the administrator to application developers and application security by making software not only robust through a protective shell but secure by design and by implementation.

4 Security for Java Platforms

The Java environment is composed by two main parts: the Java language specified in [GJSB05] and the Java virtual machine specified in [LY99]. It is designed with the assumption that no software entity* is to be trusted and therefore that each need to be checked. The first success of Java was the inception of Java Applets which enabled fully untrusted code provided by unknown web sites to be executed in a browser [DFW96]. This feature demonstrated the isolation brought by the Java Virtual Machine (JVM) between the applications and the underlying operating system. It is made possible because each internal unit of the JVM is designed to enforce such security*.

Figure 4.1 shows the architecture of a Java Virtual Machine (adapted from [COR07]).

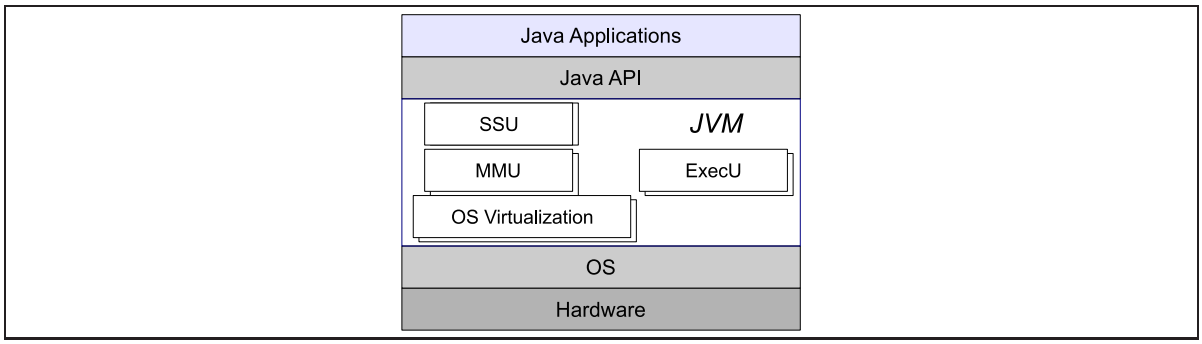


Figure 4.1: Architecture of a Java Virtual Machine

The units of the JVM are:

- *SSU*: the System Services Unit. It contains the class loader and Bytecode verifier, the thread management, management and debugging tools and timers.
- *MMU*: the Memory Management Unit. It contains the reference handling, finalization, garbage collection and fast allocation mechanisms.
- *ExecU*: Execution Unit. It contains the exception handling, Just-In-Time (JIT) Compiler, Interpreter and JNI mechanisms. It is responsible for executing runtime operations.
- *OS Virtualization* Layer Unit*: a platform*-independent abstraction layer to access host resources. It is the very module that is in charge of application virtualization.

The default Java security model is presented. Its sound design does not prevent the existence of vulnerabilities* in the JVM and in the code of Java applications. Security extensions to the JVM are then detailed. They intend to address the challenges that are brought in by the simultaneous execution of mutually untrusted components*.

4.1 The default Security Model

The security model of Java evolved from HotJava, to Netscape, to Java 2 [DFW96]. It was first designed to support the secure execution of Applets, which are potentially malicious code pieces. This is made possible by the strict language definition and its mechanisms for security enforcement. The shift towards components applications urged the support of modularity*: components from various sources can be executed on the same platform with no mutual access to their data. A comprehensive introduction is to be found in [GED03].

4.1.1 The Java Language

The Java language is meant to be a safe subset of the C language. This means that all language elements that can be exploited in C programs such as the nefarious buffer overflows or memory management activities are removed from the specification. This simplicity intends to free developers from error prone tasks and to enhance their productivity by letting him focus on the features they develop.

The principles that underlie the security of the Java language are the following [GM96]:

- Type safety.
- Automated memory management.
- Bytecode validation.
- Modularity.

Type Safety means that each language element has a given type and can only perform actions that are coherent with this type. In particular type safe languages are free from pointers which enable to manipulate memory addresses independently of their type. Type safety is enforced at three different moments: during the compilation, when the Bytecode is loaded into memory, and at runtime to ensure type coherence with libraries that are dynamically linked. Some authors consider that the Java language is not completely type safe since it enables to handle `Set`s and `Collection`s of data without regard to their actual type. Errors then lead to thrown exceptions at runtime. The solution at the language level is the use of Generics which force the data structures to only contain objects of a given type. For backward compatibility

reason, this feature is not made mandatory. A solution exist in certain SOP platforms such as Guice, that create objects in an 'extraordinarily type safe' manner (see Section 5.2.2) and thus prevent runtime exceptions which could occur due to cast errors.

Automated Memory Management consists in hiding memory allocation and de-allocation from the language. This is done by postponing these operations from the compilation phase as in C and C++ to runtime. The virtual machine is itself responsible to allocate required memory at object instantiation and to release the memory that is not longer used through a Garbage Collector (GC). Garbage collection is performed by maintaining a list of objects in use. When an object is dereferenced by all its callers, it can be removed safely. This prevents development errors. It also prevents malicious code from directly accessing the memory, which makes the resulting code more robust against attacks*. Additional mechanisms are enforced to guarantee the integrity of the memory. For instance, array bounds are checked to prevent reading or writing data outside the defined data structures. This prevents the buffer overflows and therefore eliminates one important attack vector* used for executing malicious code.

Bytecode Validation mechanisms are presented in Section 4.1.2. They guarantee that the executed Bytecode is compliant with the specification and therefore prevent the forgery of instruction suites that could not be derived from the source code by a valid compiler.

Modularity mechanisms are presented in Section 4.1.3. They enable to execute several components with no naming conflict and unintended access. In particular, they target classes that are loaded from diverse locations such as remote code servers.

The security mechanisms for the Java language mix the Software Security*, *e.g.* language features, and the Application Security* approaches in a much integrated way such as modular support. It is a powerful illustration of the complementarity of the two techniques: Software Security is enforced when possible to assert comprehensive security properties in an efficient manner. When it is not sufficient, it is complemented with Application Security mechanisms.

4.1.2 Bytecode Validation

Loaded Bytecode is considered as untrusted. It must therefore be completely validated before its execution to ensure it is compliant with the specification. This validation occurs in the Bytecode verifier of the JVM immediately before the code is loaded in memory [GM96]. It is complemented by runtime checks in two cases: when the required information is not available during validation, *e.g.* when several libraries are used together and when optimizations make runtime checks more efficient.

Figure 4.2 shows the process of Bytecode verification.

The code is generated from a `.java` source file to a Bytecode `.class` file by the compiler. Language specifications are enforced here but no guarantee exists that 1) the compiler is valid [Tho84], 2) the Bytecode has not been forged without compiler or 3) the file is not modified during its transfer through the network or any other medium. When it is loaded, the Bytecode is therefore first checked in the System Service Unit (SSU) of the JVM, before being forwarded to the Execution Unit (ExecU) for JIT compilation or interpretation.

The validation that is performed by the Bytecode verifier is made of following tasks [Sch02a]:

- *Structural correctness*: checks attribute lengths, class file validity.

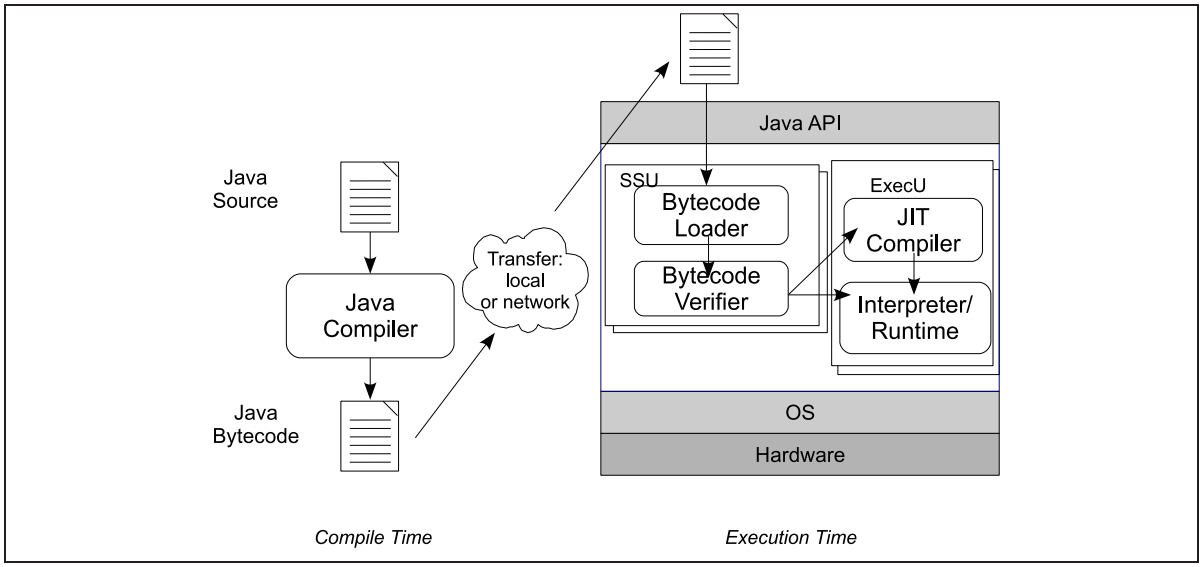


Figure 4.2: The process of Bytecode verification

- *Data type correctness*: checks that final classes are not subclassed and final methods not overridden, that all classes except `java.lang.Object` have a superclass, that all fields and methods references have legal name, classes and type signature.
- *Bytecode checks*: checks 1) static constraints: control flow correctness, validity of exception handlers; 2) static constraints: reachability of subroutines and exception handlers, data flow validity such as absence of buffer overflow and underflow.
- *Symbolic reference management during runtime*: checks the validity of current class references, type safety of method calls and field access, visibility of method calls and field access.

This approach for Bytecode validation pertains to the category of Software Security, because properties that make the code secure are deeply embedded in it. However, it is a very unflexible method that can only enforce permanent properties.

4.1.3 Modularity

Support for modular programming intends to enable the execution of mutually unknown components that do not necessarily trust each other.

Class Loaders built a subsystem of the JVM that is responsible for finding classes and making them available for execution[1]. They are organized as a hierarchy, as shown in Figure 4.3[2].

The presence of application specific class loaders is optional but required to support namespace isolation between the components. When executed in its own class loader, a component have access to the classes of its parent class loaders as well as to its own classes. This ensures that no naming conflict can occur if several components contains classes with the same

[1] http://interviewjava.blogspot.com/2007/04/what-is-class-loader-and-what-is-its.html, read on 2008/08/18
[2] http://java.sun.com/docs/books/tutorial/ext/basics/load.html, read on 2008/18/08

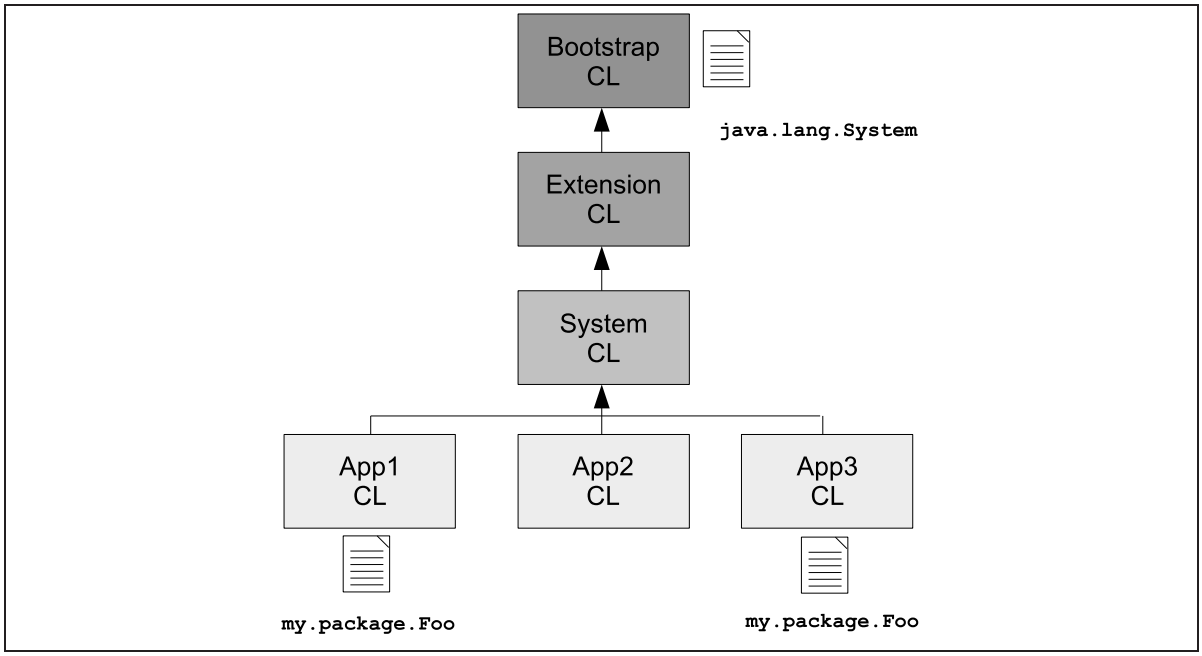


Figure 4.3: The hierarchy of class loaders

name. As shown in Figure 4.3, the component 1 and 3 can both have an internal class named `my.package.Foo` that have different implementations. They will share system classes such as `java.lang.System`.

Each class loader have the following behavior. It first checks whether the class to be loaded is already available. If this is the case, it returns it. Otherwise, it delegates the search for the new class to its parent class loader. If this request is unsuccessful, *i.e.* if none of the class loader in the hierarchy finds the class, the current class loader searches for the class itself with the `findClass()` method.

The default class loaders of the hierarchy are the bootstrap class loader, the extension class loader and the system class loader. The bootstrap class loader has access to the classes that are required to launch the JVM. In particular, it is responsible for the runtime classes (`rt.jar` archive) and the internationalization classes (`i18n.jar` archive). The extension class loader is responsible for extension classes such as the implementation of cryptographic libraries. It concerns the archive that are stored in the `lib/ext/` directory such as the `sunjce_provider.jar`, `sunpkcs11.jar` or `localedata.jar` archives in the case of the Sun JVM. The system class loader is responsible for the classpath, which is set through the `CLASSPATH` environment variable or the `-classpath` or `-cp` option of the JVM. Additional class loaders can be defined by the applications or by the framework used. For instance, the OSGi framework loads each bundle* in a dedicated class loader. Classes can be shared between the bundles and thus between the class loaders through a specific package `import` and `export` mechanism.

The obvious security benefit of class loaders is the namespace isolation which prevents one component to access another without being explicitly allowed to. Moreover, system classes are shared which prevent inconsistencies.

However, class loaders are not designed to support strong isolation. In particular, static

variables in parent class loaders are shared. For instance, the `System.out` variable in the bootstrap class loader can be accessed and modified by all classes. This can lead to unrequired interference such as denial-of-service or to data transmission that should not occur. Similarly, static synchronized calls can be exploited. Moreover, they do not support resource isolation which is the second necessary protection besides namespace safety. The last drawback of class loaders is that they can not be simply removed to uninstall a component since this could imply inconsistencies in programs executed by others. For all these reasons, class loaders build an interesting feature of Java component platforms* but can not be considered to be secure.

Protection Domains * are the solution proposed by Sun for securely executing components in parallel. This approach is also known as *Stack Based Access Control* (SBAC) since it relies on the inspection of the call stack to determine whether all callers have sufficient rights to execute a given method. It is defined in [BG97, WF98]. A technical presentation is to be found in [Sun].

Per se, a Protection Domain is the set of objects currently directly accessible by a principal* [SS73]. A Principal is the software representation of an entity (an individual, a corporation, a login, a place in the file system) to which execution rights are granted. In the case of Java component platforms, a Protection Domain thus represents all objects that can be accessed by one component. It is made of all unsensitive code which can be called seamlessly and of sensitive code to which access is granted explicitly through a permission policy. The use of protection domains in Java applications requires that the security manager of the JVM is enabled and the policy file defined according to the requirements of the applications:

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

Security checks are performed at runtime in the code of the application, the framework or the standard libraries through a call to the security manager:

```
mySecurityManager.checkPermission(
  new ServicePermission(serviceName, ServicePermission.GET));}
```

The example shows the permission check in the OSGi platform when a service* is requested. The security manager then performs following operations: it reconstructs the call stack that leads to the sensitive call and identifies related principals; it checks whether all principal have sufficient rights to perform the sensitive calls. If these rights are not granted, a `java.lang.SecurityException` is thrown.

One of the main limitations of protection domains is the important overhead they imply, especially when their number and the frequency of security checks increases. An important improvement effort has been dedicated to the original implementation [GS98]. Alternative solutions, *e.g.* through static analysis, have been proposed. However, they remain research efforts and have not succeeded at providing more efficient solutions.

Evaluation of Java Security After its release, the Java 2 language and platform have been subject to an extended analysis by the industrial and academic community [CM99]. The most important discussion arose about the conservation of the property of type safety in the presence of several class loaders. A subset of the Java language has been proved to be type safe [DE97], before an implementation error in the class name resolution was identified [Sar97].

This error has been formally modeled afterwards [JLT98] and corrected in later releases of Sun JVM.

Moreover, following limitations of the Java Stack Inspection policy are identified in the literature [ES00]:

- If a new thread is created from within a `doPrivileged` block then that thread will continue to enjoy amplified privileges, even though its code might not be within the scope of a `doPrivileged` block and even after its creator has exited from within the `doPrivileged`. This is because the new thread starts execution with a copy of its creator's call-stack (whose top frame is marked as being within the scope of a doPrivileged).
- When a class B extends some class A but does not override A's implementation of a method `foo()`, then the protection domain for A (and not B) will always be used by `checkPermission` for `foo`'s stack frame. Because B can extend A in ways that may act the semantics of `foo`, (such as by overriding other methods), one might argue that the wrong protection domain is being consulted.

4.2 Vulnerabilities

Vulnerabilities in Java-based systems* can be found in two distinct locations: the virtual machine and application code. Their disclosure acts as a strong incentive for both JVM and application developers to increase the security level* of the system.

4.2.1 Vulnerabilities in the Java Virtual Machine

Generic JVM Vulnerabilities can be identified. A comprehensive analysis of bug databases of Sun JVM and Jikes[3] is proposed by [COR06]. They can be exploited to perform at least denial-of-service exploits*. Statistics related to the origin of failures are extracted from a selected set of bug entries: 103 bugs from Sun and 28 bugs from Jikes. Consequently, only those two VMs are directly concerned by the given results. The conclusions are the following:

- Garbage collection is responsible for 73 % of the failures in the memory management unit (MMU).
- Runtime support operations such as method invocation, stack frame allocation and deallocation, exception handling and optimized Just-in-Time (JIT) compilation are responsible for 77 % of the failures in the Execution Unit (ExecU).
- Thread management is responsible for 76 % of the failures of the System Service Unit (SSU).
- Most of the failures (80 %) occur when the JVM is running under an important workload.

JVM optimizations such as garbage collection and JIT compilers are clearly identified as reliability bottlenecks. A clear trade-off appears here between performance and reliability.

These JVM failures worsen when the applications are running a long-time without interruption. This phenomenon is called *aging* and is especially present for application servers. It leads to the following behavior of an example mail application running on a Java server:

[3] http://jikesrvm.org/

- A consistent memory depletion trend (up to 50 KB/min) during periods of low garbage collector activity.
- A consistent throughput loss (up to 2.4 KB/min) has also been observed.
- The Just-In-Time compiler is responsible for a not negligible memory depletion trend (numerical value is not given).

The experiment is conducted with a mail server called James for a duration of 100 hours with a workload generator.

More exotic vulnerabilities can be exploited. For instance, memory errors can be used to abuse a virtual machine [GA03]. The attack relies on an aleatory error in a program with a very repetitive address scheme. The shift of any bit in one method leads to type unsafe behavior. The attack is sped up through physical access to the machine: the authors use a simple light bulb to perform the bit shift.

Moreover, each specific implementation of the JVM introduces new vulnerabilities. Mobile platforms are especially fragile to such attacks since they are meant to be extended through MIDlets provided by third parties. A vulnerability analysis is proposed for Sun Connected Limited Device Configuration (CLDC) by [DSTZ05]. Following flaws* are identified: SMS can be sent by malicious MIDlets, network and cryptographic errors exist in the MIDP Reference Implementation (MIDP RI), data is recorded in stores not protected from malicious attacks, no quota can be set in data storage, low level helper functions* for high level libraries are available for execution by MIDlets, and MIDlets can be transferred from a device to another by the user. These vulnerabilities can not be extrapolated to other VMs or other implementations. However, they are representative of the type of vulnerabilities that can occur in Java environments.

4.2.2 Vulnerabilities in Code

Various authors propose lists of vulnerabilities that are to be avoided and coding rules that help prevent them. Though these lists are partially overlapping, they each provide specific entries.

Source Code The first set of vulnerabilities that can be identified in the Java source code consists in counterintuitive behaviors of Java programs [Blo01, BG05] known as *puzzlers*. They concern the range limit of numbers or intricate exception and thread management. These 'strange' behaviors lead to code misunderstanding and to infinite loops or program deadlocks. An example of such an infinite loop is given in Listing 4.1. Its origin is the way the integer counter is incremented: when it reaches $Integer.MAX\_VALUE$, the count goes on with $Integer.MIN\_VALUE$.

Listing 4.1: Example of a counterintuitive infinite loop in the Java language

```
public static final int END = Integer.MAX_VALUE;
public static final int START = END - 100;
  // or any other start value
for (int i = START; i <= END; i++);
```

The reader is invited to refer to the books or to our technical report [PF07a] for selected puzzlers that turn out to introduce vulnerabilities.

The second set of vulnerabilities are real features that can be exploited to realize exploits against the JVM [Lon05]. These features are:

- Type safety: the use of the same type name for several classes can lead to confusion. This is especially true with fields that are private or protected that are made globally available for compilation reasons, for instance in nested classes.
- Public Field: clearly, no sensitive data should be stored in public fields, as they are accessible from the whole application.
- Inner Classes: since inner classes do not have explicit Bytecode support and are compiled as simple classes, their private members (fields and methods) have a package visibility to allow access by the outer class. This can be abuse by any class in the same package to access the private content of the inner class. This is illustrated in Listings 4.2 and 4.3.
- Serialization: enables to store the state of a program as a byte stream. If kept in clear, this data can be accessed from third party programs.
- Reflection: enables Java program to analyse and to modify itself.
- JVM Tool Interface (JVM-TI): is a tool for monitoring and modifying the internal state of a running JVM. It does not require specific permissions to run with the default security manager.
- Debugging: the Java platform Debugger Architecture is built on JVM-TI. It enables to monitor and modify the state and thus the data of Java programs. Since access control is not enforced, private fields can be access transparently. Its use can be prevented by the security manager.
- Monitoring and Management: the JMX management tool enables to monitor class loading, thread state, stack traces, deadlock detection, memory usage, garbage collection, operating system information. It can also introduce remote management.

Some of these features such as reflection or debugging can be protected through a security manager. Others such as remote management imply that required libraries are installed on the system under control. However, most of them stay open for exploitation.

Listing 4.2: Private inner class and methods: source code

```
public class HelloWorld
{
  ...

  private class HelloWorldPrinter
  {
    private String textHello="HelloWorld";
    private String textGoodbye="Goodbye World";

    private void sayHello()
    {
      System.out.println(textHello);
    }

    private void sayGoodbye()
    {
```

```
        System.out.println(textGoodbye);
    }
  }
}
```

Listing 4.3: Private inner class and methods: Bytecode

```
class fr.inria.ares.helloworld.HelloWorld$HelloWorldPrinter
  extends java.lang.Object{
fr.inria.ares.helloworld.HelloWorld$HelloWorldPrinter(
  fr.inria.ares.helloworld.HelloWorld,
  fr.inria.ares.helloworld.HelloWorld$1);
  Code:
    0:   aload_0
    1:   aload_1
    2:   invokespecial   #3;
//Method "<init>":(Lfr/inria/ares/helloworld/HelloWorld;)V
    5:   return

static void access$100(
  fr.inria.ares.helloworld.HelloWorld$HelloWorldPrinter);
  Code:
    0:   aload_0
    1:   invokespecial   #2; //Method sayHello:()V
    4:   return

static void access$200(
  fr.inria.ares.helloworld.HelloWorld$HelloWorldPrinter);
  Code:
    0:   aload_0
    1:   invokespecial   #1; //Method sayGoodbye:()V
    4:   return

}
```

Sun provides detailed guidelines relative to exploits that can be performed on Java applications [Sun07, Lai08]. These guidelines aim at preventing weak code by promoting good development practices. They are organized in following categories:

1. Accessibility and extensibility
    - 1-1 Limit the accessibility of classes, interfaces, methods and fields
    - 1-2 Limit the extensibility of classes and methods
    - 1-3 Understand how a superclass can affect subclass behavior
2. Input and output parameters
    - 2-1 Create a copy of mutable inputs and outputs
    - 2-2 Support copy functionality for a mutable class
    - 2-3 Validate inputs

3. Classes
    - 3-1 Treat public static fields as constants
    - 3-2 Define wrapper methods around modifiable internal state
    - 3-3 Define wrappers around native methods
    - 3-4 Purge sensitive information from exceptions
4. Object construction
    - 4-1 Prevent the unauthorized construction of sensitive classes
    - 4-2 Defend against partially initialized instances of non-final classes
    - 4-3 Prevent constructors from calling methods that can be overridden
5. Serialization and Deserialization
    - 5-1 Guard sensitive data during serialization
    - 5-2 View deserialization the same as object construction
    - 5-3 Duplicate the SecurityManager checks enforced in a class during serialization and deserialization
6. Standard APIs
    - 6-1 Safely invoke `java.security.AccessController.doPrivileged`
    - 6-2 Safely invoke standard APIs that bypass SecurityManager checks depending on the immediate caller's class loader
    - 6-3 Safely invoke standard APIs that perform tasks using the immediate caller's class loader instance
    - 6-4 Be aware of standard APIs that perform Java language access checks against the immediate caller

Another similar classification* is defined by McGraw and Felten as the '12 rules' for developing more secure Java code [MF98].

Flaw disclosures are meant to improve the quality of the concerned software. Several publications that have had an important impact such as the Java 'Hall of Shame' [Dep00], the vulnerabilities identified in web browsers by the 'Last Stage of Delirium' group [The02] are now outdated for this reason.

Bytecode A List of vulnerability types identified at the Bytecode level is given by the FindBugs tool[4] [HP04], in the 'Malicious code' category.

- EI: Expose Internal representation by returning a reference to a mutable object.
- EI2: Expose Internal representation by incorporating reference to mutable object.
- FI: Finalizer should be protected, not public.
- MS: Excessive visibility of fields and methods. Examples are static fields, non-final fields and fields of mutable types such as array and Hashtable.

These vulnerabilities are only partly redundant with previous guidelines. They provide the advantage of being identified in an automated way.

These threats call for two important shifts in the Java development community: the training of developers to make them aware of the flaws they may introduce in their applications and the extensive use of automated tools to identify at least the most common of these issues.

[4] http://findbugs.sourceforge.net/bugDescriptions.html

4.3 Security Extensions

The limitations of the default security model and the existence of vulnerabilities in Java applications lead to the development of security extensions: platform extensions, specific coding constraints and behavior injection mechanisms.

4.3.1 Platform Extensions

Platform extensions aim at isolating applications and making them accountable for the resource they consume.

Isolates The goal of isolates is to run independent programs in the same Java runtime environment while preventing interactions between them [Cza00]. The execution in an isolate should not be different than the execution in a dedicated VM from the point of view of the program. Isolates are specified in the Sun JSR 121 [Jav] and detailed in [Bry04]. The reference implementation is provided in the Sun Barcelona project[5] and following and is known as the Multi-tasking Virtual Machine (MVM) [GCLDBT03]. It is available for Solaris only.

Each isolate executes a task, *i.e.* an application with a `main` method, in a dedicated thread. This thread enforces a strong isolation with others so as to overcome the limitations of class loaders in particular: an isolate has its own system static variables, to prevent any interference (see the example of the `System.out` variable); it has its own instance of `java.lang.Class` objects; it has its own set of interned `String`s, *i.e.* immutable string values. Consequently, the communication can only be performed through IPCs or networking techniques, which prevents the realization of efficient multi-task applications with this technique.

The benefits of isolates are manyfold [Bry04]. First multi-programmed and multi-part application environments become platform-neutral which enhances the testability and the manageability of systems. Scalability is improved. Next the security is strongly improved through the combination of OS security such as process isolation and absence of data shared between applications and VM security which provides a protection against direct access to OS resources . Moreover, access to operating system mechanisms is not longer required to manage multi-application systems. Lastly resource management is made possible because each application is executed independently from others even if this is not part of the Isolate specification.

Though these important gains in the support of multi-task environments, Isolates and the MVM are so far only used in a limited manner. This is due peculiarly to their lack of convenience for production environments and the lack of transparency of the proposed programming model [Bry04, GTCF08]. Isolates are inappropriate for a certain number of widespread Java environments such as devices with restricted resources, with no address spaces isolation between processes or with no thread support. Moreover, they provide no centralized management of the Java entities in the system and are very inefficient in term of resource consumption especially at the instantiation of a new task, since all execution and management mechanisms are duplicated.

Sun Resource Management Interface [CHS+05] (RM interface) is specified by Sun JSR 284[6]. It is an extension of the isolate concept. The RM interface enables to model and monitor

[5] http://research.sun.com/projects/barcelona/
[6] http://jcp.org/en/jsr/detail?id=284

any resource type and in particular to track the resource consumption for each isolate. It is an follow up of the JRes (Java Resources) prototype [CvE98]. Its goal is to enable a trusted part of the Java system to be informed of thread creation, to state an upper limit to the resources that are consumed by a thread or set of threads and to define 'overuse call-backs', *i.e.* actions to be performed when the limits are exceeded. It is transparent for the monitored applications. It requires access to the VM. The core entities are the resource, resource domain, dispenser, consume action and reservation.

A resource is identified unambiguously by the qualified name* of its implementation class. It is modeled as a set of properties, and can be `disposable`, *i.e.* a program span where it is considered to be consumed can be identified, `reservable`, `revokable`, *i.e.* the available resource amount can be restricted without impact on the application behavior other than performance, or `unbounded`, *i.e.* no limitation exists to the consumption of this resource as for the absolute CPU time.

A resource domain is the instance of a policy regarding resource consumption. Several isolates can be bound to the same resource domain and therefore to the same policy.

A dispenser is an object that controls the quantity of a given resource that is available for resource domains and thus to computation. This control is performed based on the current usage and policy.

A consume action is executed when a request to consume a given type of resource is made. It can be persistent, *i.e.* be repeated at each similar request or not, and synchronous or not. Specific triggers which launch the consume action and call-back mechanisms which follow the action can be defined.

The reservation of a resource guarantees its availability.

The important benefit of the RM interface is that is is fully compatible with existing applications. Its main restriction is that it allows to perform very sensitive operations such as the limitation of the resources that an isolate can consume which can be exploited to performed denial-of-service attacks. This implies that the RM API is to be used conjointly with the security manager and that suitable permissions are set carefully.

Alternatives Several alternatives exist for process isolation.

KaffeOS [BHL00] is an extension of the Kaffe VM that implements isolation as well as resource management and sharing. It supports resource management through the introduction of OS processes at the VM level and the use of a user/kernel boundary which enables to split user processes that can be terminated arbitrarily from system processes that can not be.

J-Seal 2 [BHV01] is another Java micro-kernel architecture that aims at executing mobile objects that are not trusted. Resource accounting is performed for low level resources such as CPU and memory and for high level resources such as thread count.

RAJE [GS02] (Resource Aware Java Environment) is an extension of the standard Java2 platform. It aims at monitoring resource access and consumption for global resources such as CPU and memory and for resources consumed by each thread.

J-RAF (Java Resource Accounting Framework) [HK03] enables transparent monitoring of resource consumption performed through Bytecode transformation for CPU, memory and network bandwidth reification. Contrary to the other solutions for resource management, it does not involve a specific implementation of the JVM and is compatible with most Java platforms up to hardware VMs.

JMX (Java Management Extension) can also be used to perform resource management for

Java systems [Chu04]. The author proposes the JConsole which tracks available memory and OS resources such as CPU time, total and free physical memory, etc.

Similar projects exist for other target environments such as Sun PhoneME Features[7] for CDC/MIDP.

4.3.2 Static Analysis

Besides the modification of the execution platform, the enforcement of security constraints at the code level enables to improve the security properties of the applications in a important manner. Static analysis consists in verifying the compliance of the code with explicit constraints. It is usually performed at development time to enhance the quality of the code. This enables to perform costly verifications which could not be performed at runtime.

Static analyzers are often executed as a post-development mechanism. They intend to be more lightweight than proof languages such as B that enable to specify systems in a formal manner. They therefore aim at providing a reasonable trade-off between the cardinal properties of analyzers: completeness, soundness and effectiveness. Completeness means that the checked property holds for all code that is analyzed. Soundness means that no false positive, *i.e.* code excerpts that are identified as being errors but are not, are produced. Effectiveness relates to the capacity of tools of finding real bugs in software.

A comparison of several tools for static analysis of Java programs is proposed by [RAF04]. The authors consider JLint, PMD FindBugs, ESC/Java2, Bandera. The conclusion of this study is that the different tools find non-overlapping sets of bugs because they each take a different approach. Consequently no tool is sufficient in itself and all tools are complementary. Moreover, no tool can always report correct results and false positives are unavoidable.

Most tools are targeted at identifying generic bugs. Some such as FindBugs consider a subset of security bugs. One tool that is presented to the research community, JSLint, is specifically targeted at security bugs. Numerous commercial tools also exist but no precise technical data are published.

Bug Pattern Detection consists in identifying generic bugs that are described as abstracted programming constructs, the *bug patterns*. It focuses on local syntactic features of the code. This approach aims at improving the global quality of the analyzed software. This makes it more robust and less likely to let malicious users take advantage of its failures. However, this can not be considered as a security mechanism since no specific attention is paid to exploitable vulnerabilities and their prevention.

Examples of bug pattern detectors are JLint and PMD. They both target Java source code. JLint[8] also supports inter-procedural data flow analysis for detecting locks, in particular in multi-threaded programs. It is not easily extensible. PMD[9] [Cop05] supports two types of bug patterns detectors: XPath[10] expressions and Java analyzers. It contains a series of predefined bug patterns and is easily extendible. It is used for instance in the Eclipse IDE. Example of academic tools for bug pattern detection are Spoon [NP06] which analyzes the source code of Java programs, and Tom [MRV03] which analyzes the Java Bytecode.

[7] https://phoneme.dev.java.net/
[8] http://artho.com/jlint/
[9] http://pmd.sourceforge.net/
[10] http://www.w3.org/TR/xpath

Findbugs [HP04] is another tool for bug pattern detection and intra-procedural data-flow analysis. It is targeted at Java Bytecode. As previous tools is designed to identify development bad practices. It includes a small subset of bugs that are security vulnerabilities. These bugs are listed in Section 4.2.2.

FindBugs is based on four complementary analysis techniques:

- Class structure and inheritance hierarchy.
- Linear code scan with a state machine.
- Control flow analysis.
- Data flow analysis.

It is implemented using the BCEL library for Bytecode manipulation. The output of FindBugs is much smaller than the one for instance of PMD. It identifies only bug patterns that directly lead to errors when executed.

Experiments with FindBugs lead to a number of findings that highlight the benefits of such tools: 1) even well tested code written by experts contains a surprising number of obvious bugs; 2) Java has many language features and APIs which are prone to misuse; 3) simple automatic techniques can be effective at countering the impact of both ordinary mistakes and misunderstood language features. Code static analysis is shown to both increase the quality of the produced code and the developer awareness and knowledge.

JSLint [VMMF00] is the first tool dedicated to the identification of security bug patterns that is advertised to the research community. Its goal is to automatically utilize existing security knowledge to prevent bugs familiar to the security community. It enforces in particular the first 11 of the 12 rules of secure coding by McGraw and Felten presented in Section 4.2.2. Is is targeted at source code. This tool is not to be confused with JSLint[11], the JavaScript Verifier.

Annotated Code consists in expressing the constraints that variables must comply with throughout the execution. The verification is enforced at runtime when the actual value of these methods is available. It implies the insertion of additional code in the application and is strongly dependent upon the semantic of the code that is monitored. Two important examples of tools that supports annotated code are ESC/Java 2 and Bandera.

ESC/Java 2 [FLL+02] aims at extending the scope of standard static analysis mechanisms such as the Java Bytecode verifier (see Section 4.1.2). It enriches simpler analysis techniques with a theorem prover that performs formal verification of properties in the source code annotations such as preconditions, postconditions and loop invariants. A non-negligible set of bugs can be found without the need for annotations: null dereferences, array bounds errors, type cast errors, race conditions, deadlocks. It enforces modular checking, *i.e.* it does not require the source of all program libraries to be able to check a particular module.

Bandera [CDH+00] is based on model checking. Annotations in the source code express the checks that are to be performed. Several model checkers such as SPIN [Hol03] or Java PathFinder [GPC04] are supported. In the absence of annotations it verifies standard synchronization properties such as the absence of deadlocks. It is therefore not meant to identify generic security vulnerabilities. Its main limitation is that it can not check library calls, even standard ones. It is restricted to the modules under analysis.

[11] http://www.jslint.com/

4.3.3 Behavior Injection

Behavior injection consists in modifying the code to be executed to enable the enforcement of security policies in the application, at runtime. It can be performed in two manners: transparently or through weavable code. In both cases, it is independent of the semantic of the monitored methods.

Bytecode Injection enables to monitor and control the usage of resources as well as to limit functionality [CMS01]. It is performed through insertion of additional Bytecode in existing classes and is usually performed with Bytecode manipulation libraries such as ASM[12] or BCEL[13]. Two main techniques are used: class modification, *e.g.* subclassing non-final classes and method modification for control over object of final classes. It provides for instance a protection against denial-of-service attacks through excessive resource consumption or against the use of sensitive functions such as networking. The authors present two example implementations for Java Applets and for Jini proxies, *i.e.* code that is dynamically discovered, loaded and executed. This demonstrate that the approach is generic enough to be applied to unknown code in a fully automated manner. It implies a performance overhead that is estimated from 5 to 20 %.

Weavable Code Also named aspect code, it consists in integrating code excerpts at predefined cut points. It is a technology dedicated to implement non-functional features. As such security mechanisms are natural candidates for being implemented as weavable code.

To the best of our knowledge, there is little work regarding the application of Aspect-oriented Programming (AOP) to security, in particular in the Java language. A solution, AOSF, the Aspect Oriented Security Framework, is proposed by [SH00, VBC01]. It targets C programs. Their conclusions also apply to the Java language. The principles of AOSF are:

- *Separation of Concerns*: developers should not have to handle security issues.
- *Proactive stance*: weavable code is meant for security mechanisms defined and configured during the development process.
- *Global application.*
- *Consistent implementation*: of security mechanisms throughout the system.
- *Adaptability*: security policies should be flexible enough to implement various sets of security configurations.
- *Seamless integration*: security mechanisms should have no impact on the development of the application features.

The obstacle to adoption are the resistance from the development team, the learning curve for development paradigm shift, the traceability of development and the lack of tool support. The benefits of AOP for security are manyfold. It enables to replace insecure function calls with secure alternatives, to check errors automatically, to prevent buffer overflow through unsuitable function parameters (in C), to perform security logs, to insert additional access control mechanisms and to specify privileged sections in a program.

Meta-programming and Inlined Reference Monitors are also used for behavior injection. Meta-Programming provides a comprehensive framework for code injection [CV01]. It is implemented by Meta-Object Protocols, MOPs, that define a language to access and modify the

[12] http://asm.objectweb.org/
[13] http://jakarta.apache.org/bcel/

code. It provides the advantage not to require the definition of cut points and therefore to be applicable to third party components. Inlined Reference Monitor(IRM) uses code injection to flexibly enforce a wide range of security policies [ES00]. The benefit of IRM is the great flexibility it introduces in the eligible security policies. For instance, it is capable of enforcing Execution Monitoring (EM) policies which include mandatory and discretionary access control, Chinese Wall, type enforcement, and the Clark-Wilson [CW87] commercial policy.

Conclusion No static analysis code for Java is fully dedicated to the identification of vulnerabilities. Consequently, they can be considered as an important help to enhance the quality of the code and thus to improve the security level of the applications but do not bring any security guarantee. As stated with success in the FindBugs project, an important requirement in this effort is to identify the bugs patterns that are vulnerabilities rather than proposing new analysis techniques.

Figure 4.4 presents existing mechanisms for enforcing security in Java systems.

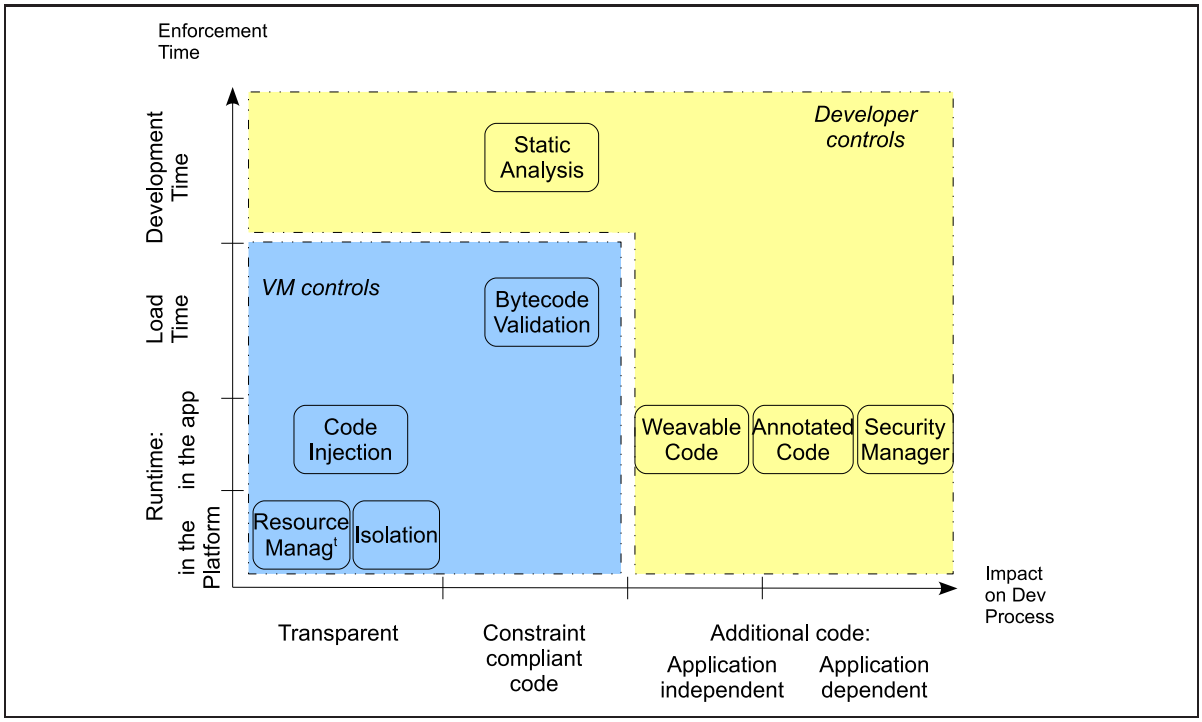


Figure 4.4: Existing protection mechanisms for Java systems

Protection mechanisms for Java are of three types: transparent mechanisms which have no impact on the development process, constraint-based mechanisms that enforce specific properties of programs developed according to the standards, and mechanisms that require the insertion of additional code. Additional code can be independent of the semantic of the application or dependent on it.

Those mechanisms are enforced throughout the life-cycle of applications: during development, at load time, or at runtime. Security mechanisms that are enforced at runtime can be driven by the virtual machine or by the application itself.

Service Oriented Programming (SOP) Platforms and Security

5

Service-oriented Programming* platforms* are execution environments* that support local communications between their components* through well-defined interfaces. The objective is to provide manageable multi-application systems* as well as re-usable applications component [HT99]. They enforce a set of architectural and design patterns to enhance software productivity and to avoid the development of *Ball of Mud* [FY97] Software.

SOP platforms are not to be confused with service-oriented Architecture (SOA) which are concerned with the loose integration of distributed systems or with service-oriented Computing (SOC) which is the research domain related to SOA [PG03]. SOA are sometimes considered to be compliant with the SOP paradigm [SVS02].

Amongst SOP platforms the OSGi platform take a particular place. It is sometimes considered as the 'universal middleware' [CK08], thanks to the support of the full life-cycle of the components, from discovery to uninstallation. It provides itself a rich support for local services* and can be used as a container for higher level SOP platforms such as Spring.

5.1 Service-oriented Programming

5.1.1 What is SOP ?

Service-oriented Programming (SOP) is a programming paradigm where software components publish and use services in a peer-to-peer manner. It is built on Object-oriented programming (OOP) and component models [BC01]. It emphasizes the possibility of improved software reuse [SVS02] and the evolution of applications at runtime.

As its name implies, SOP is based on the core concept of services. A *service* is a contractually defined behavior that can be implemented and provided by any component for use by any component, based solely on the contract [BC01].

The principles and key entities of SOP are the following ones [SVS02]:

- *Encapsulation* consists in the use of well-defined interfaces that can be made remote through *e.g.* interfaces. It maximizes the reusability, information hiding, and separation of concerns.
- A *service interface* is a well-defined software task with well-defined input and output data structures.
- A *service implementation* is the code providing the actual behavior of the service interface(s).
- A *service broker* is a registry that make services interfaces available for service clients*. It provides location transparency (for available components and for available remote hosts) and virtualization* (when several implementation languages are supported).
- *Service listeners* track the life-cycle of services such as registration and unregistration.
- *Semantic based approach* enriches service interfaces with meta-data to ease the binding between services [Ibr08] according to the execution context.

5.1.2 Service Dependency Resolution

SOP as a Design Pattern The SOP paradigm originates as the Hollywood Pattern : 'Do not call us, we will call you' [GHJV94] in which software entities are not active but reactive to external stimuli. It is also named the Dependency Injection Pattern (DIP) [Mar96] as well as Inversion of Control (IoC) [Fow04]. The difference between these different names is mainly a question of conceptual focus. SOP insists on the availability of usable services, while IoC insists on the programming shift from Object Oriented Programming (OOP) and is often bound with Dependency Injection.

Figure 5.1 shows the timeline for the definition and implementation of the *Inversion of Control* Pattern. SOP has first been popularized by the Avalon and OSGi platforms through a simple dependency lookup mechanism. It has next been extended by platforms with a more complete SOP support: Spring, Pico-container, HiveMind, Guice. These platforms are discussed in Section 5.2.

The Inversion of Control pattern can be implemented in several different manners[2] [Fow04]:

- Programmative binding consists in hard-coding the binding between components. This is the case in Guice and in Pico-container.
- Service registry lookup or contextualized dependency lookup consists in servants* publishing the services they provide by a register and clients emitting request to this register to check whether the services they need are available. It is also named the 'Whiteboard Pattern' [KH04]. This is used by the Avalon and OSGi platforms.
- Dependency injection consists in the automated resolution of the service dependencies. The objects that depend on services are populated by the framework with the services they required.
    - Programmative vs. XML configuration.

[2] http://www.picocontainer.org/injection.html

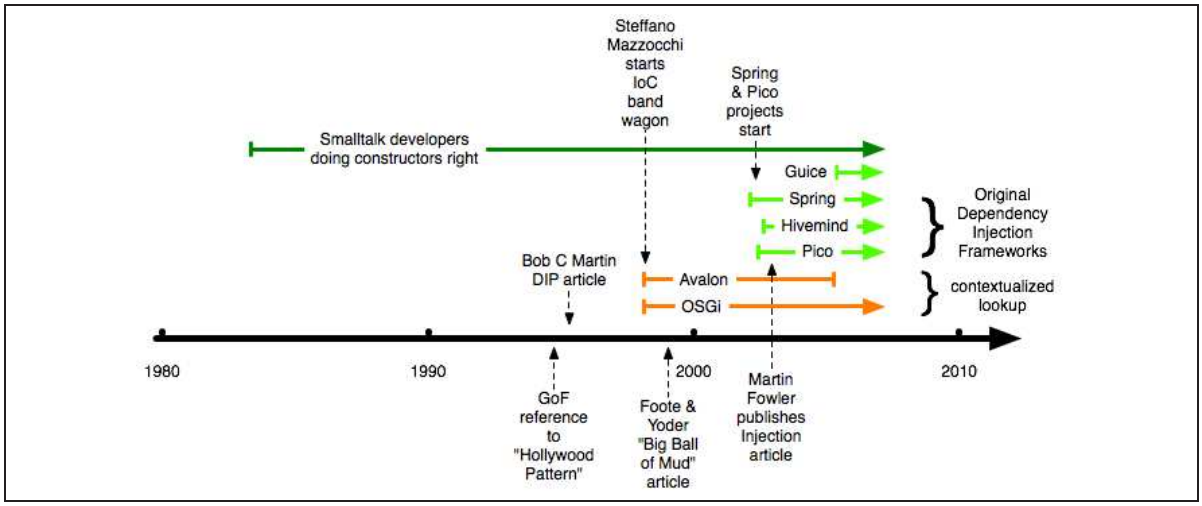


Figure 5.1: The timeline for the *Inversion of Control* pattern [from pico-container documentation[1]]

- Setter dependency injection (SDI), through `setAttribute()` methods; this is the case *e.g.* in Pico-container.
- Constructor dependency injection (CDI), through suitable object constructors; this is the case *e.g.* in Pico-container and Spring.
- Typed field injection, without a dedicated method; this is the case *e.g.* in Pico-container and Spring.
- Annotated global variable injection, which is a variation of field injection using annotations; this is the case *e.g.* in Pico-container.

Good SOP Practices In the frame of the Pico-container project, Dan North and Aslak Hellesoy identify a set of good programming practices for SOP platform[3]. They are not explicitly aiming at providing better security* or reliability, but respecting them lead to better code quality and thus potentially less security issues. Their advices consists in the commands such as: keep a consistent state at all times, have no static fields or methods, never expect or return null, fail fast - even when constructing, be easy to test, chain multiple constructors to a common place, raise checked exceptions when the caller asked for something unreasonable - e.g. open a non-existent file, raise unchecked exceptions when I can't do something reasonable that the caller asked of me - e.g. disk error when reading from an opened file, only catch exceptions that can be handled fully, only log information that someone needs to see.

It is noteworthy that these advices are fully independent of the type of implementation of the IoC pattern. This means that correctness and robustness of services are bound with the implementation of the services themselves rather than with the way they are bound together. This provides us with an useful abstraction to perform security analysis. The analysis of a given implementation of the IoC pattern keeps being valid for other implementations.

5.1.3 Contribution to Security

As for components, the security support in SOP platforms is reduced.

[3] http://docs.codehaus.org/display/PICO/Good+Citizen

Default protection mechanisms are based on the Java security manager. Openwings for instance simply relies on the Java security manager which is presented in Section 4.1.3. The OSGi platform introduces specific permissions that enable to control the life-cycle of OSGi bundles* [All07a]. It also support specific conditional permissions.

This approach does not cope with the security issues that SOP introduces. The IoC forces the developer to consider that the service client are not necessarily trustworthy to the servant this latter does not know who is calling it. However, no mechanism beyond a set of good practices is provided to ensure their security. In particular, the services of a SOP platform share resources to spare unnecessary duplication. As far as no isolation between the services is possible, this opens the way to abuses. Moreover, the sharing of services, which can be singletons by default, between several clients is not safe against malicious code or simply against unforeseen integration of the services.

This lead us to identify two security issues for SOP platforms: sound access control to shared services and resource isolation.

5.2 Existing Java SOP Platforms

5.2.1 Structure

SOP platforms share a similar architecture and to this regard differ mainly by the mechanism for service dependency resolution they implement. It is therefore possible to extract an abstracted structure that represents the features of all SOP platforms.

Figure 5.2 shows the structure of a service-oriented programming (SOP) platform.

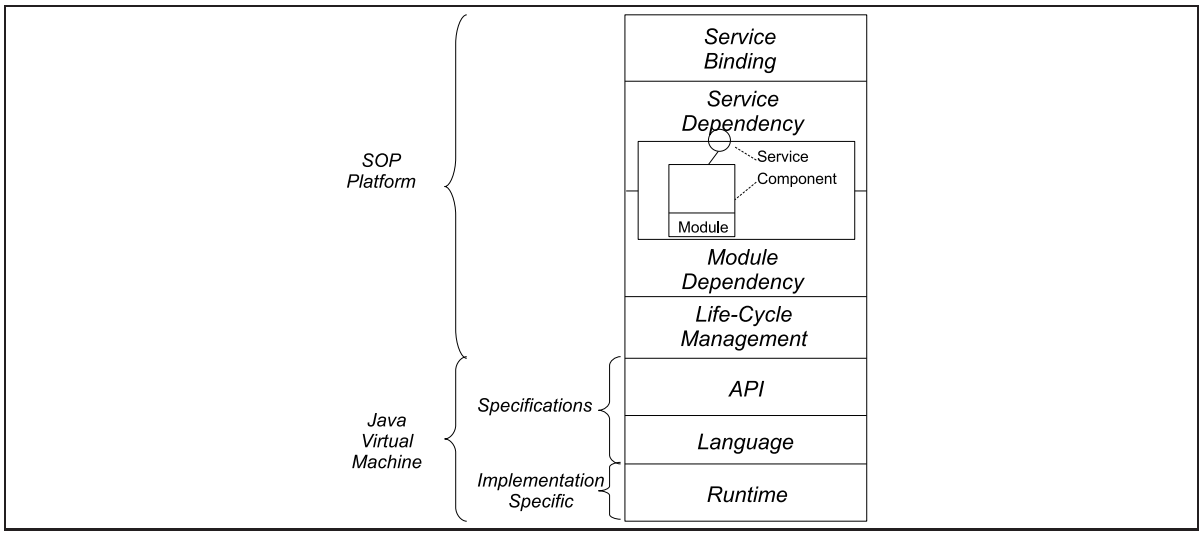


Figure 5.2: The structure of a SOP platform

A SOP platform is composed of following layers: the virtual machine and the SOP platform. The virtual machine is compound of the runtime, the language support and the standard API. The SOP platform is compound of the life-cycle management, module dependency, service dependency and service binding layers. SOP Components are executed inside the SOP platform. They typically share modules as full components or packages and communicate through services.

The virtual machine is an application-level VM. It is responsible for executing the code of the programs. The runtime provides the interpreter and JIT (Just-in-Time) compiler for the Bytecode. Language support provides language specific features such as Bytecode verification. The standard API provides classes that can be used by the applications. It differs from VM to VM, according to the VM features and to the target system: embedded devices have not the same requirements and expressiveness as desktop environments.

The SOP platform is responsible for managing service-oriented programming features, of course, but also the components themselves. The life-cycle layer is optional. It enables to install new components, to load them and to update and uninstall them. The module dependency layer is in charge of binding components together. When one component relies on classes that are provided by another one, a binding is to be created between the supplier and the user of the classes. The service dependency layer provides simple tools to bind services together such as a service registry (Whiteboard) where client components can find their dependencies through lookup. Most SOP platforms include a service binding layer that enables to define the dependencies through a definition that is external to the component themselves.

SOP components are software components containing implementation code and access code that are enriched with SOP services.

5.2.2 Examples

The most representative SOP platforms are the EJBs, Avalon, Spring, SCA, and Pico-Container. The OSGi platform is presented in Section 5.3 and is therefore not detailed here. A comparison of SOP platforms can be read in the white-paper by David Chappell[4].

Enterprise Java Beans (EJBs) are designed to support the reuse of non functional concerns such as persistence, transactional integrity and security. It is one of the first technology to support SOP. EJB 1.0 only supported remote services. EJB 2.0 introduced local services. Configuration occurs through XML descriptors. The last version, EJB3, is defined by the Java Specification Request (JSR) 220 [Sun04]. In addition to beans, it supports Plain Old Java Objects (POJOs), dependency injection, and annotations. Configuration can be performed through annotations instead of XML descriptors. EJBs rely on a service broker, the Java Naming and Directory Service (JNDI) to perform dependency resolution. They provide various types of beans which match specific types of services: stateful and stateless session beans for synchronous calls, and Message Driven Beans (MDB) for asynchronous communication. Entity beans used to represent data are replace in EJBs 3.0 by the Java persistence API for performance reasons. EJBs have brought the concept of service-oriented programming into business development. However, the first versions are perceived as heavyweight. Lightweight alternatives such as the Spring framework have been developed as an answer to these limitations. EJBs 3.0 integrate these evolutions.

Avalon [Lor01] is a framework that provides a powerful control over the life-cycle of its components along with service-based communication between them. It is an Apache project[5] that is closed since 2004. SOP is supported through a service repository, as in the OSGi

[4] http://www.davidchappell.com/HTML_email/Opinari_No15_12_05.html, read the 2008/06/27
[5] http://avalon.apache.org/

framework. Avalon suffers from serious drawbacks: each component has to implement several interfaces to be manageable, such as `Configurable`, `Startable` or `Disposable`. This introduces an important development overhead.

Spring aims at providing a lightweight application server with SOP support. It integrates libraries necessary to build web applications. It is made of a main project, the Spring Framework[6] and a set of community project such as Spring Dynamic Modules (DM)[7], which ports Spring over OSGi. Spring supports aspect-oriented programming, a data access framework, transaction management, a Model-View-Controller web framework aiming at replacing Jakarta Struts, remote access through RMI, SOAP, RMI, Corba or EJB integration, Authentication and Authorization, Remote Management, Messaging and Testing. SOP can be performed through dependency lookup and dependency injection. The main limitation of Spring is the verbose dependencies between components which are expressed through XML descriptors.

Service Component Architecture (SCA) is a framework that aims at integrating composite applications in a Service-oriented Architecture (SOA) programming style [BBB+07]. As such, it provides a binding between the SOP and the SOA world. SCA is specified by the Open Service-Oriented Architecture[8]. It is implemented by IBM Websphere, IBM Aqualogic and Apache Tuscany amongst others. SCA provides a comprehensive architectural description of applications. Those are built out of assemblies. Assemblies consist of a series of artifacts, *i.e.* composites, components, entry points, references and wires. Composites are the unit of deployment. They hold services and contain one or several components. Components contain the business logic. Entry points are available services and references are anchors to required services from other composites. Both are linked to the composite and external to components. References are bound with entry points through wires. SCA does not define services meant to be dynamically linked. However, it can be executed on top of Spring or EJB platform which makes it an extension of SOP platforms. SCA is well suited to model complex distributed component applications since it integrates the main component models as well as several communication protocols such as SOAP, JMS, JCA, RMI, RPC. It supports the expression of quality of service requirements such as security, transactions and reliable messaging. Two main limitations are identified in SCA. First, it does not address the performance question, the bottleneck of SOA applications. Next, it focuses on portability instead of interoperability which may be counter-productive and somehow recalls Corba errors.

Pico-Container [9] is a very lightweight SOP framework. It supports a wide range of service dependency resolution implementations: constructor injection, injection into setter methods, injection into annotated fields, injection into typed fields. It is meant to be used in an non obtrusive way. Its benefits are the following ones[10]: no external configuration file, no mandatory annotations, a simple life-cycle support for components (start/stop/dispose), the support of container hierarchies. As far as it is targeted at small projects Pico-container may

[6] http://www.springframework.org/
[7] http://www.springframework.org/osgi
[8] http://www.osoa.org/
[9] http://www.picocontainer.org/
[10] http://www.christianschenk.org/blog/comparison-between-guice-picocontainer-and-spring/

not be well-suited to play the role of an application server. Another of its drawbacks is the restricted support for aspect-oriented programming.

Nano-Container[11] is an extension of the Pico-Container. It adds several features: the management of tree of pico-containers and of class loaders, the support for class-name based composition through reflection, the support for XML, Groovy, Beanshell, Jython and Rhino languages, and a set of components for Web Applications. The life-cycle of the platform comprises a booter to launch the trees of Pico-Containers and a deployer to install components.

Other platforms Other SOP platforms are Guice, HiveMind, Tapestry-IoC.

Guice[12] is a Google project that intends to be small and fast, where SOP services are bound together programmatically. It claims to be 'extraordinarily type-safe'[13], *i.e.* provided services do not need to be cast. Guice intends to be fast. Repeated test have proved it to be at least 1000%, *i.e.* 10 times faster than Spring[14]. This performance implies serious limitations: annotations bind the application to the framework and make the code less portable at least at compile time.

HiveMind[15] is a micro-kernel platform designed for Tapestry IoC V4. Services are provided as POJOs and bound through dependency injection. Hivemind is considered as lacking usability, expressiveness and performances.

Tapestry-IoC v5[16] aims at overcoming the features of Spring, Guice and Hivemind that are considered hindrances. It is the container for the Tapestry project[17]. Services can be managed throughout a complete life-cycle: defined, virtual, realized, shutdown. They are bound together through dependency injection. The benefits of Tapestry-IoC v5 are the wrapping of services which enables to hide original services, to embed service configuration in the module, and to define explicit error messages. Moreover it does not rely on an XML configuration file and is thus more lightweight

SOP platforms for .Net such as ObjectBuilder[18], Castle[19] or Spring.Net[20] are built according to the same principles and are mostly ported from the Java world.

5.2.3 Security

The specification of security in the context of SOP platforms is relatively scarce. This may be due to the fact that SOP platforms can be considered as protected by several available mechanisms: the virtual machine sandbox, the use of harmless HTTP request for web servers or of encrypted communications for sensitive messages and the integration of trusted components only.

However, since modularity* and SOP ease the reuse of components, it does not seem reasonable to put a full trust in them [Mey03]. Securing the execution of components and their interactions may thus become a important requirement for SOP platforms.

[11] http://nanocontainer.codehaus.org/
[12] http://code.google.com/p/google-guice/
[13] http://smallwig.blogspot.com/2007/03/by-way-what-does-extraordinarily.html
[14] http://gorif.wordpress.com/2007/07/05/google-guice-1000-faster-than-spring/
[15] http://hivemind.apache.org/
[16] http://tapestry.apache.org/tapestry5/tapestry-ioc/
[17] http://tapestry.apache.org/
[18] http://www.codeplex.com/ObjectBuilder
[19] http://www.castleproject.org/
[20] http://www.springframework.net/

Protection mechanisms exist for the EJBs, Spring and for SCA. A comparison of the diverse Java execution environments is to be found in [HS05]. Protection mechanisms for the OSGi platform is presented in details in Section 5.3.3. All specified protection mechanisms are related to access control, *i.e.* to the domain of Application Security* as discussed in Section 3.1.2.

EJBs security relies on Java security as presented in Section 4.1 and extends it. Principals*, *i.e.* service callers and providers, are assigned roles. These roles are mapped to Java execution permissions that are enforced through the security manager. Built-in features such as authentication, authorization management and secure communication support are provided along with the EJB framework.

The security mechanisms of the EJBs have been criticized in particular by the Spring project, which intends to by-pass identified limitations: the security model is not powerful and flexible enough for all enterprise applications and the configuration can not be done at a EAR or WAR level. This implies an important configuration overhead during migrations.

Spring Security [21] is dedicated to Web Applications through authentication and authorization features. It intends to by-pass the limitations of the EJBs. It supports in particular portable configurations that can be re-used between various servers. Security policies are set at the code archive level. Authentication features are HTTP, LDAP, JA-SIG Central Authentication Service (otherwise known as CAS, which is a popular open source single sign-on system), Java Authentication and Authorization Service (JAAS) and Java Open Source Single Sign On (JOSSO). It can be integrated with the JBoss, Jetty, Resin and Tomcat containers which enables to still use Container Manager Authentication. Authorizations are handled at the level of web requests, method calls and access control for individuals.

One vulnerability* has been discovered and patched in Spring. It is not located in the framework itself but in common implementations of the web support for data access. It allowed a client to modify the data from another session[22]. Besides this flaw* that requires developers to be careful for their implementation, the Spring framework is believed to be mostly secure.

SCA Security [BBC+07] defines policies for authorization management for SCA frameworks. In particular, it supports WS-Policy [BBC+06], the standard for Web Service Security. Policies are of two types: implementation policies apply to service components and interaction policies apply to the binding of services. `policySet`s are concrete policies that apply to policy domains. They must be compliant with the `intents`, *i.e.* the requirements for individual components and interactions. The intents are also of two types: implementation intents such as authorization for access control and security identity for policy management of the authorized users, and interaction intents such as authentication, confidentiality and integrity at the message and transport layer level.

Other SOP platforms do not explicitly address security issues.

The security issues for SOP platforms are the following ones:

[21] http://static.springframework.org/spring-security/site/index.html
[22] http://searchsoftwarequality.techtarget.com/news/article/0,289142,sid92_gci1321417,00.html

- Few tools are available for controlling the security of systems based on SOP platforms, besides access control and standard secure communication protocols.
- The complexity of resulting systems is not considered as a security risk.
- SOP platforms are secured in an intuitive manner against network attacks*. However, the flexibility they allow is likely to introduce new vulnerabilities that are not considered.

5.3 The OSGi Platform

The OSGi platform is a lightweight SOP platform originally developed for embedded systems. Its powerful support for component life-cycle promotes it as a management layer for application servers. The platform is defined by the Core Specifications Release 4 [All05a] and Release 4.1 [All07a]. It is complemented by a rich set of services through the Service Compendium Specifications Release 4 [All05b] and Release 4.1 [All07b]. The OSGi platform evolves in parallel with Sun Java Specification Requests (JSRs) 277 'Java Module System'[23], 291 'Dynamic Component Support for Java SE' [24] and 294 'Improved Modularity Support in the Java Programming Language' [25]. An introduction can be read in [MK01].

We consider the OSGi platform as a prototypical SOP platform not because it is better than others but because it can be integrated with other SOP platforms. Spring Dynamic Modules (DM) provide an illustration of such an integration.

The work of this thesis is performed using OSGi Release 4 as reference, since it is the current reference for most available implementations.

5.3.1 Principles

The Platform Model The OSGi platform is compound of four layers that are running on top of a Java Virtual Machine: the Security Layer, the Module Layer, the Life-Cycle Layer and the Service Layer as shown in Figure 5.3.

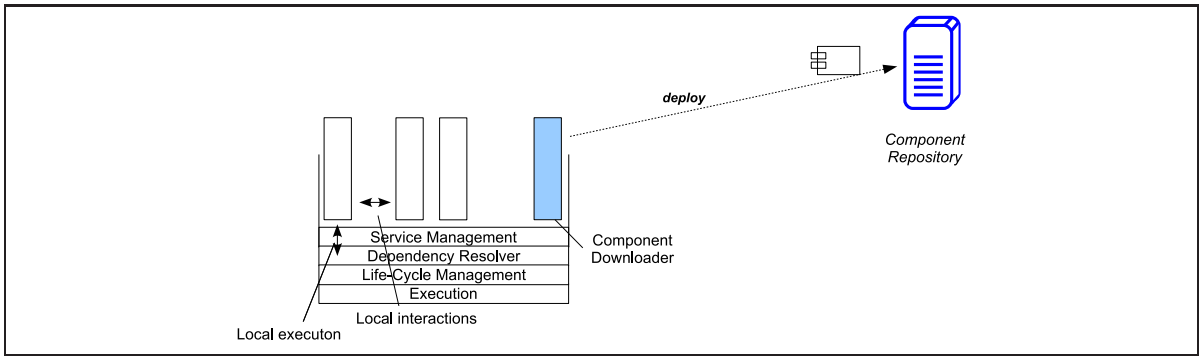


Figure 5.3: OSGi platform model

The Java Virtual Machine can be any JVM that provides an over-set of the Java Connected Device Configuration (CDC) Foundation Profile. Standard JVMs from 1.3 upwards are supported.

[23] http://jcp.org/en/jsr/detail?id=277
[24] http://jcp.org/en/jsr/detail?id=291
[25] http://jcp.org/en/jsr/detail?id=294

The Security Layer supports the validation of digital signature* of bundles and enforcement of execution permissions inside the OSGi platform. Digital signature of OSGi bundles is discussed in Section 2.1.

The Life-Cycle Layer deals with the installing, starting, stopping, updating and uninstalling of the bundles. These advanced management features are in the heart of the power of the OSGi platform. It enables loading new bundles at runtime without disturbing the execution of already installed ones.

The Module Layer acts as a dependency resolver between the bundles. Actually, each bundle is executed in a specific class loader which enables class isolation between applications. To enable the interactions between bundles, dependencies have to be explicitly defined, *i.e.* the name of the required packages are to be mentioned as metadata in the bundles. A bundle can be installed only if all its dependencies are resolved, *i.e.* if all required libraries are available. A specific type of bundle is also handled by the Module Layer: fragment bundles, which are 'slave' bundles that are used to provide configuration informations or context-dependent code to a 'Host' bundle. They cannot be used independently.

The Service Layer provides the support of service-oriented programming [BC01]. Bundles can publish services in a common `BundleContext` under the form of Java Interfaces. They are frequently enriched with specific properties that enable service search using the LDAP request format [How97]. These services can be dynamically discovered and used by other bundles without requiring that dependencies are defined at the Module Level.

A Bundle Downloader bundle (often called 'Bundle Repository' because it handles remote repositories) completes the full support of the bundle life-cycle. It is defined as an OSGi Request for Comments document [AH06]. It performs discovery and download new bundles over the Internet. Metadata format for the Bundle Repositories is named OBR v2 (Open Bundle Repository v2).

The Component Model OSGi Bundles are Java Archive (Jar) files [Sun03] extended to support life-cycle management and dependency resolution. They are composed of two main types of data: Meta-data, and resources such as Java classes, data files and native libraries. Their structure is shown in Figure 5.4.

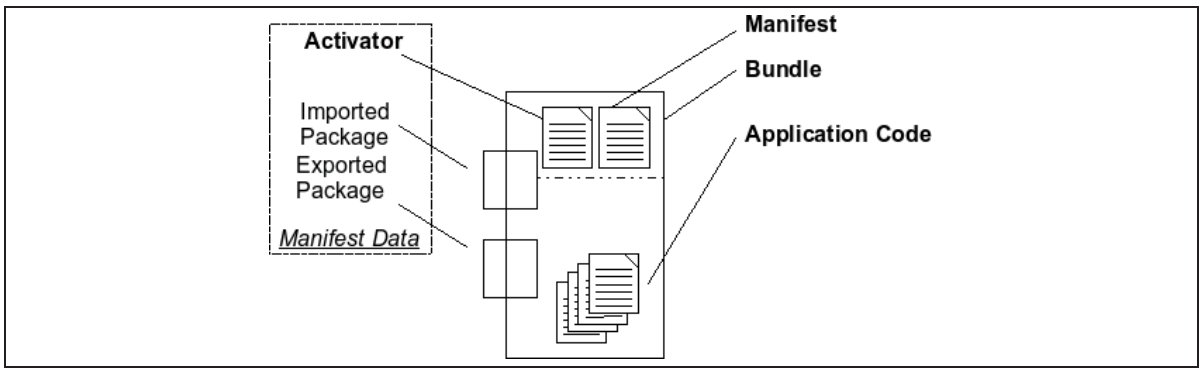


Figure 5.4: OSGi component model

Meta-data are defined in the `Manifest.MF` file of the archive. The most important information is the bundle's symbolic name, its version number, the reference of the `Activator` class and the list of imported and exported packages. The list of required and provided OSGi

services is optional. The `Activator` class is a specific Java class that performs starting activities for the bundle such as configuration and service handling. It contains `start()` and `stop()` methods called when the bundle is started and stopped. The list of imported and exported packages enables proper dependency resolution. All packages that are not exported are kept private in the bundle.

The application code is standard Java code. It can either be used by the bundle only, made available as packages where all classes can be used by third party bundles or made available as OSGi services. Application code could also contain native code as any Java application. This feature breaks all security benefits of the Java Virtual Machine and usually provides full access to the underlying operating system. It should therefore be considered an option for fully trusted code only.

A bundle can be in one of the following life-cycle phases: installed (be manageable from the platform), resolved (installed and with all dependencies resolved, after complete installation or after stopping), active (after its start), or not visible. Through the shared `BundleContext` reference, bundles can be managed by all other bundles in the platform unless specific management permissions are set.

5.3.2 Service Management

The default mechanism for service management is the service registry, or whiteboard. Other mechanisms are optional.

OSGi Service Registry The default lookup mechanism is built with a Whiteboard registry [KH04] which enables bundles to register and find services. Services can be registered with specific properties to support context aware dependency resolution. Service management is eased by a service listener mechanism: events are emitted at each modification of service state. It can also be automated through the Service Tracker, which is defined in OSGi Service Compendium [All07b]. The goal of the tracker is to relieve the bundles from themselves managing data related to the services they use. It supports customization and can be specialized to find services that match specific service name, LDAP filter or service references.

Optional service dependency mechanisms Optional service dependency mechanisms are the Service Binder, iPOJO and the Declarative Services Specification. Service Binder [CH03] is defined as part of the Gravity[26] project for managing integration of components and services as complementary mechanisms [CH04]. It performs dependency resolution according to an XML descriptor. Bundles need to be adapted to support it. Dependency injection is performed through setter classes.

iPOJO (integrated POJO - Plain Old Java Objects)[27] is a service injection framework that claims to be independent of the execution environment and is currently implemented for OSGi [EH07]. The goal of iPOJO is to enable developers to provide POJO-like classes and to let an external XML descriptor define service binding. Services are defined as POJOs that are encapsulated in a component specific container. Dependency injection is performed through typed variables.

Declarative Services Specification is defined in OSGi Service Compendium 4.1 [All07b]. It is a follow up of the Service Binder that supports backward compatibility. The management of

[26] http://gravity.sourceforge.net/servicebinder/
[27] http://felix.apache.org/site/ipojo.html

service is supported by the *Service Component Runtime* (SCR). Dependencies are expressed through a XML descriptor. Service can be accessed through event-based communication or through service lookup.

A rapid comparison of these tools with other SOP frameworks show that they follow a completely different path: most tend to get rid of XML descriptors, such as Pico-container and Guice, whereas all OSGi tools relieve the developer from service management by abstracting the service binding in such descriptors.

5.3.3 Security

The Security Layer The OSGi specification takes advantage of two complementary mechanisms to enforce security: the Java security manager and class loaders [All07a, PO08]. The Java security manager enables to control the access of bundles to sensitive methods that are provided by the JVM such as Runtime access, network or file systems API and by the OSGi framework itself such as the bundle management API. Class loaders isolate the bundles from each others: dependencies between bundles need to be explicited. They are enforced by the framework. Publication is made through package export or through service registration. Other classes, in the first case, and objects, in the second, can not be accessed.

Permission Admin The OSGi framework adds access control support for administration operations. Up to OSGi Release 4.0 excluded, `PermissionAdmin` enforces security through classical security manager checks. They can be set at runtime. Authorization criteria are the type of the action, *i.e.* the class name of the permission, the name argument of the permission, *i.e.* the target of the action and the action that is to be controlled. The authorization scheme is default-deny, *i.e.* all sensitive operations must be allowed explicitly. Authorizations are set according to the bundle signer who is liable for the behavior of the code. Following permissions are defined for the PermissionAdmin:

- `AdminPermission` which controls the actions: `class, execute, extensionLifecycle, lifecycle, listener, metadata, resolve, resource, startlevel, context`.
- `PackagePermission` which controls the actions: `export` and `import`.
- `ServicePermission` which controls the actions: `get` and `register`.
- `BundlePermission` which controls the actions: `provide, require, host, fragment`.

Conditional Permission Admin The Conditional Permission Admin extends the standard Java access control model with fine-grained, arbitrary conditions sets. These conditions can be defined by the developer and may involve the origin of the bundle, but also the state of the application or the user inputs. It also defines an explicit API for permission management to guarantee the interoperability and portability of management tools. It is introduced in the Release 4.0 of the OSGi specification.

The permissions for a given bundle are the intersection of the system permissions and the local permissions. The system permissions are defined statically for the whole platform. The local permissions are defined by the developer and stored in the bundle itself. According to the OSGi specification, the manager of the platform has to check that these local permissions will not enable the bundle to harm the platform once it is installed.

Possible trust levels for the bundles are: `untrusted`, with no or very limited rights, `trusted`, with important rights up to the system and management functions, `system`, and `manage` levels for the system access and platform management respectively.

An example of permissions for untrusted bundles is:

```
{
  (..ServicePermission "..LogService" "get" )
  (..PackagePermission "..log "import" )
  (..PackagePermission "..framework" "import" )
}
```

An example of permissions for trusted bundles from the ACME company is:

```
{
{
  [ ..BundleSignerCondition "* ; o=ACME" ]
  ( ..AdminPermission "(signer=\* ; o=ACME)" "*" )
  ( ..ServicePermission "..ManagedService" "register" )
  ( ..ServicePermission "..ManagedServiceFactory" "register" )
  ( ..PackagePermission "..cm" "import" )
}
}
```

The conditional permissions are expressed by conditions-permissions tuples. The conditions are expressed between square brackets (`[ ... ]`). When several conditions are defined, they are bound with an `AND` relationship, *i.e.* all of them must apply for the permission to be set. The permissions are expressed between parentheses (`( ... )`). So as to make the implementations of this scheme efficient, optimizations are proposed in the specification.

The properties of Conditional Permission Admin are:

Fine granularity : the developer sets the minimal required permissions himself for each bundle. They are stored in the `OSGi-INF/permissions.perm` file in the bundle archive. The OSGi security model assumes that the bundle dependencies are known at development time. The permission should enable the bundle and its dependencies to run seamlessly.

Auditability : Architects can audit the file containing local permissions to identify the authorization requirements for the bundle.

Sandboxing : all permissions that are not explicitly set are denied by default.

They are completed in the Release 4.2 Draft of the OSGi specification [OSG08] by two extensions: the ordering of the permission tuples, to prevent conflicts and ambiguities in the security policies, and the possibility of defining positive (`ALLOW`) as well as negative (`DENY`) permission types.

The limitations of conditional permissions are the following ones. First, the developer must set the permission himself, *i.e.* he must know the implementation of the dependencies of its bundles. This is plausible in commercial OSGi applications but contradictory with the possibility of runtime discovery of application and dynamic resolution of dependencies provided by the OSGi framework. Secondly, the 'least privilege' design principle (see Section

3.3.1 page 33) is not respected: if no local permission file exists, the specification defines that the bundle should run with all permissions.

Improvement of OSGi Security This security scheme introduces two main flaws: the behavior of bundles can not be controlled beyond existing permissions and no resource isolation exist between bundles. Java permissions only enable to prevent the access to methods that perform the security check themselves. This is powerful for system methods such as file system access but is of no help to control the access to sensitive methods provided by third party bundles. The lack of resource isolation allows any ill-coded bundle to consume most system resources such as CPU and memory and even disk space if it is granted sufficient rights. Several proposals exist to remedy to these flaws. First monitoring and managing computer resource usage is possible at the `Thread` level [Yam05]. However, the OSGi platform can be run on limited VMs with no thread support which makes this approach to be difficult to generalize. Next resources and access isolation can be done in a way compliant with OSGi specification by defining class loader as Isolates [GTCF08]. Since bundles are started in their own class loader, they are thus parted from each other in a natural way. A third proposition consists in monitoring the activities of bundles through an OSGi-internal IDS [HWH07]. This approach shows that the authors consider that bundles can not be completely trusted, even if their issuer is known and that preventive access control through permission is not sufficient. Its benefit is that it supports an advanced and flexible detection scheme. Its current limitation is the important performance overhead, from 50 to 70 %. This is due to the lack of optimization of the prototype which is meant to prove the validity of the detection scheme, not to be a production tool. The idea seems to be a very promising one and still needs to be implemented with reasonable performances. It is to the best of our knowledge one of the first research work that considers that installed bundles can actually be harmful.

These propositions pertain to the category of Software Security* as Java Permissions and to the category of Application Security as the resource control and IDS solutions. Software Solutions as implemented in the JVM proves to lack flexibility for dynamic platforms such as OSGi: new bundles that can be delivered by third party issuers can not be protected through permissions if they are not specifically coded to support it.

Secure Distributed Systems based on the OSGi platform The main use case of the OSGi platform is to be an application server for connected devices with limited resources. As such, security solutions to support distributed applications are a core requirement. Two security challenges are to be tackled: the secure deployment of bundles and secure connections with remote devices.

Secure deployment is supported by the OSGi specification. Our own implementation is presented in Section 2. Several alternatives are proposed, for secure service discovery [KCS05] and for bundle authentication based on Message Authentication Code (MAC) and XML security technology [LKMB05].

Enforcing secure connections with remote devices consists in providing sets of tools and protocols to guarantee the confidentiality, integrity and authentication of these connections. Recommendations for building secure OSGi infrastructure are given in the OSGi Request for Comment (RFC) 18 [OSG01]. A distributed RBAC access control model for the OSGi platform is proposed by [CMPB06].

Writing secure OSGi Bundles The actual challenge in securing OSGi systems is the development of secure bundles by people who are mostly Java developers rather than security professionals. It is therefore necessary to provide them with clear guidelines and efficient tools.

The technical requirements for building secure OSGi bundles are provided by [D'H05]. The author identifies the reason of the relative lack of security in current implementations of the platform: several trust models coexist and suitable protection mechanisms are not available for all. The OSGi platform can be used in following contexts:

- Development environment: OSGi bundles are used as COTS. This configuration implies the same risks as in any Java-based execution environment.
- Updatable platform: the platform is a closed world, bundles are all controlled by the same provider. Security policies can be provided by the bundle with no risk since it is signed. This is the model for most commercial deployments.
- Hosting platform: it allows trusted third parties to provide their bundles. This is the model supported by the OSGi R.4 release.
- Open platform: allows installation of untrusted bundles. The security model is still to be invented.

The first two configurations are closed worlds and do not introduce specific threats. The last two open the way to unknown code. The current state of the technology clearly does not enable to run fully untrusted code. Moreover, it is quite unlikely that platforms will be open to known third parties if no guarantees can be provided that go further than simply good will.

Several threats can emerge from malicious bundles or from ill-coded bundles:

- Malicious bundles can provide Trojan services such as an HTTP server or a logger that can see a lot of valuable data or Trojan classes that leak data or provide weak implementation of cryptographic protocols.
- Badly coded bundles can introduce a wide set of security risks.
    - The right set of permissions is not easy to set. Moreover, this protection mechanism suffers from the absence of repudiation and the impossibility to set resource quota.
    - Stopping a bundle may not be a trivial task when threads or non Java resources are used. Tracking and accounting user resources is a requirement to enable to fully stop the activity of a bundle when it is stopped or uninstalled.
    - Unregistered services can still be used by clients, leading to stale references. The solution is to provide a proxy object that update services when they are unregistered. However, the limitation of proxies is the performance overhead which is proportional to the number of parameters.
    - Listener management is an error prone task since faulty listeners may have methods that never returns. The solution is to catch all exceptions (inclusive `Throwable`s and `Error`s) and to warn administrators when a very serious problem is identified.

The author also proposes to introduce bundle certification to assess their security level*. Such a certification would entail an automated test suite, code review and the use of Conditional Permission Admin.

The next step after writing bundles that do not contain known security flaws is to identify the set of permissions that are necessary for them to be executed seamlessly. This is not a

trivial task since services dependencies are identified dynamically and the use of privileged code, which does not require the caller to have all permissions, introduces new threats.

A solution is proposed by the Sword4j (Security Workbench Development Environment for Java)[28] project [PHK05]. It aims at solving following difficulties: how to give just the right set of them, not too much, not too few ? What permissions are implicitly transfered by privileged code to the client ? What data are transfered between privileged code and its callers ? Which portion of code should be made privileged ?

Sword4J is an automated tool that enables to identify the required permissions as well as code excerpts that should be made privileged. It is specifically designed for the OSGi environment which makes it suitable to cope with dynamic dependencies. Sword4J is built of four parts: a static analysis engine, to identify required permissions, privileges, and 'tainted variables' (data that is passed from privileged to unprivileged code) errors, a jar inspection part, to check the validity of digital signature and permissions, a jar signer and a keystore editor. The limitation of Sword4j is the tendency to define redundant privileged code which means that developers must use it as an helper tool rather than an oracle.

Conclusion The security issues that are identified in the OSGi platform are:

- The target system has a wide attack surface*.
- Executed code is granted many rights and little control can be enforced.
- Bundles should not be trusted, even though their issuer is known.
- The security model for executing components from known third party does not provide strong guarantees so far.
- No security model exists to support the execution of fully untrusted components.

The limitations of the default Java security scheme are:

- It only enables to prevent the access to methods that perform the security check themselves.
- It does not support resource isolation.

[28] http://www.alphaworks.ibm.com/tech/sword4j

Part II

Security Analysis

A Methodology for Security Analysis of Execution Environments

6

Building secure execution environments* requires a complete methodology for assessing the existing vulnerabilities* and protection mechanisms. This can not be performed in an efficient manner through the V Secure Development Life-Cycle which is presented in Figure 3.2 page 25: it is targeted at complete applications and falls short in defining reusable security* knowledge and solutions for a given system.

The motivations for defining a specific methodology for security analysis of execution environments are first highlighted. Next, $\mathcal{SPIP}$, the ***S**piral **P**rocess for **I**ntrusion **P**revention*, is defined. Lastly, its application to our target system, Java Service-oriented Programming* (SOP) platforms*, is introduced.

6.1 Motivations

The motivation for defining a new methodology for security analysis of execution environments originates in the lack of suitable methodologies in the literature. It should support generic requirements for analysis of execution environments as well as the specific requirements for securing Java SOP platforms. To ensure that this methodology provides both reusable knowledge and powerful secure environments, it should meet several high level goals: the support of Knowledge-based Security, System-based Security, and Pro-active Security.

6.1.1 Security Analysis for Execution Environments: a *Terra Incognita* of Software Security

Whereas a great variety of techniques for enforcing Software Security* are available, very few integrated processes have been defined so far. When they exist, these processes are centered around the end-to-end development of stand alone applications. They do not provide methods and tools to build secure execution environments while controlling the security assumptions made by the underlying environment. This is required to usefully build and document a secure SOP platform.

Existing Software Security Processes The most representative processes for Software Security are the Process for Software Security Assurance*, as defined by the IATAC State-of-the-Art report [GWM+07], McGraw process for 'Building Security In' [McG06], and Microsoft Secure Development Life-Cycle (SDLC) [HL02] which are presented in Section 3.1.3, page 24. The first is an abstract process which aims at providing a systematic overview on the Software Security Assurance process. It provides a valuable framework but each step must be instantiated according to the specific requirements of the system under study. The second proposes a linear process aiming at full applications which are to be deployed as is. It is basically a more detailed version of the previous one. It is articulated around 7 specific techniques, or 'Touch-points'. The third process is an extension of a classical software development life-cycle. It has the advantage to be easy to integrate with existing development processes but is hardly tunable for more complex security tasks.

Limitations None of them provides a sufficient support for building secure execution environments, because they do target stand-alone applications. They can not be applied as is to perform the security analysis of a specific execution environment. This means that so far, application developers must assume that the environment which run their applications is fully trustworthy. In any complex system this is not the case. No methods and tools are available to help developer manage the vulnerabilities of their platforms. Moreover, they do not explicitly support the trade-off between security level* and development overhead.

6.1.2 Requirements

The requirements for a methodology for security analysis of execution environments draw from two complementary questions: what are the properties of software security for complex systems ? What are the specific requirements for our target system, SOP platforms ?

Security for complex Systems The perception of system security is evolving, in particular when complex systems such as execution environments are considered: security is no longer an absolute value; security is required for all software, not only for critical systems; security is a property of the software itself.

Security is no longer an absolute value but should be quantified and tailored according to the target system. Quantification makes sense out of the concept of 'relative security', *i.e.* security features which are suitable to the target system and not only 'best effort' features. Not every one needs full fledge security, if it can be achieved at all. Nor does every one has sufficient monetary resources or financial interest to build systems with a high level of security.

Security is required for all software and not only for highly critical systems or applications protecting high value assets. This means that costly certifications schemes such as the Common Criteria should be complemented by lightweight processes for building secure software in spite of the pressure of time-to-market and limited time and financial resources. In particular, developing secure software should be a task that standard developers should be able to perform and not only experts.

Security is a property of the software itself and not a feature. This means that software security should be considered as a part of software quality, even though techniques and risks can not be derived directly from the software quality field. In particular, security should be tightly integrated with the software development life-cycle and not restricted to specific development phases.

Security Analysis of SOP platforms In addition to high-level requirements, the methodology for security analysis should support the needs which are identified for Java SOP platforms. These requirements are elicited in Section 5.2.3, page 63 and in Section 5.3.3 page 68.

The availability of solutions to these requirements will be of use for other execution environments beyond SOP platforms.

6.1.3 Objectives

The methodology for enforcing software security analysis and development pertains to the category of *Software Security Assurance* (see Section 3.1.3). It should provide a process and tools for performing the identification of the vulnerabilities which plague the software system and for enforcing secure design as well as secure coding.

Three complementary approaches to security should be supported by our methodology in order to comply with the identified requirements: knowledge-based security, system-based security, and pro-active security.

Knowledge-based Security consists in gathering a comprehensive knowledge of the vulnerabilities of the target system before performing actual benchmarking* and development of security solutions. Knowledge here is used with the strong meaning of 'information [which can be] put to work using processes and procedures' [McG06]. A database of vulnerabilities or a catalog should be built and populated with reference and detailed description of each vulnerability. This phase is especially useful for the analysis of systems with a limited amount of available vulnerability information, in particular for commercial systems where security risks are only restrictively published. Actually, such data is available for widely deployed tools such as operating systems and common open source applications (see Section 3.2.1), but not for more specific systems. This phase of identification of the vulnerabilities is necessary to lay a sound basis for latter security analysis. It can be completed with the development of exploit code to be used for benchmarking various implementations of the target system or specific protection mechanisms. Knowledge-based security is required to make the quantification of the security status of a system possible. It is also well suited to deal with important attack surfaces*, where numerous vulnerabilities are to be managed and protected.

System-based Security consists in integrating security checks and enforcement in the execution environment. It provides an automatization of secure design by urging the designer

to exploit the security best practices and mechanisms which are provided by the system itself. The ultimate goal of system-based security is to make software developed in an insecure manner either protected by the environment or not available if it does no comply with the security policy of the environment. This obligation actually eases the task of the developers: they mainly need to build applications which cope with the architecture and with the execution environment properties rather than to shift the focus from actual development to a costly security documentation process. System-based security is therefore a way of enforcing security for all software and not only for application deserving a costly certification. It is also a good candidate for dealing with important attack surfaces and important execution right of the code since it relieves the development team from manually tracking individual vulnerabilities.

Security is then built in the execution environment itself and not in the application. Higher-level documentation processes such as Secure Development Life-Cycle (SDLC) [LH05, Noo06] can then be exploited with benefit in complement to system-based security, especially if it complies with system specific knowledge and application specific checks that can not be automatized.

Pro-active Security consists in providing systems which are *built-in secure* rather than security patches which imply an important administration overhead. It is based on *secure coding* techniques to free the application code from unproper constructs which are known to be exploitable by malicious entities. The objective is to avoid as much vulnerabilities as possible long before they would show up during execution. It is based on two complementary approaches: recommendations [Sun07, Lai08] which aim at training developers and experts, and tools such as code static analysis [RAF04] which aim at relieving the developers from painful code review which can be easily automatized. However, currently available techniques strive toward clean code - which is a great benefit for the quality of applications - but fall short in providing actual security guarantees, except in specific environments such as web applications [LL05]. Their use to prevent specific vulnerabilities is likely to bring in an important security benefit, provided they are fine-tuned according to the weaknesses which are identified during the phase of *identification of vulnerabilities*. The pro-active approach urges security to be a property of the code itself. It should be enforced as early as possible. An optimal configuration is the enforcement of protection during the development and the control on the execution environment before the execution of the suspicious code.

The principle of *knowledge-based security* is used for a long time in network security*. However, it is so far used to a limited extend in the context of applications, when one excepts widespread stable applications such as firewalls and web servers. The principle of *system-based security* is long used in the context of operating systems [SS73]. However, its application in the context of execution environments such as virtual machines or component platforms* is so far limited. The principle of *pro-active security* has experienced a more recent fame, in particular because of the increased number of target specific attacks* and the relative maturity of the Network and Operating System Security* domains.

6.2 $\mathcal{SPIP}$ - Spiral Process for Intrusion Prevention

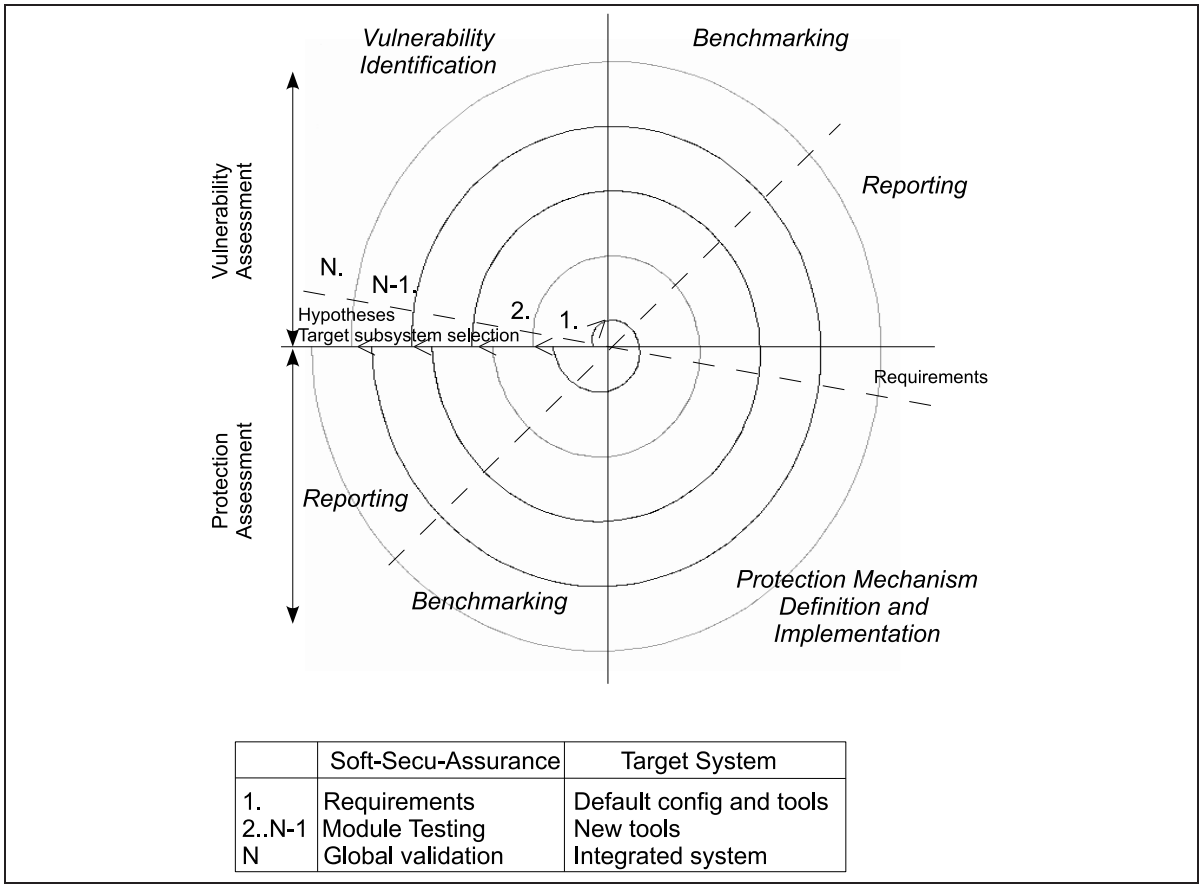
The ***S**piral **P**rocess for **I**ntrusion **P**revention*, $\mathcal{SPIP}$, is a Software Security Assurance method for complex systems. As its name implies, $\mathcal{SPIP}$ is a spiral process to be imple-

mented in a recursive manner. In the context of Java SOP platforms, its objective is to reduce at each iteration the set of unprotected APIs. The analysis course is first defined. Next, the process is detailed for one iteration. Lastly, the approach for security benchmarking is presented.

6.2.1 Analysis Course

The goal of $\mathcal{SPIP}$ is to identify and develop software protection mechanisms which can be automatized and integrated in the execution environment. It finds its inspiration in the risk oriented spiral development process defined by Boehm [Boe86] and the AEGIS spiral model for secure software development [FSH03].

Figure 6.1 shows the overview of $\mathcal{SPIP}$.



	Soft-Secu-Assurance	Target System
1.	Requirements	Default config and tools
2..N-1	Module Testing	New tools
N	Global validation	Integrated system

Figure 6.1: Overview of $\mathcal{SPIP}$ - Spiral Process for Intrusion Prevention

Principles $\mathcal{SPIP}$ targets any complex system which require the use of several complementary protection mechanisms. It is made of a set of subsequent iterations that aims at incrementally building a more secure system. Each iteration consists in the analysis of a given subset of the target subsystem to identify and solve its vulnerabilities. No continuity exists between two contiguous iterations. $\mathcal{SPIP}$ is concluded by integrating the different protection mechanisms. It implies the drawback of non negligible analysis, review and report overhead.

$\mathcal{SPIP}$ aims at relieving the analyst from tasks which can be automatized and integrated into the execution environment.

$\mathcal{SPIP}$ Iterations $\mathcal{SPIP}$ is a full-fledged process for the security analysis of complex systems. Its goals are of several types: defining system sub-sets which are ever more secure; providing data for evaluating security/cost trade-offs; supporting vulnerability identification and definition of related protection mechanisms. Each iteration provides a more hardened system, with more development and configuration constraints. $\mathcal{SPIP}$ highlights the fact that no absolute security level exists and that an adequate combination of protection mechanisms should be found for each application type.

The matching between these iterations and the standard life-cycle process for software security assurance (Figure 3.2 page 25) is the following:

- *Iteration 1*: the default system to be assessed and secured is analyzed: its vulnerabilities are characterized and reported; default configurations and protection mechanisms are assessed. This step can be viewed as a *requirement* analysis for the secure system to be defined; vulnerabilities can be considered a negative requirement. They should be resolved in the final system. Iteration 1 addresses the knowledge-based security principle.
- *Iterations 2 to N-1*: each protection mechanism is evaluated. They typically target a specific subset of the system (target API or module, or security patterns* which applies to several APIs or modules). These protection mechanisms can be existing ones which are adapted to the target system or new mechanisms which are specifically defined. Each protection mechanism is characterized by: 1) the protection it provides, and 2) the constraints it implies on the system such as development assumptions and restricted access to specific entities of the system. They should address both the system-based and pro-active security principles. Evaluation is performed through benchmarking and the results are reported. This step can be viewed as the *module design* of the secure system (see Figure 3.2 page 25): the various mechanisms build individual blocks of the final secure system. *Module analysis and testing* is performed to evaluate each individual mechanism.
- *Iteration N*: the individual protection mechanisms are integrated and the overall protection which is provided is assessed to identify its benefit as well as its potential limitations. For instance, some vulnerabilities may be to complex to detect through automated tools and still need intervention of human auditors. This step matches the *global validation* of the secure system.

6.2.2 An Iteration

Each iteration of $\mathcal{SPIP}$ is built of two main phases: vulnerability assessment* and protection assessment*. All steps need not be performed systematically. For instance, when several protection mechanisms are defined for the same set of vulnerabilities, vulnerability assessment is not repeated.

Vulnerability Assessment Its goal is to characterize the vulnerabilities which exist in the considered target system, to provide exploit code for further tests and to assess the various implementations of the target system. The steps for vulnerability assessment are the following:

1. *Selection of the target (sub)system, identification of the hypotheses and of relevant threat types.* The target system is defined by its specifications. In Java SOP platforms, its selection typically consists in identifying a given API such as the Java standard API or the OSGi API. The correct elicitation of security hypotheses is often the key factor for a successful security analysis: many security flaws* originate in broken security assumptions such as environment properties or trust level of the users [HL02]. The scope of the analysis can be restricted to focus on specific features of the system and to neglect some others. For instance, in our case, service* binding mechanisms are not considered, because they often build an optional part of Java SOP platforms. Relevant threat types can be selected, so as to focus on a particular type of attack such as outsider or insider attacks. For instance, in our case, we focus on attacks which are made possible by the installation of third party components* inside a Java platform.
2. *Identification of vulnerabilities* is performed by gathering various sources of information, as well as through independent review. The sources of information are one's own experience, the bibliography, as well as vulnerability disclosure databases. The review of the target system should be based on two complementary references: the system specifications to identify design weaknesses and the system implementations to identify implementation weaknesses.
3. *Benchmarking* of the system can be done if relevant in the context of vulnerability assessment, for instance to compare different implementations of the target system.
4. *Reporting* has two goals. First, it structures and synthesizes knowledge to support the training of developers, for instance through dedicated taxonomies. Secondly, it provides a reference of known vulnerabilities through extensive catalogs or databases. These sources can be used by auditors to certify the quality of code or by automated tools to provide technical security solutions. Reporting is typically done with the help of security taxonomies and vulnerability patterns*. These tools are presented in Sections 7.1.1 and 7.1.2.

The output data which is produced during vulnerability assessment are system specific taxonomies and a vulnerability database or catalog. This knowledge can be used as input for the protection assessment phase.

Protection Assessment Its goal is to identify a suitable protection mechanism, or to define it if it does not exist yet. Each protection mechanism typically targets a subset of the known vulnerabilities. Its efficiency and limitations are evaluated. It is quite unlikely that a *Silver Bullet* [Bro87] protection can be found, *i.e.* a protection mechanism which solves all vulnerabilities. The steps for protection assessment are the following:

1. *Identification of the security requirements* for the considered protection mechanism. These requirements are design constraints, development constraints, non-functional constraints and the vulnerabilities to be prevented. Design constraints are implied by the platform architecture. For instance, in the Java/OSGi platform, components are isolated from each others and can only communicate through shared packages and SOP services. Development constraints are implied by the type of application and the development team organization. For instance a safety critical component is not coded in the same manner as entertainment software. Non-functional constraints on the application can be performance and usability. For instance, a strong deterrent for implementing

standard Java Permissions is the fact that they imply an overhead of approximatively 30 percent. Vulnerabilities fail into two categories: the flaws which need to be patched, and the dangerous actions which require access control.

2. *Definition and implementation of the protection mechanisms* is to be performed according to the security requirements in the case existing solutions are unsufficient. It can imply one or several security patterns* (see Section 8.2 for an example).
3. *Benchmarking* consists in evaluating the efficiency of the considered protection mechanism against the identified vulnerabilities. Comparison of various implementations of the target system can also be performed if they behave differently in the presence of the protection mechanism.
4. *Reporting* consists in documenting the security patterns which are introduced or used in the protection mechanism, the constraints on the applications to be run in the secured execution environment, as well as further challenges. These constraints on applications must be available for developers so as to ensure that the programs can run seamlessly in the modified environment. For instance, the Java `SecurityManager` requires that suitable permissions are set. Static analysis approaches require that the code is able to pass pre-defined tests before being installed. Further challenges should also be identified. Since no protection mechanism is perfect, this step enables to write down the information the protection mechanism designer has about the limitations of his own tool. It is likely to save time in latter analyses, when the system is improved again.

The output data which is produced during protection assessment is the definition of new protection mechanisms, the security patterns which are applied, the result of quantitative benchmarking (for instance with the *Protection Rate* metric), the implied constraints such as development overhead, loss of functionality and performance overhead, and further challenges which are identified. The designer of an application can then rely on these data to decide whether the mechanism is worth using or too costly or restrictive.

In the case of assessment of the integration of several tools, the same process can be applied.

6.2.3 Security Benchmarking

Security Benchmarking is performed both during the vulnerability assessment and the protection assessment phases. Its goal is to quantify the security level of the individual protections mechanisms and the overall security level of the target system in its default configuration or augmented with a set of security protections. Making security measurable is a pre-requisite to make it manageable [Bla05].

We propose to perform the evaluation of the security level for a given system through an evaluation of the attack surface [HPW05] (see Section 3.2.2 page 31) of the system which matches the number of vulnerabilities which is identified in the system. So as to express the protection brought in by particular implementations of the target system or by individual protection mechanisms, we introduce the *Protection Rate* metric. It is defined in Section 7.1.3. Of course, other metrics can be used.

$\mathcal{SPIP}$ can be seen as an incremental analysis: each additional protection mechanism aims at providing a solution to vulnerabilities which are so far not protected. Each iteration therefore increases the security level of the target system. Figure 6.2 illustrates this stepwise securization effort.

The effects of protection mechanisms cumulate with each other so that the maximum

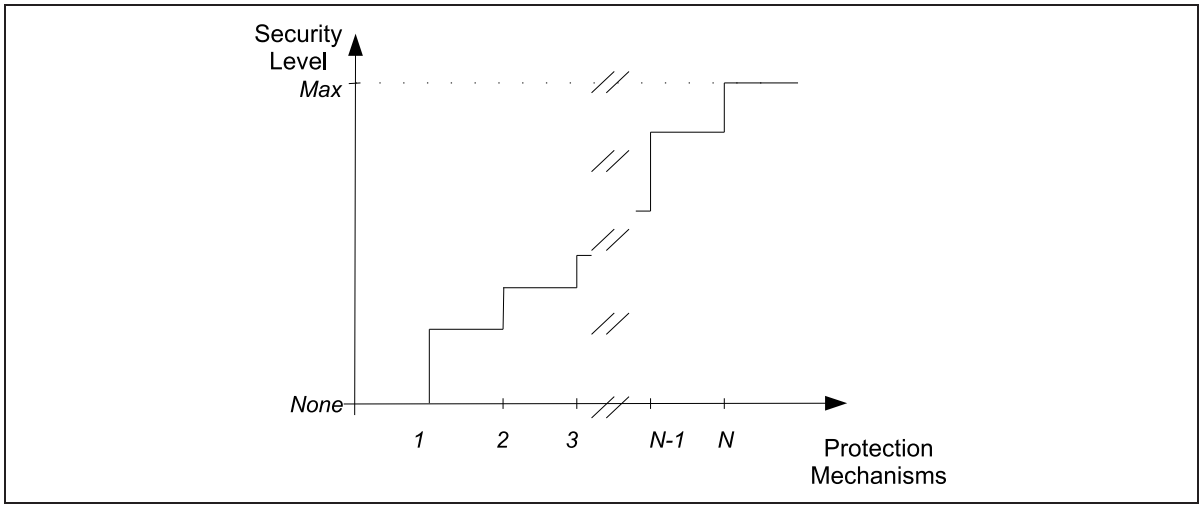


Figure 6.2: Stepwise securization of a system: the cumulated effect of complementary protection mechanisms

possible security level for this system is achieved when all mechanisms are used together. This maximum security level can be the maximum certification level or the protection of all known vulnerabilities. It does not mean that the system is completely secure, only that the maximal possible efforts have been performed to solve known vulnerabilities.

The current definition of the $\mathcal{SPIP}$ process supports following features:

- Analysis of a given target system, independently of its context.
- Identification of the vulnerabilities for a given system and comparison of various implementations of this system.
- Evaluation of the security level for each protection mechanism as well as for the integrated secure system.

Its benefits are the following:

- Developers can manage the vulnerability status of the execution environment on which their applications run. They know which vulnerabilities exist, which are protected and the cost of this protection. This is supported thanks to *Knowledge-based Security*.
- The definition of new protection mechanisms which expressively target identified vulnerabilities is integrated as part of the security analysis. System for which sufficient mechanisms are not available can therefore also be secured, even though the additional cost is important. The availability of a vulnerability database makes it possible to perform a pre-benchmark even before the mechanism is developed to evaluate its relevance. Of course, such theoretical evaluation must be confirmed afterwards through tests with actual exploits.
- The development of exploit code for each vulnerability make tests of similar execution environments or of protection mechanisms a straightforward task.
- For each protection mechanisms, the cost in term of development overhead, loss of functionality and performance overhead is explicited. This can help designers decide whether the protection is required for a particular type of application or if the constraints overweight the benefits. This is supported thanks to *Pro-active Security*.

We advocate to develop new protection mechanisms as a mandatory element of the target system under analysis to enforce *System-based Security*.

However, $\mathcal{SPIP}$ still need to be completed and has several identified limitations:

- The evaluation of the actual exposure of the vulnerabilities which is dependent on the context of exploitation is not supported.
- The cost evaluation for each protection mechanism is not supported. It can only be done when the context of exploitation is known.
- Several tool sets such as metrics and patterns for analyzing various target system types with heterogeneous objectives in term of security level are not available. It is highly likely that for different systems, for instance health care systems and online games, security requirements will vary and that specific tools are required for each specific use case.

For each of these topics, methods and tools are to be identified or defined if they do not exist.

6.3 Using $\mathcal{SPIP}$ for SOP platforms

Once a generic methodology for security analysis of complex systems is defined, it is possible to apply it to the target of our analysis: a Java SOP platform, at the example of Java/OSGi. One specific attack vector* is considered: the dynamic installation of a SOP component from a not-fully-trusted provider on a platform. The security hypothesis is that the host on which the platform runs can not be corrupted and that interactions between the applications and the outside world are protected through a standard Application Security* approach, as presented in Section 3.1.2. This means that we focus on the features which we consider to be specific to the target environment. Other attack vectors are better studied and thus do not deserve our attention here.

The implementation of the $\mathcal{SPIP}$ process for this system is first detailed. Next, experiments which are conducted are presented.

6.3.1 Process Implementation

The implementation of $\mathcal{SPIP}$ for analyzing SOP platforms is performed with the Java/OSGi platform which can be considered as a prototypical example of such technology. It is performed in three main steps: the analysis of the default target platform with standard protection mechanisms, the definition and analysis of new mechanisms which intend to solve the identified vulnerabilities and the analysis of an integrated Java/OSGi system which combines our propositions with a secure Java virtual machine, the JnJVM. A final benchmark is performed to summarize the security properties of the target system and its candidate protection mechanisms.

The following of the document is organized according to the course of $\mathcal{SPIP}$.

The default Execution Environment The first $\mathcal{SPIP}$ iteration consists in defining and performing a security analysis of a Java/OSGi system with a default configuration. Its results are presented in Chapter 7 page 87. The vulnerability assessment phase consists in identifying and documenting flaws and functions which can be exploited by malicious bundles* which

would be installed in the platform. The protection assessment phase consists in evaluating existing implementations and protection mechanisms. A first measure of the security status of common implementations of the OSGi platform, such as Felix, Equinox, Knopflerfish and Concierge, is performed so as to set a reference for latter benchmarks. A second measure of the security status of the same platforms is performed to assess the benefit of the standard protection mechanism of Java platforms: Java Permissions.

The individual Protections Mechanisms The second, third and fourth $\mathcal{SPIP}$ iterations consists in defining and assessing new protection mechanisms. The three mechanisms we introduce are *Hardened OSGi*, in Section 8.1 page 116 which is a set of recommendations for building more robust implementations of the OSGi platform, ***C**omponent-**B**ased **A**ccess **C**ontrol*, CBAC, in Section 8.3 page 124 which is an alternative to Java Permissions based on static analysis, and ***W**eak **C**omponent **A**nalysis*, WCA, in Section 8.4 page 133 which parses the code of bundles to ensure that the classes which are shared with others are free from vulnerabilities. The vulnerability assessment phase is avoided since vulnerability information stems from the analysis of the default system which needs to be improved. Each of these iterations is performed in a similar manner. First, security requirements for the default Java/OSGi platform which are to be solved are identified. Each mechanism targets a specific set of vulnerabilities. *Hardened OSGi* intends to solve vulnerabilities which are brought in by the OSGi specification, *CBAC* intends to prevent the execution of dangerous calls by unauthorized code while providing a solution for the drawbacks of Java Permissions and *WCA* intends to prevent the sharing of vulnerable code between untrusted bundles inside the platform. Secondly, the protection mechanism is defined and implemented. Thirdly, measures of the security benefits which are yielded by them are performed. Benchmarking is performed against the specific set of vulnerabilities the mechanism targets and against all vulnerabilities which plague the target platform. Comparison with the default platform and protection mechanism enables to evaluate their actual gain.

The secure integrated Execution Environment The $\mathcal{SPIP}$ iteration number five aims at solving a requirement category which is not addressed in this thesis: the resource isolation between components. Such a mechanism is necessary to prevent in particular denial-of-service attacks and requires the modification of the virtual machine itself. It is supported in particular by the JnJVM [TGCF08] which takes advantage of the isolation of OSGi bundles in class loaders to control their resource consumption. It is presented is Section 9.1, page 143. Security assessment is performed to verify that the combination of software security and virtual machine protection mechanisms is necessary, but very efficient, to provide secure SOP platforms.

The $\mathcal{SPIP}$ iteration number six provides a global overview of the security level of a Java/OSGi platform which is augmented with the various presented protection mechanisms. It is presented in Section 9.3 page 150. Such a comparison of the considered protection mechanisms and summary of the protected and unprotected vulnerabilities should help designers select the suitable mechanisms when building a particular system.

In the conclusion, Section 10.2 page 162, further requirements are identified, either to refine and extend the security tools, or to identify the vulnerabilities which exploitation can not be prevented with them.

6.3.2 Experiments

The validation of the propositions of this work is performed through experimentations. It consists in the implementation of all considered software entities, along with their evaluation. Implementation of proposed protection mechanisms is done in the way defined by $\mathcal{SPIP}$. It will therefore not be further discussed here. Experimentations related to the identification of vulnerabilities and their possible exploitation require more attention.

Condition of the Experimentations Since current production Java/OSGi SOP platforms are used in closed worlds, very few informations exist related to actual malicious bundles which would live *in the wild*. In any case, it is highly unlikely that firms which would be victims of such attacks would make advertisement for their own failure. Consequently, gathering information relative to Java/OSGi vulnerabilities must be done *in the lab*.

Development We therefore developed one or several proof-of-concept attack bundle for each suspected vulnerability so as to highlight various implementations, possible hindrances or specific preconditions for the exploitation of each vulnerability. For each platform vulnerability, a malicious OSGi bundle is built to prove its exploitability. They are 32 occurrences. Relative information is presented in the *Malicious Bundle Catalog* and in Section 7.2.1.3 of this thesis. For each Bundle vulnerability, a pair of vulnerable/malicious bundle is built. They are 33 occurrences. Relative information is presented in the *Vulnerable Bundle Catalog* and in Section 7.2.2.3 of this thesis.

The benefit of the realization of proof-of-concept attack bundles is twofold. First, the exploitability of the candidate vulnerabilities is demonstrated. It actually proves to be a very easy task since bundles which are installed in a Java/OSGi platform have very few limitation if any in the access to the JVM features, to bundles classes or to registered services objects. Secondly, these samples are available to perform tests for automated detection tools, for instance through static analysis. They build therefore a complete test suite for the protection mechanisms which are defined for the Java/OSGi platform.

Executing untrusted Java/OSGi Bundles: A Vulnerability Assessment

7

Security assessment* is performed to evaluate the security* status of the Java/OSGi platform* for a specific attack vector*: dynamic installation of a SOP component* from a not-fully-trusted provider on a platform. The results of this security assessment process are of two complementary types. First, the tools that support the security analysis process are defined and validated by the confrontation with more than 60 vulnerability* occurrences. Secondly, this information is structured according to several taxonomies* and gathered in two vulnerability catalogs: the *Malicious Bundle* catalog, which contains vulnerabilities of the platform, and the *Vulnerable Bundle* catalog, which contains vulnerabilities of the components, or bundles* in the OSGi world.

7.1 Defining Tools for Security Analysis

Assessing the security status of a complex system such as a Java platform requires that suitable tools are available. However, as few security analyses have been performed on this type of system, specific tools are not yet proposed in the literature so far. For our analysis, we derive widespread tools such as security taxonomies, vulnerability patterns*, and software security metrics. They are presented in Section 3.2 and provide a sound basis to define customized versions for specific systems.

Security taxonomies should characterize each property of the vulnerabilities. We introduced the *descriptive Vulnerability Pattern for Component platforms* to support comprehensive information related to each vulnerability to enable efficient developer training and definition of suitable security mechanisms. We define the *Protection Rate* metric to quantify the efficiency of existing protection mechanisms and to make possible the comparison of the security status between different implementations of the Java/OSGi platform.

7.1.1 Taxonomies

The objective of defining taxonomies is to provide a framework on which developers and researchers can rely to develop security mechanisms for the Java SOP platform. They are a tool that ease the generalization, communication, and application of research findings [GV95]. Their use in the computer security field show that they are a pre-condition for building secure systems, as shown in Section 3.2.1. Taxonomies must verify several properties to be useful and valid, *i.e.* to provide a sound basis both for the definition of the considered problem, here the vulnerabilities in the Java/OSGi platform and for the development of tools that are based on this knowledge. These properties are first presented. Their validity in the context of the proposed taxonomies is discussed.

To be useful, taxonomies must be explanatory and predictive [Krs98]:

- *Explanatory* means that they should clarify the relationships between their elements.
- *Predictive* means that their structure should reflect the knowledge of a specific domain - in our case the security related properties of the Java/OSGi platform - and that loopholes in this knowledge are easy to identify. For instance, the taxonomy for the *Implementations of Vulnerabilities in the Java/OSGi platform*, in Table 7.2, reflects the structure of the platform under study. If for instance no vulnerability is identified in the Module Layer, it is more likely that they have been overlooked rather than the module layer would be free of any vulnerability.

To be valid taxonomies must satisfy several properties: objectivity, determinism, repeatability and specificity [Krs98]:

- *Objectivity* means that each feature must be identified from the object known and not from the subject knowing. The attribute being measured should be clearly observable.
- *Determinism* means that there must be a clear procedure that can be followed to extract the feature.
- *Repeatability* means that several people independently extracting the same feature for the object must agree on the value observed.
- *Specificity* means that the value for the feature must be unique and unambiguous.

Taxonomies for characterizing vulnerabilities in the Java/OSGi platform are presented in Section 7.2.1.2. Taxonomies for characterizing vulnerabilities in Java/OSGi bundles are presented in Section 7.2.2.2. Both are built in a bottom-up approach: properties of the vulnerabilities are first observed then classified. When possible, the possible value of the properties are drawn from the relevant specifications.

The validation of our taxonomies is obtained by the confrontation between the two vulnerability catalogs. The fact that the taxonomies developed for platform vulnerabilities are exploitable with minor extensions for component vulnerabilities shows that they actually represent an objective and repeatable view of the system under study. We claim that the taxonomies are deterministic since they are based on the system specifications. Specificity is verified for each feature since the taxonomy properties can not take ambiguous values.

7.1.2 The descriptive Vulnerability Pattern for Component Platforms

The *descriptive Vulnerability Pattern for Component Platforms*, or shortly *descriptive Vulnerability Pattern*, is presented extensively in [PF07a]. It aims at gathering all information relative to a vulnerability that can be abused by malicious components that are install in a Java/OSGi platform to understand it and prevent its exploitation. It extends widespread vulnerability patterns to adapt them both for development support - whereas most patterns are designed for information only - and for developer training - whereas others are targeting system administrators. Patterns similar to the The *descriptive Vulnerability Pattern* are the VEDEF Pattern [ACDM01] and the OVAL Pattern [Mar05].

The structure of the *descriptive Vulnerability Pattern* is the following:

- Vulnerability Reference, to provide the unique identifier of the vulnerability and a proper classification* according to the taxonomies that are introduced in Sections 7.2.1.2 and 7.2.2.2.
- Vulnerability Description, to enable a more verbose presentation of the behavior of the vulnerability, as in the Morbray pattern [MM97].
- Protections, being actual or potential ones that may be efficient.
- Vulnerability Implementation, the development status of the proof-of-concept malicious bundles.

The 'Vulnerability Reference' Section contains the name of the vulnerability, its origin and the value for each feature that is represented through a dedicated taxonomy. Main sources for Java related vulnerabilities are the FindBugs database[1], the Sun Guidelines for secure coding [Sun07], as well as the *Malicious-Bundle* Amazones project [2] which is the development project we set up that contains proof-of-concept bundles for all vulnerabilities addressed in this study. The only source for OSGi related vulnerabilities is the *Malicious-Bundle* project, since we do not know of other similar research or development projects that concern this specific platform. The taxonomies of interest are the following: technical implementation of the vulnerability, location of the malicious payload, targets and consequences of attacks* based on these vulnerabilities, introduction time, and exploitation time.

[1] http://findbugs.sourceforge.net/bugDescriptions.html

[2] The code is available with Subversion on the INRIA Forge: svn checkout svn://scm.gforge.inria.fr/svn/maliciousosgi

The 'Vulnerability Description' Section contains more detailed explanation related to the type of vulnerability and the way it is exploited. A general description, the precise preconditions, the attack process, and the description of the consequence of attacks based on the vulnerability are given. Reference to related patterns - the famous *See also* entry - concludes the section.

The 'Vulnerability Implementation' Section contains information related to the proof-of-concept implementation of the vulnerability: the reference of the code (name of the bundle archive), the OSGi Profile on which the tests have been performed (in the vulnerability catalogs: J2SE-1.5 and J2SE-1.6), the date of implementation for reference, the test coverage (the percentage of the known variants of the vulnerability that have actually been implemented), and the list of the OSGi platform implementations that are vulnerable.

The 'Protections' Section identifies existing and potential security mechanisms, their life-cycle enforcement point, as well as mechanisms to prevent attacks, and possible reactions to an attack based on the considered vulnerability.

The descriptive Vulnerability Pattern for an example vulnerability, the *Management Utility Freezing - Infinite Loop* vulnerability, is given in Table 7.1. This vulnerability enables an attacker to freeze the management utility of the platform by providing a component which starter method (*e.g.* the `Activator.start()` method in OSGi) does not return.

For further examples, the reader is invited to refer to the INRIA Research Reports that contain the *Malicious Bundle* [PF07a] and the *Vulnerable Bundle* [PF08c] catalogs.

7.1.3 The Protection Rate Metric

Protection assessment requires that tools for quantifying the security level* of a system are available. The requirements are twofold: the efficiency of each security mechanism as well as the robustness of whole platforms must be quantified. To the best of our knowledge, no such metric is available so far. We therefore define the *Protection Rate* metric [PF].

The *Protection Rate* (PR) metric has a goal similar to the coverage metric presented in Section 3.2.2 page 31, which expresses the percentage of faults that are contained by fault-tolerance mechanisms. It targets security mechanisms. It represents the percentage of the known vulnerabilities that are protected by such a mechanism.

The *Protection Rate* is defined as the quotient of the Attack Surface* [HPW05] (see Section 3.2.2 page 31) that is protected through the considered security mechanism and the Attack Surface of the reference system, without any protection. It is also expressed as the complement of the quotient of the actual Attack Surface for the system to be evaluated and the Attack Surface of the reference system. In our example, the reference system is an idealized OSGi platform that contains all vulnerabilities that are identified both in the specifications and in common implementations. The system to be evaluated is a concrete implementation of the OSGi platform, possibly with some security mechanisms enabled. The Protection Rate can be expressed as:

$$PR = \Big(1 - \frac{Attack\ Surface\ of\ the\ evaluated\ System}{Attack\ Surface\ of\ the\ Reference\ System}\Big) * 100 \tag{7.1}$$

Vulnerability Reference

- **Vulnerability Name:** Management Utility Freezing - Infinite Loop
- **Extends:** Infinite Loop in Method Call
- **Identifier:** Mb.osgi.4
- **Origin:** Ares research project 'malicious-bundle'
- **Location of Exploit Code:** Bundle Activator
- **Source:** OSGi Platform - Life-Cycle Layer (No safe Bundle Start)
- **Target:** OSGi Element - Platform Management Utility
- **Consequence Type:** Performance Breakdown; Unavailability
- **Introduction Time:** Development
- **Exploit Time:** Bundle Start

Vulnerability Description

- **Description:** An infinite loop is executed in the Bundle Activator.
- **Preconditions:** -
- **Attack Process:** An infinite loop is executed in the Bundle Activator.
- **Consequence Description:** Blocks the OSGi Management entity (the Felix, Equinox or Knopflerfish (KF) shell; when launched in the KF graphical interface, the shell remains available but the GUI is frozen). Because of the infinite loop, most CPU resource is consumed.
- **See Also:** CPU Load Injection, Infinite Loop in Method Call, Stand Alone Infinite Loop, Hanging Thread.

Protection

- **Existing Mechanisms:** -
- **Enforcement Point:** -
- **Potential Mechanisms:** Static Code Analysis; Resource Control and Isolation - CPU; OSGi Platform Modification - Bundle Startup Process (launch the bundle activator in a separate thread to prevent startup hanging).
- **Attack Prevention:** -
- **Reaction:** -

Vulnerability Implementation

- **Code Reference:** fr.inria.ares.infiniteloopinmethodcall-0.1.jar
- **OSGi Profile:** J2SE-1.5
- **Date:** 2006-08-24
- **Test Coverage:** 10%
- **Known Vulnerable Platforms:** Felix; Equinox; Knopflerfish; Concierge
- **Known Robust Platforms:** SFelix

Table 7.1: Example of the descriptive vulnerability pattern: Management utility freezing - infinite loop

The Protection Rate metric enables to compare several similar platforms, for instance several implementations of the same specifications, by expressing the rate of protection that each of the platforms provides when compared to the set of vulnerabilities that are identified. Moreover, individual security mechanisms can be evaluated by calculating the Protection Rate for an OSGi platform that is run with the considered protection. The *Protection Rate* metric still has some limitations. It does only represent the rate of known vulnerabilities that are protected in a given flavour of the considered system: unknown vulnerabilities can not be taken into account. Moreover, it does not reflect the relative importance of the vulnerabilities, *e.g.* according to the impact of the attacks that exploit them.

7.2 Vulnerability Assessment of Java/OSGi Systems and Applications

7.2.1 Vulnerabilities in the Java/OSGi Platform

7.2.1.1 Examples of Attacks against the Java/OSGi Platform

Example attacks against the Java/OSGi platform can in particular exploit the standard JVM API, the OSGi Life-Cycle layer, the OSGi Service layer, or application code [PF].

Standard JVM API One attack that exploits the standard JVM API uses the platform life-cycle management functions to stop the whole runtime by performing a `System.exit(0)` call. Whereas such a call is innocuous in a stand-alone application which just kills itself, it leads to a radical denial-of-service attack against all installed applications in a component platform. This attack is avoidable by using Java permissions. This highlights the fact that, in spite of their limitations (see section 7.3.3), Java permissions are necessary to protect Java component platforms - unless an alternative is provided.

Another attack exploits the standard JVM API, namely the `Thread` management API. A vulnerability of the VM implementations can be exploited by recursively creating `Thread`s. It is due to the fact that `Threads` are processes that go on executing (and recursively creating themselves here) in spite of errors in other locations of the virtual machine. Up the the Sun 1.5 JVM included, `StackOverflowError`s that are generated by numerous thread creations are not caught properly, and lead to a `OutOfMemoryError` and a virtual machine crash. The JamVM 1.4.5 Virtual Machine is also vulnerable to this attack. The vulnerability is not reproducible with recursive objects creation since, in this case, the `StackOverflowError` is properly caught. It corrected in the Sun JVM 1.6.

Listing C.1 in Appendix C gives an implementation of the *Recursive Thread Creation* vulnerability.

OSGi Life-Cycle Layer Several attacks exploit features and weaknesses of the OSGi Life-Cycle layer. This layer brings flexilibity in the platform since it enables the installation and management of bundles at runtime. It actually also makes the vulnerabilities of the platform exploitable by enabling, under some conditions, the installation of third party code without further control.

One attack against an unsecured Java/OSGi platform consists in interfering with the life-cycle of third party bundles through the `BundleContext` API. For instance, it is possible to install a new bundle, to stop or to uninstall available ones. In this case, the attack is

due to the abusive execution of a legitimate system function by a malicious entity. Listing C.2 in Appendix C gives an implementation of the *Launch a Hidden Bundle* attack where a malicious bundle loads another one that is hidden inside its own archive and starts it.

This attack can be prevented through the use of Java permissions or an alternative mechanism.

Another example of attack that exploits the OSGi Life-Cycle layer consists in freezing the OSGi management tool through a non-returning bundle activator, which is the class used to start and stop the bundle. This example has been presented in Table 7.1 to illustrate the descriptive Vulnerability Pattern. Each management tool, such as the OSGi shell, executes all commands in the same `Thread`. In particular, the `BundleActivator.start()` method of a bundle is executed. If it freezes, *e.g.* when looking for an unavailable network resource or simply if an infinite loop occurs, all subsequent calls to the management tool are also blocked. In this case, the attack is due to a flaw in the platform architecture.

Listing 7.1 gives an implementation of the *Infinite Loop in Bundle Activator* attack.

Listing 7.1: Example of malicious code in an OSGi bundle: infinite loop in bundle activator

```
public class InfiniteStartupLoopActivator
   implements BundleActivator
{
   public void start(BundleContext context){
      System.out.println("Bundle InfiniteStartupLoop started");
      while(true){}
   }

   public void stop(BundleContext context){
      System.out.println("Bundle InfiniteStartupLoop stopped");
   }
}
```

This attack can be prevented by starting each `Activator` object in a specific `Thread`. Of course, this requires that the JVM supports multi-threading, which is not a prerequisite of the OSGi specification.

OSGi Service Layer An example of an attack that exploits the OSGi Service layer is the *Numerous Service Registration* attack. In most cases, a bundle registers a couple of services. When the bundle is stopped, all services must be unregistered before the stopping process can be fulfilled. The unregistration mechanism has a cost similar to the registration. Numerous services can be provided, for instance through a `while` loop, when the same service is published with varying configurations. The duration of unregistration then grows significantly. It can lead to a denial-Of-service attack. DoS is effective through important resource consumption when the registration process takes place but also when the bundle is to be stopped or when the whole platform is shut down. In some OSGi implementations, such as `Concierge`, the *Numerous Service Registration* attack leads to the freezing of the whole platform, which can no longer be used and needs to be re-started.

This attack can be prevented by setting a limitation in the number of registered services. For instance, 50 services can be registered by any bundle. This is far more as usually required, does

not imply a significative performance overhead and guarantees that the maximum duration of the registration and unregistration processes remains almost unnoticeable.

Application Code The last example of attacks by malicious bundles exploits features of the application code, without requiring access to the platform itself, by exploiting language features. Such attacks are due to lack of algorithm safety in Java, for instance through the exhaustion of CPU resources or injection of memory load. This vulnerability is referenced by [BG05].

Listing C.3 in Appendix C gives an implementation of the *Memory Load Injection* attack, which consumes an important part of the available memory. It can lead to an `OutOfMemoryError` if other processes also require memory.

No standard mechanism exists to prevent such attacks. Research efforts such as Proof-Carrying-Code [NL96, Nec97] define promising techniques but no Java compiler is made publicly available[3] [CLN00].

7.2.1.2 Taxonomies for characterizing Platform Vulnerabilities

The prevention of vulnerabilities implies to have precise knowledge about them. Such characterization can be done through the 5 Ws: what, who, where, why, and when, and structured through taxonomies. The *What* is related to the subject of the taxonomies, namely vulnerabilities of the Java/OSGi platform. This is the only invariant property. The *Who* identifies the entity* that performs the action, in our case the malicious payload in the Java/OSGi component. The *Where* identifies the location of the vulnerability that is exploited in the target system, *i.e.* the way it is implemented. This property can also be considered to be the *source* of the vulnerability. The *Why* identifies the target of the exploitation, *i.e.* the entity that is impacted and its objective, *i.e.* the consequence of the attack based on the vulnerability. The *When* is related to the time points where the vulnerability is introduced in the system and where it can be exploited. To provide a complete knowledge about one vulnerability, each field is to be informed.

The taxonomies that characterize platform vulnerabilities are now presented. The taxonomy for the *Location of the malicious payload in a Bundle* describes the place where attacks that are based on the considered vulnerabilities are launched. Its goal is to identify the element of the bundle that can be analyzed to identify such attacks. The values that build up this taxonomy are the following:

- *Bundle Archive*: the vulnerability is exploited through the format of the archive such as its size, its structure.
- *Manifest File*: the vulnerability is exploited through specific values of the meta-data that are contained in the `Manifest.mf` file.
- *Activator*: the vulnerability is exploited through the `Activator` class which is used to launch the bundle and to configure it.
- *Application Code*: the vulnerability is exploited by the code of the bundle.
    - *Native Code*: the vulnerability is exploited through a call to native code.
    - *Java Code*: the vulnerability is exploited through the Java code itself.
    - *Java API*: the vulnerability is exploited through calls to some Java API method.

[3] Mail exchange with George Necula, 2006/03/06 and 2006/03/27.

Entity	Layer	Property	Flaw/ Function	Occurrences
Operating System			Function	2
JVM	Runtime	Runtime stopping methods	Function	2
		Thread Management	Function	3
	API	ClassLoader	Function	3
		Reflection	Function	3
		File Handling	Function	1
		Native Code Execution	Function	2
	Language	No algorithm safety	Flaw	4
OSGi	Life-Cycle Layer	Proper removal	Flaw	1
		Bundle Management	Function	2
		Invalid Activator	Flaw	2
		Invalid Archive	Flaw	3
	Module Layer	Fragments	Function	3
		Invalid Metadata	Flaw	3
	Service Layer	No control on service registration	Flaw	2
		Invalid Workflow	Flaw	1

Table 7.2: Taxonomy: implementations of the vulnerabilities in the Java/OSGi platform [The vulnerabilities that are considered here are those that can be exploited by a malicious component installed in the platform. Vulnerabilities are classified according to the entity in which they occur, the layer in this entity, and the property in this layer. Each property can be either a flaw, or a function. A flaw is an error in the implementation that can be patched without impacting the proper system behavior. A function is a feature exploited by other parts of the application to perform specific task.]

    - *OSGi API*: the vulnerability is exploited through calls to some OSGi API method.
- *Bundle Fragment*: the vulnerability is exploited through the `Bundle Fragment` feature which allows a bundle to gain access to all classes it contains.

The taxonomy *Implementations of the Vulnerabilities in the Java/OSGi Platform* describes the way vulnerabilities are exploited in the vulnerable system. Its goal is to identify the element of the system that need to be modified or protected to prevent the exploitation of each vulnerability. Vulnerabilities can be of two type: flaw* or function*. A flaw is an error in the implementation that can be patched without impacting the proper system behavior. A function is a feature that is exploited by other parts of the application to perform specific tasks. The vulnerability occurs when these tasks are dangerous, for instance if they provide access to sensitive resources or enable to perform denial-of-service attacks. Access to functions must be granted to trustworthy code only.

Table 7.2 shows the taxonomy for the *Implementations of the Vulnerabilities in the Java/OSGi Platform*.

The total number of identified vulnerabilities that plague the Java/OSGi platform is 32. The total number of occurrences that are given in this table is bigger than this, since some vulnerabilities imply the use and abuse of several functions together. For instance, `ClassLoaders` and `Reflection` are often used together, as well as `Java Native Interface` (JNI) and operating system functions.

The three main entities that contain vulnerabilities are the operating system, the Java Virtual Machine, and the OSGi framework. The concerned layers in the JVM are the runtime engine which executes classes, the API which provides standard libraries to applications, and the language itself. The concerned layer in the OSGi framework are the Life-Cycle layer, the Module layer, and the Service* layer. Each of these layers contains specific flaws and functions that are detailed in the table.

The taxonomy for the *Targets of Attacks* describes the victims of the attacks that exploit the vulnerabilities of the Java/OSGi platform. Its goal is to identify the entities that are impacted by such attacks to rank the importance of the vulnerabilities. A vulnerability which exploitation threatens the whole platform is more serious that one that targets a specific service. This does not mean that, according to the applicative context, the latter can not lead to more damages, for instance through financial or reputation loss, but gives an approximation of the potential impact of the vulnerability. The potential targets of attacks in the Java/OSGi platform are the following:

- *Platform*: the whole platform is impacted: it is typically either unavailable or suffers from important performance loss. In any case, all components that are installed are also directly impacted.
- *Java Element*: a specific element of a Java program is impacted.
  - *Class*: a specific class in a component.
  - *Object*: a specific object, that is shared between several clients* components.
- *OSGi Element*: a specific element of the OSGi framework is impacted.
  - *Package*: a specific package is the target.
  - *Service*: a specific service is the target.
  - *Bundle*: a whole bundle and thus the resource it provides are the target; specific bundles such as Fragment bundles and Extension Bundles pertain to this category.
  - *Management Utility*: the management utility, which can be a shell, a graphical user interface, or an automated management tool, is the target. Such attacks usually impact all subsequent management operations.

The taxonomy for the *Consequences of attacks* that exploit vulnerabilities of the Java/OSGi platform describes the type of impact that the attacks have. It should be used together with the *Targets of Attacks* taxonomy. Its goal is to evaluate the impact of the attacks. Like the previous taxonomy, it enables to rank the importance of the vulnerabilities. The type of consequences of an attack can take following values:

- *Unavailability*: the impacted entity is no longer available. If the attack target is the platform, all dependent entities, *i.e.* the whole system, are unavailable.
- *Performance Breakdown*: the impacted entity experiences a loss of performance. If the attack target is the platform, all dependent entities run with less resources.

- *Undue Access*: the impacted entity is accessed in an undue manner. Since access to Java code in general does not enable to distinguish between read and write access, data can be read AND modified.

The two last taxonomies that characterize vulnerabilities of the Java/OSGi platform represent the temporal properties of the vulnerabilities: the introduction time, and the exploitation time of the vulnerability. This information is required to identify the moment when preventive mechanisms can be efficiently introduced in the target system.

The possible values for the *Introduction Time* of the vulnerabilities are the following:

- *Platform Design/Implementation*: the vulnerability is induced by the platform issuer, either through design or through implementation weaknesses.
- *Development*: the vulnerability is introduced during the development of the bundles.
- *Bundle Metadata Generation*: the vulnerability is introduced during the generation of the bundle meta-data.
- *Bundle Digital Signature**: the vulnerability is introduced during the digital signature of the bundle.
- *Bundle Installation*: the vulnerability is introduced at the moment of installation of the bundle.
- *Service Publication/Resolution*: the vulnerability is introduced during the process of service publication or resolution.

The possible values for the *Exploitation Time* of the vulnerabilities are the following:

- *Bundle Download*: the vulnerability is exploited consecutively to the download of the bundle.
- *Bundle Installation*: the vulnerability is exploited when the bundle is installed.
- *Bundle Start*: the vulnerability is exploited when the bundle is started.
- *Bundle Execution*: the vulnerability is exploited when the bundle is executed, for instance when the code it provides is called.

These six complementary taxonomies enable to characterize with precision the properties of a vulnerability in an SOP platform such as OSGi.

7.2.1.3 The *Malicious Bundle* Vulnerability Catalog

Protecting a system against malicious actions imply to know in a systematic manner what the threat is. Since very few work have been done so far to analyze the security properties of the Java/OSGi platform, it is necessary to make an inventory of the vulnerabilities of this platform. Such an inventory relies on two strong bases: the Java vulnerabilities that are quite well documented and the tools we defined to support vulnerability assessment, that provide a complete methodology for analysis and documentation. The *Malicious Bundle* vulnerability catalog is first presented. It contains 32 occurrences. Next, the catalog is analyzed to identify the predominant characteristics of the vulnerabilities. In particular, the origins of the vulnerabilities, and attack targets and consequences are discussed.

The motivation for building a vulnerability catalog is to make the result of this vulnerability analysis available for the community. It can be used for training, as reference for developers who wish to build more secure OSGi applications, or for platform developers that intend

to build more robust environments. The *Malicious Bundle* vulnerability catalog is built up as follows. First, all entries are identified, from the literature, our own experience, and discussion with practitioners. For each entry, a proof-of-concept bundle is developed, to show that the vulnerability can actually be exploited. The documentation is done through the *descriptive Vulnerability Pattern*, which contains information relative to the characterization of the vulnerabilities (through the taxonomies), its description, the actual and potential protections, as well as the implementation status of the proof-of-concept bundles. An overview of the catalog entries can be seen in Appendix 7.2.1.3. The whole catalog is published as an INRIA Research Report [PF07a].

Based on this vulnerability catalog, it is possible to better understand the security properties of most widespread open-source implementations of the OSGi platform. First, the origin of the vulnerabilities are analyzed to identify the type of intrusion techniques that are exploited and the relative responsibilities of the Java and OSGi environments. Next, targets and consequences of the attacks are detailed to enable the focus on the more serious attacks when protections are developed.

Analyses are performed through automated parsing of the XML representation of the Vulnerability Patterns (see Appendix F of our technical report [PF07a]). Life-Cycle related information is not considered further here, though they are available. Actually, they do not bring in information that is of outstanding interest for the assessment analysis.

Origin of the Vulnerabilities Protecting a system against the vulnerabilities it contains imply that their causes and their relative importance are known. Two main aspects are to be considered: the part of the system where the vulnerabilities are located, and the type of intrusion technique that is used to exploit them.

A raw partition of the system can be made between the Java Virtual Machine and the OSGi platform. This distinction enables to identify the relative liability of each part and to know which amount of the vulnerabilities is specifically due to the OSGi platform. Out of the thirty-two (32) vulnerabilities that are presented in the catalog in Section 7.2.1.3, eighteen (18) are are implied by the JVM, *i.e.* 56 %. The number of vulnerabilities that are implied by the OSGi platform is 14 out of 32, *i.e.* 44 %. This means that most of the vulnerabilities that are identified in this study are due to the JVM itself and therefore that other Java-based component platforms* such as J2EE and MIDP are very likely to suffer from the same threats.

The relative importance of the intrusion techniques in an OSGi platform are shown in Figure 7.1. The classification that is used as reference is the Neumann and Parker Classification [NP89] which defines nine categories of intrusion techniques that can be used against computing systems. Categories 3 to 8 are related to software attacks, other ones to non technological and hardware attacks. Category 3 concerns masquerading, which is not relevant here as far as no access control system is considered. Categories 4 to 8 are the following: 4) setting up subsequent misuse, 5) bypassing intended control, 6) active misuse of resources *i.e.* write or execution access to the system or resource (CPU, memory, disk space) consumption, 7) passive misuse of resources *i.e.* read-only access by a malicious bundle, and 8) misuse resulting from inaction.

The distribution of the vulnerabilities in the Neumann and Parker categories is the following. One vulnerability can pertain to several categories.

Most vulnerabilities are *active misuse of resources* - 25 of them - which means that a great deal of the vulnerabilities are directly implied by actions that are performed by the malicious

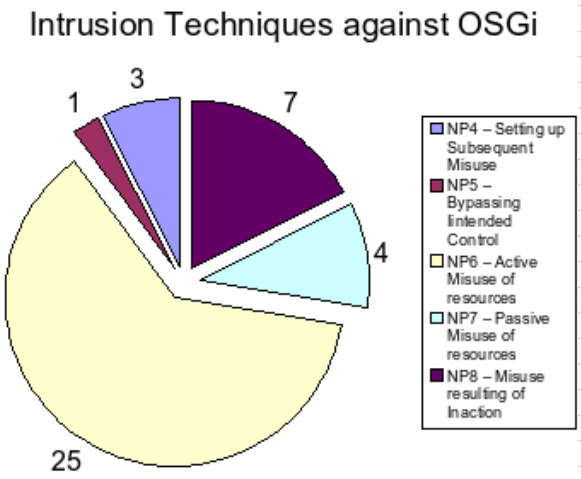


Figure 7.1: Relative importance of intrusion techniques in the Java/OSGi platform [The number of vulnerabilities in the *Malicious Bundle* catalog is given for each category.]

bundles. Preventing these actions through a convenient access control mechanism would thus bring an important improvement in the security status of the OSGi platform.

The number of *misuses resulting from inaction* is 7. These vulnerabilities are due to the fact that the OSGi platform has not yet been designed to withstand attacks against it and that therefore very few counter-measures are available.

The number of *passive misuse of resource* is 4, mainly undue read access. Since writing and reading access inside programs both occur through method calls, this kind of vulnerabilities can be prevented through the same mechanisms than active misuse, *i.e.* access control.

The number of vulnerabilities that consist in *setting up subsequent misuse* is 3. They are related to bundle management and fragments and are solved by OSGi permissions.

The last type of vulnerability is *bypassing intended control*. The unique occurrence is due to the fact that JVM digital signature validation algorithm does not exactly match the OSGi specification. A patch is presented in [PF07b] and is available on the SFelix Web Site.

Vulnerabilities that originate in the JVM cause the majority of identified threats in the OSGi platform. The OSGi specification itself is not free of such threats, as very few efforts have been so far done to make it secure. Though other types of weaknesses exist, most vulnerabilities are due to the absence of default access control mechanism, both at the Java and OSGi level.

Attacks: Targets and Consequences Each vulnerability can be exploited to set up attacks against a platform. The objective of the analysis of their characteristics is to identify the ones that represent major threats to the system so as to express priorities in the process of securing the OSGi platform. Two properties of the attacks are relevant to identify these priorities: the platform elements that are targeted and the type of consequence they have.

The two main targets are the OSGi platform itself and the OSGi elements. The platform is the execution and management environment for the applications it contains. Consequently, attacking the platform means that all of the applications that are running on it are impacted. The OSGi elements represent the code that is executed inside the platform. These elements

can be impacted with a varying granularity: whole bundles can be attacked, single services, or packages. Attacks that target a specific element have impact on the element itself and to the elements that interact with it, but unrelated elements are not disturbed. A specific type of sensitive element is the platform Management Utility which is a single point of failure since its unavailability prevents the subsequent management of any new or already installed bundle.

The most serious attacks are those that target the whole platform. The relative importance of attacks that target specific elements depends on the number of interactions the victim element has with others and of the application type: the unavailability of an entertainment service has not the same consequence that the unavailability of a health-care service.

The second characteristic of attacks is the consequence they have on the system under aggression. Figure 7.2 shows a dedicated taxonomy to reflect the specificity of the OSGi platform. A comparison with a mainstream classification, the Lindqvist Classification [LJ97], is also given for reference.

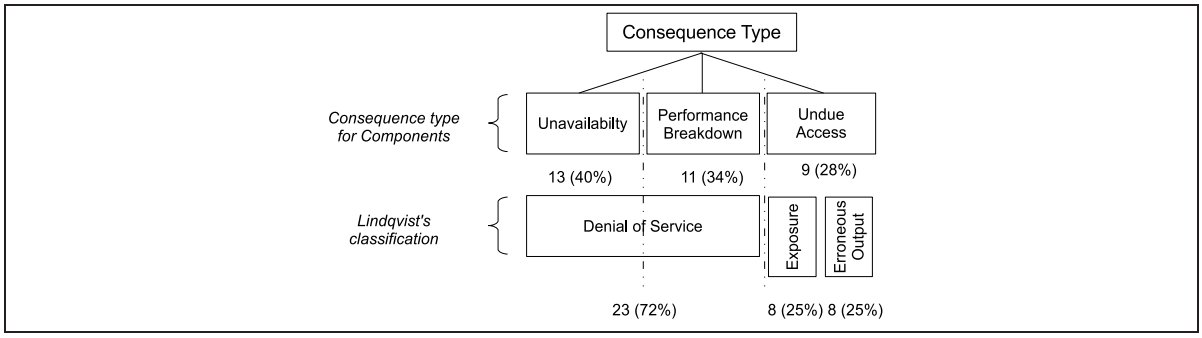


Figure 7.2: Taxonomy for consequence type in the Java/OSGi platform

The types of attack consequences for the OSGi platform are unavailability, performance breakdown and undue access. Here, the target elements are not considered. One vulnerability can have several consequences. Unavailability is the consequence of 13 of the vulnerabilities, *i.e.* represents 40% of them. Performance breakdown concerns 11 vulnerabilities, *i.e.* 34% of them. Undue access in read or write mode represents 9 vulnerabilities, *i.e.* 28%. Lindqvist classification establishes a distinction between the Exposure (*i.e.* read-only) and the Erroneous Output (*i.e.* read-write) undue access type. This reflects security properties of networks and operating systems and is less relevant in our case: access to a given method of a bundle enables both read and write access. It does not depend on access control properties. It also considers all denial-of-service (DoS) attacks as a single type, whereas in the OSGi context unavailability and performance breakdown are clearly caused by different kinds of attacks.

The type of attack consequence that causes the most serious threats is highly dependent on the kind of application. For instance, Health-Care Systems are very sensitive to unavailability, whereas undue access is of the utmost importance in banking systems. In both cases, the wider the attack is, the most damages are caused. However, it is important to note that a recent trend in computer security is that aggressions are targeted at very specific victims and that a precise aggression to an element that is a single point of failure can cause even more damage than when the whole platform is attacked in a brute force manner.

7.2.1.4 Requirements for Protection Mechanisms

The vulnerability assessment of the Java/OSGi platform enables to identify the security requirements for this environment. The requirements are related to the robustness of the platform or the constraints that the platform puts on the components it executes. They are the following:

- For the OSGi framework itself a hardened implementation is necessary to patch identified flaws and proper access control is required to prevent the exploitation of dangerous functions.
- For the Java execution environment, proper access control is required to prevent the exploitation of dangerous calls to the API or the runtime.
- A solution to contain resource consumption is necessary. This can be achieved either through resource isolation between the components, of through algorithmic safety.

The vulnerabilities that should be prevented with the highest priority are the vulnerabilities that lead to the full unavailability of the platform, through a call to `System.exit(0)` or through a JVM crash, but also the vulnerabilities that lead to *Undue Access*, in particular *Erroneous Output*, since it is frequent that data modifications are more damageable than unavailability. Vulnerabilities that lead to *Data Exposure* and *Performance Breakdown* can be considered to have a smaller priority even though they should not be neglected.

7.2.2 Vulnerabilities in Java/OSGi Bundles

The vulnerability assessment process for the Java/OSGi Bundles brings two major results: taxonomies to characterize the properties of the vulnerabilities of the target system and the *Vulnerable Bundle* catalog that gathers all these vulnerabilities [PF08a]. The availability of the catalog enables to perform a first analysis to identify the origin of the vulnerabilities, *i.e.* the weaknesses in the bundles, the types of attacks that are based on them, *i.e.* their dangerousness, and the efficiency of existing security mechanisms, which prove, as in the case of the platform vulnerabilities, to be fully unsufficient.

7.2.2.1 Examples of Attacks against Java/OSGi Bundles

Three categories of attacks against bundles can be identified, according to the level of access that is then required: *Stand-alone applications*, to which no access to the code is provided, *Class-sharing components* where access to the class Bytecode is provided, and *Service-oriented Programming* components* where access to objects published by other components is provided.

Stand-alone Applications On way to obtain access to application data without direct access to the code is the exploitation of serialized data, which is often stored either on a file system or sent without protection across a network. Since the *serialization* vulnerability is not specific to SOP platforms, we will not discuss it further. However, this highlights the fact that no component should serialize objects without ensuring that their data is properly protected, *e.g.* through encryption.

Class-sharing platforms Class-sharing occurs in almost all software environments where libraries are used to perform recurrent tasks. In the Java/OSGi platform, class-sharing is obtained by exporting and importing packages from a bundle.

Attacks that exploit class-sharing consists in impacting the behavior of applications through access or modification of the class data. For instance, static variables that are available to third party bundles can be modified. Another example of an attack that exploits class-sharing is the freezing of *Synchronized Code* blocks in static methods. Whenever a *Synchronized Code* block can be freezed, either by freezing a method that it depends on or by relying on material errors, all subsequent calls to this static method also freeze.

An example of the *Synchronized Code* attack is shown in Figure 7.3 as an UML sequence diagram. The scenario is the following: A component for digitally signing documents, `ScaSigner` is installed on a handheld device such as a mobile phone. A pre-installed software component provides an interface for accessing the SIM chip of the device, which executes among others cryptographic operations bound with the digital signature. A client component, `Ecom`, emits requests for digital signatures of specific documents.

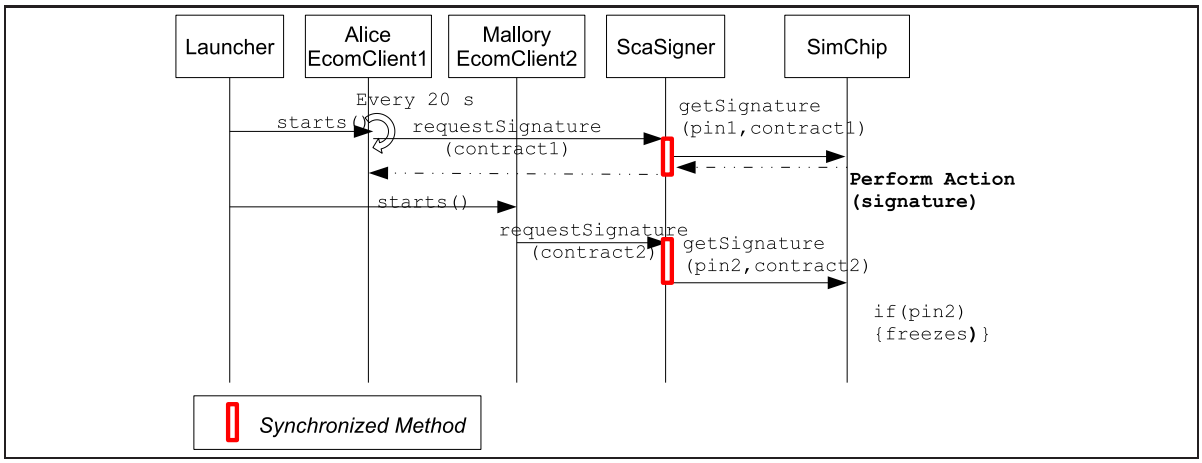


Figure 7.3: An example scenario for an attack against a synchronized method: sequence diagram

The synchronized method, `requestSignature()`, is provided by the `ScaSigner` class. This service relies on another one, `SimChip`, which provides a `getSignature()` method. A default valid scenario is executed by Alice, which is a benevolent component that signs a contract every 20 seconds. The attack is performed as follows. First, the `SimChip.getSignature()` method hangs when it is provided a specific `pin2` value for the `pin` parameter. This hanging condition can be replaced by a denial-of-service attack against a valid implementation of the `SimChip` service. The Mallory component is aware of the malfunction of the vulnerable `SimChip` service. It can therefore trigger its freezing (*i.e.* transmit the pin data with `pin2` value). It performs the malicious call to the `ScaSigner` service, which in turn calls the `SimChip` service, which hangs. As a consequence, Alice as well as any other client that call the `ScaSigner.requestSignature()` method will hang.

This attack can also be performed against *SOP* objects without requiring that the synchronized method be static. Two solutions exist to prevent the attack. Either all dependencies of the `synchronized` statement must be trusted and validated to prevent freezing, or the `synchronized` statement must be banned from class-sharing platforms to prevent such abuse.

SOP platforms Attacks against SOP components exploit the access they obtain to objects from third party bundles to modify them, retrieve information, modify the behavior of the applications, or perform denial-of-service. Since services are often provided as singleton objects, they are shared between client components.

Such an attack is the *Malicious Inversion of Control through overridden Parameters* attack. It can occur when a service takes non-final classes as parameter. An example of this attack is given in Figure 7.5 as UML sequence diagrams. The scenario is the same as in previous example (see Figure 7.3).

Figure 7.4 presents the normal execution process of contract signature.

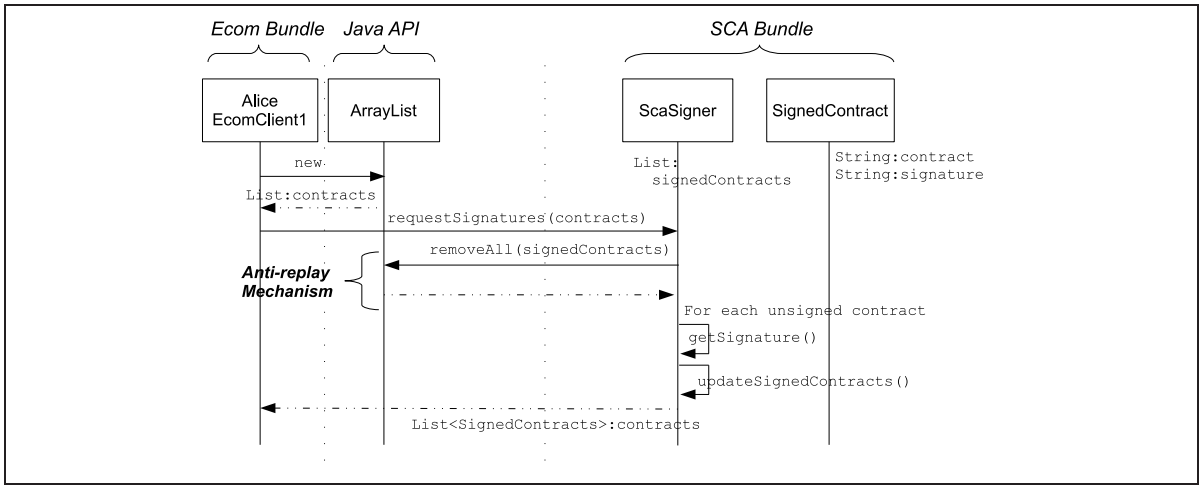


Figure 7.4: An example of a service with vulnerable method: sequence diagram

The scenario is the following. First, Alice `EcomClient1` creates the `contracts` object as an object of type `List<SignedContract>`, by using the `ArrayList` implementation of the required interface. `SignedContract` objects contain variables for both contract data and contract signature. They are initialized with contract data only. Next, Alice `EcomClient1` sends the `List` as a parameter to the `ScaSigner` service, through the `requestSignature(List)` method. So as to prevent replay attacks, contracts that are already signed are not signed again. This requires to store them in the `ScaSigner` service as the `signedContracts` class variable. Removing old contracts from the `contracts List` is done by calling the `List.removeAll(signedContracts)` method. In the benevolent case, the implementation is provided by the Java `ArrayList` API. The `ScaSigner` then signs each new contract and sends them back to the Alice `EcomClient1` object.

Figure 7.5 presents the process of attack against contract signature with *Malicious Inversion of Control through overridden Parameters*.

In this case, the malicious implementation of the `EcomClient`, `Mallory EcomClient2`, provides its own implementation of the List interface, `MaliciousArrayList`, which extends the original `ArrayList` class. The difference between `ArrayList` and `MaliciousArrayList` lies in the implementation of the `removeAll()` method. In the malicious case, all objects that are passed as parameter of the `removeAll()` method are stored, before calling the original `ArrayList.removeAll()` implementation. At the end of the signature process, `Mallory EcomClient2` thus owns the content of the signed contract objects, *i.e.* the valid contracts that have been performed by the `ScaSigner` beforehand - if any - but also all the matching

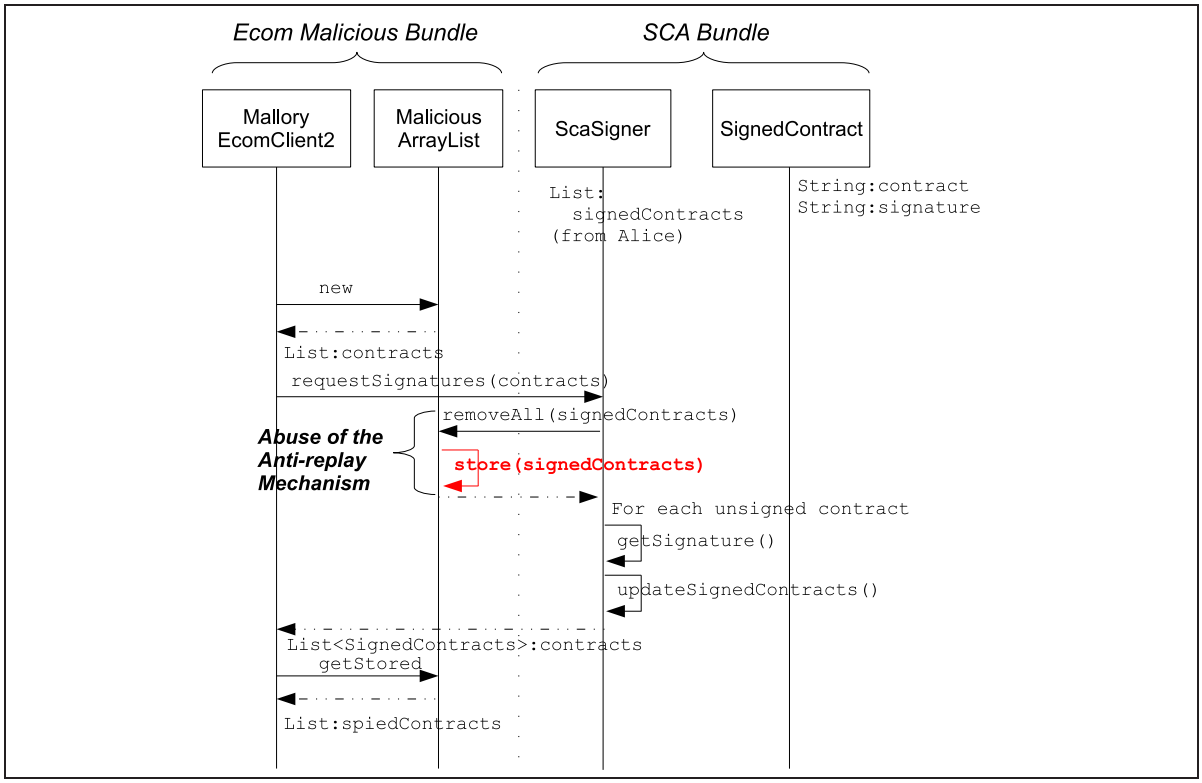


Figure 7.5: An example of a service with vulnerable method that is abused through malicious inversion of control: sequence diagram

contract signatures.

The *Malicious Inversion of Control through overridden Parameters* attack has two preconditions. First, services must provide methods with non-final parameters. Next, the victim service must call methods on these parameters and possibly passes to them some sensitive information. The given example is deliberately exaggerated: it is quite unlikely that any careful developer would handle back digital signatures to unknown code. It nonetheless highlights the potential risks that exist on ill-protected bundles.

Two solutions can be identified against this attack. The first solution would be to limit the use of *Singleton* objects to services that are fully stateless. In the example case, this does not solve the problem since information from previous transactions are required to prevent replay attacks. The second solution would be to harden the implementation of registered services by programming them with important constraints. One such constraint is the fact that all method parameters of registered services should be either basic types or final classes. The exact set of these constraints is still to be determined.

7.2.2.2 Taxonomies for characterizing Bundle Vulnerabilities

As they describe the vulnerabilities that can be exploited through installed components in a manner that is similar to the vulnerabilities of the platform, the taxonomies for the *Vulnerable Bundles* catalog are similar to the taxonomies that are defined for the catalog of *Malicious Bundles* exploiting platform vulnerabilities. In most case, no or very limited modification

is required. The shift of focus from the vulnerable platform towards vulnerable components imply to redefine the *Implementation* and the *Consequence of the Attack* taxonomies. This work is published in [PF08a].

In the *Target of attack* taxonomy, the 'Java Element' type is specific to the *Vulnerable Bundles* catalog. Actually, components can make some of their classes and their members, or some of their objects and their members, accessible from other components.

The taxonomy for *Implementations of the Bundles Vulnerabilities* describes the way vulnerabilities are opened in components. Its goal is to identify the element of the components that need to be modified or protected in order to prevent the exploitation of each vulnerability.

Table 7.3 gives the taxonomy for the Implementations of the Bundles Vulnerabilities in the Java/OSGi platform.

The vulnerabilities are classified according to the category of applications that can suffer from them. The three main such categories are stand-alone applications, class-sharing systems, and object-sharing systems. Stand-alone applications do not make their code available to third party code. Class-sharing systems enable components to share the definition of classes with others. This is the case for instance in the OSGi framework with the `import` and `export` meta-data for packages. Object-sharing systems, typically SOP platforms, enable a component to make an instance of an object available to third party components that act as clients. If such objects are shared between the clients, these latter can use them to communicate in an undue manner or to interfere with each other in ways that are not foreseen by the component developers. In specific cases, vulnerabilities can plague both class-sharing and object-sharing systems. This is the case of synchronization which can be abused to freeze all subsequent calls to the `synchronized` method. If this method is `static`, the vulnerability can be exploited in class-sharing systems. Otherwise, only object-sharing systems are vulnerable. Each of these categories contains specific implementations that are detailed in the table.

The taxonomy for *Consequences of the Attacks* describes the outcome of the exploitation of component vulnerabilities. Its goal is to highlight the actual risks that exist in ill-coded components and to rank them.

Table 7.4 gives the taxonomy for the Consequences of the Attacks that exploit Vulnerabilities in Java/OSGi Bundles Interactions.

The two consequence types are Undue Access and denial-of-service (DoS). Contrary to platform vulnerabilities, performance breakdown can not be achieved by attacking bundles directly. Undue access can be either access to data that should be kept internal to the component, or by-passing existing security mechanisms such as calls to the security manager. Each of these category contains specific sub-consequences that are detailed in the table.

7.2.2.3 The *Vulnerable Bundle* Vulnerability Catalog

These vulnerabilities exploit features of the Java language. They are therefore not bound with the OSGi platform but to the way Java elements are shared inside this platform. Most of other Java SOP platforms are therefore highly likely to contain the same vulnerabilities. An overview of the catalog entries in provided in Appendix 7.2.2.3. There are contains 33 occurrences.

Origin of the Vulnerabilities The repartition of the vulnerabilities in the defined categories is the following. Stand-alone application experience one occurrence of vulnerability out of 33, *i.e.* 3% of the total. Class-sharing systems are concerned with 14 occurrences, *i.e.* 42

Attack Vector	**Implementation**			**Occurrences**
Component Interactions	Stand Alone App.	Serialization		1
	Class Sharing	Exposed Internal Representation	Mutable element in static variable	2
			Reflection	3
			Fragments	2
			No suitable control	2
		Avoidable Calls to the Security Manager	At instantiation	4
			In method call	5
	Class Sharing or SOP	Synchronization		2
	SOP	Exposed Internal Representation	Returns reference to mutable element	2
			No suitable control	4
		Flaws in Parameter Validation	Unchecked parameter	3
			Checked parameter without copy	1
			Checked and copied parameter	4
			Non final parameter	2
		Invalid Workflow		1

Table 7.3: Taxonomy: implementations of the component vulnerabilities in the Java/OSGi platform
[The vulnerabilities that are considered here are those that can be exploited by a malicious component that is installed in the platform.]

Attack Consequence	Sub-Consequence	Interaction Category
Undue Access	Access to internal Data	Class Sharing and SOP - Exposed Internal Representation Class Sharing - Fragments
	By-pass Security Check	Class Sharing - Avoidable Calls to the Security Manager SOP - Flaws in Parameter Validation
DoS	Method unavailability	Class Sharing and SOP - Synchronization SOP - Invalid Workflow

Table 7.4: Taxonomy: consequences of the attacks that exploit vulnerabilities in Java/OSGi component interactions

%. Object-sharing systems are concerned with 16 occurrences *i.e.* 49 %. Two occurrences of vulnerabilities, that are bound with synchronization, concern both class-sharing and object-sharing systems. This tantamounts to 6 %. This shows that vulnerabilities in class-sharing and in object-sharing systems have a similar importance. Both need to be addressed.

The intrusion types are of three different categories in the Neumann and Parker's classification [NP89]. 9 occurrences concern the *NP5-by-passing intended control* category, *i.e.* 27 %. 23 occurrences concern the *NP6-active misuse of resource* category, that is to say writing access, *i.e.* 70 %. 1 occurrence concerns the *NP7-passive misuse of resource*, that is to say reading access, *i.e.* 3 %. This shows that the lack of protection for writing access is the cause of most vulnerabilities in Java components. Next to it, the proper enforcement of security checks is the second threat type. Read-only access is only a marginal vulnerability in the case of components.

Attacks: Targets and Consequences The target of attacks that exploit component vulnerabilities can be objects, classes, the platform, and configuration data. The most important target is built by the objects that are in memory. This is due to the fact that the assets that components can protect or make available are data they store. This is true for the vulnerabilities that are specific to object-sharing environments but also for several vulnerabilities that originate in class-sharing mechanisms, for instance through static variables. The number of occurrences for this category is 24, *i.e.* 73 %. The second target set is built by the classes themselves which represent 6 vulnerabilities, *i.e.* 18 %. Marginal targets are the platform, which can be forced to execute code outside of the component life-cycle through `Shutdown hooks`, with 2 vulnerabilities, *i.e.* 6 %, and configuration data, which can be leaked through excessively verbose exception, with 1 vulnerability, *i.e.* 3 %.

A great majority of these attacks can lead to erroneous output since access to the code of another component imply in almost cases that the values of its variables can be impacted. 30 occurrences of vulnerabilities, *i.e.* 91 %, concern this case. 2 attacks lead to denial-of-service, *i.e.* 6 %, and 1 to exposure of data,*i.e.* 3 %.

7.2.2.4 Requirements for Protection Mechanisms

The vulnerability assessment for Java/OSGi bundles enables to identify the security requirements that should be enforced on the applicative code. The requirements are related to the robustness of components against malevolent third party components. They are the following:

- SOP Workflows should be secured and mandatory.
- Component code should be properly isolated from each other, to avoid undue modification of their data. This implies a clean encapsulation, whose criteria are given by the *Exposed internal representation* and *Flaws in parameter validation* vulnerability categories.
- Effective access control should be enforced, *i.e.* not be by-passable.
- No data leak should occur in component, in particular through serialization and exceptions.
- `Synchronized` calls should be avoided unless it is possible to guarantee that their dependencies do not freeze.

The vulnerabilities that should be prevented with the highest priority are the one that enable malicious components to perform actions that they should not or to impact the behavior of other components without control. Security protections that are thus urgently required are effective access control and proper isolation between bundles. Data leak and localized freezing should be prevented, with less priority. The securization of SOP Workflows would only provide marginal improvement relative to the vulnerabilities that are identified here. However, Workflow specific security issues have not been considered. It is likely that secure Workflows could improve the global security status of SOP platforms but this question requires further research to identify requirements, constraints and possible solutions.

The agregation of the information from the *Malicious Bundles* and the *Vulnerable Bundles* taxonomies provides an overview of the vulnerability categories that can occur in systems that are based on SOP platforms. The vulnerabilities are of four origins: the underlying operating system, the Java Virtual Machine, the SOP platform itself and the components. Figure 7.6 presents the taxonomy of vulnerability categories in the Java/OSGi platform.

Since this taxonomy is an overview of data that has already been presented, it will not be explained in details again. Worth to mention is the fact that it integrates vulnerabilities that are mentioned by the literature such as JVM bugs in optimized modules, or the security issues that originate in service binding mechanisms and that are difficult to control due to their optional implementation is platforms such as OSGi.

The integration of data from the two vulnerability catalogs close the vulnerability assessment phase for the default target system of the $\mathcal{SPIP}$ process. A first protection benchmarking must be performed, so as to identify the security level that is provided by default implementations and security mechanisms against the vulnerabilities that are identified here.

7.3 Protection Assessment of Java/OSGi Systems and Applications

The second phase of the $\mathcal{SPIP}$ process is protection assessment. It consists in evaluating security mechanisms that intend to protect the system from the vulnerabilities that are identified in the first phase. Since the actual state of the available system is to be assessed first, no extra definition of new security mechanism is to be done during the first iteration of $\mathcal{SPIP}$.

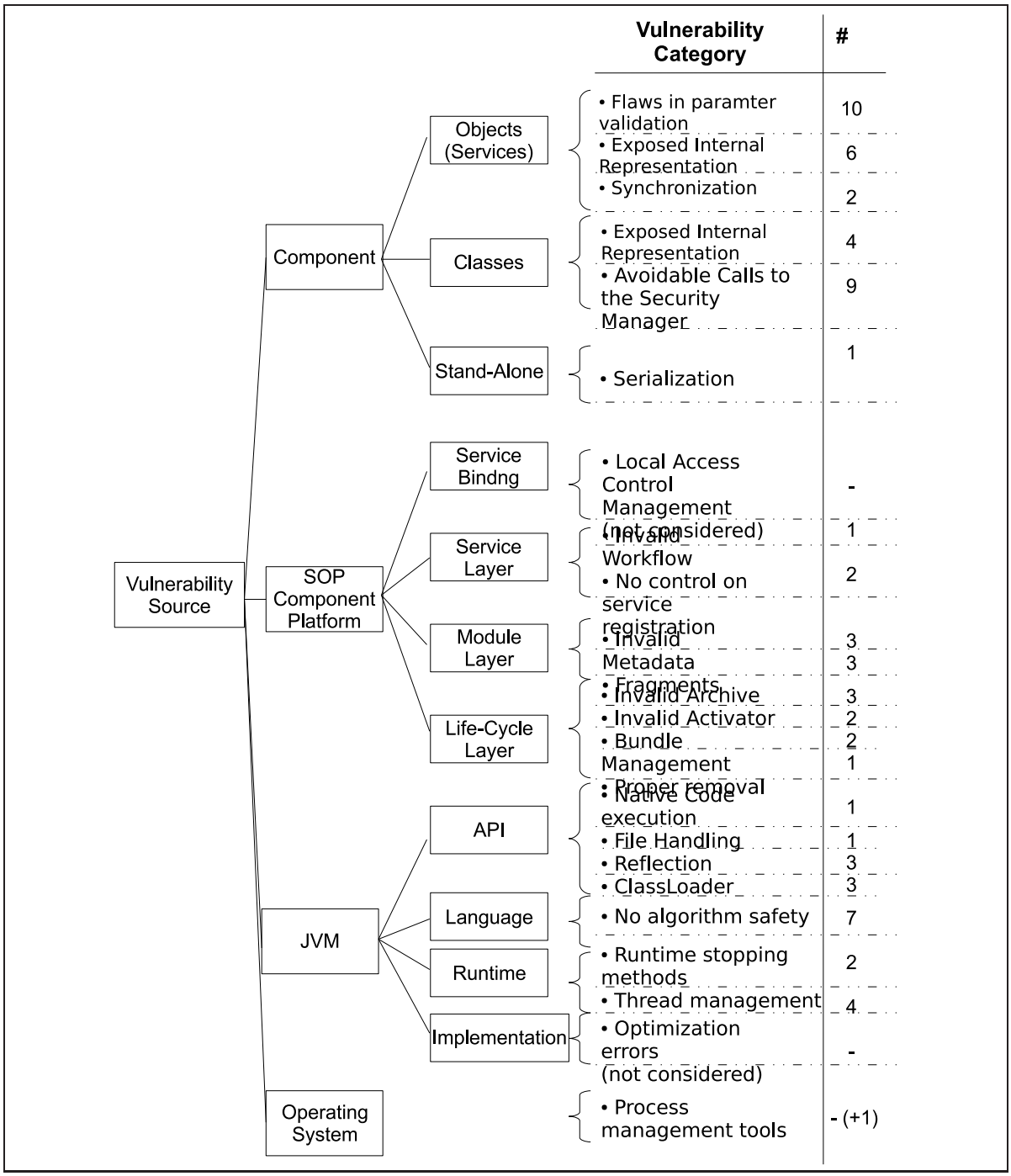


Figure 7.6: Taxonomy: vulnerability categories in the Java/OSGi platform

7.3.1 Protection Assessment of standard Implementations of the OSGi platform

The quantitative evaluation of the security level is performed with the help of the *Protection Rate* metric. The security level provided by default configurations of various implementations of the OSGi framework is measured. Next, the benefit that is brought in by Java permission is quantified.

The Protection Rates for the various implementations of the OSGi platform are given in Table 7.5.

Platform Type	**# of Protected Flaws**	**# of Known Flaws**	**Protection Rate**
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 %

Table 7.5: Protection rate for mainstream OSGi platforms

The number of known flaws varies to reflect the features actually supported by each implementation. The security manager, which is responsible of enforcing security checks that are defined by Java permissions, proves to bring a not negligible security improvement for the Java/OSGi platform.

However, even though Java permissions greatly enhance the protection rate of each implementation, it can not be considered as satisfactory: 56 % (for Felix), or even 45 % (for Equinox) of unprotected vulnerabilities are more than required to attack a system.

7.3.2 Protection Assessment of Java/OSGi Bundles

The vulnerabilities that occur in components also need to be protected: they represent more than the half of the documented vulnerabilities. Developers will be interested in knowing whether their application is safe and not only whether the underlying platform can be harmed.

In the case of components, Java permissions are far less efficient: only 3 out of 33, *i.e.* 9 % of the vulnerabilities can be protected in this manner. One of the important source of vulnerabilities for the *Vulnerable Bundles* catalog is the FindBugs project[4]. One would thus expect that the 9 vulnerabilities that are identified from there would be protected. According to the tests that are performed with the proof-of-concept bundles, this is not the case for any of the concerned occurrences. This is due to the fact that FindBugs aims at providing good practices in Java programming. It therefore advocates pattern that can be exploited without

[4] http://findbugs.sourceforge.net/

restricting the expressive power of the code. In the case of the vulnerability assessment for SOP Components, a very tight definition of vulnerability has been used, namely that fact that any access in reading, writing, or denial-of-service, is considered to be harmful. The use case for both approach is not identical.

The only tool that remains for ensuring the security of developed component is manual review. It has the advantage to tackle complex coding patterns that prove to be vulnerabilities in our target environment. However, it is very costly and does not provide a full guarantee even though all vulnerabilities can theoretically be identified.

7.3.3 Evaluation of Java Permissions

Performances Java permission are an efficient standard for enforcing access control in Java-based systems. However, they suffer from several serious drawbacks. First, they imply an important performance overhead, because the call stack must be rebuilt at each security check. Though important effort have been dedicated to minimize this overhead [GS98], it causes many production systems to withdraw the use of the security manager to preserve acceptable user experience and overall performance[5]. Example of this overhead is shown in Figure 7.7, where the duration for platform start is printed for several application profiles. Here, overhead goes from 0 second, where no check is performed, to 30 %, when a complex application is launched and when numerous checks have to be performed. This is the case for the `managedplatform` and `sfjarsigner` profiles which launch the necessary bundles to manage an OSGi platform through the JMX protocol and to use the SF-Jarsigner[6] respectively tool respectively. Secondly, security checks are performed in the code and can not be customized at runtime. Basically, each method that considers itself as dangerous performs its own check, through an explicit call to the `SecurityManager`. Consequently, it is not possible to declare a third party method as dangerous, for instance if a bundle is identified as not trustworthy. This approach is satisfactory for systems where all code archives are under control of the administrator but proves to be unsufficient if new code is loaded dynamically from the environment. Thirdly, runtime time check has another drawback in addition to performance overhead: it forces the application to abort, or asks for specific permissions to users that are security unaware. Both behaviors can lead to security weaknesses which is obviously not the goal of Java permissions.

Table 7.6 shows the protection rate brought in by Java permissions for the platform and the Java component vulnerabilities.

It improves the platform security but falls short in protected Java components from each others.

Requirements These considerations encouraged us to identify an alternative mechanism to permission, that are also called *Stack-based Access Control*: the *Component-based Access Control* (see section 8.3) which aims at providing security mechanisms that are more suitable for Java platforms such as OSGi. Moreover, the vulnerability catalogs show that isolation between bundles can not be obtained with mere access control mechanisms and that they are plagued with an important number of vulnerabilities if the development is not properly validated. Therefore, interactions between bundles at the package and at the service level are

[5] Discussions with the Open Web Applications Security Project (OWASP) Switzerland Group, 2007-07-24, among others.

[6] http://sf-jarsigner.gforge.inria.fr/

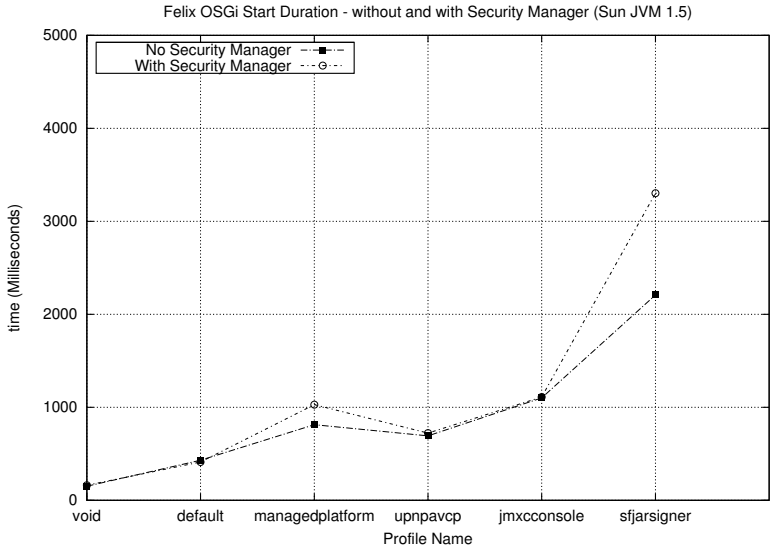


Figure 7.7: Performance of Felix OSGi platform without and with the Java security manager

Security Mechanism	**# of Protected Flaws**	**# of Known Flaws**	**Protection Rate**
Java Permissions (platform vulns)	13	32	41 %
Java Permissions (component vulns)	3	33	1 %
Java Permissions (all vulns)	16	65	25 %

Table 7.6: Protection rate for the Java security manager

to be protected, for instance through the Weak Component Analysis (WCA) tool (see section 8.4).

Conclusion Default OSGi platforms are very unsecure: even though specific implementations such as Equinox are to some extend a bit more robust than the others, they open an important attack vector by installing components from third party providers with almost no isolation. The standard security mechanism, Java permissions, provides a first improvement. It is well suited to perform access control. However, it has several drawbacks: it is not extensible for pre-existing classes, for instance to control the access to a method that is discovered to be dangerous in a particular context. Moreover, it implies an important performance overhead which makes it difficult to use in resource constraint environments. New tools that harden Java SOP platforms therefore need to be defined, to enhance the security status of both the platform and the components.

Part III

Hardening the Java/OSGi SOP Platform

Mechanisms for secure Execution in the Java/OSGi Platform

8

The security* analysis of the default distributions of the Java/OSGi platform* provides us with a detailed knowledge of the requirements that should be enforced if one intends to build a secure OSGi system*. A set of protection mechanisms is set forth, which goal is precisely to solve the identified vulnerabilities* in the OSGi platform implementations, in the access control model for Java environments, and in the component* code. These propositions build the iterations 2, 3 and 4 of the *SPIP Spiral Process for Intrusion Prevention*, presented in Section 6.2.

The protection mechanisms that are defined fall in the two categories for construction of secure software presented in Section 3.3: Architecture and Design, and Secure Coding. *Architecture and Design* solutions for the secure OSGi system are Hardened OSGi, the *Install Time Firewall* security pattern* and the CBAC (***C****omponent-****B****ased* ***A****ccess* ***C****ontrol*) security model. The *Secure Coding* solution for secure OSGi applications is WCA (***W****eak* ***C****omponent* ***A****nalysis*) which guarantees the robustness of component code.

Hardened OSGi [PF] aims at solving the vulnerabilities that originate in the OSGi platform implementations. The *Install Time Firewall* security pattern highlights the architectural modification we advocate to enhance the security status of SOP platforms: the verification of security properties of components immediately before their installation. This pattern is used for both the CBAC model and the WCA tool.

CBAC [PF08b] is an access control model that is enforced through static analysis at install time. Its goal is to restrict the access to sensitive code at the package, class or method level to which each component can access. Sensitive code can be provided by the platform or by other components. Its execution can so far only be controlled if access control is pre-defined through hard-coded permissions, or on a coarse grained level, since access to full packages or services* can be prevented by OSGi permissions. CBAC intends to solve the features of the Java security manager that appear to be ill-suited in SOP platforms, such as the important resource overhead, as presented in Section 7.3.3.

WCA automatizes code review for building vulnerability free components. It sets constraints on the code that components share with others. In particular, shared classes should be free of the vulnerabilities that can be exploited on classes, and shared objects, *i.e.* SOP services, from vulnerabilities that can be exploited on objects, as presented in Section 7.2.2.

For each mechanism, analysis and testing is performed through the $\mathcal{SPIP}$ method: benchmark* is based on the *Protection Rate* metric and tests are done through the sample malicious and vulnerable bundles* (see Section 6.3.2).

8.1 Hardened OSGi

The first proposition in the Architecture and Design category for secure software construction is *Hardened OSGi*, a set of recommendations for enhancing the implementations of the OSGi platform. It is presented in [PF07a] and [PF]. The objective is to patch flaws* that affect the SOP platform, as shown in Figure 7.6 page 109. In particular, the Life-cycle, Module and Service layers should be protected. The service binding process is not considered, because it builds an optional part of the OSGi framework.

These recommendations are first listed. Their implementation requires to slightly modify the OSGi API. The performance footprint is measured. The protection enhancement is quantified to evaluate the proposition.

8.1.1 Recommendations

Following modifications should be introduced in the implementations of SOP platforms, and in particular in the OSGi framework, to prevent known weaknesses. For each recommendation, the reference section of the OSGi R4 specification is given. They are presented according to the OSGi platform layer (see Table 7.2) that is concerned. The references of prevented attacks* as defined in the *Malicious Bundle* catalog in Appendix B.1 are given.

Module layer The recommendation for a hardened Module Dependency layer for SOP platforms, at the example of the OSGi Module layer, intends to make bundle dependency management more robust.

- *Do not reject harmless unnecessary metadata* in particular duplicate imports during the bundle dependency resolution process, but simply ignore them and install the concerned bundle (see *OSGi R4 par. 3.5.4*). This prevents the attack 'Duplicate Package Import' (mb.osgi.1) of the *Malicious Bundle* catalog.

Life-Cycle Layer Recommendations for a hardened Life-cycle Management layer for SOP platforms, at the example of the OSGi Life-Cycle layer, intend to protect the platform from the installation of malicious or ill-formed bundles as well as to guarantee that the uninstallation process is performed in a clean manner.

- *Check component size before download*, if a component download facility such as an OBR client [AH06] is available. The components should be installed only if the required storage space is available. This prevents the attack 'Big Component Installer' (mb.archive.2). This security mechanism can be used in conjunction with the 'maximum storage size' recommendation at the platform level (see next item).
- *Control the cumulated size of loaded components* that are actually installed on the platform. The check is performed immediately before the installation. It is based on a platform property, '`secureosgi.max.storage`'. Alternatively, a maximum storage size for all data stored on the local disk, such as component archives and files created by the component during its execution, could be set. The related specification paragraph is *OSGi R4 par. 4.3.3*. This prevents the attack 'Big Component Installer' (mb.archive.2).
- *Check digital signature* at install time* through a dedicated library. The check should not rely on the Java built-in validation mechanism since this latter is not compliant with the OSGi R4 specification [PF07b] (Section 2.1; see *OSGi R4, par. 2.3*) which is better suited for extensible platforms. This prevents the attack 'Invalid Digital Signature Validation' (mb.archive.1). Moreover, the standard Java validation is enforced at runtime which enables to install components with invalid signature (*e.g.* if one class if modified) and causes runtime failures (when this class is used).
- *Launch the component activator in a separate Thread*, to decouple the management process of the application starting and configuration process (see *OSGi R4 par. 4.3.5*). This prevents the attacks 'Management Utility Freezing - Infinite Loop' (mb.osgi.4) and 'Management Utility Freezing - Hanging Thread' (mb.osgi.5). It is only possible in JVMs that support threads.
- *Remove all component data at uninstallation* rather than when the platform is stopped. The related paragraph of the OSGi specification is *OSGi R4 par. 4.3.8*. This prevents the 'Zombie Data' attack (mb.osgi.8).

Service Layer The recommendation for a hardened Service Dependency layer for SOP platforms, at the example of the OSGi Service layer, intends to make the Service-oriented Programming* (SOP) feature of the platform more reliable.

- *Limit the number of registered services* for each SOP component. The value is set for instance through the `secureosgi.max.service.number` system property. It can be set *e.g.* to 50. The related paragraph of the OSGi specification is *OSGi R4 par. 5.2.3*. This prevents the attacks 'Numerous Service Registration' (mb.osgi.10) and 'Freezing Numerous Service Registration' (mb.osgi.11).

In the context of the OSGi platform, several attacks relative to the bundle archive, the bundle manifest, the bundle activator and the OSGi API are prevented through this set of modifications. In particular, memory exhaustion due to the installation of big bundles or unclean uninstallation, or denial-of-service against the platform management utility and the service registry are prevented. Installation of maliciously modified bundles is also prevented.

These recommendations can be used in any OSGi implementation to make it more robust in presence of malicious or simply ill-coded bundles.

8.1.2 Implementation

A prototype for Hardened OSGi, *Hardened Felix*, has been built based on the Felix Apache implementation[1] of the OSGi R.4 platform to validate the usability of these recommendations.

Impact on the OSGi API The control of storage size through the *Check bundle size before download* and *Control the cumulated size of bundles* recommendations requires to extend the OSGi core [All05a, All07a] and service APIs [All05b, All07b]:

- In the Class `BundleContext`, a method `getAvailableStorage()` is defined to support both recommendations.
- In the class `org.osgi.service.obr.Resource`, a method `getSize()` is defined to support the 'Check bundle size before download' recommendation. This method relies on the `size` entry of the bundle meta-data representation (usually a XML file).

Other modifications are strictly limited to the OSGi framework itself.

Performances To be acceptable as a security mechanism for production systems, *Hardened Felix* should imply only a limited performance overhead. Three phases of the life-cycle of the bundles are impacted by the Hardened OSGi recommendations: the installation, the service registration, and the uninstallation.

Figure 8.1 shows the performances of Hardened Felix without a Java security manager during the installation phase, which concentrates most of the modifications.

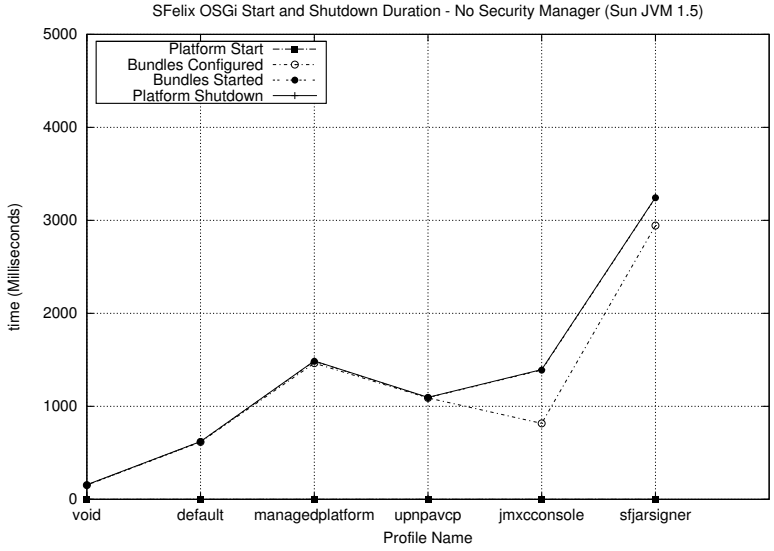


Figure 8.1: Performances of Hardened Felix

[1] http://felix.apache.org

These performances can be compared with those of the default Felix implementation, in Figure 7.7 page 112. The `void` profile is not impacted because recommendations concern the installation of bundles. Other aspects of the behavior of the platform are not impacted. The profiles: '`default`', 'JMX `managedplatform`' and '`upnpav` control point' contain bundles with reduced initial configuration, *i.e.* a limited amount of code in the `BundleActivator.start()` method. They experience an overhead of approximatively 20 %, due to the control of digital signature validation. The profiles: '`jmxconsole`' and '`sfjarsigner`' contain bundles with important initial configuration, *i.e.* a non negligible amount of code in the `BundleActivator.start()` method. The graph shows a decoupling of the duration of bundle installation and of bundle start, because the bundle activator is started in a dedicated thread. This modification leads to more availability of the platform management tool. The overhead is due mainly to the verification of bundle digital signature. It is therefore bound with the enforcement of the OSGi specification for security, which is not available on default distributions such as Felix. Actual overhead of the Hardened OSGi recommendations at start time is negligible.

Overhead in other phases of the bundle life-cycle such as service registration and bundle uninstallation is expected to be very low since it mostly concerns the verification of counters. The 'Remove all bundle data at uninstallation' recommendation is slightly more costly since it involved access to the file system but occurs during uninstallation only. It therefore does not impact directly the performance or availability of active bundles.

8.1.3 Results of Security Benchmarking

The Hardened OSGi recommendations intend to solve vulnerabilities that originate in the OSGi framework implementation. The precise evaluation of its benefits and limitations is required to show that its benefits overweight its cost and to identify the vulnerabilities that it can not prevent. This evaluation is performed with the *Malicious Bundle* catalog as reference. This means that only platform vulnerabilities are considered. Hardened OSGi does not intend to prevent component vulnerabilities.

Protection Rate for Hardened OSGi The Protection Rate (PR) for Hardened OSGi platforms are given in the Table 8.1 for the following configurations: theoretical PR value for the specification, PR value for Hardened Felix, PR value for Hardened Felix, PR value for Hardened Felix with Java permissions, and PR value for an hypothetical implementation of Hardened Equinox with Java permissions.

Hardened OSGi recommendations prevent one quarter of all the platform vulnerabilities. Used together with the standard security mechanism for Java environments, the Java permissions, they protect the system from two third to more than 70 % of the vulnerabilities, according to the considered implementation.

Further requirements As long as no combination of security mechanisms is available for the Java/OSGi platforms that prevents all identified vulnerabilities, further effort is required. The relative importance of remaining Java and OSGi-specific vulnerabilities are discussed for two configurations: without, and with Java permissions.

Vulnerabilities of Hardened Felix without Java permissions that are specific to the OSGi platform are bound with OSGi functions such as bundle management, OSGi services and fragments. Almost all of these can be protected through OSGi permissions.

Platform Type	# of Protected Flaws	# of Known Flaws	Protection Rate
Hardened OSGi (HO) Spec.	8	32	25 %
HO + perms	21	32	66 %
Hardened Felix	8	32	25 %
Hardened Felix with permissions	21	32	66 %
Hardened Equinox with perms (expected)	22	31	71 %

Table 8.1: Protection rate for the Hardened OSGi platform
[The evaluation is performed with the *Malicious Bundle* catalog as reference, *i.e.* only platform vulnerabilities are considered. Hardened OSGi does not intend to prevent component vulnerabilities.]

One single vulnerability of Hardened Felix with Java permissions is bound with the OSGi platform, *i.e.* 9 %: the 'Erroneous value of Manifest attributes'. It is actually more a configuration error than an raw vulnerability but is considered as a kind of denial-of-service because it can in several cases lead to the failure of the bundle installation.

OSGi vulnerabilities can be considered to be fully patched by the Hardened OSGi recommendations, when suitable Java and OSGi permissions are set. However, Java vulnerabilities keep being open and make OSGi platforms at risk. The remaining major attacks that can occur in a Hardened Felix platform with Java permissions set are the following: platform crash through recursive thread creation, uncontrolled CPU or memory resource consumption, as well as hanging thread, decompression bomb and excessive size of the manifest file.

Relative to default OSGi platforms, Hardened OSGi recommendations bring an important step towards building secure systems. When used together with Java permissions, Hardened Felix enables the development of platforms that have a high Protection Rate. Hardened Felix with permissions has a Protection Rate of 66 %, and Equinox with the same mechanisms provides a Protection Rate of 71 %. However, the important overhead of permissions acts as a deterrent for their actual use in production system. They can therefore not be considered as an sufficient solution. An alternative mechanisms is required.

8.2 The *Install Time Firewall* Security Pattern

The next propositions in the Architecture and Design category for secure software construction are implementation of the *Install Time Firewall* security pattern. This pattern targets extensible component platforms*. The principle is the following: components are validated after they have been downloaded and before they are installed on the platform. This pattern builds the basis of our subsequent propositions: *Component-based Access Control* (CBAC) presented in Section 8.3, and *Weak Component Analysis* (WCA) in Section 8.4.

Table 8.2 gives the full representation of the *Install Time Firewall* security pattern.

Reference

Name: Install Time Firewall

Extends: -

Also Known As: -

Definition date: 2008/05/21

Related Patterns: Component-based Access Control, Weak Component Analysis

Problem

Context: The *Install Time Firewall* pattern is relevant for component platforms that support component life-cycle management. The installation phase of components in the system is extended by security checks.

Scope: This pattern is applicable for all component platforms with life-cycle management facilities.

Purpose: Security checks at runtime are often costly, and strongly impair user experience. This has several consequences. When security mechanisms are on, systems behave less efficiently. However, this fact has often as the pragmatic consequence that security is simply turned off in production systems. The objective is to reduce or withdraw runtime checks.

Intent: Since more and more systems enable to manage the life-cycle of components, the *Install Time Firewall* pattern intends to exploit the installation phase of the life-cycle to perform security checks on the component archive. Such checks include metadata validations or code static analysis.

Motivation: Installation is not part of the execution of a component, and is often a process that imply several time-costly operations, such as component discovery, download, or the standard digital signature validation. Performing additional checks is likely to add a negligible overhead to a process that is not time critical.

Solution

Applicability: The platform must have a reference on the software component that is to be checked, and this reference can not be modified after the check by an external party.

Structure: The platform first gets a reference on a software component. Then, is passes it to a security checker. The checker retrieves the security policy, if is does not yet have it, and checks the component according to this policy. Figure 8.2 shows the UML sequence diagram for the *Install Time Firewall* security pattern.

Participants: The entities that participate in the *Install Time Firewall* pattern are the following: the component platform, the checker library, the policy database. The software component is a passive entity that is being checked.

Collaborations: -

Consequences: Components that are valid according to the policy are installed. Others are rejected. Variations can imply modification of the runtime behavior of the system according to verification results. Verification occurs in a negligible amount of time and does not impact the behavior of the system when it is executed. The limitations is the number of false positives, and the fact that install time analysis sometimes needs to be complemented by runtime checks for finer verifications.

Implementation: Integrity of the platform, the checker library and the security policy must be guaranteed.

Known Uses: Digital Signature Validation; Proof-Carrying-Code (PCC); CBAC; WBA.

Table 8.2: The *Install Time Firewall* security pattern

The Problem The problem it tackles is the following. Components that are dynamically installed from the environment are usually not provided with any security guarantee. When these components are installed in security sensitive environments - and any production platform is, as well as any PC since its owner is responsible for the possible attacks that would be launched from it - their probable behavior should be evaluated and compared with explicit policies. The motivation for performing these checks at install time is that runtime security checks are often costly and impair the user experience. Standard Java security is therefore frequently removed from the execution environments* for convenience reasons. Since the installation process is performed only once and involves several costly operations such as download and validation of digital signature, the additional performance overhead of the *Install Time Firewall* pattern is not likely to be annoying. It can be unnoticeable for checks that are not overly too complex.

The Solution The pattern can be applied as soon as a reference to the component is available. To prevent TOCTOU (Time Of Check to Time Of Use) attacks, no external party should be able to modify this reference afterwards. It implies the participation of several software entities: the component platform, that launches the check; the checker module, that performs it; and the policy database, that provides security policies the component should be compliant with. All components that are valid are installed. All components that are not valid according to the policy are rejected. The limitation of this pattern is that static analysis, which can be performed on the component code and metadata, necessarily produces false positives with regard to runtime analysis. Implementations of this pattern should guarantee that the rate of false positives is kept as low as possible.

Figure 8.2 shows the UML sequence diagram for the *Install Time Firewall* security pattern.

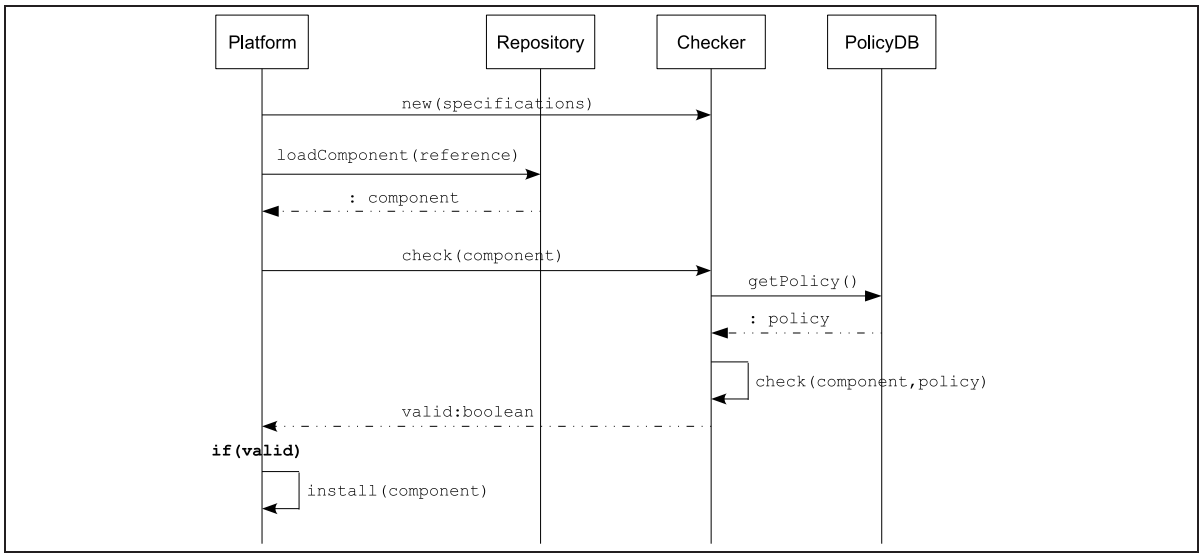


Figure 8.2: The *Install Time Firewall* security pattern: sequence diagram

The interactions between the concerned entities are the following. First, the platform creates a new checker object. When the component is loaded, its sends a request to the checker to perform the evaluation. At the first evaluation request, the checker loads the policy. It can then perform the security check of the component according to this policy. If

this check is negative, *i.e.* no security policy violation is identified, the component is installed.

8.3 Component-based Access Control - CBAC

The second proposition in the Architecture and Design category for secure software construction is the ***C**omponent **B**ased **A**ccess **C**ontrol* (CBAC) security model. Its goal is to restrict access to dangerous method calls to authorized components only. It is presented in [PF08b].

8.3.1 Requirements

The objective of CBAC is to provide an access control mechanism to all sensitive functions* that are identified in the Java Virtual Machine and in the SOP platform itself, as shown in Figure 7.6 page 109.

Unprotected Vulnerabilities The first category of sensitive functions which access should be controlled entails the functions that lead to denial-of-service attacks if the components are not developed in a careful manner and well tested. This is the case of `Threads` which can lead to platform crash and call hanging but also of native code execution which lets the component execute any utility that is provided by the underlying operating system to abuse it or to attack the SOP platform itself.

The second category of sensitive functions which access should be controlled entails the functions that allow the components to access and alter system resources configurations. In the Java Virtual Machine, the use of class loading, reflection and file manipulation are concerned. In the SOP platform, the administration functions of the life-cycle, module and service layer are concerned. Examples of such functions are respectively the installation of bundles, the use of fragment bundles and the registration of services.

In addition to these vulnerabilities that are presented in Section 7.2.1.3, components themselves can provide sensitive or dangerous methods. The administrator of the platform should be able to define an access control policy that limits the access to component methods to trustworthy code only or to prevent for all the access to methods that are identified as malicious. Moreover, would a given method be identified as dangerous, it should be possible to fully prevent its access without requiring that the method itself enforces the call, as is usual in Java permissions.

An Alternative to Java Permissions Most of the sensitive methods that are provided by the JVM standard API as well as by SOP platforms such as OSGi can be protected by Java permissions. However, they fail in being an efficient security solution, as discussed in Section 7.3.3. Their deficiencies are 1) the runtime overhead, up to 30 %, 2) the programmative definition of sensitive methods, and 3) the runtime failure they imply. The runtime overhead induces an important loss in user experience. Programmative definition of sensitive methods implies that each sensitive method declares itself as such. This is likely not what malicious code does and makes the Java permissions approach valid for access control in benevolent applications only. Failed security checks lead to errors during the execution or urge the user to grant required execution rights, often without a precise understanding of their consequences.

Our access control method should therefore have following properties. It should imply little to no runtime overhead. It should support declarative policies for arbitrary code. Lastly, it should not discover unresolved rights at runtime.

8.3.2 Definition of CBAC

The CBAC security model is defined to provide a solution to the identified requirements for access control in SOP platforms.

The CBAC Security Pattern The *Component-based Access Control* security pattern

The *Component-based Access Control* security pattern aims at enforcing access control at the code level by identifying the method calls that are performed in the component. Direct calls, performed in the component, as well as indirect calls, performed in dependencies, are considered. This security mechanism is applicable for all component platforms where an installation phase can be identified to enforce code analysis and verification. The objective is to replace the Java permissions security model, also known as *Stack-based Access Control*, while resolving its limitations. The CBAC analysis is performed as follows for each component. The description, as list of methods, of the calls that are performed in the component are extracted through static analysis. The security verification is performed in two steps: first, the calls that are directly performed in the component are verified; next, the calls that are performed in the component dependencies are checked. A call can be executed if it is innocuous or if it is both sensitive and granted for all of its call stack. Since a component is provided by a given issuer, it builds a security domain (see Section 4.1.3). Static analysis enables to perform checks without impairing the runtime performances of the applications. Explicit policies enable to extend the calls that are considered as sensitive, what is not possible with Java permissions.

The structure of the Component-based Access Control model is shown in Figure 8.3 as an UML sequence diagram.

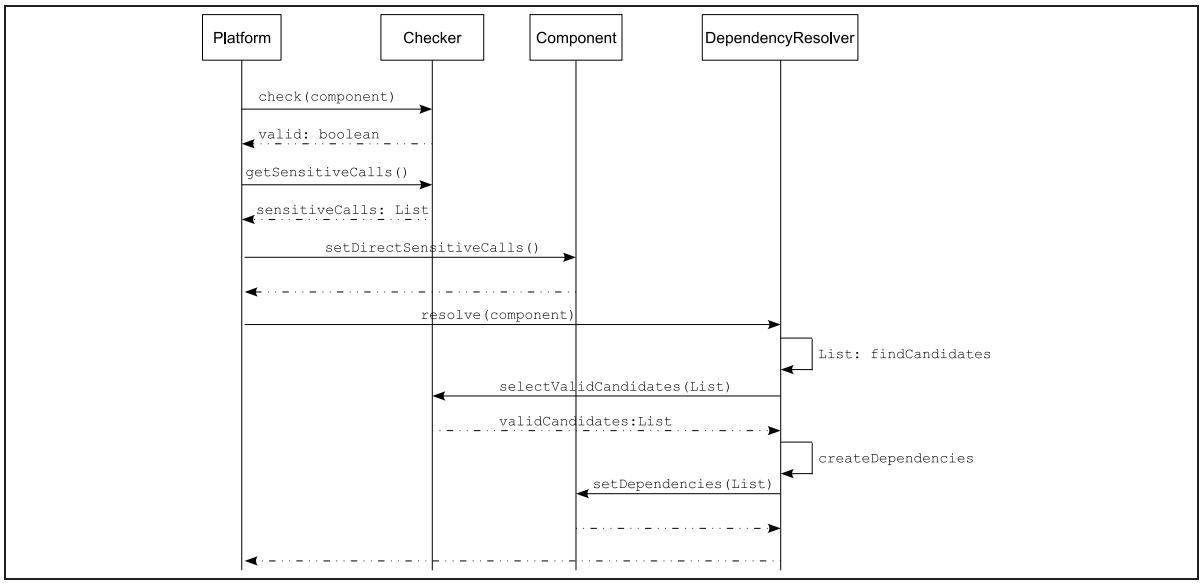


Figure 8.3: The *Component-based Access Control* security pattern: sequence diagram

Components that are valid according to the CBAC policy are installed. Others are rejected. The limitation of CBAC is that static analysis implies a certain rate of false positive when compared to runtime analysis and thus a rejection rate that is higher as would be required to prevent the dangerous calls that are actually performed.

Hypotheses The CBAC security model is valid when the two following hypotheses are valid:

- The component platform itself is not modified *i.e.* the process of access right verification cannot be tampered with. This can be obtained through the use of a Trusted Computing Base (TCB) [AFS97].
- Each component contains a valid digital signature which guarantees that no modification has been done to the component archive and that the component signer name is unique and known by the platform[2].

Formalization Following parameters are defined to describe security policy-related entities in an extensible component platform:

pf **:** the Component platform,
b_i **:** the ID of the considered bundle,
$\{b\}_i$ **:** the list of bundles on which the bundle i depends,
$C_{S_{pf,b_i}}$ **:** the sensitive calls performed by the bundle i directly to the platform, or directly to the bundles on which it depends,
p_i **:** the provider of the bundle b_i,
A_{p_i} **:** the set of authorized sensitive calls for the provider p of the bundle i,
PSC_{b_i} **:** the set of Performed Sensitive Calls by the bundle i, directly to the platform or via method calls to other bundles,
$PSC_{\{b\}_i}$ **:** the set of Performed Sensitive Calls by the bundles on which the bundle i depends. $PSC_{\{b\}_i} = \sum_{b_j \ in \ \{b\}_i} PSC_{b_j}$.

- A component N is valid if:

$$b_i, p_i, pf \vdash A_{p_i} \wedge PSC_{\{b\}_i}, \neg PSC_{\{b\}_i} \tag{8.1}$$
$$with \ PSC_{b_i} = C_{S_{pf,b_i}} \vee PSC_{\{b\}_j} \tag{8.2}$$

This means that a bundle can be installed when the calls that are made directly to the platform or to other bundles and all calls that are made via other bundles are either sensitive and allowed by the current policy or innocuous.

This can be demonstrated with the following argument through recursion. Suppose that the set of Performed Sensitive Calls for the bundle k, PSC_{b_k}, is available for all bundles k that are already installed on the gateway. The bundle i which has dependencies to a set of bundle b_j can be installed if its execution does not break the access control policy through direct or indirect calls. The value of PSC_{b_i}, the set of Performed Sensitive Calls for the bundle i, can then be extracted. The demonstration of this theorem with Sequent Calculus is given in the Appendix of the related paper [PF08b] and available on the web[3].

8.3.3 Implementation

The first step in validating the proposed security model is to implement it so as to evaluate its feasibility and the impact is has on performances. The implementation of the CBAC Model is performed on the Felix implementation of the OSGi framework.

[2] In particular, this implies that default Java Archive signature verification tools should not be used [PF07b].
[3] http://www.rzo.free.fr/parrend08cbac.php

Integration with OSGi The integration of the CBAC checker library with the OSGi platform is done through a call to the `OSGiSecurityChecker.check()` (which is called `Checker.check()` in the security pattern, Figure 8.3) that takes place immediately before the actual installation of the bundle on the Java/OSGi platform. In Felix, this check is performed at the end of the creation of the `BundleArchive` object, and precisely at the end of the `revise()` method. This ensures that all data that is required to check the component is available, including the component itself and that, in case of error, the platform is rolled back to the state before the operation (installation or update of a bundle) began.

The verification of the compliance between a component and the CBAC policy requires that the issuer of the component is known, for instance through the component digital signature, as presented in Section 2.1. The CBAC mechanism can not be enforced if the digital signature is not available.

Implementation Choices The goal of the CBAC library is twofold: to handle the policies to be enforced, and to extract related data from the component Bytecode through static analysis. Policy management is developed in an ad-hoc manner. Static Bytecode analysis is performed with the ASM library [4] [BLC02]. It aims at extracting the list of sensitive calls that are performed by the components. ASM is selected because it is much smaller than other libraries for Bytecode manipulation such as BCEL [5] or SERP[6], and because its performances are better by several orders of magnitude [7].

An earlier prototype has also been built using the FindBugs framework which often implies an overhead of more than 100% in performance because of the indirection between the SOP platform and the Bytecode analysis library. This overhead is mainly due to the fact that FindBugs is a development tool and not a tool for securing execution environments. In this context, expressivity and flexibility is privileged over performance.

Policy Management Since CBAC intends to replace Java permissions, the CBAC policy is expressed according to a very similar syntax. The objective is to relieve developers from as much learning efforts as possible.

Table 8.3 shows an example of a CBAC policy file.

A CBAC policy file is defined for the whole platform. It is compound of two parts: the list of sensitive methods and Manifest attributes, and the execution grants for each bundle signer, *i.e.* the sensitive methods and Manifest attributes the signer is allowed to execute.

The list of sensitive methods contains the qualified name* of packages, classes and methods that are considered as sensitive. The sensitivity property is hierarchical. If a given package is sensitive, all the classes it contains are sensitive. If a given class is sensitive, all the methods is contains are sensitive. The keyword `<init>` is used to denote the constructor method, as in the Bytecode language. The list of sensitive Manifest attributes contains the name of the attributes that are sensitive, such as `Fragment-Host` which enables a bundle to attach to another (see Section 5.3.1). The default values for sensitive methods should cover all methods that are protected through Java permissions, as well as all methods that are identified as sensitive in the *Malicious Bundle* catalog. The identification of sensitive

[4] http://asm.objectweb.org/
[5] http://jakarta.apache.org/bcel/
[6] http://serp.sourceforge.net/
[7] http://james.onegoodcookie.com/?p=106, in addition to the conference paper by the authors [BLC02]

```
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;
  ...
};
```

Table 8.3: Example of a CBAC policy file

methods from the *Vulnerable Bundle* catalog would require the support of single method names such as `readObject` and `writeObject`, to protect the access to serialization.

The execution grants contain the list of attributes and methods that the related signer can access. In Java permissions, the `Principal`* for a code archive can be either the signer or the code location on the network or on the filesystem. The principal for CBAC can only be the signer of the archive. This enables to guarantee that the archive has been issued by the signer itself and that the trust that can be put on the component is the same that is put on the signer itself. Of course, several grant entries can exist in the CBAC policy, because bundles are likely to be provided by various signers.

Management tools enable to check the validity of bundles or to extract the policy they require. The `cbac_validity.sh` tool checks the validity of a bundle according to the policy, without taking dependencies into account. The `cbac_requirement.sh` tool provides the CBAC policy requirements for a bundle or set of bundles, and for a given list of sensitive methods. They are currently both implemented as Linux shell scripts.

The access control check for indirect calls is performed as shown is Figure 8.3: the list of performed sensitive calls is stored for each by the platform through a `DependencyResolver` object. Bytecode analysis for each bundle is performed when it is installed or updated, and needs not be performed again when new bundles that depends on them are installed.

Performances One of the objective of CBAC is to resolve the performance limitations of Java permission. The overhead implied by CBAC should therefore be reduced, so as to makes it a credible alternative to the Java standard.

Figure 8.4 shows the duration of the CBAC check only and Figure 8.5 shows the performance of Digital Signature validation and CBAC check which are to be performed together to ensure the validity of the analysis.

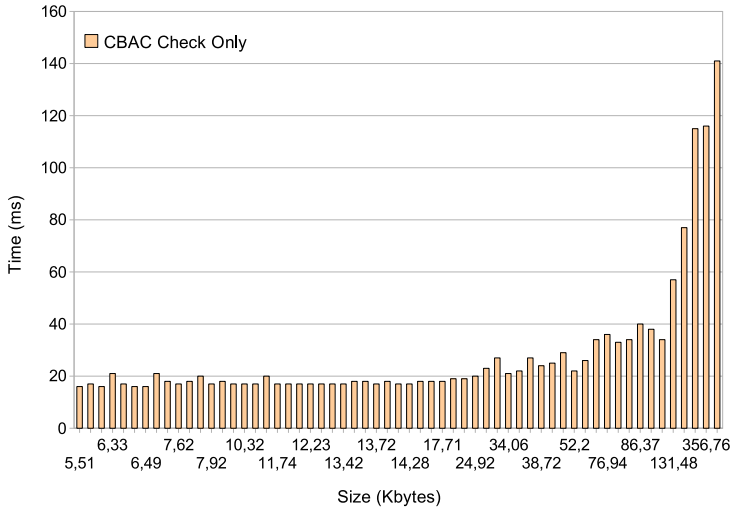


Figure 8.4: Performances of OSGi security: CBAC check only

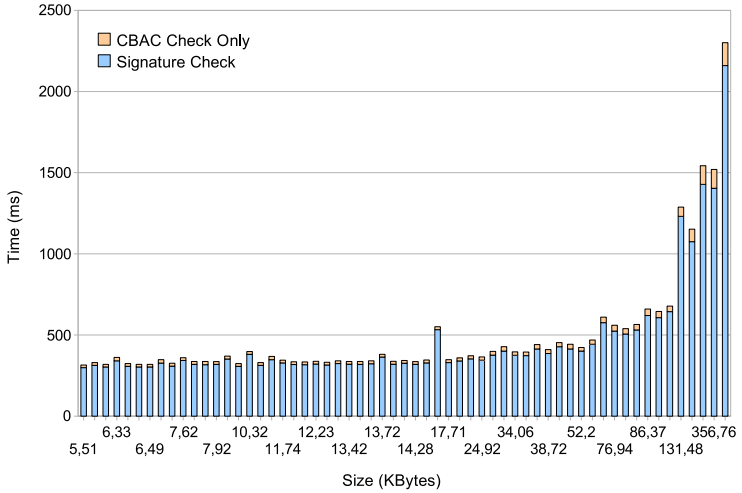


Figure 8.5: Performances of OSGi security: Digital signature and CBAC checks

These graphs highlight the fact that for a limited number of sensitive methods (which is usually the case), the overhead implied by CBAC is negligible when compared to the duration of the digital signature check. Note that the abscissa is not linear but represents the size of the various bundles that are available in the Felix distribution of the OSGi platform.

For most bundles, that are smaller that 30 KBytes, the verification time for CBAC only does not exceed 20 ms. For bundles until 100 KBytes, it is less than 40 ms. For the bigger bundles, that weight a couple of hundred KBytes, the verification time does not exceed 140 ms.

For the bundles that weight 70 KBytes, the verification time, including the verification of digital signature, is less than 500 ms. This concerns the vast majority of sample bundles. For bundles under 100 KBytes, the verification time does not exceed 600 ms. In rare cases, where bundles weight several hundred KBytes, the verification time is beyond one second. The worst case example is 2.3 seconds.

These figures show that the performance cost of CBAC is very limited, especially when compared with the verification of the digital signature. Any SOP platform that intends to put any control on the security status of the executed code should verify the digital signature of bundles[8]. Moreover, this overhead is restricted to the installation phase of components. It therefore does not impact the runtime behavior.

Advanced Features To provide a proper protection of OSGi applications, several complementary features have to be defined. They are not implemented in the first CBAC protype but are required to support an efficient use of this security mechanism.

The first feature is *Privileged Method Calls*. Privilege calls means that a given component can execute sensitive calls, even though the initial caller of the method does not own sufficient rights. This is for instance the case of logging mechanisms. A component from a less trusted provider can log its action on a file through the platform logger without having access rights to the file system.

The second advanced feature which can easily be provided by the CBAC model thanks to its declarative nature is the support of both *Positive and Negative permissions*. Currently, negative permissions are set by identifying a given method call as 'sensitive'; positive permissions are set by allowing a given signer to execute some methods. A more flexible expression could be introduced in the future, in particular to support negative permissions for a given signer without impacting the policies for other bundle providers.

The last required feature is to support *Access Control for Service Calls*. The OSGi platform support Service-oriented Programming (SOP) [BC01] and thus calls through services that are published inside the platform. Since the services are resolved at runtime, a runtime mechanism is to be defined that enforces the CBAC policy for these service. Otherwise, package level access control can be by-passed through service calls.

8.3.4 Results of Security Benchmarking

The validation of the proposed security model is performed through security benchmarking, to check whether requirements are met.

Prevented Attacks The CBAC security model prevents the exploitation of the following vulnerabilities. The references of vulnerabilities according to their definition in the *Malicious Bundle* catalog in Appendix B.1, and in the *Vulnerable Bundle* catalog in Appendix B.2, are given.

[8] This is not true in the case of code that is proved to be correct such as Proof-Carrying Code. However, the availability of this technique does not allow so far to use it for production systems, at least in the Java world.

Security Mechanism	# of Protected Flaws	# of Known Flaws	Protection Rate
Hardened OSGi (HO)	8	32	25 %
CBAC	16	32	50 %
Java Permissions (Perms)	13	32	41 %
HO + CBAC	24	32	75 %
HO + Perms	21	32	66 %

Table 8.4: Protection rate for the CBAC protection mechanism, and comparison with the Java Permissions alternative
[The evaluation is performed with the *Malicious Bundle* catalog as reference, *i.e.* only platform vulnerabilities are considered. The results are given for the Felix/CBAC implementation.]

- 'CPU load injection' (mb.native.1) when it is performed through native code.
- `System.exit()` (mb.java.1) and `Runtime.halt(mb.java.2)`.
- 'Recursive Thread Creation' (mb.java.3), 'Hanging Thread' (mb.java.4) and 'Sleeping Bundle' (mb.java.5) when the use of the `Thread` API is forbidden.
- 'Big File Creator' (mb.java.6) when the use of the `File` API is forbidden.
- 'Code Observer' (mb.java.7), 'Component Data Modifier' (mb.java.8) and 'Hidden Method Launcher' (mb.java.9) when the use of the `Reflection` API is forbidden.
- 'Launch a Hidden Bundle' (mb.osgi.6) and 'Pirate Bundle Manager' (mb.osgi.7) when the use of the OSGi `BundleContext` API is forbidden or restricted.
- 'Execute Hidden Classes' (mb.osgi.12), 'Fragment Substitution' (mb.osgi.13) and 'Access Protected Package through split Packages' (mb.osgi.14) if the use of `Fragments` is forbidden.
- 'Shutdown Hook' (vb.java.class.4) if the use of `Shutdown hooks` are forbidden.
- 'Expose Internal Representation - Serialized Sensitive Data' (vb.java.1), 'Cloning' (vb.java.class.9) and 'Deserialization' (vb.java.class.10) could be prevented if CBAC is extended to support the execution of methods based on their (short) name. This feature is currently not implemented.

A recapitulative table for platform vulnerabilities and the suitable protection mechanisms can be found in Table 1 from the related technical report [PF07a].

Protection Rate for CBAC The Protection Rate for CBAC and other security mechanisms for the Java/OSGi platform is given in Table 8.4: PR value for Hardened OSGi (see Table 8.1), PR value for CBAC, PR value for Java permissions, and two combinations: PR value for Hardened OSGi + CBAC, and PR value for Hardened OSGi + Java permissions.

All results are given for our implementation of CBAC over Felix.

CBAC protects from 50 % of the vulnerabilities from the *Malicious Bundle* catalog. Combination of several mechanisms give the maximal protection level that can be achieved with available tools. The use of Hardened OSGi together with CBAC provides a protection rate of

75 %. In comparison, Hardened OSGi together with Java permissions provides a protection rate of 66 %.

CBAC prevents the exploitation of 1 single bundle vulnerability out of 33 in the *Vulnerable Bundle* catalog.

Comparison with Java Permissions CBAC is designed to solve several drawbacks of Java permissions. It therefore presents better properties what relates to runtime overhead, policy expression, and absence of runtime failures. Runtime overhead is prevented by the execution of all verifications at install time. Policy are expressed in a full declarative manner which enables to set any method as vulnerable even after the application has been released or installed. Since components that do not comply with the CBAC policies are rejected before being installed, all components that are on the platform are executed seamlessly, with risk of runtime failure.

Moreover, the security benchmarking with the Protection Rate metric shows that CBAC has a slightly better protective power than Java permissions.

As a runtime mechanism, Java permissions keep being more accurate. A call that is present in the code but never executed will be rejected by CBAC but be unnoticed with permissions. This can cause problems for instance with libraries that contain an important amount of code that is not used in a specific context.

Limitations The proposed model for CBAC is an efficient step towards the development of more secure applications. However it has some drawbacks and does not prevent all vulnerabilities.

The main drawback of the current implementation is the coarse granularity of the security checks. All calls that are contained in the code are taken into account for policy compliance, even if they are never called. An extension of the model using Control Flow Graphs [BJMT01] could be an efficient solution for this problem.

Another drawback which is easier to patch is the fact that the declaration of sensitive methods that are not bound to a specific class is not supported. Vulnerabilities that could be prevented by this extension exist in the *Vulnerable Bundle* catalog, such as the `readObject()` or `writeObject()` methods that are called during the serialization process.

Seven platform vulnerabilities stay unprotected by the Hardened OSGi and CBAC security mechanisms. They are bound with consumption of resources such as CPU and memory, and the lack of algorithmic safety in the Java language. Resource consumption control can be done by taking advantage of the fact that each bundle is started in a dedicated class loader to isolate it. Such control can only be enforced inside the JVM and can not be done through the Software Security* techniques. The JnJVM provides such an isolation of components based on class loaders [TGCF08]. The integration of this secure JVM with our hardened OSGi platform is presented in Section 9.1.

The last limitation of CBAC is that it contains a flaw that enable to by-pass the access control mechanism. This flaw is namely the 'Malicious inversion of control through overridden parameter', that is presented in Section 7.2.2.1. When a component calls another one, it can provide its own implementation for all parameter classes that are not final. In particular, they can provide an implementation that performs a call back to the initial class. CBAC (contrary to Java permissions for instance) can not identify such calls. It is therefore necessary to put additional constraints on the code that is shared between components, to prevent the abuse

of this flaw - and of other flaws that are presented in the *Vulnerable Bundle* catalog. This effort leads us to the definition of *Weak Component Analysis*, WCA.

8.4 Weak Component Analysis - WCA

Our proposition in the *Secure Coding* category for secure software construction is the ***W**eak **C**omponent **A**nalysis* (WCA) code validation tool. Its goal is to ensure that installed components are free from common vulnerabilities that can be exploited by other components. It is based on the taxonomy* for implementation of component vulnerabilities presented in Table 7.3 page 106. Some vulnerabilities can be exploited when they are located in shared classes such as exported packages from OSGi bundles, some when they are located in shared objects such as SOP services, some can be exploited independently of their location in the component. WCA is presented in [Par09].

8.4.1 Requirements

Secure coding is required to ensure the validity of secure design: high level protection mechanisms can otherwise be bypassed though the exploitation of local weaknesses. This fact is highlighted by the *Vulnerable Bundle* vulnerability catalog. Almost none of its entries are protected by the protection mechanism we propose, nor by standard protection mechanisms such as the Java security manager.

Moreover, static analysis tools such as FindBugs and JSLint (see Section 4.3.2 page 52) enforce coding best practices but do not consider the exploitability of vulnerability. We claim that the development of secure components imply to enforce very strict constraints on code that is accessible to third party components, *i.e.* shared classes and shared objects, to prevent uncontrolled access from one component to another. The inner code of the component should be written in a clean manner but is less sensitive.

One special case is to be considered. Some components share all of their classes. This is the case of libraries but also of OSGi bundles that act as fragment Host (see Section 5.3.1 page 65). All classes are then shared classes and should be free from related vulnerabilities.

Finally, the analysis should be performed on the Bytecode level to enable the verification of Java components for which the source code is not available.

8.4.2 Definition of WCA

The WCA Security Pattern The *Weak Component Analysis* security pattern aims at preventing exploitable vulnerabilities in the installed components. It is applicable for all component platforms that execute components sharing code resources with different exposition types: sharing of objects, sharing of classes, and internal classes. Internal classes must be analyzed since they can contain language constructs that are identified as potential vulnerabilities such as `synchronized` code. Since different vulnerabilities flaw code according to its exposition, component classes should be written accordingly. The objective is to prevent malicious components from impacting the behavior of others and from gaining access to their internal state. Dubious interactions are those that 1) do not occur through proper encapsulation, and 2) enable to impact in an unintended manner the behavior of components that do not take part in the current interaction. The realization of WCA is motivated by the fact

that available tools only enforce generic best practices for coding without regard to the actual threat. They are therefore likely to be not strict enough for highly exposed code.

WCA analysis requires that the actual exposition of each class in the component is known: which classes are kept internal, which classes are shared as such, and which classes are shared as objects, for instance as service. Identification of vulnerabilities can be performed through static analysis. WCA policy defines the vulnerabilities that are to be identified and the type of reaction for each of them. If WCA is used during development, it typically generates a report with all identified vulnerabilities. If it is used to perform code check at install time, it typically rejects invalid components.

The structure of the WCA Security Pattern is shown in Figure 8.6.

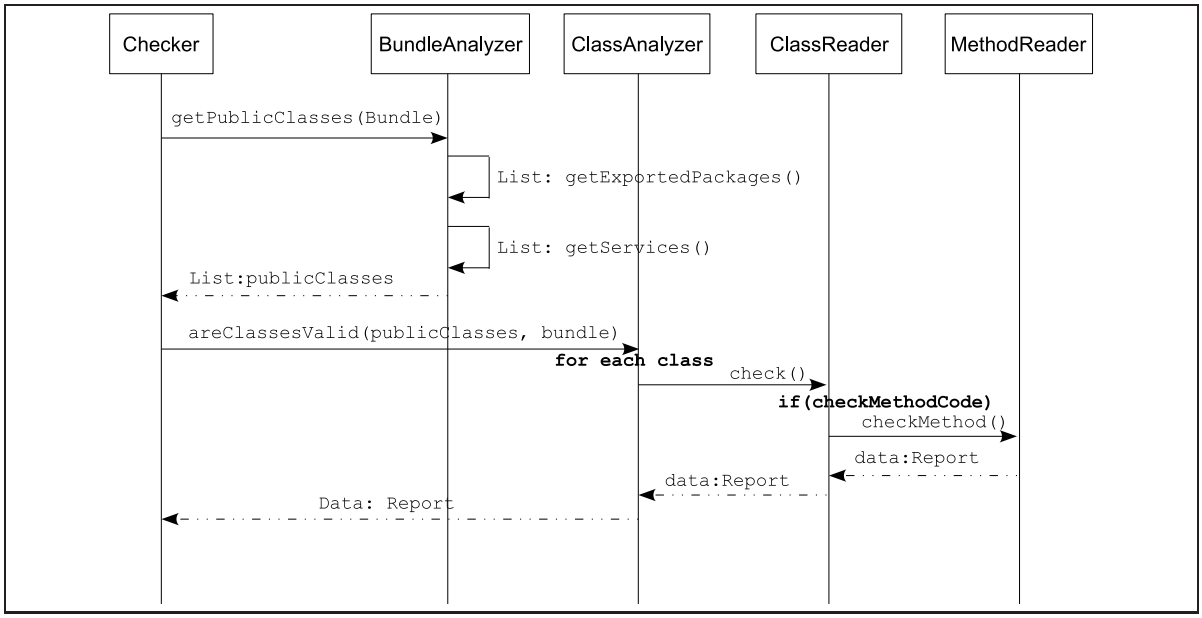


Figure 8.6: The structure of the WCA Security Pattern

The limitation of WCA is that it is not possible to identify vulnerabilities from code or Bytecode in the general case through static analysis. Only vulnerabilities that can be expressed can be identified. Others are neglected.

Hypotheses The WCA security model is valid when the following hypothese is valid:

- The component platform itself is not modified *i.e.* the process of access right verification cannot be tampered with.

The WCA formal Vulnerability Pattern Target flaws are expressed as a formal vulnerability pattern which defines the set of properties that characterizes a vulnerability. The patterns are compared with the structure of the classes and the code of their methods. When all properties of a pattern occur in the same class, the related vulnerability is identified. The formal vulnerability pattern is composed of three parts: the vulnerability reference, the related message, and the characterization of the vulnerability.

Listing 8.5 gives a concrete example of the formal Vulnerability Pattern*, for the vulnerability *Synchronized method call*.

```
 <vs:vulnerability>
    <vs:vulnerabilityRef>
      <vs:catalog_id>vb</vs:catalog_id>
      <vs:src_ref>java</vs:src_ref>
      <vs:type>class</vs:type>
      <vs:id>15</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 (Vulnerability can be exploited with class-sharing only
through a static call).
    </vs:message>
    <vs:exposition>sharedClasses</vs:exposition>
    <vs:location>allCode</vs:location>
    <!--vs:weaknessScope>publicClasses</vs:weaknessScope-->
    <vs:method>
      <vs:access>synchronized</vs:access>
    </vs:method>
  </vs:vulnerability>
```

Table 8.5: Example of the formal Vulnerability Pattern: Synchronized Method Call

This vulnerability is referenced as the vulnerability `vb.java.class.15` (see Appendix B.2 page 179). It concerns components that have classes exposed as 'shared classes', and occurs if a method located in any class of the component (`allCode`) is `synchronized`.

Policy Management Policy management in WCA consists in setting the type of reaction when a vulnerability is discovered. The reactions can be: `ignore`, `warning`, or `reject`. It can be set for each individual vulnerability. The maximum reaction level is set according to the origin of the related class. Classes can be provided either by the current component, by dependency components, by the classpath, or in the standard JVM archives.

Table 8.6 shows the example of a WCA policy file.

In this example, the reaction level for the vulnerability `vb.java.class.14` is set to `warning`. The maximum reaction level for the classes provided in the current component (`current_bundle` property, since WCA implementation is OSGi specific so far) is also set to `abort`.

8.4.3 Implementation

The fist step in validating the proposed security model is to implement it so as to evaluated its feasibility and the impact it has on performances. The implementation of WCA is realized as a stand-alone command line tool. It provides a `BundleVulnerabilityChecker` Java class to makes its integration as a library into existing systems easy.

```
<!-- policy entries -->
  <vcp:policyEntries>
    <vcp:policyEntry>
      <vcp:vulnerabilityRef><!-- vulnerability reference -->
        <vcp:catalog_id>vb</vcp:catalog_id><!-- tobe released -->
        <vcp:src_ref>java</vcp:src_ref>
        <vs:type>class</vs:type>
        <vcp:id>14</vcp:id>
      </vcp:vulnerabilityRef>
      <vcp:level>warning</vcp:level><!-- could be: ignore|warning|abort -->
    </vcp:policyEntry>

  <!-- force max level entries  -->
  <vcp:forceMaxLevel>
  <vcp:maxLevel>
    <vcp:maxLevel>
      <vcp:origin>current_bundle</vcp:origin>
      <vcp:level>abort</vcp:level>
    </vcp:maxLevel>
  </vcp:forceMaxLevel>
```

Table 8.6: Example of the Policy for Reaction to Vulnerability

WCA in the Component Life-Cycle The WCA tool is made available through two different versions: `bundlevulchecker.xml`, based on the XML expression of the formal vulnerability pattern, and `bundlevulchecker.hardcoded` which implements the same policy through Java programming to limit the performance overhead due to the analysis of XML policies as well as the memory footprint bound with the presence of additional libraries.

`bundlevulchecker.xml` aims at providing flexibility for developers. The set of vulnerabilities that are looked after are defined in a dedicated XML file. They can therefore be selected easily. Current implementation of the WCA tool only enables this selection. More mature versions should support the definition of arbitrary vulnerabilities.

`bundlevulchecker.hardcoded` aims at providing a better performance for validation of the component at install time. To ensure the coherence between both versions, the class enforcing the vulnerability analysis in `bundlevulchecker.hardcoded`, `GenVulnerability`, is generated automatically from the XML file for vulnerability definition.

Implementation Choices The implementation of WCA requires the support of following information: the exposition of the classes in the component, the representation of the classes through a dedicated meta-model. Moreover, a library for static code analysis must be selected.

The conditions for the realization of WCA is that the exposition of each class from the component under analysis is known. In the case of OSGi, shared classes are the classes contained in exported packages. Shared services can not be easily deduced from code, at least in the general case. We therefore advocate to add a specific `ServiceImpl` entry in the metadata which references the service and related implementation classes. This makes the

current implementation of WCA not fully compatible with default implementation of OSGi bundles due to this additional metadata entry.

The representation of the classes is performed through a complete meta-model of Java classes. In particular, the entities `MetaClass`, `Attribute`, `MetaMethod`, `Parameter` are defined and extracted for each class under analysis. They are used to perform the pattern matching with the formal vulnerability patterns.

Bytecode analysis is performed with the ASM library as in the case of the implementation of the CBAC model. The library provides a relative lightweight support for Bytecode handling and proves to have better performances than other available libraries.

The size of the libraries for the WCA implementations together with their dependencies is 437 KBytes for the `bundlevulchecker.xml` version and 373 KBytes for the `bundlevulchecker.hardcoded` version. The difference lies essentially in the absence of XML handling libraries in the second one. The ASM library itself is 319 KBytes big. This let assume that many classes that are not used by WCA could be pruned for instance to provided a lightweight implementation for embedded platforms.

Performances The two versions of WCA pursue different goals: `bundlevulchecker.xml` is a development tool, and `bundlevulchecker.hardcoded` is intended to be executed immediately before the component is installed. Consequently, the performance aspect is less critical for the first version, and is more important for the second one. The performances are extracted for three configurations: `bundlevulchecker.hardcoded` in `abort` mode, *i.e.* the verification aborts as far a vulnerability is found, `bundlevulchecker.hardcoded` in `warning` mode, *i.e.* a vulnerability report is printed when the whole component is checked, and `bundlevulchecker.xml` in `warning` mode. All tests are performed with the `check` option, which enables to analyze the method code. Code analysis is necessary for a couple of vulnerabilities, and implies a non-negligible overhead. Tests results are therefore worse case tests.

Figure 8.7 shows the performance of a WCA check for sample vulnerable bundles, according to the number of public classes. Public classes are classes that pertain to the 'shared classes' or 'shared object' category. The samples classes are test classes that each contain one specific vulnerability from the 'Vulnerable Bundle' catalog.

The overhead implied by the reading of XML files is 20 ms. It is observed when 0 public classes exist in the component, and therefore when no code analysis is performed. For components with few public classes, the analysis overhead is between 10 and 20 ms for `bundlevulchecker.hardcoded`, and between 20 and 40 ms for `bundlevulchecker.xml`. When the number of public classes increase, the overhead grows in an important manner, as shows the last sample. One can note that the analysis duration is not proportional to the number of public classes. This is due to the fact that the analysis depends on many factors: type of public class, 'shared class' or 'shared object', internal complexity and size of the class.

Figure 8.8 shows the performance of a WCA check for actual bundles, namely the bundles from the Apache Felix trunk code repository, according to the number of public classes.

The analysis for an arbitrary set of real OSGi bundles confirms the observation from Figure 8.7: the duration generally grows as the number of public classes do, but is not proportional to it. Most analyses do not exceed 200 ms. However, in several cases, analysis takes more that 400 ms, and more that 1 second for one example. These important values are directly correlated with the size of the component under analysis. Moreover, for large

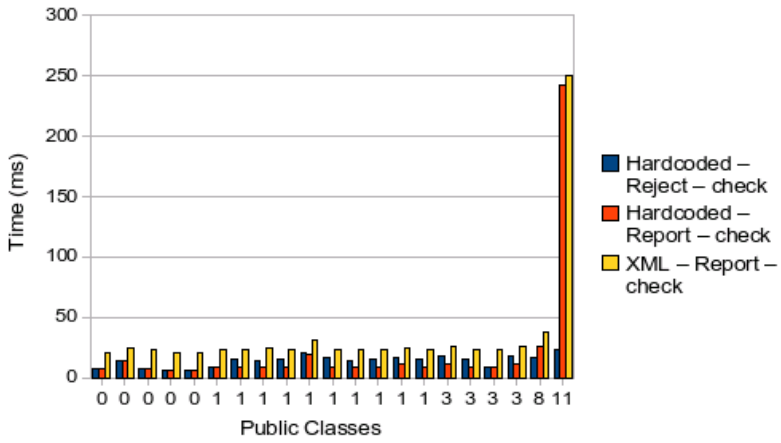


Figure 8.7: Performances of WCA check for sample vulnerable bundles

analyses, the performance of `bundlevulchecker.hardcoded` is worse than the performance of `bundlevulchecker.xml`. Consequently, it still provides a benefit in term of used disks space, but have a negative impact on the system reactivity.

Figure 8.9 shows the performance of a WCA check for actual bundles, namely the bundles from the Apache Felix trunk code repository, according to the component size, which is an indirect indicator of the total number of classes in a component and of its complexity (number of methods, attributes, size of the methods that are analyzed).

A stronger correlation between component size and analysis duration can be observed. However, there is still no proportionality. Though in most case the WCA verification requires only a short time, it can not be considered as negligible, and may even imply a long delay. It should be used carefully as an install time mechanism, since it is likely to have an impact on the performances of the whole system, in particular in term of CPU availability.

Advanced Features WCA makes possible to install only components that are known to be free of common vulnerabilities. However, its performance impact can be important, since the whole code of public classes and internal classes must be analyzed. Moreover, automated analysis through the formal vulnerability pattern only partially covers the set of known vulnerabilities.

We therefore propose to define a WCA certificate to enable the costly analysis to be performed during the development. The WCA certificate is made of two sections: the `Manual-Review` section, which enable reviewers to express the vulnerabilities they have looked after, and the `Automated-Review` which contains the list of vulnerabilities that are not present in the code, according to the WCA tool. Such certificate require that the component signer is liable for the validity of these assertions.

Figure 8.7 shows an example of a WCA certificate, for a component that is checked for class-level vulnerabilities only.

Vulnerabilities in this class are numbered from 1 to 14. Vulnerabilities in objects are not verified: it is therefore likely that some of them exist in this component. The name of the signer is repeated for information only. It is actually redundant with the information that is contained in the X.509 certificate of the signer.

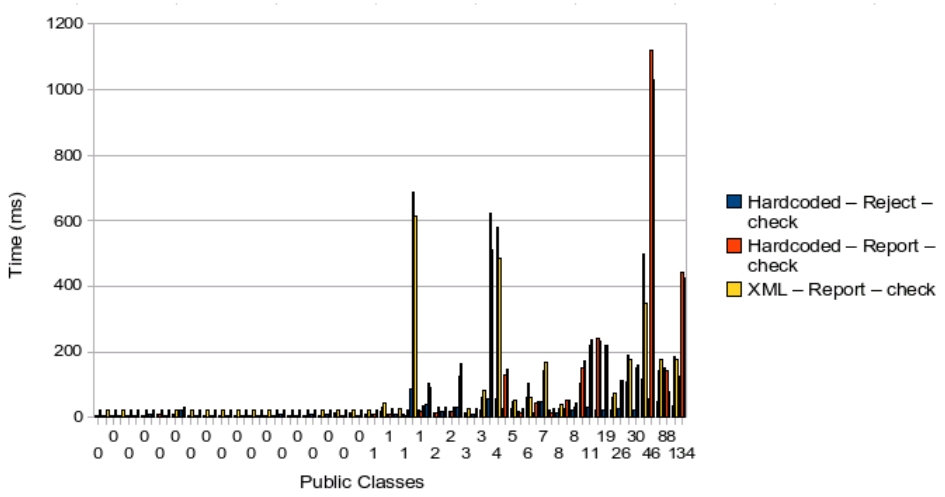


Figure 8.8: Performances of WCA check for Felix trunk bundles

```
Signer-Name: CN=Alice, OU=Wonderland Unit, O=Wonderland, L=Nowhere, ST
  =Rhone-Alpes, C=FR
Manual-Review: vb.java.class.4, vb.java.class.6, vb.java.class.7, vb.j
  ava.class.8, vb.java.class.9, vb.java.class.10, vb.java.class.14
Automated-Review: vb.java.class.1, vb.java.class.2, vb.java.class.3, v
  b.java.class.5, vb.java.class.11, vb.java.class.12, vb.java.class.13
```

Table 8.7: Example of a WCA Certificate

8.4.4 Results of Security Benchmarking

The validation of the proposed security mechanism is performed through security benchmarking, to check whether the requirements are met.

Prevented Attacks The WCA security mechanism prevents the exploitation of the following vulnerabilities. The reference of vulnerabilities according to their definition in the *Vulnerable Bundle* catalog in Appendix B.2 are given.

- 'Stores Mutable Object in static Variable' (vb.java.class.1).
- 'Stores Array in Static Variable' (vb.java.class.2).
- 'Non Final Static Variable' (vb.java.class.3).
- 'Private nested Classes and Attributes made protected' (vb.java.class.5).
- 'Finalize Method' (vb.java.class.13).
- 'Synchronized Method' (vb.java.class.15).
- 'Synchronized Code' (vb.java.class.16).
- 'Returns Reference to Mutable Object' (vb.java.object.1).
- 'Returns Reference to Array' (vb.java.object.2).
- 'Non Final non Private Field]' (vb.java.object.4).
- 'Non Final Parameters - Malicious Implementation' (vb.java.object.15).

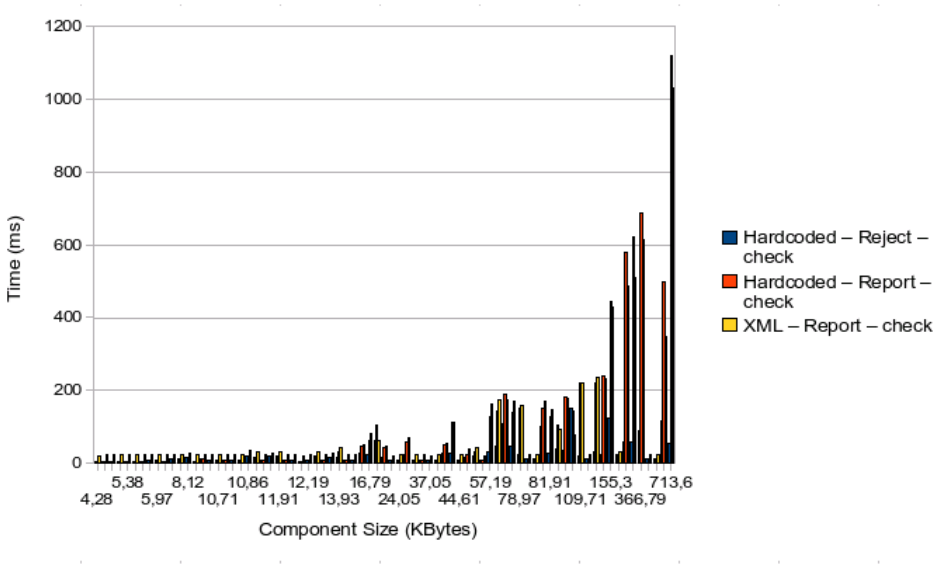


Figure 8.9: Performances of WCA check for Felix trunk bundles, according to bundle size

- 'Non Final Parameters - Inversion of Control' (vb.java.object.16).

A recapitulative table for all vulnerabilities and the suitable protection mechanisms can be found in Table 6 from the related technical report [PF08c].

Protection Rate for WCA The Protection Rates for the mechanisms enabling to prevent vulnerabilities in OSGi bundles are given in Table 8.8. These mechanisms are WCA of course, CBAC, the Java security manager. Since all vulnerabilities from the *Vulnerable Bundle* catalog can not be identified in an automated manner, manual review is also considered. However, since it relies on human intervention, it can not be considered as a guarantee of absence of the vulnerabilities of interest.

CBAC protects from 2 vulnerabilities. Up to 3 more could be prevented if CBAC would support the definition of method 'short names', `readObject()`), as sensitive methods, which is not the case in the current implementation.

Limitations The proposed WCA tool is an efficient step towards the development of secure Java components. In particular, it is to the best of our knowledge the only one that takes the exposition of classes and the actual set of exploitable vulnerabilities into account. However, it has some restrictions, and does not prevent all vulnerabilities.

The limitations of the current WCA model are the following ones. First, the use of static analysis together with pattern matching enables to identify only a subset of the known vulnerabilities. A more complete definition of the formal vulnerability pattern is required. In particular, it should support the expression of state machines to analyse method Bytecode, control flow graphs (CFG) and data flow graphs (DFG). These artifacts are used for instance by FindBugs, but are hardcoded. They are consequently hard to validate and to reuse. The expression of complex vulnerabilities can be performed for the source code with XPath, such as in PMD. However, the design of the JVM stack does not enable to reuse this approach

Security Mechanism	**# of Protected Flaws**	**# of Known Flaws**	**Protection Rate**
CBAC	2	33	6 %
WCA	12	33	36 %
Java Permissions (Perms)	3	33	9 %
Manual Review	33*	33	100* %
WCA + Manual Review	33*	33	100* %
*Security level achieved through manual review can not be considered as being an actual guarantee: human auditors can find any type of bugs they know, but are not likely to find them all.			

Table 8.8: Protection rate for the WCA and complementary protection mechanisms [The evaluation is performed with the *Vulnerable Bundle* catalog as reference, *i.e.* only component vulnerabilities in the context of the OSGi platform are considered.]

directly for Bytecode. Next, standard vulnerability patterns such as those defined for FindBugs or JSLint, should be supported in addition to our *Vulnerable Bundle* catalog. This would ensure that known vulnerabilities are all taken into account, and would help improve the overall quality of code validated through WA. Lastly, the abstract definition of vulnerabilities implies that they are specific to object oriented languages, but not limited to Java. The model could therefore be extended to support other languages such as C#, C++, Delphi. Of course, language specific features should then be taken into account.

The limitations of the current WCA implementation are the following ones. First, it is specific to the OSGi platform. Other Java SOP component types such as Spring component, or EJB, could also be supported with a minimal refactoring work. Next, WCA validation implies in certain cases an important validation overhead. This could be limited by the implementation of the proposed WCA certificate scheme to enable costly verification to be performed by trusted issuers during the development. The last implementation improvement that is required is the development of plug-ins for integrating WCA in common development environments such as Eclipse or Maven.

Conclusion So as to solve the identified vulnerabilities in the Java/OSGi platform and related components, we propose a set of protection mechanisms: the Hardened OSGi recommendations, to patch the features of the OSGi platforms that can be abused, CBAC, Component Based Access Control, to support a flexible access control model and overcome the limitations of the Java security manager, and WCA, Weak Component Analysis, to identify the exploitable vulnerabilities in the code of SOP components.

An integrated secure Java/OSGi Execution Environment

9

The realization of a secure execution environment* require that the different protection mechanisms are integrated together, that vulnerabilities* that were not addressed in this thesis are prevented, and that the components* themselves are coded according to the identified constraints.

Security benchmarking* is performed to identify the relationship of the proposed mechanisms with existing Java protection mechanisms and to assess their efficiency.

9.1 Hardened OSGi over a secure Java Virtual Machine

Software security* mechanisms provide a great improvement in the security level* of the Java/OSGi platform*. However, all vulnerabilities can not be protected at the level of the SOP platform and in the component code. In particular, resource isolation is a system* issue. It builds a different area of research and could not be addressed in the time frame of this thesis.

9.1.1 Resource Isolation for OSGi Platforms

Selection of a secure JVM The architecture of OSGi applications, where each bundle* is started in a dedicated class loader, provides a sound isolation based on the namespaces. It prevents uncontrolled interactions between the bundles. However, shared system data such as system static variables (*e.g.* `System.out`) or resources such as CPU and memory can not be protected.

A secure JVM for executing OSGi applications should therefore provide following features:

- It should isolate bundles from each other in term of memory and CPU consumption.
- It should support the execution of legacy OSGi applications, inclusive the communication between bundles.

The Sun MVM is not suitable for this tasks, since its isolates can communicate through system IPC only, and since each process is defined by a `main` method. Consequently, bundle communication can not be supported, and the MVM can not deal with bundle activators. The KaffeOS VM requires that the processes are defined as kernel or user processes. It can neither deal with bundle activators.

The JnJVM [TGCF08] is a recent prototype implementing isolation of resources and static variables based on class loaders. It is explicitly designed to be a secure execution environment for the OSGi platform. As such, it supports the execution and the transparent management of bundles. The main limitation of the JnJVM for executing legacy OSGi bundles is that static variables are no longer shared at the system level, to prevent in particular the modification of system variables. As the SOP paradigm intends to make the components communicate through service* exclusively, it is possible to consider that no communication through static variable should occur, and therefore that this limitation only enforces a programming best practice. The JnJVM is currently under development at the Regal team of the LIP6 Laboratory, Paris[1]. It proposed to adapt the concept of Isolates, defined by the JSR 121, to support isolation of system resources between the OSGi bundles [GTCF08]. Each isolate is executed in a **domain**.

Domains are an hybrid of standard Java class loaders (see Section 4.1.3 page 42) and Isolates from JSR 121 (see Section 4.3.1 page 50) which enables OSGi applications to run unmodified in a protected way. They are so far only available for the JnJVM Virtual Machine.

The principles of security* domains are the following:

- Each bundle has its private environment.
- Class loading properties are preserved, in particular the possibility of user-defined class loader.
- Direct method invocation between bundles is preserved, contrary to Java Isolates that only support calls through IPCs.

Their implementation combines lightweight isolation and resource accounting. Lightweight isolation means that domains share the system classes, have private access to the class they load (as with standard class loaders). They have their own system static variable, `java.lang.Class` instances, and interned Strings, as Java Isolates do. Consequently, classes are either loaded by a domain, which is the case for all the classes of a given bundle, or shared as common facilities, *e.g.* system classes, system libraries and the classpath. Resource accounting encompasses CPU and memory consumption. Memory consumption is checked at allocation opcodes in the applications and at allocations that are performed by the runtime.

The JnJVM The JnJVM is a full-fledged implementation of JVM Specifications [LY99]. It is actually a set of scripts that can generate adaptative virtual machines in a flexible manner [OTF05, TGCF08].

As such, it can be considered as a virtualized virtual machine, *i.e.* a virtual machine that enable the dynamic adaptation of the behavior of the VM itself to enhance performances

[1] http://vvm.lip6.fr/projects_realizations/jnjvm/

[Bac04]. This dynamism could be extended to the instruction set at the hardware level and to the scheduler. The first implementation of the JnJVM was based on the Virtual Virtual Machine Project[2] which shares common goals with [Bac04], and on the Low Level Virtual Machine (LLVM) project [LA04].

An adaptative virtual machine is a VM that can be modified at runtime, to adapt to the system context (*e.g.* available resources) or to specific applications (*e.g.* by extending the VM when required). Flexible generation of the VM means that it can be set up according to the foreseen execution context. Modules such as Garbage Collector or Just-in-Time (JIT) compiler can be replaced or customized. The JnJVM is a research prototype with performances that are comparable with the Kaffe Virtual Machine. `JnJVM` is a recursive acronym standing for JnJVM is not a virtual machine.

9.1.2 Implementation

No modification to the OSGi platform is required: the domains of the JnJVM are designed to support the execution of unmodified OSGi bundles. Hardened Felix is already integrated with complementary protection mechanisms such as CBAC and WCA. Is is therefore sufficient to start the secure implementation of the OSGi platform atop the JnJVM.

Figure 9.1 shows the architecture of a secure Java/OSGi implementation: Hardened OSGi with CBAC over the JnJVM Virtual Machine.

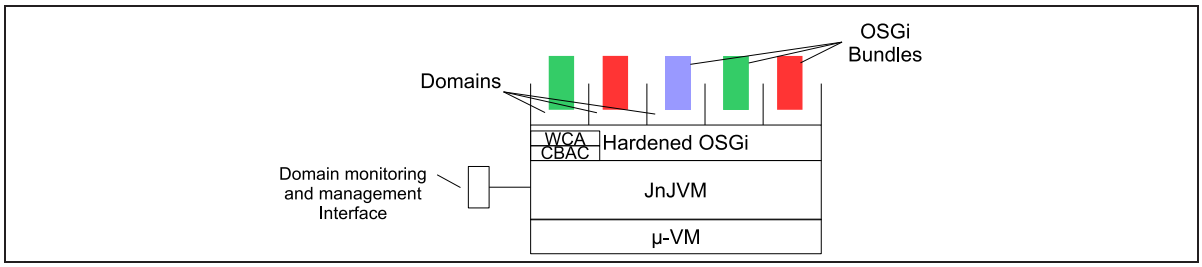


Figure 9.1: Hardened OSGi with CBAC over the JnJVM Virtual Machine

The application layer is the following. The Micro Virtual Machine [OTF05] is a configurable virtual machine that provides a minimal basis for execution of Java applications. It can be customized through a Lisp-like language to replace core features such as the Just-In-Time (JIT) compiler or Garbage Collector or to extend it. It is written in C++. The Micro Virtual Machine runs VMLets which are executable configuration file. One such VMLet is the JnJVM which can be enriched with the 'domain' security model presented hereafter. Our secure OSGi platform is run atop the JnJVM.

9.1.3 Results of Security Benchmarking

Prevented Attacks* The JnJVM with security domains prevents the exploitation of the following vulnerabilities. The references of vulnerabilities according to their definition in the *Malicious Bundle* catalog in Appendix B.1, and in the *Vulnerable Bundle* catalog in Appendix B.2, are given.

[2] http://vvm.lip6.fr/

Security Mechanism	# of Protected Flaws	# of Known Flaws	Protection Rate
JnJVM	14	32	44 %
HO + JnJVM	18	32	56 %
Perms + JnJVM	22	32	69 %
CBAC + JNJVM	22	32	69 %
HO + Perms + JnJVM	30	32	94 %
HO + CBAC + JnJVM	30	32	94 %

Table 9.1: Protection rate for the OSGi platform over JnJVM for platform vulnerabilities [The evaluation is performed with the *Malicious Bundle* catalog as reference, *i.e.* only platform vulnerabilities are considered. The results are given for the Hardened Felix over JnJVM configuration.]

- 'CPU load injection' (mb.native.1) which is prevented by monitoring CPU times of each service and setting suitable limits.
- 'Stand Alone Infinite Loop' (mb.java.11).
- 'Infinite Loop in Method Call' (mb.java.12).
- 'Memory Load Injection' (mb.java.10) which is prevented by dedicating a memory allocator to each service so that the system knows how much memory a service consumes.
- 'Exponential Object Creation' (mb.java.13), with a memory allocator that is tuned accordingly.
- 'Cycle between service' (mb.osgi.9), in a similar way to the previous item.
- `System.exit()` (mb.java.1) and `Runtime.halt(mb.java.2)`.
- 'Recursive Thread Creation' (mb.java.3) which is prevented by monitoring the number of new threads.
- 'Hanging Thread' (mb.java.4) and 'Sleeping Bundle' (mb.java.5).
- 'Synchronized Method' (vb.java.15) and 'Synchronized Block' (vb.java.16).

It prevents in particular the exploitation of vulnerabilities bound with `Thread` management and resource consumption.

Protection Rate for the integrated system Table 9.1 show the security benchmarking results for the OSGi platform over JnJVM for the vulnerabilities of the OSGi platform.

The JnJVM alone accounts for a protection rate of 44 %.

When combined either with a Java security manager or Component-based Access Control (CBAC), it provides a protection rate of 69 %. The interesting fact here is that both combination protect from the same set of vulnerabilities, although the Java security manager and CBAC do not protect from the same vulnerabilities. This is due to their redundancy with the JnJVM isolation mechanism. Consequently, designers can choose to use Java permissions or CBAC according to their precise requirements. If performance is privileged, CBAC will be the correct match. If the user is allowed to grant new rights to the applications to enable the execution of some sensitive operation or if sensitive methods are optional or rarely used, the security manager is a better choice. In both case, the JnJVM introduce a finer management

Security Mechanism	**# of Protected Flaws**	**# of Known Flaws**	**Protection Rate**
Hardened OSGi (HO) + Perms	24	65	37 %
JnJVM	14	65	22 %
HO + Perms + JnJVM + WCA	44	65	68 %
HO + CBAC + JnJVM + WCA	44	65	68 %
HO + CBAC + JnJVM + WCA + Manual Review	65*	65	100* %
*Security level achieved through manual review can not be considered as being an actual guarantee: human auditors can find any type of bugs they know, but are not likely to find them all.			

Table 9.2: Protection rate for Hardened OSGi with CBAC and WCA over JnJVM for platform and component vulnerabilities
[The evaluation is performed with the *Malicious Bundle* and the *Vulnerable Bundle* catalogs as reference. The results are given for the Hardened Felix over JnJVM configuration.]

for sensitive operations such as thread handling: it can control the number of threads to avoid excessive use and does not need to completely prevent them. Moreover it prevents platform shutdown and crashes which is an important improvement for the system stability.

Together with Hardened OSGi (HO), these solutions provide a protection rate of 94 %. If errors in the manifest settings are not considered as vulnerabilities, the protection rate can be considered to amount to $30/31 = 97$ %.

Table 9.2 show the security benchmarking results for the OSGi platform over JnJVM for the vulnerabilities of the OSGi platform and components.

An important effort is still required to realize an Hardened OSGi platform with robust bundles. In particular, bundle vulnerabilities are only partially prevented so far. Moreover, management tools are still lacking to control individual security mechanisms.

The integration of the JnJVM with our protection mechanisms nonetheless provides an important benefit in term of achieved security level. It supports the power of isolates while keeping a programming model fully compliant with the OSGi specifications.

9.2 Developing secure Java/OSGi Bundles

Developing secure Java/OSGi bundles implies to comply with the constraints identified so as to pass successfully the CBAC and WCA validation processes. The validation of bundle code through static analysis is performed in two of the bundle life-cycle phases: at the end of development, and at installation time.

9.2.1 Identified Constraints

Dangerous method calls are the calls that are protected through Java permissions, OSGi-specific permissions, and the additional calls that we identify as sensitive in the context of SOP component platforms*. They should be used in a manner as limited as possible, by trusted component providers only.

They are the following ones:

- Security related actions,
- File management actions,
- Some Graphical User Interface (GUI) management actions,
- Reflection,
- Access to the `Runtime` and `System` classes,
- Thread management,
- Network related action,
- Database access actions,
- Bean management actions,
- Class loading actions,
- OSGi management actions.

Based on the presented experiments and classifications* of vulnerabilities in Java/OSGi component interactions, following recommendations can be emitted to component developers. Security constraints should be enforced at two level: the component level, *i.e.* the application architecture, and the public code level, *i.e.* the code that components make available to others.

Components should:

- only have dependencies on components they trust,
- never used synchronized statements that rely on third party code,
- provide a hardened public code implementation following given recommendations.

Shared Classes should:

- provide only final static non-mutable fields,
- set security manager calls during creation in all required places, at the beginning of the method: all constructors, `clone()` method if the class is cloneable, `readObject(ObjectInputStream)` if serializable,
- have security checks in final methods only,

Shared Objects (*e.g.* SOP Services) should:

- only have basic types and serializable final types as parameter,
- perform copy and validation of parameters before using them,
- perform data copy before returning a given object in a method. This object should also be either a basic type or serializable,
- not use Exception that carry any configuration information, and not serialize data unless a specific security mechanism is available,
- never execute sensitive operations on behalf of other components.

The actual implementation of these recommendations should be compliant with the security requirements of the system where the components are deployed, and therefore with the

existing security policy. For instance, some components can be allowed to access the network, or to contain `synchronized` code if their implementation and dependencies are provided by trustworthy code issuers.

9.2.2 Security in the Software Development Life-Cycle

Software security assurance* can be enforced throughout the development process of Java components, in particular OSGi bundles, through the proposed code validation tools: the CBAC engine and the WA tool. The secure development life-cycle of components is not to be confused with $\mathcal{SPIP}$, which is a process for building secure execution platforms. Moreover, the recommendations we propose here do not take into account the integration of components, *i.e.* the properties of the resulting software architecture. It focuses on the properties of the component code.

As stated in the original claim of this thesis, in Section 1.1 page 3, the objective of developing secure component is to enforce security as early as possible in the life-cycle while taking the constraints of extensible SOP platforms into account. Consequently, two development options are to be considered:

1. Target system and related security policies are known: the security policies are to be enforced in the component code.
2. Off-the-Shelf components*: the target system and related security policies are not known. The development should enforce maximal protection: vulnerabilities that are 'flaws'* are to be prevented; vulnerabilities that are sensitive 'functions'* are to be used only if required. This is the case of sensitive functions defined in the CBAC model, and of `synchronized` code in the WCA mechanism.

CBAC policies enforce access control requirements. WCA policies enforce code quality requirements.

Figure 9.2 shows the publication process for Hardened OSGi Applications (development of constraint compliant code, validation of the Access Control policies, deployment, component verification according to constraints and policies, installation).

The secure development life-cycle SDLC for components is the following one:

- Developer training.
- Code development.
- Code analysis and correction (CBAC, WCA). This step should be simultaneous with the previous one.
- Code manual review and WCA certification.
- Archive signature. This step should be done in an atomic process integrating previous one to prevent code modification after the validation.
- Archive publication.
- Archive discovery and download.
- Automated verification (CBAC, WCA).
- Installation, execution.

This SDLC considers only protection mechanisms studied in this thesis, CBAC and WCA. It does not intend to define a generic framework. In particular, it is generally recommended to run all available tools for vulnerability identification through static analysis, since they all covert non fully overlapping sets of vulnerabilities.

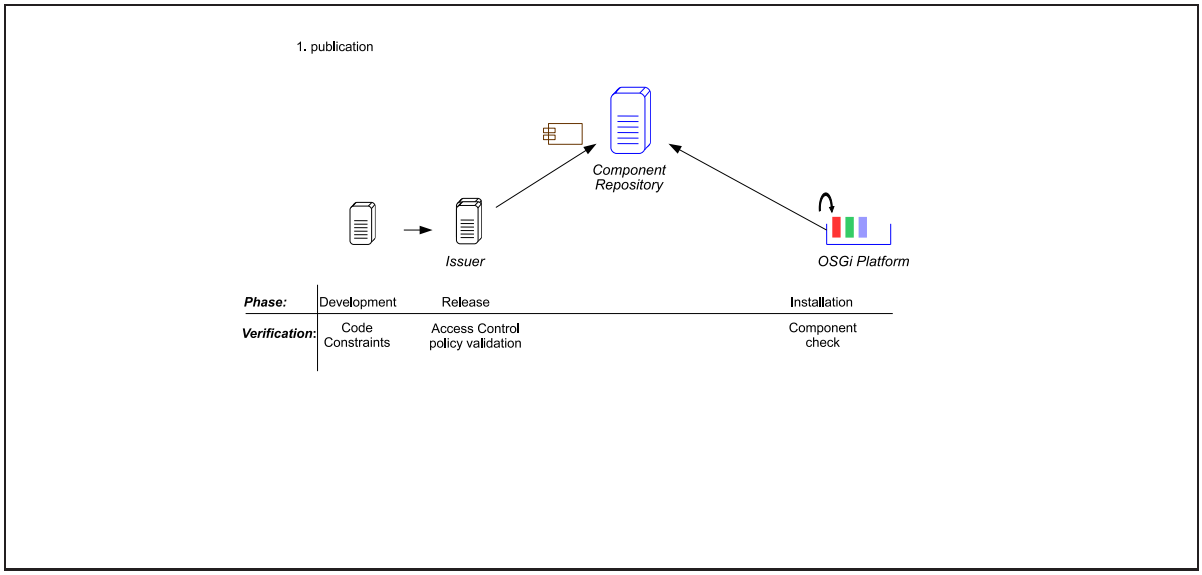


Figure 9.2: The publication process for Hardened OSGi applications

9.3 Security Benchmarking of the integrated System

The final step of the $\mathcal{SPIP}$ process is the realization of the security benchmarking of the integrated system, with all identified and newly defined security mechanisms. The types of the defined mechanisms are compared with existing mechanisms for Java components. The protection they provide are documented.

9.3.1 Protection Mechanisms for Java Systems

Figure 9.3 presents existing and proposed mechanisms for enforcing security in Java systems. It complements the figure 4.4 page 55 with our own propositions.

Hardened OSGi is a Platform-level protection mechanism which is transparent for the developed component. CBAC and WCA implies that the code is compliant with the policy constraints. Both mechanisms are enforced at install time.

9.3.2 Protection against Vulnerabilities

The security improvement brought in by our propositions is now summarized. First, a qualitative overview of the contribution of each tool to the realization of a secure OSGi-based environment is provided. Next, an overview of the the quantitative benchmarks presented throughout this document is given.

The proposed protection mechanisms intend to prevent the vulnerabilities we identified. These vulnerabilities can be of two kinds: flaws, which have to be patched, and dangerous functions for which a suitable access control mechanism is required.

Figure 9.4 shows an overview of the type of vulnerabilities that are addressed by the proposed protection mechanisms.

The vulnerabilities are found in two main locations: the platform and the components. The access control scheme is part of the platform, since it is enforced by it. Platform vulnerabilities are related to the access control model, to the implementation of the SOP platform, and to

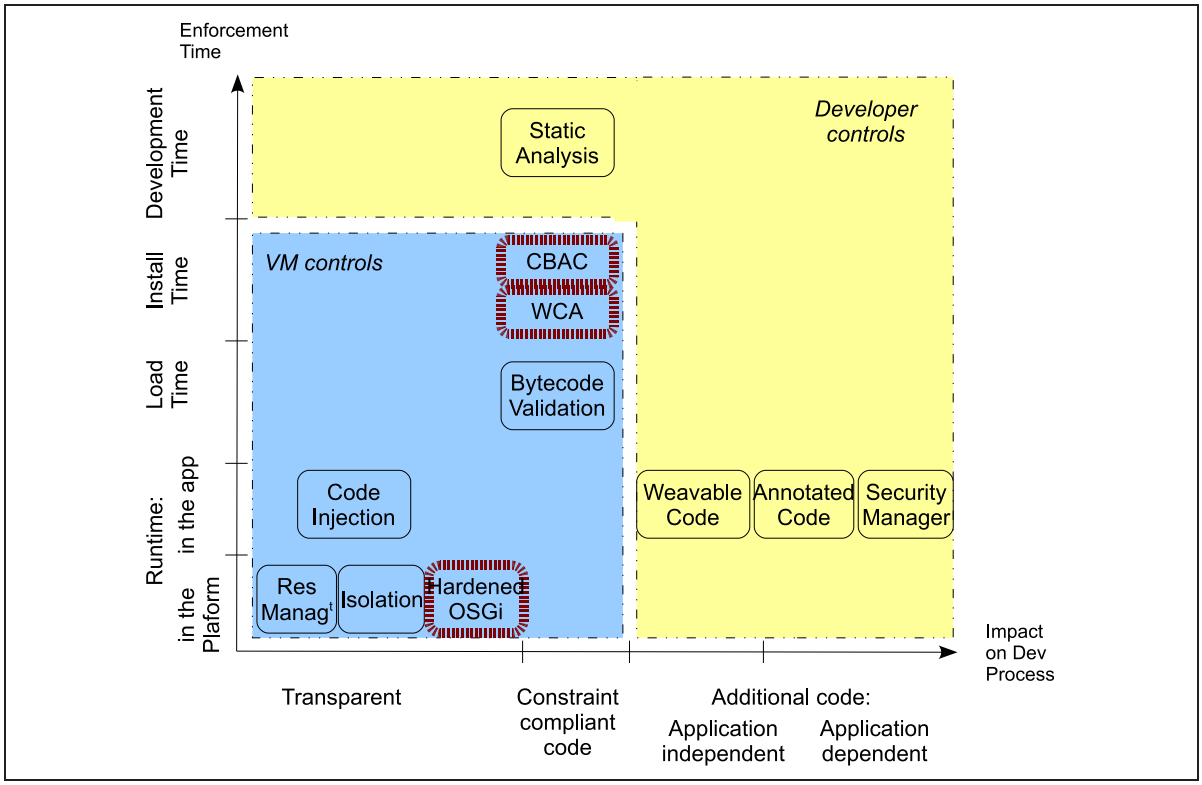


Figure 9.3: Existing and proposed protection mechanisms for Java systems

the JVM. The related solutions are the CBAC model, the Hardened OSGi recommendations which are our own proposition, and the JnJVM with is an external proposition respectively. Component vulnerabilities are located in the internal classes, in the shared classes or in the shared object. The WCA tool, and CBAC marginally, enable to tackle an important number of them. However, manual review is still required to identify complex vulnerabilities which discovery can not be automated easily.

These protection mechanisms should be extended with new vulnerabilities as their are discovered and documented.

The global results of the performed security benchmarking are now expressed for the three types of OSGi platforms: the specification compliant default OSGi platform with a standard JVM, the Hardened OSGi platform with a standard JVM, and the Hardened OSGi platform executed on the top of a JnJVM. For each platform, the protection provided by our proposition is quantified.

Following figures are implementations of the generic figure 6.2 page 83.

Figure 9.5 shows the security benchmark for a default OSGi platform with each of our Software Security propositions.

The default OSGi platform does not provide any protection. CBAC, WCA for shared classes and WCA for shared objects enable to achieve a protection rate of 50 %. Manual code review is required for ensuring a higher security level, but does not provide any formal guarantee.

Figure 9.6 shows the security benchmark for Hardened OSGi with each of our Software Security propositions.

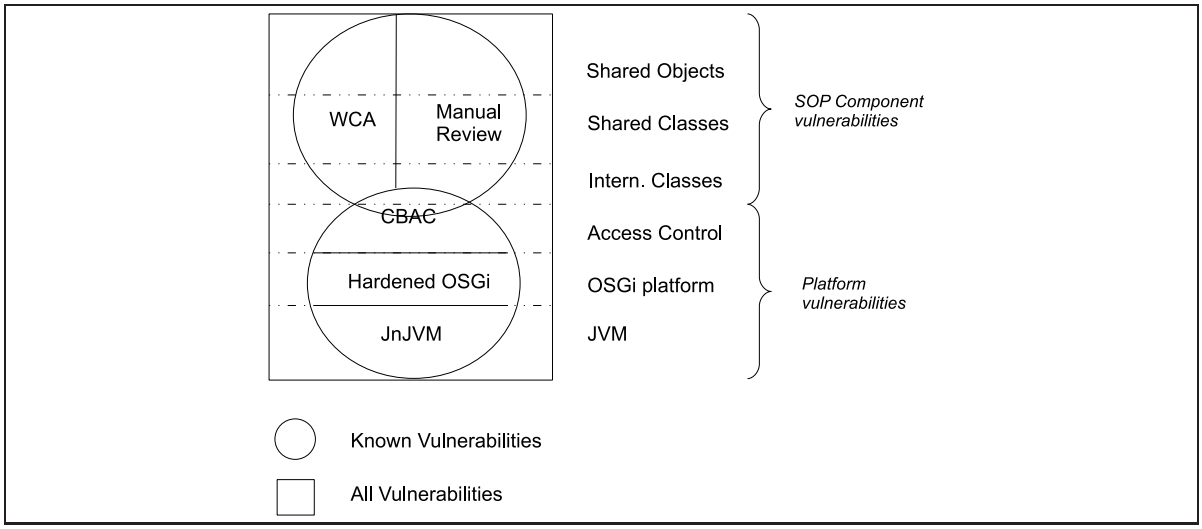


Figure 9.4: The type of vulnerabilities in a SOP platform

The Hardened OSGi platform provides *per se* a protection rate of slightly more that 10 %. CBAC, WCA for shared classes and WCA for shared objects enable to achieve a protection rate of 60 %.

The vulnerabilities that are not covered are:

- Decompression bomb,
- Erroneous value of manifest attribute (which leads to denial-of-service of the faulty component, but can not be exploited by malicious component to harm other entities of the system),
- Exponential object creation,
- Cycle between services,
- The complex component vulnerabilities identified in the *Vulnerable Bundle* catalog.

The heterogeneity of these vulnerabilities imply that a specific security mechanism is to be defined for each of them.

Manual code review is required for ensuring a higher security level, but does not provide any formal guarantee.

Figure 9.7 shows the security benchmark for Hardened OSGi over JnJVM with each of our Software Security propositions.

The Hardened OSGi platform executed on top of the JnJVM provides a protection rate of almost 30 %. CBAC, WCA for shared classes and WCA for shared objects enable to achieve a protection rate of almost 70 %. Manual code review is required for ensuring a very high security level, but does not provide any formal guarantee. Moreover, vulnerabilities that are neither identified by the tools nor known by the developers can not be identified.

Conclusion The protection mechanisms we propose provide a great improvement in the security level of OSGi applications. These results are encouraging, and are an incentive to develop tools with production level quality to let developers and architects build secure OSGi-based environments.

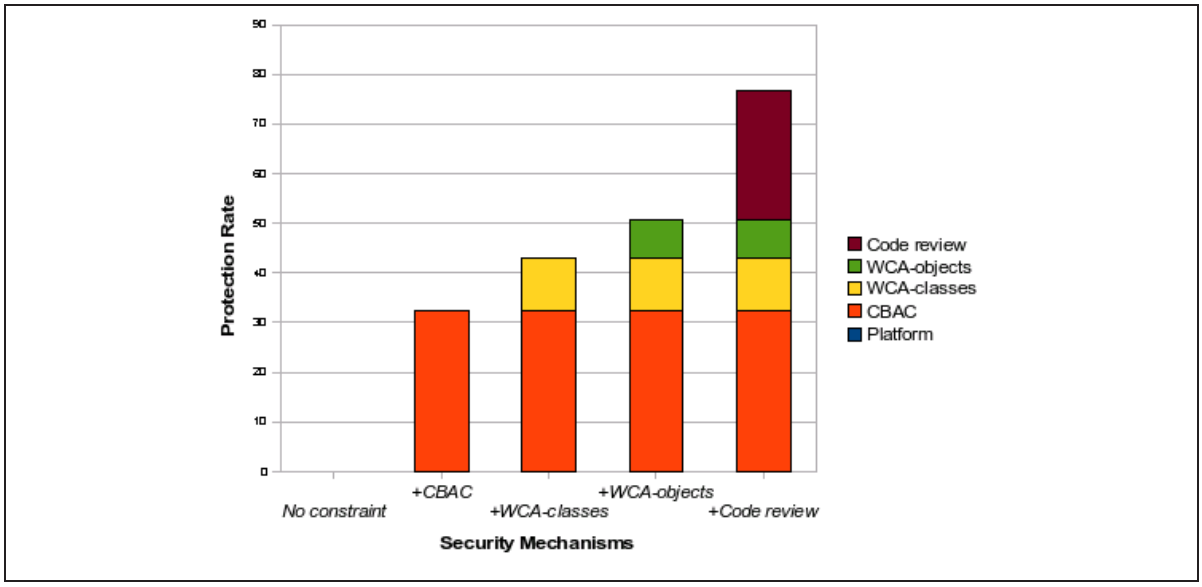


Figure 9.5: Security benchmark for a default OSGi platform

Figure 7.6 presents the taxonomy* of vulnerability categories in the Java/OSGi SOP platform, along with the related security mechanisms we developed.

Most vulnerabilities are fully protected through the set of mechanisms we presented. Some stay partially unprotected, and require further efforts to automate their prevention, in particular vulnerabilities in the component code. The simpler cases can be protected by applying state-of-the art techniques to the set of vulnerabilities we defined.It is likely that more complex cases can only be tackled with manual review, or with new static analysis methods.

We can consider that the research effort presented here provides a complete set of solutions to the identified security weaknesses, even though it requires to be deepened to provide a Java SOP platform that can be considered as sufficiently secure to be exploited in production environments.

Of course, the reference set of vulnerabilities should be extended as new ones are discovered, to enable to keep the proposed solutions up to date.

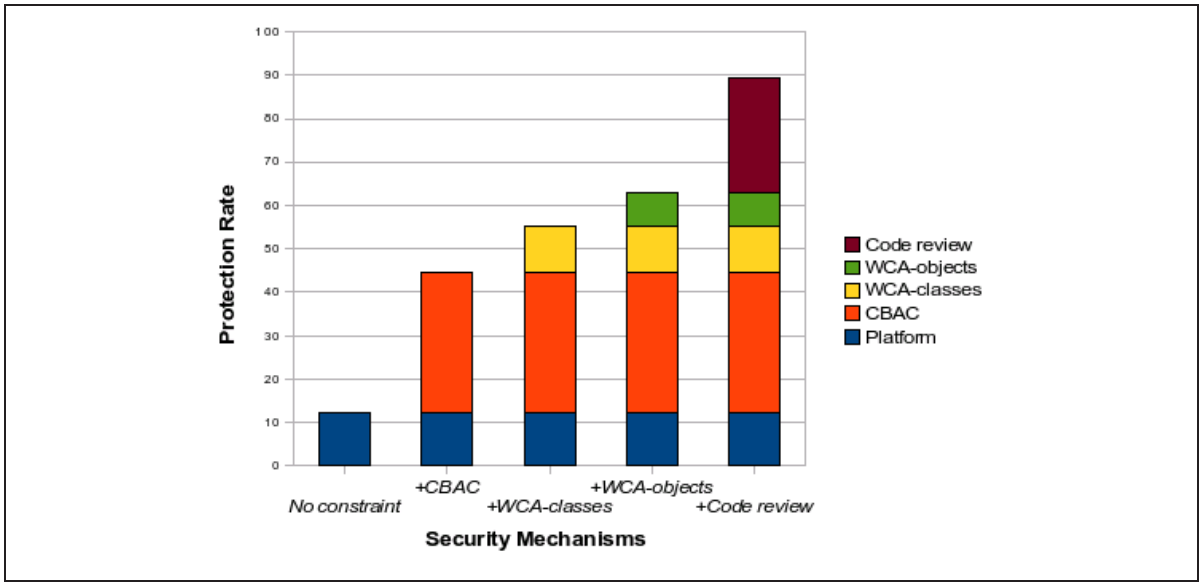


Figure 9.6: Security benchmark for Hardened OSGi

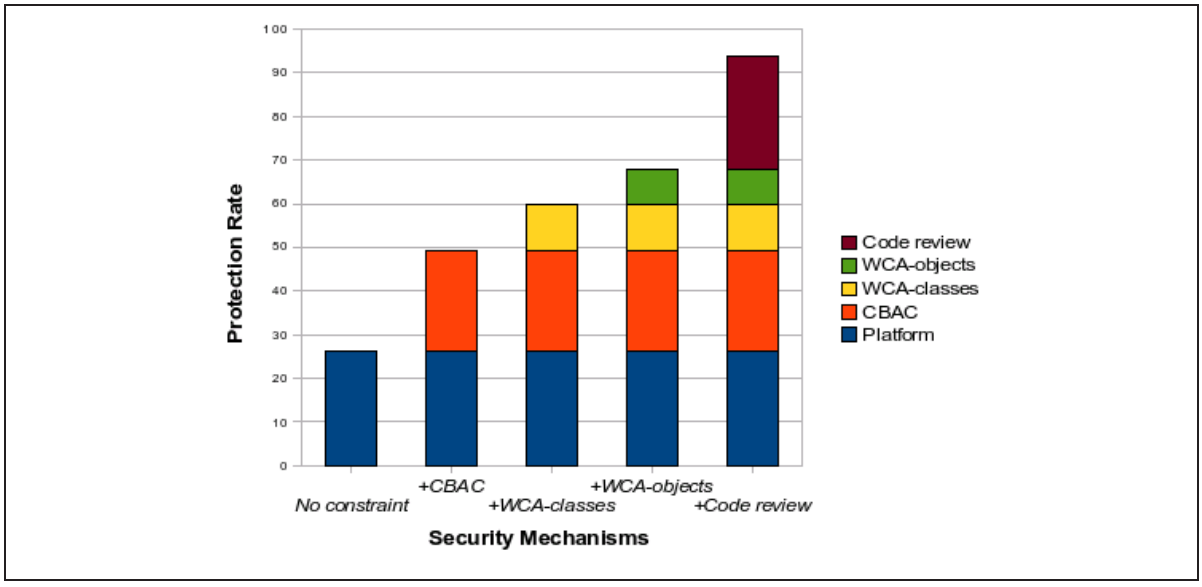


Figure 9.7: Security benchmark for Hardened OSGi over JnJVM

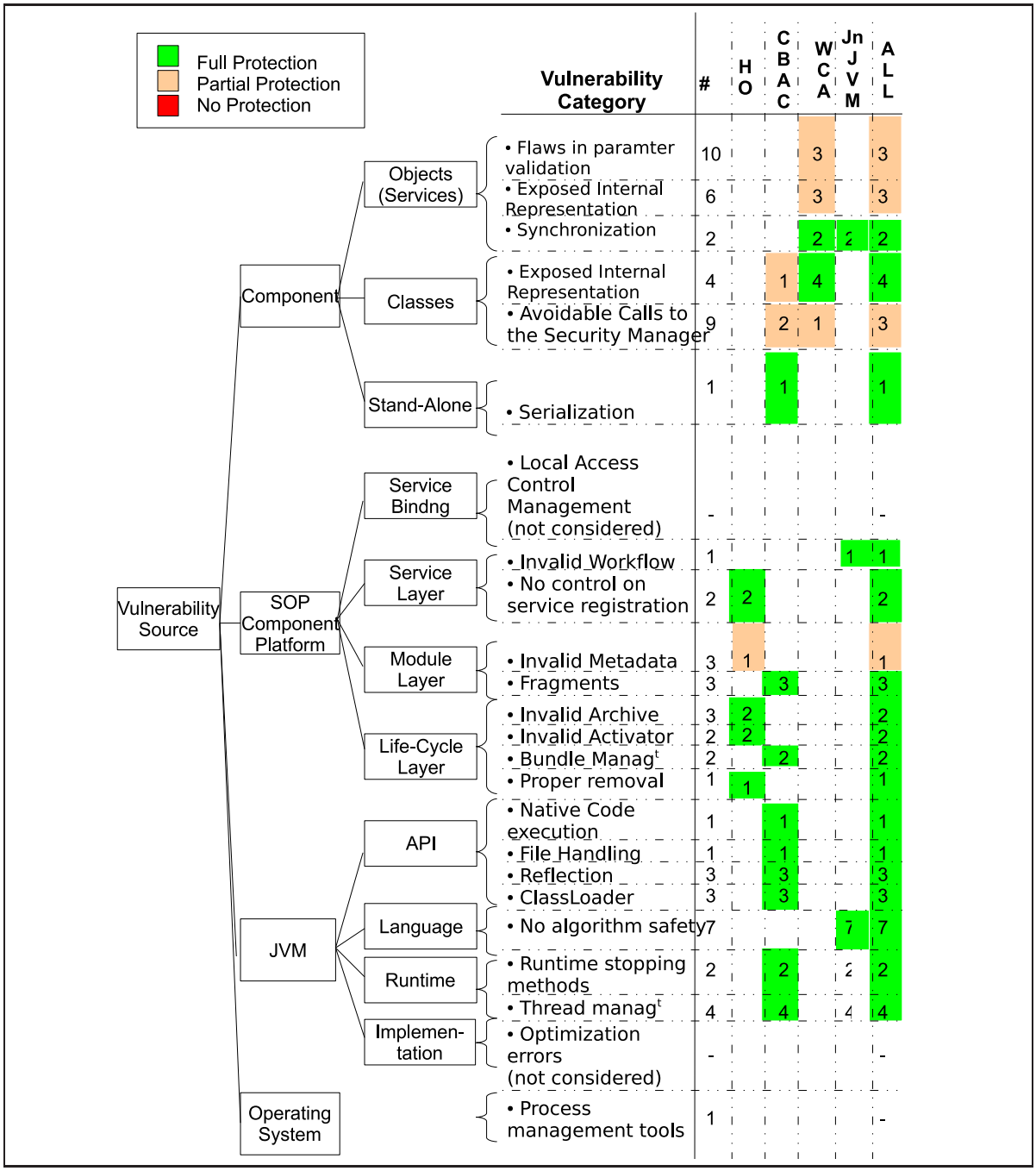


Figure 9.8: Taxonomy: vulnerability categories and related security mechanisms in the Java/OSGi platform

Part IV

Conclusions and Perspectives

10 Conclusions and Perspectives

10.1 Synthesis

This thesis presents a comprehensive security* analysis of a specific SOP platform*, the Java/ OSGi platform, as well as a set of potential solutions to the identified vulnerabilities*. It focuses on one attack vector* against the target system*: the installation of malicious components*. Two types of attacks* are considered: attacks against the platform, and attacks against other components.

The objective is to identify the challenges that need to be tackled to make the 'open dynamic application' vision a reality, and to propose first solutions: Component integration is currently performed at design time only, because very limited guarantees can be set on the code itself. The exploitation of the full potential of SOP platforms, and of the OSGi platform in particular, would require that secure integration at runtime is supported.

This section provides an overview of the presented research effort and its relationship to the state of the art in the domain of Software Security* for Java SOP platforms. Developments introduced throughout the document are also summarized.

10.1.1 Methodology for Security Analysis

The proposed methodology for security analysis, $\mathcal{SPIP}$, intends to be an original approach to the problem of building secure systems: whereas common methodologies define development processes focusing on the realization of 'one-shot' secure applications, $\mathcal{SPIP}$ focuses on the realization of secure execution environments* which can be used to build families of secure applications. The resulting execution environment, along with the related security information such as the result of security assessment* of the various security solutions, aims at providing tools for application developers and integrators. This increases the reusability of protection mechanisms, and is therefore likely to increase their stability. Automatization and generalization of the security infrastructure relieve the developers from the configuration overhead and therefore reduce security risks caused by misconfigurations. Application developers and integrators can then choose the protections that match their requirements and control the security risks they choose to address or to neglect.

For these reasons, we believe that the $\mathcal{SPIP}$ process pave the way for more secure applications. However, it still has several limitations. First, we did not yet have had the opportunity to work with application developers to experiment whether the process of selecting protection mechanisms actually match existing requirements. This is in particular due to the lack of maturity of the proposed technical solutions. Next, it lacks genericity. We did not have had the opportunity to experiment $\mathcal{SPIP}$ in the frame of other projects as the one presented here.

Analysis tools as well as metrics that are more expressive than the Protection Rate are likely to be necessary. Lastly, the $\mathcal{SPIP}$ process does not involve enough information to make the selection of suitable protection mechanisms possible. In particular, the cost involved with the use of such mechanisms, as well as the cost involved with the absence of protection for certain type of vulnerabilities, should be expressed to enable architects to take balanced choices.

$\mathcal{SPIP}$ is therefore in the state of an early research proposition. More effort is required to make its adoption on a wider scale possible.

10.1.2 A secure Execution Environment

The proposed secure execution environment, *Hardened OSGi* over the JnJVM, intends to provide a comprehensive solution for executing sensitive dynamic applications. It emphasizes the complementarity of the software security and system security approach. Software security is enforced in particular through code analysis at installation. System security is enforced by the component platform as well as by the underlying secure JVM. The component platform should be implemented according to the *Hardened OSGi* recommendations we provide. A secure JVM which must comply with the specific requirements of OSGi applications. This requirements let us select the JnJVM environment, which is designed to take advantage of the class loader structure of the OSGi platform. The JnJVM provides an important benefit for building secure Java/OSGi platforms in particular in term of resource isolation.

The integration of the two solutions, *Hardened OSGi* and a secure JVM is the demonstration that each of them addresses a part of the security problem of Java/OSGi applications. It highlights the need of a radical evolution of the security paradigm for Java environments and identifies one possible evolution: components should be isolated from each other, but still be able to communicate through local method calls. The current security manager is by far not sufficient to build secure Java SOP applications.

The limitation of this proposition is due to the fact that no real complex application involving components from mutually untrusted developers has been realized. The benchmarks were limited to the verification of the robustness of the integrated execution environment in the presence of our proof-of-concept OSGi attack bundles*. Further research is therefore required to identify the impact on the programming model, if any, and to protect the interactions between the bundles which are not protected by this security solution.

The benefit of the standard Java security manager has not been subject to enough attention to enable us to emit criticism. However, several of its features seem to be ill-suited for dynamic applications: the lack of flexibility, the risk of runtime errors, the performance overhead. It nonetheless keeps being a powerful tool for controlling access to the pre-defined sensitive methods in the JVM, and to developer-defined sensitive methods *e.g.* in libraries. It avoids the exploitation of a non negligible number of dangerous functions* in SOP platforms such as the OSGi platform.

10.1.3 Secure Components

We propose to perform the control of the interaction between bundles through two complementary protection mechanisms: CBAC (Component-based Access Control) and WCA (Weak Component Analysis). They enable to identify dangerous bundles based on the sensitive action they perform, with CBAC, and vulnerable bundles that can be exploited by others, with WCA. These tools enable to assess in an automated manner the security status of compo-

nents without requiring any human intervention or even the availability of the source code. They shift the focus of the security analysis from the process, as most production tools do, to the product which is assessed directly. WCA in particular enables to identify vulnerabilities that are really exposed, supporting focused code analysis. These tools should thus enable to evaluate the security properties of a component that is discovered dynamically from the environment.

The limitation of these propositions are the lack of completeness of both solutions. The CBAC mechanism fails into identifying generic malicious bundles, especially because it does not support algorithmic safety. It should therefore be completed with techniques such as proof-carrying-code. Moreover, as an access control mechanism, it does not provide a very fine-grained access control: the methods contained by dependency bundles are taken into account even if they are never called. Refinement of the analysis through control flow graphs (CFG) or data flow graphs (DFG) are likely to provide better results. The WCA mechanism fails into identifying a comprehensive set of vulnerabilities in components. The current formal vulnerability pattern* should be completed with state machines, CFG and DFG to enhance the quality of the detection. Moreover, it relies on a catalog of known vulnerabilities rather than on an abstract definition of these vulnerabilities, which prevents the automated discovery of new weaknesses. For both tools, further research is required in order to make production tools out of the existing research prototypes.

10.1.4 Development

Figure 10.1 shows an overview of our development and scientific contributions.

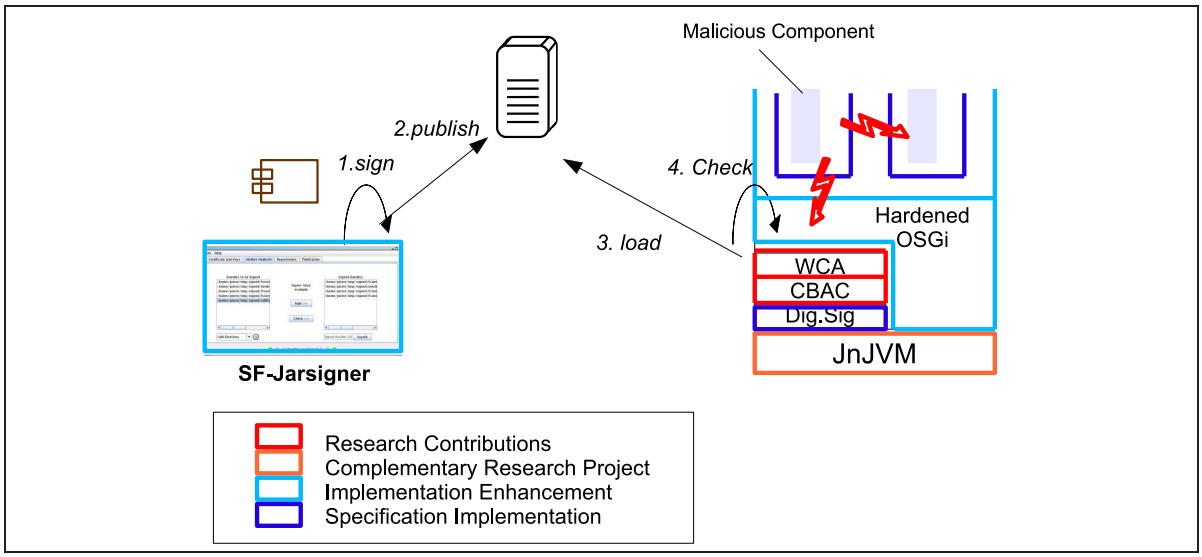


Figure 10.1: Overview of development and scientific contributions

The original research contributions in the context of SOP platforms are the CBAC and WCA models and tools, as well as the identification of platform and component vulnerabilities. They are integrated with another research contribution from the LIP6 laboratory in Paris, the JnJVM, which provides the underlying secure virtual machine. The propositions for enhancing existing specifications are *Hardened OSGi* and the SF-Jarsigner, which enables to

sign and publish OSGi bundles. Implementations that are compliant with the specifications are a library for digitally signing and verifying digital signature* (Dig. Sig) of OSGi bundles, and the integration of bundle-specific Java permissions.

Table 10.1 shows the summary of developed software along with its size and number of Lines of Code (LoC).

Projet	**Code size**	**# of LoC**
Digital Signature	30.5 KBytes	330
Java Protection Domains for Felix	-	83
SF-JarSigner	44.5 KBytes	557
Hardened OSGi	-	224
CBAC	21 KBytes	577
WCA	60.5 KBytes	2026

Table 10.1: Developed Software
[Lines of Code (LoC) are counted as Non Commented Source Statements (NCSS). The size of the Digital Signature and SF-Jarsigner tools include pre-set configuration such as cryptographic keys. Size for the implementations of Java protection domains and Hardened OSGi is not given, since the code is integrated with existing OSGi implementations such as Apache Felix.]

The number of bundles which have been developed as proof-of-concept for the *Malicious Bundle* and the *Vulnerable Bundle* catalogs is 155. This number encompasses the various implementations of the malicious bundles as well as the couples of malicious/vulnerable bundles for the second catalog. Extracting the total number of Lines of Code (LoC) for these bundles is not meaningful, since most code is compound of standard code for OSGi bundles such as the activation code. The active code is in most case very succinct.

10.2 Perspectives

Further efforts are required to build production systems based on secure OSGi platforms. Both research work and industrial development are needed.

10.2.1 Open Requirements for building Secure Execution Environments

Research requirements concern both the methodology for security analysis, and the development of secure SOP platforms. They have been identified throughout this document.

The methodology for security analysis, $\mathcal{SPIP}$, should be refined in following ways:

- It should be validated against other target systems, and if required extended, to makes it technology independent. It is likely that specific tools such as metrics, vulnerability patterns and security patterns* are required for each type of target system.
- Finer security metrics are required. In particular, a metric to perform vulnerability assessment based on the dangerousness of attacks is missing. Such a metric should take the type of application into account: health care systems, banking systems and online games to not have similar security requirements; for instance, a heart-disease monitor

can not afford to suffer from complete denial-of-service, whereas a financial transaction server will not allow integrity damage.
- The cost computation should be considered explicitly to support the trade-off between a high security level* which prevents abuses and thus latter patches, and development overhead and constraints which requires developer training, limited functionalities, additional security review and tests.

Research requirements for building secure SOP platforms are the following ones. They include the integration of available research tools, such as a Trusted Computing Platform (TPM) for guaranteeing platform integrity, and the development of new ones to tackle identified challenges.

All attack vectors against OSGi systems should be mitigated (see Section 1.2.3 page 8):

- Bundle Deployment; tools are presented in this document.
- Application Interfaces, both GUI and remote interfaces.
- Attacks against the JVM and OSGi data.
- Execution of malicious bundles inside the platform, for which a set of tools are defined here.

In particular, these open requirements can be addressed through the following proposals:

- A Hardened JVM is required that provides protection against attacks from the local host, *e.g.* through JVM-TI or modification of system resources on the file system.
- Hardened OSGi should be extended to withstand modifications of data and resources on the file system such as platform code and applicative profiles.
- The proposed tools for assessing the security status of individual components need to be completed.
- An architectural approach to integrate components with a known security status in a secure manner is required.
- An integrated tool suit for building secure Java components is still to be developed.
- A secure SOP Workflow framework is to be defined. It should be mandatory for components that communicate through services*.
- Standard secure remote communication protocols and architecture validation tools should be defined and made available.

Moreover, our work opens the way for Bytecode analysis for security of non-Java languages that are compiled to Java Bytecode, such as Python with the Jython framework and Ruby with the JRuby framework. Since analyses are performed at the Bytecode level, such analysis should be supported. However, the impact of the original source language on the security status of the applications is hardly known.

Recent evolution of service-based computing consists in accessing remote services instead of local execution of components. This is called the 'software-as-a-service' approach. This opens new security challenges: if one component of the SOP platform is replaced by a stub to a service executed in a remote component, what trust can be put on it ? What are the technical guarantees that can be set ?

10.2.2 Open Requirements for Industrial Use Cases

Requirements for industrial development concern security mechanisms that are well known but not available so far. Their availability would make it possible for the industrial users

of the OSGi platform to take the best of the platform. It is likely that at least part of the identified required tools are providers by commercial vendors.

5 OSGi Security Profiles The OSGi platform is used in many different environments and application types. Therefore, several OSGi Security Profiles can be identified, which each provide a solution for a given use case.

The OSGi security profiles that we have identified so far are the following:

- Specification
- Life-cycle
- Management
- Critical applications
- Multi-user applications

For each profile, specific requirements are defined, and suitable implementation should be available.

Profile 1: OSGi R4 Specifications OSGi R4 Specifications is the basic security profile. The requirements are the following:

- Proper implementation of the specifications. For instance, OSGi Permissions, and OSGi Conditional Permissions are currently not available in the Apache Felix implementation.
- Simple OSGi Security Configuration Management for simple use of the specification-defined security mechanisms. Ready to use permissions profiles should be defined, that match specific application types.
- Negative permissions (preventing an operation for a given principal*) are currently missing, since OSGi permission management is fully based on Java permissions, that supports positive permission (allowing an operation) only.

The state of existing implementations is as follows.

- SFelix is an implementation of the bundle signature validation process, provided as a patch for the Apache Felix 1.0 implementation of the OSGi platform. It guarantees that no unvalid or unsigned bundle can be installed on the platform. SFelix is developed by the INRIA-ARES project.

Profile 2: Life-Cycle The Life-Cycle of OSGi bundles is not addressed by the OSGi R4 specification. However, it is supported in most if not all implementations. The Life-Cycle of OSGi bundles is compound of following steps: bundle packaging, bundle publication (by the issuer), bundle discovery, bundle download, bundle installation, and bundle execution (by the OSGi Platform). However, a Life-Cycle long security control is so far only partially available.

The state of existing implementations is as follows. SF-Jarsigner is a tool for signing and publishing bundles onto an OBR v2 repository. SF-Jarsigner is developed by the INRIA-ARES project, as a complement for the SFelix platform.

The requirements are as follows.

- Tools for management of public and private keys are to be defined, according to the needs of real projects.

Profile 3: Remote Management of OSGi Gateways OSGi Platforms need to be managed remotely. This can be done through JMX based console, such as the MOSGI console in the Felix project. So far, no security mechanisms exists for such environments, which prevents their exploitation over the Internet.

The requirements for secure OSGi management are as follows.

- JMX management over secure channels.
- Key management for OSGi remote access.
- Confidential deployment.

Profile 4: Critical Client* Server applications based on the OSGi platform The dynamic nature of OSGi platforms make them a potential candidate for being an execution environment for evolutive applications. For instance, critical applications such as billing terminals and related banking servers could take advantage of the technology, provided that the enforced security model can be proved to be sufficient.

The requirements for OSGi Critical Applications are the following ones:

- Isolation between providers inside the OSGi Gateway.
- Transaction Support for (e.g. monetary) transactions.
- Secure message-based communications between gateways and servers.
- Certification of Security procedure in the case of bank card applications which protect data, PIN codes and keys.

The Requirements relative to the development of critical applications are the following ones:

- Cohabitation of applications.
- Ease of development.
- No compromission of security.

The security profile 'Critical Applications' shows that the OSGi Platform has a great potential for building highly secure applications, both because of built-in security features and because of the associated development model. However, the technology is clearly not yet mature enough to support such applications.

Profile 5: Multi-User Applications The 5th profile, 'Multi-User Applications', requires that proper access control mechanisms are available.

Associated requirements are the following ones:

- Access isolation between bundles - validated access control model.
- Limited performance overhead.
- Compile time or install time verification when possible.

Conclusion Besides the identification of vulnerabilities and the proposition of several protection mechanisms for the OSGi platform, and SOP platforms in general, the open requirements in term of research and of development are summarized here. They should be considered in the context of the final claim of our work.

10.3 Conclusions on the Thesis' Argument

The technological conclusion of this work is that, in spite of the actual state of available implementations, the OSGi platform can actually be a very secure execution environment if sufficient development effort is provided. This imply to continue the development effort to release tools with production level quality.

The original methodological argument of this thesis, presented in Section 1.1 page 3, can be revised according to the presented experiments:

> Techniques from the Software Security domain at the platform level greatly enhance the security level of extensible SOP platforms, and their integration in the development life-cycle provides the additional benefit of supporting seamless developer training.
>
> These techniques should be used together with System Security* techniques at the Virtual Machine level to provide maximal protection.

This thesis highlights three important challenges for building secure extensible SOP platform:

1. A Secure Development Life-Cycle (SDLC) process should be defined to tackle application-specific threat prevention. It should be supported as far as possible by automated tools.
2. Tools with production level quality are required to support the proposed Software Security process,
3. Secure implementations of the Java Virtual Machine and other VMs with production-compliant performances are required.

This work leads me to believe in the efficiency of *Mandatory Software Security* both in term of security properties of complex software systems, and in term of developer awareness and training.

Mandatory Software Security for component-based systems can be defined as follows:

> Recent execution environments enforce software quality through a mandatory programming model. For instance, the OSGi platform provides component isolation and a clean dependency resolution process.
>
> Software Security should be similarly enforced in the execution environment itself by dedicated modules as a mandatory non functional property of the software components.

Part V

Appendix

Definitions

Application Security : the domain of security* that deals with the prevention and detection of attacks* against applications through post-development analysis and protection mechanisms (black-box testing, sandboxing, runtime monitoring, policies management and enforcement).

Attack : actions that attempt to defeat the expected security status of a system. The goal of attack can be the technical exploit, or the obtention of direct benefits.

Attack Surface : the scope of functionalities that is available to application users, particularly unauthenticated users, in a software environment. This includes, but is not limited to: user input fields, protocols, interfaces, services*[1].

Attack Vector : a path or means by which a hacker (or cracker) can gain access to a computer or network server in order to deliver a payload or malicious outcome. Attack* vectors enable hackers to exploit system* vulnerabilities*, including the human element. Examples of attack vectors are viruses, e-mail attachments, Web pages, pop-up windows, instant messages, chat rooms, and deception[2].

Black Hat : term used to describe a hacker (or cracker) who breaks into a computer system or network with malicious intent[3].

Bundle : the unit of deployment and installation of code in the OSGi platform* [All07a]. A bundle typically contains Java classes, exports some packages for use by third party bundles as libraries, and provides service-oriented programming* (SOP) services*.

Classification : systematic arrangement in groups or categories according to established criteria [MW08].

Client : an application or system* that accesses a local or remote service* through a well-defined protocol. Service providers are called 'servants'* when they are located in the same system than the client, 'servers' otherwise.

[1] from http://en.wikipedia.org/wiki/Attack_surface, 2008/07/29
[2] from http://searchsecurity.techtarget.com/dictionary/definition/1005812/attack-vector.html, 2008/07/29
[3] from http://searchsecurity.techtarget.com/sDefinition/0,,sid14_gci550815,00.html, read on 2008/08/20

Component : a system* element offering a predefined service* or event, and able to communicate with other components[4]. The following five criteria for what a software component shall be are: multiple-use, non-context-specific, composable with other components, encapsulated i.e., non-investigable through its interfaces, a unit of independent deployment and versioning [SGM02].

Component Platform : a platform* that is dedicated to the management and execution of a specific type of components*. Examples are J2EE for EJB, OSGi for OSGi bundles*, .Net for Assemblies.

Digital Signature : a type of asymmetric cryptography used to simulate the security properties of a handwritten signature on paper. Digital signature schemes consist of at least three algorithms: a key generation algorithm, a signature algorithm, and a verification algorithm. A signature provides authentication of a "message"[5].

Entity : An entity is something that has a distinct, separate existence, though it need not be a material existence[6]. We use the term entity to describe any element of a computing system* that interacts with others.

Execution Environment : A software routine that accepts commands as input and causes them to be executed. Execution environments exist within operating systems and may be an option within applications[7]. Commands can be simple or be made of full programs. Examples of execution environments are virtual machines, OS, and any application supporting plug-ins (*e.g.* Firefox and other Web browsers).

Exploit : a piece of software, a chunk of data, or sequence of commands that take advantage of a bug, glitch or vulnerability in order to cause unintended or unanticipated behavior to occur on computer software[8].

Flaw : a developmental, internal, human-made, software, nonmalicious, nondeliberate, permanent fault [ALRL04].

Function : Behavior of a program that is part of its specification. A function can be a vulnerability* if it can be abused by malicious entities.

Modularity : the property of modular programming. Modular programming is a software design technique that increases the extent to which software is composed from separate parts, called modules[9].

[4] from http://en.wikipedia.org/wiki/Component-based_software_engineering, 2008/07/29
[5] from http://en.wikipedia.org/wiki/Digital_signature, 2008/07/29
[6] http://en.wikipedia.org/wiki/Entity, 2008/07/29
[7] from http://www.techweb.com/encyclopedia/defineterm.jhtml?term=execution+environment, 2008/07/29
[8] from http://en.wikipedia.org/wiki/Exploit_(computer_security), read on 2008/08/20
[9] from http://en.wikipedia.org/wiki/Modularity_(programming), 2008/07/29

Network Security : the domain of security* that deals with the prevention and detection of attacks* through protocols, appliances and monitoring tools that are located in the network (firewalls, IDS, etc.).

Off-the-Shelf Component : generic software component*, that contains fixed functionality [Voa98], and can be obtained as is. Examples are COTS (Commercial OTS), open source libraries, legacy software ...

Platform : the basic technology of a computer system*'s hardware and software that defines how a computer is operated and determines what other kinds of software can be used[10].

Principal : Software representation of an entity* (an individual, a corporation, a login, a place in the file system) to which execution rights are granted[11].

Protection Domain : the set of objects directly accessible by a principal* [SS73].

Qualified Name : an unambiguous name that specifies which object, function, or variable a call refers to absolutely[12].

Reference Vulnerability Information (RVI) , or Refined Vulnerability Information: extensive databases that maintain up to date information related to known software vulnerabilities [PF07a].

Security : a composition of the following attributes: confidentiality (the absence of unauthorized disclosure of information), integrity (absence of unauthorized system* alterations) and availability (readiness for correct service, for authorized actions only) [ALRL04].

Security Assessment : the qualitative and quantitative evaluation of the security properties of a system*. The goal of assessment is to gather information relative to a given system.

Security Assurance Software Security Assurance (SSA) is the process of ensuring that software is designed to operate at a level of security that is consistent with the potential harm that could result from the loss, inaccuracy, alteration, unavailability, or misuse of the data and resources that it uses, controls, and protects[13].

Security Benchmarking : the quantitative evaluation of a system*. The goal of benchmarking is to compare various flavours of a system, or to compare a given implementation against a reference implementation. Security benchmarking is a subset of security assessment*.

Security Level : Quantification of the security status of a given computing system. It is a relative measure aimed at comparing various flavours of the system. In this thesis, we propose to use the *Protection Rate* as a metric for expressing the security level.

[10] http://www.answers.com/topic/platform
[11] from http://java.sun.com/j2se/1.5.0/docs/api/java/security/Principal.html
[12] from http://en.wikipedia.org/wiki/Fully_qualified_name, read on 2008/08/20
[13] from http://en.wikipedia.org/wiki/Software_Security_Assurance

Security Pattern : a well-understood solution to a recurring information security problem. They are patterns in the sense originally defined by Christopher Alexander [Ale77] (the basis for much of the later work in design patterns and pattern languages of programs), applied to the domain of information security[14].

Servant : a component* that provides local services* to other components executed in the same execution environment*, which act as its clients*. The term servant is used to make a distinction with the term 'server' which often imply remoting technologies.

Service : a contractually defined behavior that can be implemented and provided by any component* for use by any component, based solely on the contract [BC01]. This term is often used implicitly for remote services using technologies such as the Web Services.

Service-oriented Programming : a programming paradigm where software components* publish and use services* in a peer-to-peer manner. It is built on object-oriented programming (OOP) and component models [BC01].

Software Security : the domain of security* that is concerned with the realization of provably secure software systems*. The term *Realization* means that security is to be enforced throughout the software development life-cycle. *Provably secure* means that the security level of the resulting software system can be measured to assess the actual benefit brought in by the security effort and to compare it with similar systems.

System : A group of interacting, interrelated, or interdependent elements forming a complex whole[15]. The term 'System' is also used to denote operating system, distributed system or any coherent set of software and hardware.

System Security , or Operating System Security: the domain of security* that deals with the prevention and detection of attacks* against an operating system. The ultimate goal of attacks against an OS is to 'own' it, *i.e.* to have full administration rights on it.

Taxonomy : the theoretical study of a classification*, including its bases, principles, and rules. The term *taxonomy* is also used to appoint the classifications that are built according to such principles [Krs98, MW08].

Virtualization : a broad term that refers to the abstraction of computer resources. In this thesis, we are concerned with Application Virtualization, *i.e.* software technologies that improve the portability, manageability and compatibility of applications by encapsulating them from the underlying operating system on which they are executed[16].

[14] from http://www.scrypt.net/ celer/securitypatterns/, 2008/07/29
[15] from http://www.thefreedictionary.com/system, 2008/07/29
[16] from http://en.wikipedia.org/wiki/Application_Virtualization, 2008/07/29

Vulnerability : a weakness in a computing system* that could be exploited to gain unauthorized access to information or to disrupt critical processing [DM90], cited by [Krs98]. Unauthorized access means that the access is either forbidden by the system specifications, or that it constitute a violation of the expectations of users, administrators, and designers [Krs98]. A vulnerability can be either a flaw* or a dangerous function*.

Vulnerability Pattern : Design Pattern that documents software vulnerabilities*.

White Hat : White hat describes a hacker (or cracker) who identifies a security weakness in a computer system or network but, instead of taking malicious advantage of it, exposes the weakness in a way that will allow the system's owners to fix the breach before it is can be taken advantage by others[17].

[17] from http://searchsecurity.techtarget.com/sDefinition/0,,sid14_gci550882,00.html, read on 2008/08/20

B Vulnerability Catalog Entries

B.1 The *Malicious Bundle* Catalog

Bundle Archive

Vulnerabilities that originate in the Bundle Archive are due to flaws in the structure of the archive. They are not bound with the content of the archive, and are therefore not specifically bound to the Java world.

Invalid Digital Signature Validation (mb.archive.1) a bundle which signature is NOT compliant to the OSGi R4 Digital Signature is installed on the platform; bundles with lacking or additional malicious classes can then be installed,

Big Component Installer (mb.archive.2) remote installation of a bundle which size is bigger than the available device memory; this results in rapid disk space exhaustion,

Decompression Bomb (mb.archive.3) the Bundle Archive is a decompression Bomb (either a huge file made of identical bytes, or a recursive archive); this leads to memory exhaustion.

Bundle Manifest

Vulnerabilities that originate in the Bundle Manifest are due to flaws in the expressed Metadata. They are bound either to the way the JVM handles the Manifest, or to OSGi-defined properties.

Duplicate Package Import (mb.osgi.1) a package is imported twice (or more) according to manifest attribute 'Import-Package'. In the Felix and Knopflerfish OSGi implementations, the bundle can not be installed. This is due to the OSGi platform.

Excessive Size of Manifest File (mb.osgi.2) a bundle with a huge number of (similar) package imports (more than 1 MByte); this behavior originates in the virtual machine (in particular, SUN JVM and JamVM are concerned), and leads to a temporary freezing of the platform when the Manifest is loaded into memory,

Erroneous values of Manifest attributes (mb.osgi.3) a bundle that provides false meta-data, for instance an non existent bundle update location. This is bound with OSGi Metadata.

Bundle Activator

Vulnerabilities that originate in the Bundle Activator are bound to OSGi bundle starting process.

Management Utility Freezing - Infinite Loop (mb.osgi.4) an infinite loop is executed in the Bundle Activator ; this freezes the process that has launched the starting of the bundle, and consumes most of the available CPU,

Management Utility Freezing - Thread Hanging (mb.osgi.5) a hanging thread in the Bundle Activator makes the management utility freeze. Already installed bundles are not impacted.

Bundle Code (Native)[1] Vulnerabilities that originate in Native Code are bound to the possibility that exists in the Java Virtual Machine to execute code outside of the JVM. They are therefore not specific to the Java world.

Runtime.exec.kill (mb.native.1) a bundle that stops the platform through an OS call,

CPU Load Injection (mb.native.2) a malicious bundle that consumes an arbitrary amount (up to 98%) of the host CPU. Other processes on the platform experience an important loss of performance.

Bundle Code (Java API)

Vulnerabilities that originate in Java API are due to features that are provided through the Java Classpath, *i.e.* libraries that are provided along with the JVM.

System.exit (mb.java.1) a bundle that stops the platform by calling 'System.exit(0)',

Runtime.halt (mb.java.2) a bundle that stops the platform by calling 'Runtime.getRuntime.halt(0)',

Recursive Thread Creation (mb.java.3) The platform is brought to crash by the creation of an exponential number of threads,

Hanging Thread (mb.java.4) Thread that makes the calling entity hang through interlocking (service, or package),

Sleeping Bundle (mb.java.5) a malicious bundle that goes to sleep during a specified amount of time before having finished its job,

Big File Creator (mb.java.6) a malicious bundle that creates a big (relative to available resources) files to consume disk memory space,

Code Observer (mb.java.7) a component that observes the content of another one through reflection; code and hard-coded configurations can be spied,

Component Data Modifier (mb.java.8) a bundle that modifies data (i.e. the value of the `public static` attributes of the classes) of another one through reflection,

Hidden Method Launcher (mb.java.9) a bundle that executes (through reflection) methods from classes that are not exported or provided as service. All classes that are referenced (directly or indirectly) as class attributes can be accessed. Only public methods can be invoked.

Bundle Code (Java Language)

Vulnerabilities that originate in Java Language are bound with language-level features, in particular Object Orientation. They can therefore be relevant to other Object-Oriented Languages, and some are general enough to concerns any programming language.

[1] These Bundles are specifically targeted to the OSGi platform. All native code attacks against the operating system or other applications are not considered here.

Memory Load Injection (mb.java.10) a malicious bundle that consumes most of available memory; if other processes are executed that require memory, `MemoryError`s are caused. This vulnerability concerns any multi-process system without resource isolation.

Stand Alone Infinite Loop (mb.java.11) a void loop in a lonesome thread that consumes much of the available CPU. This vulnerability concerns any multi-process system without resource isolation.

Infinite Loop in Method Call (mb.java.12) an infinite loop is executed in a method call (at class use, package use); this vulnerability concerns any programming language.

Exponential Object Creation (mb.java.13) Objects are created in an exponential way, and force the call to abort with a `StackOverflowError`. This vulnerability as such is specific to Object-Oriented languages, but variants can be built that use procedure recursions in any programming language.

Bundle Code (OSGi API)

Vulnerabilities that originate in the OSGi API are due to features that are provided by the OSGi platform.

Launch a Hidden Bundle (mb.osgi.6) a bundle that launches another bundle it contains (the contained bundle could be masqueraded as a 'MyFile.java' file); any program can therefore be hidden.

Pirate Bundle Manager (mb.osgi.7) a bundle that manages others without being requested to do so (for instance, stops, starts or uninstalls the victim bundle).

Zombie Data (mb.osgi.8) Data stored in the local OSGi data store are not deleted when the related bundle is uninstalled. It thus becomes unavailable and consumes disks space (especially on resource constraint devices). This is due to the implementation of the OSGi platform.

OSGi Services and Bundle Fragments

Vulnerabilities that originate in the OSGi API are due to the specific type of interactions that exist inside the OSGi platform. Those that are due to the Service-oriented Programming (SOP) [BC01] paradigm are relevant to any SOP platforms.

Cycle Between Services (mb.osgi.9) a cycle exists in the services call; this vulnerability is related to SOP,

Numerous Service Registration (mb.osgi.10) registration of a high number of (possibly identical) services through an loop; this vulnerability is related to SOP,

Freezing Numerous Service Registration (mb.osgi.11) registration of a high number of (possibly identical) services through an loop, that make the whole platform freeze in the Concierge implementation; this vulnerability is related to SOP.

Execute Hidden Classes (mb.osgi.12) a fragment bundle exports a package from its Host that this latter do not intend to make visible. Other bundles can then execute the classes in this package,

Fragment Substitution (mb.osgi.13) a specific fragment bundle is replaced by another, which provides the same classes but with malicious implementation,

Access Protected Package through split Packages (mb.osgi.14) a package is built in the fragment that have the same name than a package in the host. All package-protected classes and methods can then be accessed by the fragment, and thus exported to the framework.

B.2 The *Vulnerable Bundle* Catalog

Stand-Alone Applications Vulnerabilities

Expose Internal Representation - Serialized Sensitive Data (vb.java.1) All data in a serialized object can be read. In particular, security checks that may exist in the code are no longer enforced. No sensitive data must be stored in serializable objects.

Class Sharing Vulnerabilities - Exposed Internal Representation

Stores Mutable Object in static Variable (vb.java.class.1) A method stores a reference to a mutable object in a static variable. Internal Data of the victim object can be read and/or modified.

Stores Array in Static Variable (vb.java.class.2) A method stores a reference to a array in a static variable. Process. Internal Data of the victim object can be read and/or modified.

Non Final Static Variable (vb.java.class.3) A method keeps a reference to a static non final static object. Internal Data of the victim object can be read and/or modified.

Shutdown Hook (vb.java.class.4) Shutdown Hooks enable to execute code when the platform is stopped. In particular, this implies that components can execute code after they have been uninstalled.

Private nested Classes and Attributes made protected (vb.java.class.5) Private nested classes and attributes are made protected at compilation. Consequently, OSGi Bundle Fragments can be exploited to access the target package through the 'Split Package' vulnerability, and access the private Class or Attribute as a protected one.

Class Sharing Vulnerabilities - Avoidable Calls to the Security Manager

Override Method (vb.java.class.6) Security
Checks that are performed in overridable methods can be by-passed by rewriting the methods.

Privileged Execution of Code provided by the Caller (vb.java.class.7) Privileged Code Execution must be restricted to code provided by the privileged bundle. If code origin is not properly controlled, less trusted bundles can provide their own code for Privileged Execution.

Privileged Execution of Code provided by the Caller - Class Loader Privileges (vb.java.class.8) Privileged Code Execution must be restricted to code provided by the privileged bundle. If code origin is not properly controlled, less trusted bundles can provide their own code for Privileged Execution. Privileged Execution is granted for several calls according to the current class loader (Reflection, Library Loading).

Cloning (vb.java.class.9) Calls to the 'clone' method enable to create a new instance of a class without calling the constructor, which often contains security checks such as calls to the security manager.

Deserialization (vb.java.class.10) Deserialization enables to create a new instance of a class without calling the constructor, which often contains security checks such as calls to the security manager.

Call Overrideable Method in Constructor (vb.java.class.11) Calling non-final methods in constructor enable sub-classes to access to partially initialized instances of the objects, and break security and configuration assumptions that are made in the superclass.

Call Overrideable Method in Clone Method (vb.java.class.12) Calling non-final methods in clone method enable sub-classes to access to partially initialized instances of the objects, and break security and configuration assumptions that are made in the superclass.

Finalize Method (vb.java.class.13) Methods on a Class that is protected through a security manager can be called by creating a subclass that, after creation abortion, performs calls on the partially initialized object during finalization.

Shutdown Hook (vb.java.class.14) Shutdown Hooks enable to execute code when the platform is stopped. In particular, this implies that components can execute code after they have been uninstalled. Moreover, if a security check is performed in the constructor after static global variable have been initialized, their value can be accessed.

Class or Object Sharing - Synchronization

Synchronized Method (vb.java.class.15) A method is synchronized, to as to avoid the execution of the same method by two different clients (used in particular in case of access to resources). 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.

Synchronized Code (vb.java.class.16) A
method contains a synchronized block, so as to avoid the execution of the same method by two different clients (used in particular in case of access to resources). 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.

Object Sharing Vulnerabilities - Exposed Internal Representation

Returns Reference to Mutable Object (vb.java.object.1) A method returns a reference to a mutable object. Internal Data of the victim object can be read and/or modified.

Returns Reference to Array (vb.java.object.2) A method returns a reference to a array. Internal Data of the victim object can be read and/or modified.

Visibility (vb.java.object.3) A method keeps a reference to a variable with too much visibility. Internal Data of the victim object can be read and/or modified.

Non Final non Private Field (vb.java.object.4) A method keeps a reference to a static non final non private object. Internal Data of the victim object can be read and/or modified.

No Wrapper (vb.java.object.5) Variables can be accessed on the object without wrapper methods, which prevent the execution of security or parameter checks. Variables can be accessed directly on the object.

Information Leak through Exceptions (vb.java.object.6) Exception Messages often contain data that describes the configuration of the system. These data should not be propagated to external callers, unless it directly concerns caller input.

Object Sharing Vulnerabilities - Flaws in Parameter Validation

Unchecked Parameters - Malicious Program Abuse - Java Code (vb.java.object.7) Unchecked parameters in bundle public code (OSGi Services or Exported Packages) can be exploited to execute malicious code.

Unchecked Parameters - Malicious Program Abuse - Native Code (vb.java.object.8) Unchecked parameters in bundle public code (OSGi Services or Exported Packages) can be exploited to execute malicious code, especially native code that does not provide any security guarantees.

Unchecked Parameters - Accidentally unsupported Value (vb.java.object.9) Unchecked parameters in bundle public code (OSGi Services or Exported Packages) can lead to unexpected program behavior if constraints on their values are not enforced.

Parameters Checked without Copy (vb.java.object.10) A parameter that is checked without being copied beforehand can be modified after validation and lead a TOCTOU (Time of Check To Time of Use) attack.

Copied and Checked Parameters - Fake Clone Method (vb.java.object.11) Copying parameters before their validation can be worthless if the copy is done through an overridden 'clone' method that is implemented partially or with a malicious objective.

Copied and Checked Parameters - Fake Copy Constructor (vb.java.object.12)Copying parameters before their validation can be worthless if the copy is done through a fake copy constructor that is implemented partially or with a malicious objective.

Copied and Checked Parameters - Uncomplete Copy - State Omission (vb.java.object.13) Copying parameters before their validation can be unsufficient if some states are omitted.

Copied and Checked Parameters - Uncomplete Copy - Mutable States (vb.java.object.14) Copying parameters before their validation can be unsufficient if some states are mutable.

Non Final Parameters - Malicious Implementation (vb.java.object.15) Non-final parameters in bundle public code (SOP Services or Exported Packages) can be exploited to execute malicious code, possibly exploiting internal data of the victim bundle.

Non Final Parameters - Inversion of Control (vb.java.object.16) Non-final parameters in bundle public code (SOP Services or Exported Packages) can be exploited to execute malicious code through inversion of control, ie. actions in the malicious bundle can be triggered through the victim bundle, possibly leaking data. Data from the victim bundle can be exploited. Certain type of access control mechanisms can be by-passed.

C Listings

C.1 Examples of Attacks against the Java/OSGi Platform

Listing C.1 gives an implementation of the *Recursive Thread Creation* vulnerability.

Listing C.1: Example of malicious code in OSGi bundle: recursive thread creation

```
public class Stopper extends Thread{

    Stopper(int id, byte[] payload)
    {
        this.id=id;
        this.payload = payload;
    }
    public void run()
    {
        System.out.println("Stopper id: "+id);
        Stopper tt = new Stopper(++id, payload);
        tt.start();

        Stopper tt2 = new Stopper(++id, payload);
        tt2.start();

        Stopper tt3 = new Stopper(++id, payload);
        tt3.start();
    }
}
```

Listing C.2 gives an implementation of the *Launch a Hidden Bundle* attack, where a malicious bundle loads another one, that is hidden inside its own archive, and starts it.

Listing C.2: Bundle that launches a bundle that is hidden inside it

```
public void start(BundleContext context)
{
    try
    {
        //get data for the created bundle
        String fileName = "bigflowerpirat-1.1.jar";
```

```
            byte[] buffer = this.getBundleResourceData("/"+fileName);

            //create a new data file with loaded data
            File file = this.newDataFile(fileName, buffer, context);

            //install new bundle
            String location = file.getPath();
            System.out.println(location);
            Bundle b = context.installBundle("file://"+location);
            b.start();
        }
        catch(Exception e){e.printStackTrace();}

        System.out.println("Malicious BundleLoader started");
    }
```

Listing C.3 gives an implementation of the *Memory Load Injection* attack, which consumes an important part of the available memory. It can lead to an `OutOfMemoryError` if other processes also require memory.

Listing C.3: Example of malicious code in OSGi bundle: memory load injection

```
private void stressMem(int size)
{

    System.out.println("Eating " + size + " bytes of memory");

    this.memEater = new byte[size];
    for (int i=0 ; i<size ; i++)
    {
       this.memEater[i] = 0;
    }
}
```

Bibliography

[ACD+75] R. P. Abbott, J. S. Chin, J. E. Donnelley, W. L. Konigsford, S. Tokubo, and D. A. Webb. Security analysis and enhancements of computer operating systems. Technical report, National Bureau of Standards, Washington DC, Institute for Computer Sciences and Technology, December 1975.

[ACDM01] Jimmy Arvidsson, Andrew Cormack, Yuri Demchenko, and Jan Meijer. TERENA's incident object description and exchange format requirements. IETF RFC 3067, February 2001.

[AFS97] William A. Arbaugh, David J. Farber, and Jonathan M. Smith. A secure and reliable bootstrap architecture. In *IEEE Symposium on Security and Privacy*, pages 65–71, 1997.

[AH06] OSGi Alliance and Richard S. Hall. Bundle repository. OSGi Alliance RFC 112, 2006.

[Ale77] Christopher Alexander. *A Pattern Language*. Oxford University Press, 1977. ISBN-13: 978-0195019193.

[All05a] OSGi Alliance. OSGi service platform, Core Specification release 4. Draft, July 2005.

[All05b] OSGi Alliance. OSGi service platform, Service Compendium release 4. Draft, July 2005.

[All07a] OSGi Alliance. OSGi service platform, Core Specification release 4.1. Draft, May 2007.

[All07b] OSGi Alliance. OSGi service platform, Service Compendium release 4.1. Draft, April 2007.

[ALRL04] Algirdas Avizienis, Jean-Claude Laprie, Brian Randell, and Carl Landwehr. Basic concepts and taxonomy of dependable and secure computing. *IEEE Transactions on Dependable and Secure Computing*, 01(1):11–33, 2004.

[Arn73] Thomas F. Arnold. The concept of coverage and its effect on the reliability model of a repairable system. *IEEE Transactions on Computing*, 22(3):251–254, 1973.

[Bac04] David F. Bacon. The Virtualized Virtual Machine: The next generation of virtual machine technology. In *Workshop Language Runtimes '04, at OOPSLA'04 ,IBM T.J. Watson Research Center*, 2004.

[BAT06] Anil Bazaz, James D. Arthur, and Joseph G. Tront. Modeling security vulnerabilities: A constraints and assumptions perspective. In *2nd IEEE International Symposium on Dependable, Autonomic and Secure Computing (DASC'06)*, pages 95–102, 2006.

[BBB+07] Michael Beisiegel, Henning Blohm, Dave Booz, Mike Edwards, Oisin Hurley, Sabin Ielceanu, Alex Miller, Anish Karmarkar, Ashok Malhotra, Jim Marino, Martin Nally, Eric Newcomer, Sanjay Patil, Greg Pavlik, Martin Raepple, Michael Rowley, Ken Tam, Scott Vorthmann, Peter Walker, and Lance Waterman. SCA service component architecture - assembly model specification, version 1.00, March 2007.

[BBC+06] Siddharth Bajaj, Don Box, Dave Chappell, Francisco Curbera, Glen Daniels, Phillip Hallam-Baker, Maryann Hondo, Chris Kaler, Dave Langworthy, Anthony Nadalin, Nataraj Nagaratnam, Hemma Prafullchandra, Claus von Riegen, Daniel Roth, Jeffrey Schlimmer, Chris Sharp, John Shewchuk, Asir Vedamuthu, Umit Yalcinalp, and David Orchard. Web Services policy 1.2 - framework (WS-policy). W3C Member Submission, public draft release, April 2006.

[BBC+07] Michael Beisiegel, Dave Booz, Ching-Yun Chao, Mike Edwards, Sabin Ielceanu, Anish Karmarkar, Ashok Malhotra, Eric Newcomer, Sanjay Patil, Michael Rowley, Chris Sharp, and Umit Yalcinalp. SCA policy framework v1.00, March 2007.

[BC01] Guy Bieber and Jeff Carpenter. Introduction to Service-Oriented Programming (rev 2.1). OpenWings Whitepaper, April 2001.

[BCHM99] David W. Baker, Steven M. Christey, William H. Hill, and David E. Mann. The development of a common enumeration of vulnerabilities and exposures. In *Second International Workshop on Recent Advances in Intrusion Detection*, 1999.

[BG97] Dirk Balfanz and Li Gong. Experience with secure multi-processing in Java. Technical Report 560-97, Department of Computer Science, Princeton University, September 1997.

[BG05] Joshua Bloch and Neal Gafter. *Java Puzzlers - Traps, Pitfalls and Corner Cases*. Pearson Education, June 2005. ISBN-13: 978-0321336781.

[BH78] Richard Bisbey and Dennis Hollingworth. Protection analysis: Final report. Technical Report ARPA ORDER NO. 2223, ISI/SR-78-13, Information Sciences Institute, University of Southern California, May 1978.

[BHL00] Godmar Back, Wilson C. Hsieh, and Jay Lepreau. Processes in KaffeOS: isolation, resource management, and sharing in Java. In *4th conference on Symposium on Operating System Design & Implementation (OSDI'00)*, Berkeley, CA, USA, 2000. USENIX Association.

[BHV01] Walter Binder, Jarle Hulaas, and Alex Villazon. A portable resource control in Java: The J-Seal2 approach. In *16th ACM OOPSLA*, October 2001.

[BJMT01] Fréderic Besson, Thomas Jensen, Daniel Le Metayer, and Tommy Thorn. Model checking security properties of control flow graphs. *Journal of Computer Security*, 9(3):217 – 250, January 2001.

[BK98] East B. Kaliski, RSA Laboratories. PKCS-7: Cryptographic message syntax, version 1.5. IETF RFC 2315, March 1998.

[BKF06] Paul E. Black, Michael Kass, and Elizabeth Fong, editors. *Workshop on Software Security Assurance Tools, Techniques, and Metrics*, February 2006.

[Bla05] Paul E. Black. Software assurance metrics and tool evaluation. In *International Conference on Software Engineering Research and Practice (SERP'05)*, June 2005.

[BLC02] Eric Bruneton, Romain Lenglet, and Thierry Coupaye. ASM: a code manipulation tool to implement adaptable systems. In *Adaptable and Extensible Component Systems*, November 2002.

[Blo01] Joshua Bloch. *Effective Java Programming Language Guide*. Addison-Wesley Professional, 2001. ASIN: B000OZ0N5I.

[Boe86] Barry Boehm. A spiral model of software development and enhancement. *SIGSOFT Software Engineering Notes*, 11(4):14–24, 1986.

[Boy04] C. Boyapati. *SafeJava: A Unified Type System for Safe Programming*. PhD thesis, Massachusetts Institute of Technology, Department of Electrical Engineering and Computer Science, February 2004.

[Bro87] Frederick P. Brooks. No silver bullet: Essence and accidents of software engineering. *Computer*, 20(4):10–19, April 1987.

[Bru06] Glenn Brunette. Toward systemically secure IT architectures. Sun BluePrints OnLine, February 2006. Sun Microsystems, Inc.

[Bry04] Ciaran Bryce. 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.

[CBR03] William R. Cheswick, Steven M. Bellovin, and Aviel D. Rubin. *Firewall and Internet Security: Repelling the Wily Hacker*. Addison-Wesley, 2003. ISBN-13: 978-0201633573.

[CDH+00] James C. Corbett, Matthew B. Dwyer, John Hatcliff, Shawn Laubach, Corina S. Păsăreanu, Robby, and Hongjun Zheng. Bandera: extracting finite-state models from Java source code. In *22nd International Conference on Software Engineering (ICSE '00)*, pages 439–448. ACM, 2000.

[CH03] Humberto Cervantes and Richard S. Hall. Automating service dependency management in a service-oriented component model. In *ICSE CBSE*, 2003.

[CH04] Humberto Cervantes and Richard S. Hall. Autonomous adaptation to dynamic availability using a service-oriented component model. In *26th International Conference on Software Engineering (ICSE '04)*, pages 614–623. IEEE Computer Society, 2004.

[Che06] Brian Chess. Metrics that matter: Quantifying software security risk. In *Workshop on Software Security Assurance Tools, Techniques, and Metrics*, February 2006. NIST Special Publication 500-265.

[Chr05] Steven M. Christey. The preliminary list of vulnerability examples for researchers (PLOVER). In *NIST Workshop Defining the State of the Art of Software Security Tools, Gaithersburg, MD*, August 2005.

[Chr06] Steven M. Christey. Open letter on the interpretation of "vulnerability statistics". Bugtraq, Full-Disclosure Mailing list, January 2006.

[CHS+05] Grzegorz Czajkowski, Stephen Hahn, Glenn Skinner, Pete Soper, and Ciarán Bryce. A resource management interface for the Java™platform. *Software Practice & Experience*, 35(2):123–157, 2005.

[Chu04] Mandy Chung. Using JConsole to monitor applications. *Sun Developer Network Whitepaper*, December 2004. http://java.sun.com/developer/technicalArticles/J2SE/jconsole.html.

[CK08] Dave Chappell and Khanderao Kand. Universal middleware: What's happening with OSGi and why you should care. http://soa.sys-con.com/node/492519, read on 2008/07/31, February 2008.

[CL00] David Carney and Fred Long. What do you mean by COTS ? Finally, a useful answer. *IEEE Software*, 17(2):83–86, March/April 2000.

[CLN00] Christopher Colby, Peter Lee, and George C. Necula. A proof-carrying code architecture for Java. In *12th International Conference on Computer Aided Verification (CAV'00)*, pages 557–560, London, UK, 2000. Springer-Verlag.

[CM99] David M. Chess and John F. Morar. Is Java still secure? In *Virus Bulletin Conference*, October 1999.

[CMPB06] Eun-Ae Cho, Chang-Joo Moon, Dae-Ha Park, and Doo-Kwon Baik. Access control policy management framework based on RBAC in OSGi service platform. In *Sixth IEEE International Conference on Computer and Information Technology,*, 2006.

[CMS01] Ajay Chander, John C. Mitchell, and Insik Shin. Mobile code security by Java bytecode instrumentation. In *DARPA Information Survivability Conference & Exposition II (DISCEX '01)*, 2001.

[Cop05] Tom Copeland. *PMD Applied*. Centennial Books, November 2005. ISBN: 0-9762214-1-1.

[COR06] Domenico Cotroneo, Salvatore Orlando, and Stefano Russo. Failures classification and analysis of the Java Virtual Machine. In *26th IEEE International Conference on Distributed Computing Systems (ICDCS'06)*, 2006.

[COR07] Domenico Cotroneo, Salvatore Orlando, and Stefano Russo. Characterizing aging phenomena of the Java Virtual Machine. In *26th IEEE International Symposium on Reliable Distributed Systems (SRDS 2007)*, pages 127–136, Washington, DC, USA, 2007. IEEE Computer Society.

[CV01] Denis Caromel and Julien Vayssiere. Reflections on MOPs, components, and Java security. In *European Conference on Object-Oriented Programming (ECOOP'2001)*, volume 2072 of *LNCS*, pages 256–274. Spinger Verlag, 2001.

[CvE98] Grzegorz Czajkowski and Thorsten von Eicken. JRes: a resource accounting interface for Java. *SIGPLAN Notice*, 33(10):21–35, 1998.

[CW87] D. D. Clark and D. R. Wilson. A Comparison of Commercial and Military Computer Security Policies. In *1987 IEEE Symposium on Security and Privacy*, pages 184–194. IEEE Computer Society Press, 1987.

[CW02] Hao Chen and David A. Wagner. MOPS: an infrastructure for examining security properties of software. Technical Report CSD-02-1197, Berkeley, CA, USA, 2002.

[Cza00] Grzegorz Czajkowski. Application isolation in the Java^{TM}Virtual Machine. In *15th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, 2000.

[DE97] Sophia Drossopoulou and Susan Eisenbach. Java is type safe - probably. In *ECOOP'97 - Object-Oriented Programming*, volume 1241/1997 of *LNCS*, pages 389–418. Springer Berlin / Heidelberg, 1997.

[Dep00] Department of Computer Science for University of Arizona. The Java hall of shame, September 2000. http://www.cs.arizona.edu/sumatra/hallofshame/.

[DFW96] Drew Dean, Edward W. Felten, and Dan S. Wallach. Java security: From HotJava to Netscape and beyond. In *IEEE Symposium on Security and Privacy (SP'96)*, page 190. IEEE Computer Society, 1996.

[D'H05] Michel D'Hooge. Guidelines to improve the robustness of the OSGi framework and its services against malicious or badly coded bundles. In *OSGi Alliance Developer Forum*, 2005.

[Dij72] Edsger Dijkstra. The humble programmer. Turing Award Lecture; published in: Communication of the ACM n. 15, 1972.

[DM90] Longley Dennis and Shain Michael. *Data & Computer Security: Dictionary of Standards, Concepts, and Terms.* Macmillan Stockton Press, 1990. ISBN-13:978-0849371103.

[DSTZ05] Mourad Debbabi, Mohamed Saleh, Chamseddine Talhi, and Sami Zhioua. Security evaluation of J2ME CLDC embedded Java platform. *Journal of Object Technology*, 5(2):125–154, 2005.

[EH07] Clément Escoffier and Richard S. Hall. Dynamically adaptable applications with iPOJO service components. In *6th International Symposium on Software Composition (SC 2007)*, Braga, Portugal, March 2007.

[ES00] Ulfar Erlingsson and Fred B. Schneider. IRM enforcement of Java stack inspection. In *IEEE Symposium on Security and Privacy*, 2000.

[FD06] Olivier Festor and Sam D'Haeseleer. D B3.4 - Specification of residential gateway configuration. Muse IST Project n 026442 Deliverable, October 2006. Chapters 5 to 9 written by Pierre Parrend and Stéphane Frénot.

[FLL+02] Cormac Flanagan, K. Rustan M. Leino, Mark Lillibridge, Greg Nelson, James B. Saxe, and Raymie Stata. Extended static checking for Java. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, 2002.

[Fow04] Martin Fowler. Inversion of control containers and the dependency injection pattern, January 2004. http://www.martinfowler.com/articles/injection.html.

[FSH03] Ivan Flechais, Angela M. Sasse, and Stephen Hailes. Bringing security home: A process for developing secure and usable systems. In *ACM/SIGSAC New Security Paradigms Workshop*, 2003.

[FY97] Brian Foote and Joseph Yoder. Big ball of mud. In *Fourth Conference on Patterns Languages of Programs, PLoP'97*, 1997.

[GA03] Sudhakar Govindavajhala and Andrew Appel. Using memory errors to attack a virtual machine. In *IEEE Symposium on Security and Privacy*, 05 2003. Oakland.

[GCLDBT03] Grzegorz Czajkowski, Laurent Daynès, and Ben Titzer. A Multi-User Virtual Machine. In *Usenix*, pages 85–98, 2003.

[GED03] Li Gong, Gary Ellison, and Mary Dadgeforde. *Inside Java 2 Platform Security - Architecture, API Design, and Implementation, Second Edition.* Addison-Wesley, 2003. ISBN-13: 978-0201787917.

[GHJV94] Erich Gamma, Richard Helm, Ralph Johnson, and John M. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software.* Addison-Wesley Professional Computing Series. Addison Wesley Professional, 1994. ISBN-13: 978-0201633610.

[GJSB05] James Gosling, Bill Joy, Guy Steele, and Gilad Bracha. *Java Language Specification, Third edition.* Addison-Wesley Professional, June 2005. ISBN-13: 978-0321246783.

[GM96] James Gosling and Henry McGilton. The Java language environment - a white paper. Technical report, Sun Microsystems, Inc., 1996.

[GMPS97] Li Gong, Marianne Mueller, Hemma Prafullchandra, and Roland Schemers. Going beyond the sandbox: An overview of the new security architecture in the Java Development Kit 1.2. In *USENIX Symposium on Internet Technologies and Systems*, 1997.

[GPC04] Dimitra Giannakopoulou, Corina S. Pasareanu, and Jamieson M. Cobleigh. Assume-guarantee verification of source code with design-level assumptions. In *ICSE '04: Proceedings of the 26th International Conference on Software Engineering*, pages 211–220, Washington, DC, USA, 2004. IEEE Computer Society.

[GS98] Li Gong and Roland Schemers. Implementing protection domains in the Java Development Kit 1.2. In *Network and Distributed System Security Symposium*, 1998.

[GS02] Frédéric Guidec and Nicolas Le Sommer. Towards resource consumption accounting and control in Java: a practical experience. In *Workshop on Resource Management for Safe Language, ECOOP 2002*, June 2002.

[GTCF08] Nicolas Geoffray, Gaël Thomas, Charles Clément, and Bertil Folliot. Towards a new isolation abstraction for OSGi. In *First Workshop on Isolation and Integration in Embedded Systems (IIES 2008)*, pages 41–45, April 2008.

[GV95] Robert L. Glass and Iris Vessey. Contemporary application-domain taxonomies. *IEEE Software*, 12(4):63–76, 1995.

[GW05] Michael Gegick and Laurie Williams. Matching attack patterns to security vulnerabilities in software-intensive system designs. *ACM SIGSOFT Software Engineering Notes*, 30(4), July 2005.

[GWM+07] Karen M. Goertzel, Thoedore Winograd, Holly L. McKinley, Lyndon Oh, Michael Colon, Thomas Mcibbon, Elaine Fedchak, and Robert Vienneau. *Software Security Assurance: a State-of-The-Art Report (SOAR).* Information Assurance Technology Analysis Center (IATAC) and Data and Analysis Center for Software (DACS), July 2007.

[Hal01] Stuart D. Halloway. *Component Development for the Java Platform.* Addison-Wesley, 2001. ISBN-13: 978-0201753066.

[HK03] Jarle G. Hulaas and Dimitri Kalas. Monitoring of resource consumption in Java-based application servers. In *10th Workshop HP OpenView University Association*, July 2003.

[HL98] John D. Howard and Thomas A. Longstaff. A common language for computer security incidents. Technical Report SAND98-8667, Sandia National Laboratories, USA, October 1998.

[HL02] Michael Howard and David LeBlanc. *Writing Secure Code, Second Edition.* Microsoft Press, Redmond, WA, 2002. ISBN-13: 978-0735617223.

[HLV05] Michael Howard, David LeBlanc, and John Viega. *19 Deadly Sins of Software Security.* McGraw-Hill Osborne Media, July 2005. ISBN-13: 978-0072260854.

[HM04] Greg Hoglund and Gary McGraw. *Exploiting Software - How to Break Code*. Addison-Wesley, Peason Education Inc., 2004. ISBN-13:978-0201786958.

[Hol03] Gerard J. Holzmann. *The SPIN Model Checker: Primer and Reference Manual*. Addison-Wesley Professional, September 2003. ISBN-13: 978-0321228628.

[Hou04] Russell Housley. Cryptographic Message Syntax (CMS). IETF RFC 3852, July 2004.

[How97] T. Howes. The string representation of LDAP search filters. IETF RFC 2254, December 1997.

[How06] Michael Howard. A process for performing security code reviews. *IEEE Security and Privacy*, 4(4):74 – 79, July 2006.

[HP04] David Hovemeyer and William Pugh. Finding bugs is easy. In *ACM SIGPLAN Notices*, volume 39, pages 92 – 106, 2004. COLUMN: OOPSLA onward.

[HPW05] Michael Howard, Jon Pincus, and Jeanette M. Wing. *Computer Security in the 21st Century*, chapter Measuring Relative Attack Surfaces, pages 109–137. Springer, March 2005.

[HS05] Almut Herzog and Nahid Shahmehri. An evaluation of Java application containers according to security requirements. In *14th IEEE International Workshops on Enabling Technologies: Infrastructure for Collaborative Enterprise (WETICE?05)*, 2005.

[HT99] Andrew Hunt and David Thomas. *The Pragmatic Programmer: From Journeyman to Master*. November 1999. ISBN-13: 978-0201616224.

[HWH07] Chi-Chih Huang, Pang-Chieh Wang, and Ting-Wei Hou. Advanced OSGi security layer. In *21st International Conference on Advanced Information Networking and Applications Workshops (AINAW '07)*, pages 518–523, Washington, DC, USA, 2007. IEEE Computer Society.

[Ibr08] Noha Ibrahim. *Spontaneous Integration of Services in Pervasive Environments*. PhD thesis, INSA-Lyon, October 2008. To be defended.

[Jav] Java Community Process. Jsr 121 - Application Isolation API Specification. http://www.jcp.org/en/jsr/detail?id=121.

[JLT98] T. Jensen, D. Le Métayer, and T. Thorn. Security and dynamic class loading in Java: A formalisation. In *IEEE International Conference on Computer Languages*, pages 4–15, Chicago, Illinois, 1998.

[Joh78] Stephen C. Johnson. Lint, a C program checker. In *Comp. Sci. Tech. Rep*, pages 78–1273. Murray Hill, 1978.

[Jon99] Capers Jones. *Applied software measurement: assuring productivity and quality*. McGraw-Hill, Inc., New York, NY, USA, 1999.

[KCS05] Antonio Kung, Danny De Cock, and Hans Scholten. Using OSGi for secure service discovery. In *OSGi Alliance Developer Forum*, 10 2005.

[KETEH01] Darrell M. Kienzle, Matthew C. Elder, David Tyree, and James Edwards-Hewitt. Security patterns repository version 1.0, 2001. Report for DARPA Contract F30602-01-C-0164.

[KH04] Peter Kriens and BJ Hargrave. Listeners considered harmful: The 'whiteboard' pattern, revision 2.0. Technical report, OSGi Alliance, August 2004.

[Kim04] Hangkon Kim. A framework for security assurance in component based development. In *Computational Science and Its Applications (ICCSA)*, 2004.

[Krs98] Ivan Victor Krsul. *Software Vulnerability Analysis*. PhD thesis, Purdue University, May 1998.

[KZZ06] Jingfei Kong, Cliff C. Zou, and Huiyang Zhou. Improving software security via runtime instruction-level taint checking. In *1st workshop on Architectural and system support for improving software dependability (ASID '06)*, pages 18–24, New York, NY, USA, 2006. ACM.

[LA04] Chris Lattner and Vikram Adve. LLVM: A compilation framework for lifelong program analysis & transformation. In *International Symposium on Code Generation and Optimization (CGO'04)*, March 2004.

[Lai08] Charlie Lai. Java insecurity: Accounting for subtleties that can compromise code. *IEEE Software*, 25(1):13–19, 2008.

[LBMC94] Carl E. Landwehr, Alan R. Bull, John P. McDermott, and William S. Choi. A taxonomy of computer program security flaws, with examples. In *ACM Computing Surveys*, volume 26, pages 211–254, September 1994.

[LC05] Mikael Lofstrand and Jason Carolan. Sun's pattern-based design framework: the service delivery network. Sun BluePrints OnLine, September 2005. Sun Microsystems, Inc.

[LH05] Steve Lipner and Michael Howard. The trustworthy computing security development lifecycle. Microsoft Corporation Whitepaper, March 2005. Security Engineering and Communications, Security Business and Technology Unit, Microsoft Corporation.

[LJ97] Ulf Lindqvist and Erland Jonsson. How to systematically classify computer security intrusions. In *IEEE Symposium on Security and Privacy*, pages 154–163, May 1997.

[LKMB05] Hee-Young Lim, Young-Gab Kim, Chang-Joo Moon, and Doo-Kwan Bai. Bundle authentication and authorization using XML security in the OSGi service platform. In *Fourth Annual ACIS International Conference on Computer and Information Science*, pages 502 – 507, 2005.

[LL05] V. Benjamin Livshits and Monica S. Lam. Finding security vulnerabilities in Java applications with static analysis. In *USENIX Security Symposium*, 2005.

[Llo05] Wes J. Lloyd. A Common Criteria based approach for COTS component selection. *JOURNAL OF OBJECT TECHNOLOGY, Special issue: 6th GPCE Young Researchers Workshop 2004,*, 4(3):27–34, 2005. Special issue: 6th GPCE Young Researchers Workshop 2004,.

[Lon05] Fred Long. Software vulnerabilities in Java. Technical Report CMU/SEI-2005-TN-044, Carnegie Mellon University, October 2005.

[Lor01] Berin Loritsch. Developing with Apache Avalon. Apache Avalon Project, Developper Guide, December 2001.

[LY99] Tim Lindholm and Frank Yellin. *The Java^{TM} Virtual Machine Specification (2nd Edition)*. Prentice Hall PTR, 1999. ISBN-13: 978-0201432947.

[Mar96] Robert C. Martin. The dependency inversion principle. *C++ Report*, 8, 1996.

[Mar05] Robert A. Martin. Transformational vulnerability management through standards. *CrossTalk, The Journal of Defense Software Engineering*, pages 12–15, May 2005.

[McC02] Chris McCown. Framework for secure application design and development: Foundations, principles and design guidelines. GIAC Certification: Practical Assignment v1.4, November 2002. White Paper, http://www.sans.org.

[McG04] Gary McGraw. Software security. *IEEE Security & Privacy Magazine*, 2(2):80–83, March-April 2004.

[McG06] Gary McGraw. *Software Security - Building Security In*. Pearson Education, London, January 2006. ISBN-13: 978-0321356703.

[McI69] M. Douglas McIlroy. Mass produced software component. In *NATA Conference on Software Engineering*, pages 138–155, 10 1969.

[MEL01] Andrew P. Moore, Robert J. Ellison, and Richard C. Linger. Attack modeling for information security and survivability. Technical Report CMU/SEI-2001-TN-001, CMU/SEI, March 2001.

[Mey03] Bertrand Meyer. The grand challenge of trusted components. In *International Conference on Software Engineering (ICSE'25))*, May 2003.

[MF98] Gary Mcgraw and Edward Felten. Twelve rules for developing more secure Java code. *JavaWorld.com*, January 1998.

[MK01] Dave Marples and Peter Kriens. The Open Services Gateway initiative: an introductory overview. IEEE Communications Magazine, December 2001.

[MM97] Thomas J. Mowbray and Raphael C. Malveau. *Corba Design Patterns*. John Wiley & Sons, January 1997. ISBN-13: 978-0471158820.

[MRV03] Pierre-Etienne Moreau, Christophe Ringeissen, and Marian Vittek. A pattern matching compiler for multiple target languages. In G. Hedin, editor, *12th Conference on Compiler Construction*, number 2622 in LNCS, pages 61–76. Springer-Verlag, May 2003.

[MTS04] Mark S. Miller, Bill Tulloh, , and Jonathan Shapiro. 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.

[MW08] Meriam-Webster. Meriam-webster online repository, 2008. http://www.merriam-webster.com/.

[nas92] Software assurance standard. Technical Report Standard No. NASA-STD-2201-93, National Aeronautics and Space Administration (NASA), Wahsington,DC, November 1992.

[Nec97] George C. Necula. Proof-carrying code. In *Conference Record of POPL '97: The 24th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 106–119, Paris, France, jan 1997.

[NL96] George C. Necula and Peter Lee. Safe kernel extensions without run-time checking. In *Second Symposium on Operating Systems Design and Implementation (OSDI '96)*, October 1996.

[Noo06] Davis Noopur. Secure software development life cycle processes. Technical Report CMU/SEI-2005-TN-024, Software Engineering Institute, July 2006.

[NP89] Peter G. Neumann and Don B. Parker. A summary of computer misuse techniques. In *12th National Computer Security Coifererice*, page 3961107, Baltimore, Maryland, USA, October 1989.

[NP06] Renaud Pawlakand Carlos Noguera and Nicolas Petitprez. Spoon: Program analysis and transformation in java. Technical Report 5901, INRIA, Mai 2006.

[OSG01] OSGi Alliance. Security architecture specification. OSGi Alliance RFC 18, June 2001.

[OSG08] OSGi Alliance. Osgi service platform release 4 version 4.2 - early draft. Revision 1, August 2008.

[OTF05] Frédéric Ogel, Gaël Thomas, and Bertil Folliot. Support efficient dynamic aspects through reflection and dynamic compilation. In *20th Annual ACM Symposium on Applied Computing (SAC'05, PSC)*, pages 1351–1356, Santa Fe, New Mexico, USA, March 2005.

[OWA07] OWASP Foundation. Owasp testing guide - V 2, 2007. Creative Commons Attribution-ShareAlike 2.5 License.

[OWA08] OWASP Foundation. Owasp code code guide - RC 2, 2008. Creative Commons Attribution-ShareAlike 2.5 License.

[Par09] Pierre Parrend. Enhancing automated detection of vulnerabilities in java components. In *Forth International Conference on Availability, Reliability and Security (AReS 2009)*, Fukuoka, Japan, March 2009.

[PF] Pierre Parrend and Stéphane Frénot. Security benchmarks of osgi platforms: Toward hardened osgi. *Software: Practice & Experience*. Accepted for publication.

[PF06a] Pierre Parrend and Stéphane Frénot. Dependability for component systems deployment. Poster, first EuroSys Conference, April 2006.

[PF06b] Pierre Parrend and Stéphane Frénot. Secure component deployment in the OSGi^{TM}release 4 platform. Technical Report RT-0323, INRIA, June 2006.

[PF07a] Pierre Parrend and Stéphane Frénot. Java components vulnerabilities - an experimental classification targeted at the OSGi platform. Research Report RR-6231, INRIA, 06 2007.

[PF07b] Pierre Parrend and Stéphane Frénot. Supporting the secure deployment of OSGi bundles. In *First IEEE WoWMoM Workshop on Adaptive and Dependable Mission- and bUsiness-critical mobile Systems, Helsinki, Finland*, June 2007.

[PF08a] Pierre Parrend and Stéphane Frénot. Classification of component vulnerabilities in Java service oriented programming (SOP) platforms. In *Conference on Component-based Software Engineering (CBSE'2008)*, volume 5282/2008 of *LNCS*, Karlsruhe, Germany, October 2008. Springer Berlin / Heidelberg.

[PF08b] Pierre Parrend and Stéphane Frénot. Component-based access control: Secure software composition through static analysis. In *Software Composition (SC'2008)*, volume 4954/2008 of *LNCS*, pages 68–83. Springer Berlin / Heidelberg, March 2008.

[PF08c] Pierre Parrend and Stéphane Frénot. More vulnerabilities in the Java/OSGi platform: A focus on bundle interactions. Technical Report RR-6649, INRIA, September 2008.

[PF08d] Pierre Parrend and Stéphane Frénot. Vérification automatique pour l'exécution sécurisée de composants java. *Numéro spécial de la revue L'Objet - Composants, Services et Aspects : techniques et outils pour la vérification*, 2008.

[PG03] M. P. Papazoglou and D. Georgakopoulos. Service-Oriented Computing. *Communications of the ACM*, 46(10), 2003.

[PGFU07] Pierre Parrend, Samuel Galice, Stéphane Frénot, and Stéphane Ubeda. Identity-based cryptosystems for enhanced deployment of OSGi bundles. In *IARIA International Conference on Emerging Security Information, Systemsand Technologies (SecurWare)*, October 2007.

[PHK05] Marco Pistoia, Ted Habeck, and Larry Koved. Enabling Java 2 runtime security with eclipse plug-ins - analyzing security requirements for OSGi-enabled platforms. In *OSGi Alliance Developer Forum*, 10 2005.

[PO08] Karl Pauls and Marcel Offermans. Building secure OSGi applications. Tutorial at EclipseCon'2008, 2008. http://www.eclipsecon.org/2008/?page=sub/&id=300.

[RAF04] Nick Rutar, Christian B. Almazan, and Jeffrey S. Foster. A comparison of bug finding tools for Java. In *The 15th IEEE International Symposium on Software Reliability Engineering (ISSRE'04)*, pages 245–256, November 2004. Saint-Malo, Bretagne, France.

[Ran96] Brian Randell. The 1968/69 nato software engineering reports. In *Dagstuhl-Seminar 9635: "History of Software Engineering" Schloss Dagstuhl*, August 1996.

[RF07] Yvan Royon and Stéphane Frénot. Multiservice home gateways: Business model, execution environment, management infrastructure. *IEEE Communications Magazine*, October 2007.

[RFMP06] David G. Rosado, Eduardo Fernandez-Medina, and Mario Piattini. Comparison of security patterns. *International Journal of Computer Science and Network Security*, 6(2):139–146, 2006.

[Rom01] Sasha Romanosky. Security design patterns, November 2001. http://www.cgisecurity.com/lib/securityDesignPatterns.html.

[RPF+07] Yvan Royon, Pierre Parrend, Stéphane Frénot, Serafeim Papastefano, Humberto Abdelnur, and Dirk Van de Poel. Multi-service, multi-protocol management for residential gateways. In *BroadBand Europe*, December 2007.

[Sar97] Vijay Saraswat. Java is not type-safe. AT&T Research Whitepaper, August 1997.

[Sch96] Bruce Schneier. *Applied Cryptography: Protocols, Algorithms, and Source Code in C*. Wiley; 2nd edition, October 1996. ISBN-13:978-0471117094.

[Sch00] Fred B. Schneider. Enforceable security policies. *ACM Transactions on Information System Security*, 3(1):30–50, 2000.

[Sch02a] Marc Schoenefeld. Security aspects in Java bytecode engineering. In *Blackhat Briefings 2002, Las Vegas*, 2002.

[Sch02b] Markus Schumacher. Security patterns and security standards - with selected security patterns for anonymity and privacy. In *European Conference on Pattern Languages of Programs (EuroPLoP)*, Irsee, Germany, 2002.

[Sch03] Markus Schumacher. *Security Engineering With Patterns: Origins, Theoretical Models, and New Applications*. LNCS. September 2003. ISBN-13: 978-3540407317.

[SGM02] Clemens Szyperski, Dominik Gruntz, and Stephan Murer. *Component Software - Beyond Object-Oriented Programming*. Addison-Wesley / ACM Press, 2002. 589 pages.

[SH00] Viren Shah and Frank Hill. An aspect-oriented security framework: Lessons learned. In *AOSDSEC Workshop. in conjunction with AOSD 04, Lancaster, U.K.*, 2000.

[SH05] Robert C. Seacord and Allen Householder. A structured approach to classifying security vulnerabilities. Technical Report CMU/SEI-2005-TN-003, Carnegie Mellon University - Software Engineering Institute, January 2005.

[Sof08] Software Assurance Forum for Excellence in Code (SAFECode). Software assurance: An overview of current industry best practices. SAFECode Whitepaper, February 2008.

[Sri06] Manoj Srivastava. Security enhanced virtual machines - an introduction and recipe, April 2006.

[SS73] Jerome H. Saltzer and Michael D. Schroeder. The protection of information in computer systems. In *Fourth ACM Symposium on Operating System Principles*, October 1973.

[SSLA06] Xinyue Song, M. Stinson, R. Lee, and P. Albee. An approach to analyzing the Windows and Linux security models. In *5th IEEE/ACIS International Conference on Computer and Information Science (ICIS-COMSAR)*, 2006.

[Ste06] Martijn Stevenson. Asbestos: Operating system security for mobile devices. Master's thesis, Massachusetts Institute of Technology, May 2006.

[Sun] Sun Microsystems, Inc. Permissions in the Java^{TM}2 SDK. Online documentation. http://java.sun.com/j2se/1.4.2/docs/guide/security/permissions.html.

[Sun03] Sun Microsystems, Inc. JAR file specification. Sun Java Specifications, 2003. http://java.sun.com/j2se/1.5.0/docs/guide/jar/jar.html.

[Sun04] Sun Microsystems, Inc. Enterprise JavaBeans specification documentation 3.0 final release. Sun JSR 220 FR Final Release, 2004.

[Sun07] Sun Microsystems Inc. Secure coding guidelines for the Java programming language, version 2.0. Sun Whitepaper, 2007. http://java.sun.com/security/seccodeguide.html.

[SVS02] Alberto Sillitti, Tullio Vernazza, and Giancarlo Succi. Service oriented programming: A new paradigm of software reuse. In *7th International Conference on Software Reuse (ICSR-7)*, pages 269–280. Springer-Verlag, 2002.

[TCM05] Katrina Tsipenyuk, Brian Chess, and Gary McGraw. Seven pernicious kingdoms: A taxonomy of software security errors. *IEEE Security & Privacy*, 3(6):81–84, November/December 2005.

[TGCF08] Gaël Thomas, Nicolas Geoffray, Charles Clement, and Bertil Folliot. Designing highly flexible virtual machines: the JnJVM experience. In *Software: Practice & Experience*. John Wiley & Sons, Ltd., 2008.

[THB06] Andrew S. Tanenbaum, Jorrit N. Herder, and Herbert Bos. Can we make operating systems reliable and secure? *IEEE Computer*, 39(5):44–51, May 2006.

[The02] The Last Stage of Delirium. Research Group. Java and Java Virtual Machine. security vulnerabilities and their exploitation techniques. In *Black Hat Briefings*, 2002.

[the08] Software that makes software better. *The Economist*, March 2008. http://www.economist.com/search/displaystory.cfm?story_id=10789417, read on 2008/08/01.

[Tho84] Ken Thompson. Reflection on trusting trust. *Communication of the ACM*, 27(8):761–763, August 1984.

[VBC01] John Viega, J.T. Bloch, and Pravir Chandra. Applying aspect-oriented programming to security. *Cutter IT Journal*, 14(2):31–39, February 2001.

[VBKM00] John Viega, J. T. Bloch, Yoshi Kohno, and Gary McGraw. Its4: A static vulnerability scanner for C and C++ code. In *16th Annual Computer Security Applications Conference (ACSAC '00)*, page 257. IEEE Computer Society, 2000.

[VGMC96] Jeffrey M. Voas, Anup K Ghosh, Gary McGraw, and F. Charron. Defining an adaptive software security metric from a dynamic software failure tolerance measure. In *11th Annual Conference on Computer Assurance (COMPASS'96)*, pages 250–263, 1996.

[VM01] John Viega and Gary McGraw. *Building Secure Software - How to Avoid Security Problems the Righ Way*. Number 528 in Professional Computing Series. Addison-Wesley Professional, 2001. ISBN-13: 978-0201721522.

[VMMF00] John Viega, Gary McGraw, Tom Mutdosch, and Edward W. Felten. Statically scanning Java code: Finding security vulnerabilities. *IEEE Software*, 17(5):68–74, 2000.

[VNC+06] Paulo E. Verissimo, Nuno F. Neves, Christian Cachin, Jonathan Poritz, David Powell, Yves Deswarte, Robert Stroud, , and Ian Welch. Intrusion-tolerant middleware: The road to automatic security. *IEEE Security & Privacy*, 4(4):54–62, July/August 2006.

[Voa98] Jeffrey M. Voas. COTS software: The economical choice? *IEEE Software*, 15(2):16–19, March/April 1998.

[WF98] Dan S. Wallach and Edward W. Felten. Understanding Java stack inspection. In *IEEE Symposium on Security and Privacy, Oakland, CA, USA*, pages 52–63, May 1998.

[Whe03] David A. Wheeler. *Secure Programming for Linux and Unix HOWTO*. 2003. GNU Free Documentation License (GFDL).

[XA06] Yichen Xie and Alex Aiken. Static detection of security vulnerabilities in scripting languages. In *15th conference on USENIX Security Symposium (USENIX-SS'06)*, pages 13–13, Berkeley, CA, USA, 2006. USENIX Association.

[Yam05] Ikuo Yamakasi. Monitoring and managing computer resource usage on OSGi frameworks. In OSGi Alliance, editor, *OSGi Alliance Developer Forum*, 10 2005.

[YWM08] Nobukasu Yoshioka, Hironori Washizaki, and Katsuhisa Maruyama. A survey on security patterns. *Progress in Informatics*, 5:35–47, 2008.