SOFTWARE—PRACTICE AND EXPERIENCE
Softw. Pract. Exper. 2009; **39**:471–499
Published online 10 November 2008 in Wiley InterScience (www.interscience.wiley.com). DOI: 10.1002/spe.906

Security benchmarks of OSGi platforms: toward Hardened OSGi

P. Parrend[1,*,†] and S. Frenot[2]

[1]*Software Engineering, FZI Forschungszentrum Informatik, Haid-und-Neu-Strasse 10-14, 76131 Karlsruhe, Germany*
[2]*INRIA Amazones/CITI, INSA-Lyon, F-69621, France*

SUMMARY

OSGi platforms are extensible component platforms, i.e. they support the dynamic and transparent installation of components that are provided by third party providers at runtime. This feature makes systems built using OSGi extensible and adaptable, but opens a dangerous attack vector that has not been considered as such until recently. Performing a security benchmark of the OSGi platform is therefore necessary to gather knowledge related to the weaknesses it introduces as well as to propose enhancements. A suitable *Vulnerability Pattern* is defined. The attacks that can be performed through malicious OSGi components are identified. Quantitative analysis is then performed so as to characterize the origin of the vulnerabilities and the target and consequences of the attacks. The assessment of the security status of the various implementations of the OSGi platform and of existing security mechanisms is done through a metric we introduce, the *Protection rate* (PR). Based on these benchmarks, OSGi-specific security enhancements are identified and evaluated. First recommendations are given. Then evaluation is performed through the PR metric and performance analysis. Lastly, further requirements for building secure OSGi platforms are identified. Copyright © 2008 John Wiley & Sons, Ltd.

Received 11 February 2008; Revised 15 September 2008; Accepted 17 September 2008

KEY WORDS: software security assurance; software vulnerabilities; security benchmark; OSGi component framework; component platform; dependability

1. INTRODUCTION

The OSGi platform is a Java-based component platform. It enables managing several applications on the same virtual machine as well as installing, starting, stopping and uninstalling the so-called

*Correspondence to: P. Parrend, Software Engineering, FZI Forschungszentrum Informatik, Haid-und-Neu-Strasse 10-14, 76131 Karlsruhe, Germany.
†E-mail: parrend@fzi.de

Contract/grant sponsor: ANR-LISE Project; contract/grant number: ANR-07-SESU_007

bundles at runtime. These features characterize what we call *Extensible Component Platforms*. Other examples are Java MIDP [1], Android[‡], which is built on a Java/Linux stack and the .Net Platform by Microsoft[§].

Runtime extensibility provides a radical improvement in applications' flexibility and management. However, it also opens a brand new attack vector: the dynamic and transparent installation of bundles that may be provided by unknown or at least uncontrolled providers. In particular, bundles are automatically installed to provide all dependencies of an application that is being installed. This attack vector can be exploited in two ways. Either malicious developers act in the development team of the bundle provider or uncontrolled providers publish malicious bundles that are dynamically installed on the victim platform. The consequences can be undue access to other bundles, or to the platform itself, or denial of service (DOS), for instance in case of massive resource consumption. These threats highlight the limitations of the Java security model for multi-application platforms, as discussed in Section 2.3.

In order to improve the security status of existing OSGi platforms, a set of requirements are identified. First, a better knowledge of the security risks that are bound with OSGi is necessary. Few works have been carried out on the subject so far. Second, performing a security benchmarking of the OSGi platform is mandatory to make developers aware of the actual weaknesses of the development environment they are working on. Benchmarking is performed on the most widespread open source implementations of OSGi because of their availability. Similar analysis could be performed for other implementations, in particular industrial ones, by executing the malicious bundles we developed and verifying whether the implementations resist the attack. Thirdly, recommendations for building more secure implementations of the OSGi platform are required so as to patch the identified weaknesses and to not let them be open to attacks.

The remainder of the paper is organized as follows. Section 2 defines the OSGi platform and gives an overview of its security strengths and weaknesses. Section 3 presents the catalog of vulnerabilities that we identified in the OSGi platform, as well as security benchmarking results for open source implementations. Based on these data, recommendations for building Hardened OSGi platforms are given and evaluated in Section 4. Section 5 concludes this study.

2. THE OSGI PLATFORM

Performing security benchmarking for a given system implies that this system is properly understood, and that its key security properties are identified.

The OSGi platform is an overlay to the Java Virtual Machine (JVM) that supports the packaging of applications with their own Life-Cycle. Applications are provided as 'bundles', which are executed in dedicated class loaders, and can thus be isolated from each other. The bundles can be managed, i.e. installed, started, stopped, updated and uninstalled, through a dedicated shell. With OSGi, Java is no longer a single-application execution engine, but behaves more like an operating system that supports the execution of several, potentially interdependent, applications.

[‡] http://code.google.com/android/.
[§] http://msdn.microsoft.com/netframework/.

DOI: 10.1002/spe

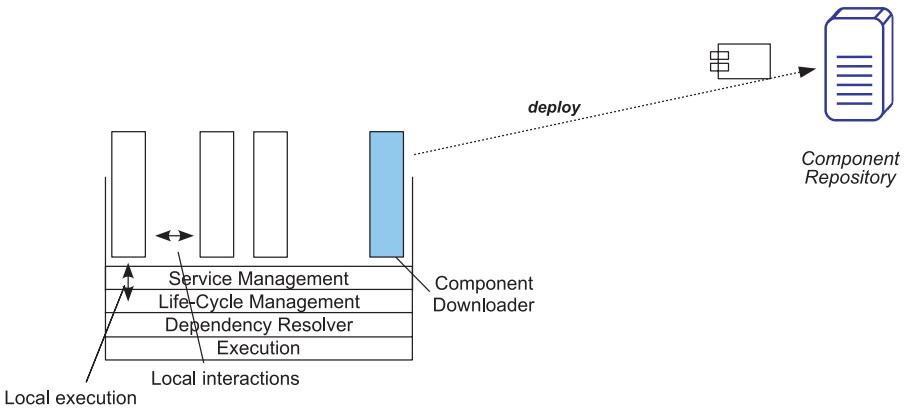

Figure 1. Platform model for OSGi.

Core elements of OSGi-based systems, the platform itself and the *bundles* are therefore first defined. Next, these definitions are formalized as taxonomies to support systematic analysis. Third, an overview of known strengths and weaknesses of the OSGi platform is provided.

2.1. Structure

The structure of OSGi-based systems is defined in OSGi specifications Release 4.1 [2]. These specifications present the architecture of the OSGi platform and define the format of bundles.

2.1.1. The platform

The OSGi platform is a compound of four layers that are running on top of a JVM: the security layer, the module Layer, the Life-Cycle Layer and the service layer (Figure 1).

The JVM can be any JVM that provides an over-set of the Java connected device configuration (CDC) foundation profile[¶]. Standard JVM from 1.3 upwards are supported.

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 very similar to the digital signature of Jar files. However, verification criteria are stronger, so specific verification tools have to be used [3].

The module layer acts as a dependency resolver between the bundles within the platform. Actually, each bundle is executed in a specific class loader, which enables class isolation between the various applications. In order to enable the interactions between bundles, dependencies have to be explicitly defined, i.e. the names 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

[¶] http://java.sun.com/javame/technology/cdc.

DOI: 10.1002/spe

are 'slave' bundles that are used to provide configuration informations or context-dependent code to a 'host' bundle. They cannot be used independently.

The Life-Cycle Layer deals with the installing, starting, stoping, updating and uninstalling of the bundles. These advanced management features are 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 service layer provides the support of Service-Oriented Programming [4]. Bundles can publish services in a common *Context* under the form of Java interfaces. They are frequently enriched with specific properties that enable service search using the LDAP request format [5]. These services can be dynamically discovered and used by other bundles without requiring dependencies being 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 [6]. It performs discovery and download new bundles over the Internet. Metadata format for the Bundle Repositories is named OBR v2 (Open Bundle Repository v2).

2.1.2. The bundles

OSGi bundles are Java Archive (Jar) files [7] extended to support specific operations such as 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 2.

Meta-data are defined in the **Manifest** 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 the standard Java code. It can be either 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

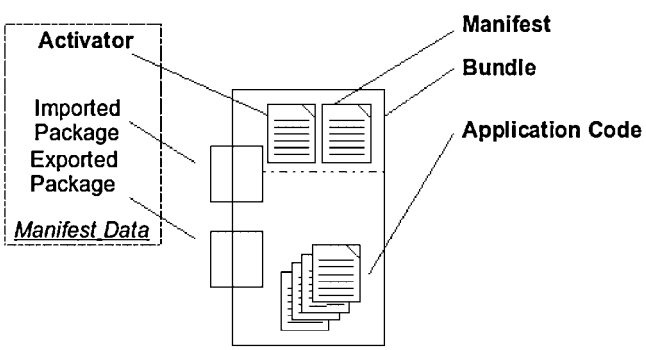

Figure 2. Component model for OSGi.

DOI: 10.1002/spe

feature breaks all security benefits of the JVM, 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 of 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.

2.2. Taxonomies for OSGi-based systems

Systematic security analysis of a system implies that a model of this system is available. We therefore define two taxonomies that represent the core elements of an OSGi-based system, the platform and the bundle.

The taxonomy of the elements of an OSGi platform is presented in Figure 3. It is subdivided into three subsystems: the operating system, the JVM and the OSGi platform. The JVM can be subdivided between the Runtime, which is the execution engine, and the Classpath, which is built up by the standard API classes. The JVM is specified by Lindholm and Yellin [8]. The OSGi platform is built up by the module layer, the Life-Cycle Layer and the service layer. The security layer is split between these three internal layers.

The taxonomy of the elements of OSGi bundle elements is presented in Figure 4. These elements can be classified into two main categories: intra-bundle structure and inter-bundle interactions. The intra-bundle structure is composed of the following items: bundle archive, Manifest file, Activator Class, Native Code, Java Language, Java API calls and OSGi API calls. Inter-bundle-specific interactions can be performed through OSGi services or OSGi fragments. Interactions through packages export and import are not explicitly considered since they can be considered either as related to Meta-data (Manifest file) or as standard Java calls.

2.3. Security strengths and weaknesses of OSGi platforms

The OSGi platform is deemed to be highly secure by its proponents. Actually, it has several built-in features that makes it a serious candidate for developing secure execution platforms. However, the

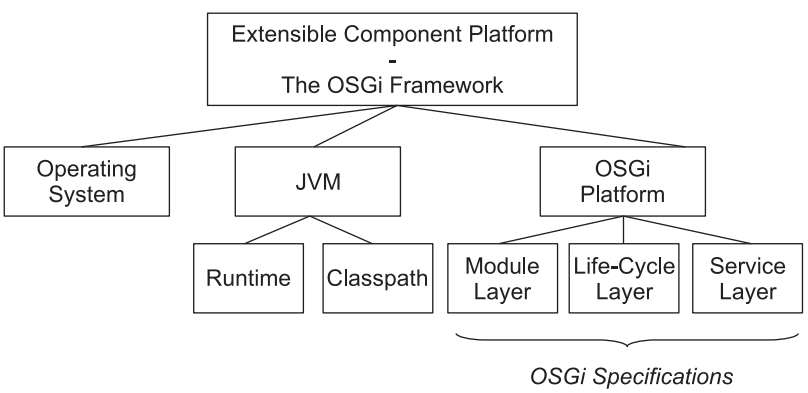

Figure 3. Taxonomy of the elements of an OSGi platform.

DOI: 10.1002/spe

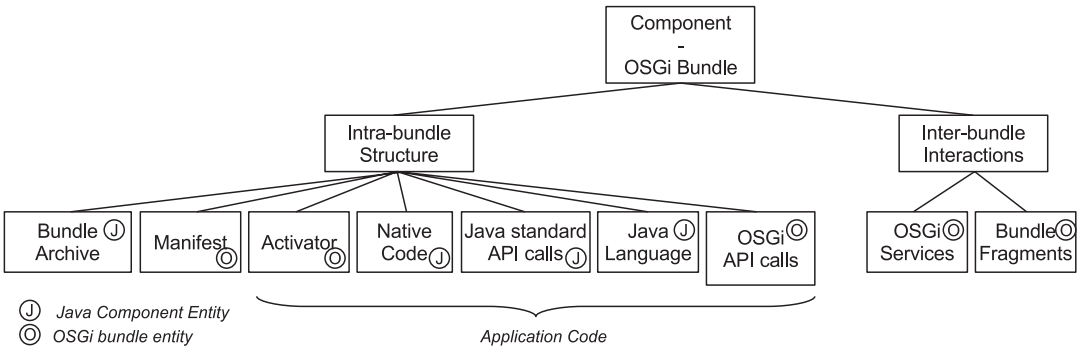

Figure 4. Taxonomy of the elements of an OSGi bundle.

state of current implementations shows that this matter is still poorly addressed in spite of the strong security requirements of the common OSGi use cases.

2.3.1. Why is OSGi potentially secure

The two major arguments in favor of the strong potential of OSGi as a secure execution platform are the following: First, OSGi is executed on top of a JVM. Second, it has been designed to support a proper namespace isolation between bundles.

The Java language and Virtual Machine is one of the execution environment that has been subject to extensive tests and validation since its publication (see, for instance, [9–11]). It has been originally designed to support the safe execution of fully untrusted code, such as Web Applets [12]. Four mechanisms support this feature [13]: Type Safety of the language, Garbage Collection, which prevents user-defined memory management, which is error prone, Bytecode verification to ensure that executed programs are compliant with the language specification, and Sandboxing, which prevents the access to sensible resources such as the network or file system. Two additional mechanisms are meant to support secure modular programming: secure class loaders, which make it possible to load several modules that cannot interact with each other, and customizable permissions, which enable flexible Sandboxing by allowing only necessary rights to the code that needs it [14]. Optional tools can also be used for building secure Java environments: Isolates and resource management. Isolates [15] are defined by the Java Specification Request (JSR) group 121. Their objective is to support parallel execution of several Java applications with the same level of isolation that is available for applications running on different Virtual Machines. One implementation of this isolation API is the Java Multi-User Virtual Machine (MVM), which is released as a research prototype by SUN in the Barcelona Project [16]. Isolates are complemented with the Resource Consumption Management (RCM) API, proposed by the JSR 284 [17]. The RCM API supports the management of usual system resources, such as CPU and heap CPU, but also enables defining application specific resources. Resource Management policies can be defined for each modeled resource.

On top of this sound basis, the OSGi platform extends the component-based approach to permissions. First, it provides a clean implementation of class loaders, which prevents the application

DOI: 10.1002/spe

designer from misusing or by-passing them. Secondly, it defines specific permissions that enable full control of the interactions between the bundles: dependencies at the package level and at the service level can be authorized or prevented. Management capability from one bundle to the others can also be restricted.

This clean management model makes it possible to claim that the OSGi platform is security aware, and that it provides advanced security features that are available on few others. However, we will show that these security efforts in the specifications are not sufficient to provide a secure programming model for a multi-application platform.

2.3.2. Why is OSGi currently not secure

As presented in the taxonomy for the elements of an OSGi platform, security flaws can originate in three parts: the Operating System, the JVM and the OSGi platform itself.

Flaws in Operating Systems are out of the scope of this study. A reference for the Unix System can for instance be found in [18]. It will therefore not be further discussed, all the more as the Java Sandbox is considered since the inception of Applets as a good protection against malicious code. Unless it is explicitly authorized, no bundle can have access to the underlying system.

Flaws in the Java Language and Virtual Machine can be divided into two categories. The first one is built up by raw flaws that enable code to harm the system. The second is related to the security model of Java. This latter has originally not been designed for multi-application systems and proves to fall short in providing strong isolation between mutually untrusted applications.

Several unexpected and somewhat illogical behavior is identified in the Java Language [19,20]. This can lead to unsafe behavior, difficult to track infinite loops and memory leaks. Moreover, several failures are reported to take place in JVMs when they are under heavy workload [21]. In particular, optimizations such as Just-in-Time compilers or Garbage Collectors are important sources of weaknesses. Both of these failure types can be used to launch DoS attacks against any Java execution Platform.

Java has been designed for secure execution of single applications, such as Web Applets [12]. It supports Sandboxing of untrusted applications, i.e. access isolation from the rest of the system. It also enables to prevent DOS attacks, insofar as an application consuming a lot of memory or CPU can be killed together with the JVM with no further impact on other applications. This cannot be achieved with multi-application platforms, where killing the platform implies that all running applications are also shut down.

Current security mechanisms for secure modular systems are an extension of these mechanisms for single-application support. In particular, namespace isolation is supported thanks to the class loading mechanism, but resource isolation, i.e. isolation of available CPU and memory is only supported in specific implementations of the JVM, such as the Multi-User Virtual Machine [16], or through the RCM API. Consequently, requirements for proper multi-applications Java-based system are only partially addressed so far.

Flaws in the OSGi platform have hardly been studied so far. Although the platform has been subject to an important security effort, the flexibility brought into systems do not come without new risks. The actual security model that is currently supported therefore needs to be clarified.

As an execution platform that targets networked environments, the OSGi platform has been subject to an important effort to enhance its security. Specifications introduce a specific security layer to support the verification of the digital signature of bundles and to manage OSGi specific execution

DOI: 10.1002/spe

permissions [2]. Moreover, recommendations for building secure distributed OSGi-based systems are detailed in a dedicated OSGi Request for Comments [22]. Nonetheless, these specifications deal with the integration of secure remote connection mechanisms rather than with secure execution of code. In particular and to the best of our knowledge, no implementation of the specified security layer is so far available in open source implementations of the platform. An alternative security layer is proposed by Huang *et al.* [23]. It involves an OSGi-internal Intrusion Detection System. By tracking individual calls, the 'Advanced OSGi Security Layer' (AOSL) is a powerful tool for preventing many attacks. One limitation of the study is the lack of optimization: the performance overhead of AOSL is greater than 50%. However, the fact that little effort has been spent also means that an important enhancement can still be brought in. The second limitation is the fact that AOSL is built on top of JVM Profiler Interface [24], which is deprecated and should be replaced by JVM Tool Interface [25]. Differences between default Java specifications and OSGi ones are not yet considered in mainstream projects, for instance, what are the criteria of validation of digital signature of bundles [3].

Specific security implications of OSGi features have also been neglected so far. In particular, the bundle Life-Cycle support and the Bundle Repository tool enable the dynamic and transparent installation of bundles provided by possibly unknown developers through public repositories. This flexilibity is the very enhancement brought by the OSGi platform, but it opens a new and easily exploitable attack vector. Understanding the risks it implies is necessary to adapt the use cases of the OSGi platforms to the security status that is actually provided. This is what this study is intended to do.

The security model that is currently supported by OSGi-based systems is as follows: A platform must only accept signed bundles if it wants to have any control on the origin of the executed code. The bundle signer is thus liable for any actions of the bundles it provides. Since bundles are signed but never encrypted, no confidential information should be transmitted under this form. Any system that requires both OSGi bundle management flexibility and more complete security guarantees must use *ad hoc* security mechanisms.

2.3.3. Conclusion

OSGi-based systems can be divided into two main elements: the platform itself and the bundles. Although it is designed with several efficient embedded security mechanisms, the platform is likely to contain vulnerabilities as any software system would. Bundles can contain code that exploits these vulnerabilities to harm the system where they are installed.

As the use of OSGi increases in environments such as Smart Homes or Web Servers, the potential security flaws may soon be an efficient way of performing costly attacks. A better knowledge of existing flaws in the current specifications and open source implementations of the OSGi platform is therefore required.

3. SECURITY BENCHMARKING OF OSGI PLATFORMS

Security benchmark of existing OSGi platforms is required to gather knowledge that can be exploited to enhance their security status. Requirements are twofold: building a catalog of the vulnerabilities that exist in the platform, and performing a benchmarking of the open source OSGi platforms

DOI: 10.1002/spe

against catalog's vulnerabilities. We merely perform the analysis of open source implementations of the OSGi platform, because our goal is to help in enhancing them, and not to criticize commercial products. This keeps the benchmark vendor-neutral, and provides proprietary system providers with sufficient data to perform their own benchmarks. These latter are often tuned for specific environments and benchmarks should take these constraints into account, whereas open source projects do not make such assumption. Moreover, the access to the code of such platforms is not necessarily free or legal.

Building such a catalog requires a Vulnerability Pattern to be available that standardizes the data that is gathered for each vulnerability and supports security benchmarking for the considered platform type, i.e. the OSGi platforms. This pattern is first introduced. Next, malicious bundles that threaten OSGi elements are listed. Lastly, security benchmarks of the most current implementations of the OSGi platform are presented.

3.1. The vulnerability pattern for security benchmarking of OSGi platforms

Gathering information relative to vulnerabilities implies that a common pattern is available and defines the type of data necessary to handle them. Existing patterns and languages for vulnerability documentation are first discussed. Since none of them that supports security benchmarking for OSGi platforms is available, a dedicated *Vulnerability Pattern* is introduced. Its use is highlighted through an example.

3.1.1. Patterns and languages for documenting vulnerabilities

If the use of patterns in Software Engineering is commonplace since the 'Gang of Four' book [26], the application of this approach to security engineering is more recent [27]. Patterns and languages for documenting vulnerabilities can be parted into three categories: informative patterns, for documentation only, vulnerability and exposure patterns for operational support, which are integrated into administration tools and languages for vulnerability assessment and resolution, which allow advanced vulnerability processing.

- Informative patterns are meant for broad disclosure of vulnerability data. They provide the minimal information that is required by systems administrators to perform security update. Example of such patterns are the Rocky Heckman pattern[‖], the CERT (Computer Emergency Response Team) pattern and the CIAC[**] (US Department of Energy) pattern.
- *VEDEF* (Vulnerability and Exposure Description and Exchange Format) is a set of machine-readable descriptions of the vulnerabilities and of the remediation process. In order to be exploitable, they should contain the following data fields [28]: description of the platform(s) that is(are) affected, description of the nature of the problem, description of the likely impact if the Vulnerability and/or Exploit were triggered, available means of remediation, disclosure restrictions. One of the *de facto* standard is the MITRE CVE[††] (Common Vulnerability and Exposures) pattern. Other examples are the XML Common Format for Vulnerability

[‖] http://www.rockyh.net/.
[**] http://www.ciac.org/ciac/index.html.
[††] http://cve.mitre.org/.

DOI: 10.1002/spe

Advisories, by the EISPP [29], the Common Advisory Interchange Format (CAIF), by the RUSCERT[‡‡], and the Advisory and Notification Markup Language (ANML), by OpenSec. More specific propositions, for instance, the Application Vulnerability Description Language (AVDL), by OASIS, are defined for web applications.
- The third type of patterns is defined to support both assessment and remediation processes. One example is the OVAL Language (Open Vulnerability and Assessment Languages) [30].

The VEDEF Vulnerability Pattern type contains several useful information for performing OSGi platform security benchmarking such as the description of the vulnerabilities, of the affected platforms and of the means of remediation. OVAL is very similar to our proposition insofar as it supports penetration testing. The main difference is that OVAL is used as a support in a series of security assessment and remediation tools, whereas our *Vulnerability Pattern* only supports assessment and analysis, and is not integrated in tools. However, no existing pattern supports component-platform-specific properties such as Life-Cycle information or benchmarking specific information such as reference to attack code implementation.

3.1.2. The vulnerability pattern for security benchmarking of OSGi platform

It falls in the category of vulnerability and exposure description for operational support, with extension to take penetration testing as well as component Life-Cycle into account. It contains the following sections:

- Vulnerability Reference, to provide the unique identifier of the vulnerability, and a proper classification according to the taxonomies that are introduced in Sections 2.2 and 3.3.
- Vulnerability Description, to enable the detailed presentation of the behavior of the vulnerability, as in the Morbray pattern [31].
- Protections, being actual or potential ones that may be efficient.
- Vulnerability Implementation, the development status of the proof-of-concept malicious bundles.

The benefits of this Vulnerability Pattern are as follows: It is the only vulnerability pattern that considers information related to the Life-Cycle of components, which is actually surprising with regard to the development of COTS-based systems. It includes the reference to one or several proof-of-concept implementations of malicious bundles. It supports the elicitation not only of available security mechanisms, but also of potential ones, so as to support the identification of required development efforts for security protections when a specific platform is benchmarked. Lastly, specific values are predefined for each field of the 'Vulnerability Reference' section, according to the taxonomies defined in this study. This provides a common language for describing vulnerabilities in a coherent manner.

However, the Vulnerability Pattern shows some limitations. It could be integrated into existing patterns, such as the OVAL one, as a platform-specific extension. This would make its integration with security management tools using OVAL possible, but it would also introduce an important

[‡‡]http://cert.uni-stuttgart.de/files/caif/requirements/split/requirements.html.

DOI: 10.1002/spe

development requirement to adapt tools to this OSGi-specific extension. This is clearly out of the scope of this paper.

3.1.3. Example

In order to highlight the role of the defined Vulnerability Pattern, we now present an example of vulnerability: the 'Management Utility Freezing—Infinite Loop' vulnerability.

This vulnerability consists in the presence of an infinite loop in the activator of a given Bundle, which causes the platform management tool (often an OSGi shell) to freeze. The presence of infinite loops as a vulnerability is given by Bloch in 'Java Puzzlers—Traps, Pitfalls and Corner Cases', puzzlers 26–33 [20]. The matching Pattern is first given, and then explained.

The 'Management Utility Freezing—Infinite Loop' is referenced under the identifier 'mb.osgi.4' ('malicious bundle catalog—originates in the OSGi platform itself—number 4). This vulnerability is an extension of the 'Infinite Loop in Method Call' one. It has been identified in the frame of the research project 'Malicious Bundles' of the INRIA Ares Team.

The location of the malicious code that performs the attack is the Bundle Activator. Its source is the Life-Cycle Layer of the OSGi platform, which is not robust against such a vulnerability. Its target is the Platform Management Utility, which can be either the OSGi shell or a graphical interface such as the Knopflerfish graphical user interface (GUI). This vulnerability has a two-fold consequence: the method does not return, so that the caller also freezes; and the infinite loop consumes most of the available CPU, which causes the existing services to suffer from a serious performance breakdown.

This vulnerability is introduced during development, and exploited at bundle start time.

Related Vulnerability Patterns are 'Management Utility Freezing—Hanging Thread', which also targets the Management Utility, 'Infinite Loop in Method Call', 'CPU Load Injection', 'Stand-alone Infinite Loop' which have the same consequence of performance breakdown and the 'Hanging Thread', which also freezes the calling thread.

No specific protection currently exists. Two potential solutions have been identified. The first one consists in launching every Bundle Activator in a new Thread, so as not to block the caller if the activator hangs. The second solution would enable to prevent invalid algorithms to be executed: static code analysis techniques such as Proof Carrying Code [32] or similar approaches can provide formal proves of code well formedness.

Its reference implementation is available in the OSGi bundle named 'fr.inria.ares.- infiniteloopin-methodcall-0.1.jar'. The test coverage is 10%, since 10 types of infinite loops have been identified and only one has been implemented. The test bundle has been tested on the following implementations of the OSGi platform: Felix, Equinox, Knopflerfish and Concierge. The only robust Platform against this attack is our Hardened Felix Platform, which is a prototype meant to enhance the current Felix implementation. Hardened Felix is presented in Section 4.

3.2. Threatening OSGi elements through malicious bundles

The various vulnerabilities that threaten OSGi-based system are now presented.

As any vulnerability list, this one does not pretend to be exhaustive, but intends to be as complete as possible. In particular, the identification is based on the taxonomies we presented in Section 2.2.

DOI: 10.1002/spe

Analysis is performed in a systematic way, i.e. all elements of the OSGi platform and of the bundles are carefully studied to identify security holes.

A classification is established according to the origin of the attack within the malicious bundle: Bundle Archive, Bundle Manifest, Bundle Activator, Bundle Code (Native), Bundle Code (Java), Bundle Code (OSGi API), Bundle Fragments. For each vulnerability, its name, its identifier, a quick description as well as its consequences are given.

Full descriptions of the vulnerabilities according to the Vulnerability Pattern are available in the related Technical Report [33]. Proof-of-concept implementation of the malicious bundles is available for most vulnerabilities under simple request to the authors. Features that can be protected by a Security Manager [14] are considered as vulnerabilities, as they can be exploited to abuse the system. Moreover, performance overheading the use of Java Permissions force many developers not to use the Security Manager and its associated Java Permissions.

3.2.1. 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): If the verification of digital signature of the bundles relies on Java tools only, a bundle whose signature is NOT compliant to the OSGi R4 Digital Signature can be installed; for instance, classes can be added or removed without notice.

Big Component Installer (mb.archive.2): Remote installation of a bundle whose size is bigger than the available device memory; this results in rapid disk space exhaustion. This vulnerability can also be exploited by installing an important number of bundles on the platform.

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.

3.2.2. Bundle manifest

Vulnerabilities that originate in the Bundle Manifest are due to flaws in the expressed Meta-data. 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 the manifest attribute 'Import-Package'. In the Felix and Knopflerfish OSGi implementations, the bundle cannot 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 MB); 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 a non-existent bundle update location. This is bound with OSGi Meta-data.

3.2.3. Bundle activator

Vulnerabilities that originate in the Bundle Activator are bound to the OSGi bundle starting process.

DOI: 10.1002/spe

Table I. Example of a malicious code in OSGi bundle: infinite loop in Bundle Activator.

```
public class InfiniteStartupLoopActivator implements BundleActivator{
        public void start(BundleContext context){
                System.out.println("Bundle InfiniteStartupLoop started");
                while(1==1){}
        }
        public void stop(BundleContext context){
                System.out.println("Bundle InfiniteStartupLoop stopped");
        }
}
```

Management Utility Freezing—Infinite Loop (mb.osgi.4): An infinite loop is executed in the Bundle Activator (see Table I); 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.

3.2.4. Bundle code (Native)[§§]

Vulnerabilities that originate in Native Code are bound to the possibility that exists in the JVM to execute the 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 execution 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.

3.2.5. 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 execution platform is led to crash by the creation of an exponential number of threads (see Table II).

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. If this behavior occurs in a registered service, this can cause a DOS against the caller.

[§§] These bundles are specifically targeted to the OSGi platform. All native code attacks against the Operating System or other applications are not considered here.

DOI: 10.1002/spe

Table II. 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();
    }
}
```

Big File Creator (mb.java.6): A malicious bundle that creates a big (relative to available resources) file 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 a service. All classes that are referenced (directly or indirectly) as class attributes can be accessed. Only public methods can be invoked.

3.2.6. 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 concern any programming language.

Memory Exhaustion (mb.java.10): A malicious bundle that consumes most of available memory (see Table III); if other processes are executed an require memory, MemoryErrors occur. 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.

DOI: 10.1002/spe

Table III. Example of malicious code in OSGi bundle: memory load injection (with 'size' parameters in the same order of magnitude than the total memory).

```
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;
    }
}
```

Table IV. 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");
}
```

3.2.7. 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 stored 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) (see Table IV).

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.

DOI: 10.1002/spe

3.2.8. OSGi services and bundle fragments

Vulnerabilities that originate in the OSGi API are due to the specific type of interactions that exist within the OSGi platform. Those that are due to the SOP [4] paradigm are relevant to any Service-Oriented Programming (SOP) platform.

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 a loop; this vulnerability is related to SOP.

Freezing Numerous Service Registration (mb.osgi.10): Registration of a high number of (possibly identical) services through a 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 does not intend to make visible. Other bundles can then execute 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 has the same name as a package in the host. All package-protected classes and methods can then be accessed by the fragment, and thus exported to the framework.

This catalog of vulnerability builds the knowledge base required so as to perform OSGi platforms benchmarking. The availability of such a knowledge base is advocated in particular in the NIST Samate project [34].

3.3. Security Benchmarking for open source implementations of the OSGi platform

Based on this vulnerability catalog, it is now possible to assess the security status of most widespread open source implementations of the OSGi platform. First, the origin of the vulnerabilities is 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 more serious attacks when protections are developed. Since no suitable metric exists to express them, these properties are discussed based on the frequency of occurrence of each of their values. The definition of metrics that could measure the impact of attacks is a complex task that should take the specific application context into account: certain failures can be tolerated in a multi-player game whereas they are not acceptable in health-care systems. Lastly, quantitative assessment and comparison of the various implementations of the OSGi platform is performed. The Protection rate (PR) metric is introduced to support it.

Life-Cycle-related information is not considered further here, although it is available. Actually, they do not bring in information of outstanding interest for the benchmarking analysis.

3.3.1. 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.

DOI: 10.1002/spe

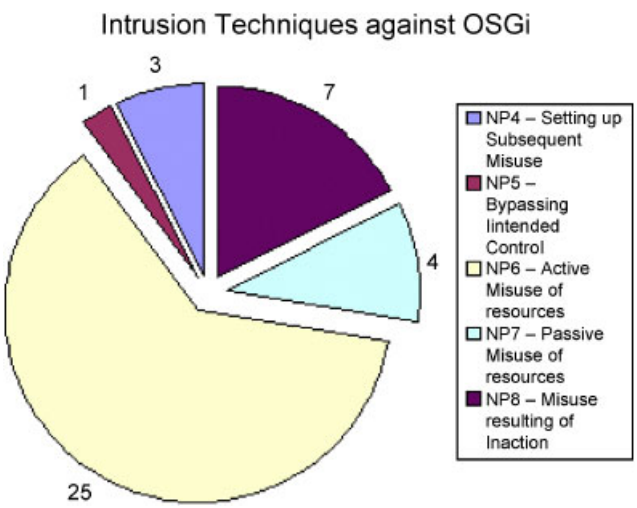

Figure 5. Relative importance of intrusion techniques in an OSGi platform.

A raw partition of the system can be made between the JVM and the OSGi platform. This distinction enables the identification of the relative liability of each part and to know which amount of the vulnerabilities is specifically due to the OSGi platform. Out of the 32 vulnerabilities that are presented in the catalog in Section 3.2, 18 are implied by the JVM, i.e. 56%. The number of vulnerabilities that is implied by the OSGi platform is 14 out of 32, i.e. 44%. This means that most of the vulnerabilities identified in this study are due to the JVM itself, and therefore that other Java-based Extensible Component Platforms such as MIDP also suffer from the same threats.

The relative importance of the intrusion techniques in an OSGi platform is shown in Figure 5. The classification used as reference is the Neumann and Parker Classification [35], which defined nine categories of intrusion techniques that can be used against computing systems. Categories 3–8 are related to software attacks, and the others 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 4–8 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.

Figure 5 shows the distribution of the vulnerabilities in the Neumann and Parker categories. 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 bundles. Preventing these actions through 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 very few counter-measures are available.

The number of *passive misuse of resource* is 4, mainly undue read access. Since writing and reading accesses inside programs both occur through method calls, these kind of vulnerabilities

DOI: 10.1002/spe

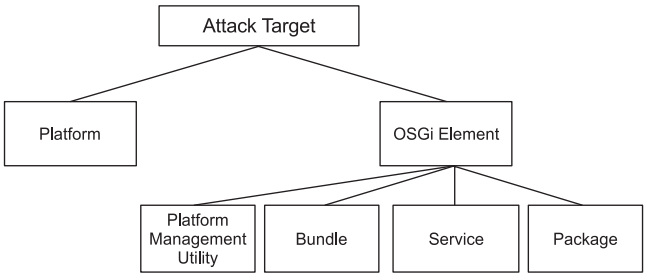

Figure 6. Taxonomy for attack targets in the OSGi platform.

can be prevented through the same mechanisms as the one used with active misuse, i.e. access control.

Three types of vulnerabilities consisting in *setting up subsequent misuse* exist. They are related to bundle management and fragments and are resolved by OSGi permissions.

The last type of vulnerability is *bypassing intended control*. The unique occurrence is due to the JVM digital signature validation algorithm that does not exactly match the OSGi specifications. A patch is presented in [3] 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 made so far to make it secure. Though other types of weaknesses exist, most vulnerabilities are due to the lack of default access control mechanism, both at the Java and OSGi level.

3.3.2. 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 targeted platform elements and their type of consequence.

Figure 6 shows the taxonomy of existing attack targets in OSGi-based systems. 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 its running applications are impacted. The OSGi elements consist in all the code executed in the platform. These elements can be impacted with a varying granularity: any element of the bundles can be attacked such as single services or packages. Attacks that target specific elements have impact on the element itself and on 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

[¶¶] http://sfelix.gforge.inria.fr/.

DOI: 10.1002/spe

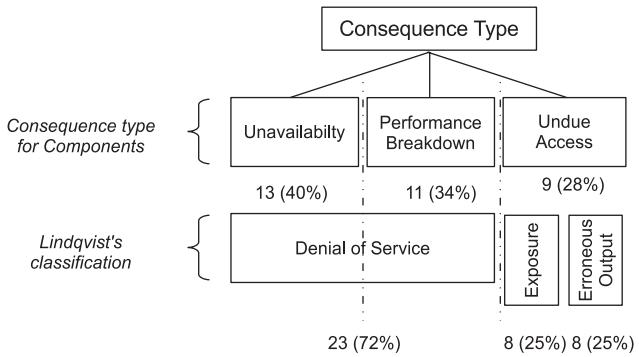

Figure 7. Taxonomy for consequence type in the OSGi platform.

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 has.

The second characteristic of attacks is the consequence they have on the attacked system. To reflect the specificity of the OSGi platform, a dedicated taxonomy is introduced, as shown in Figure 7. As comparison with a mainstream classification, the Lindqvist classification [36], is also given for reference.

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. 40% of them. Performance breakdown concerns 11 vulnerabilities, i.e. 34% of them. Undue access in read or write mode represents nine 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 is less relevant in our case: access to a given method of a bundle enables both read and write access, and does not depend on access control properties. It also considers all 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's 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, the greater the damages caused. However, it is important to note that a recent trend in software 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.

3.3.3. Quantitative security assessment through the PR Metric

Security benchmarking requires the availability of tools for quantitative security assessment so as to support evaluation and comparison of the open source implementations of the OSGi platform.To the best of our knowledge, no such metric is so far available.

DOI: 10.1002/spe

A metric for fault-tolerant systems exists that addresses quantitative assessment and benchmarking: the *coverage* metric [37]. Its goal is to express the percentage of faults that are contained by fault-tolerance mechanisms. We therefore define the similar *PR* metric, which represents the percentage of the known vulnerabilities protected by a given security mechanism. Several implementations of the OSGi platform are compared through their respective PR against the set of vulnerabilities we identified for the OSGi specifications.

The PR is based on the Attack Surface metric [38]. The Attack Surface is the number of functionalities and flaws that can be exploited for a given system. In our study, the Attack Surface represents the total number of vulnerabilities that are identified in the OSGi platform. Its value is 32. The PR is relative to the set of vulnerabilities considered. Since no inventory can be considered as closed, a reference for the considered vulnerabilities is to be expressed. For instance, here, the values of the PR are relative to the 'Malicious Bundles' catalog of vulnerabilities [33].

The PR is defined as the quotient of the Attack Surface that is protected through any security mechanism and the Attack Surface of the reference system. 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 case, 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 real implementation of the OSGi platform, possibly with some security mechanisms on. The PR can be expressed as

$$PR = \left(1 - \frac{\text{Attack Surface of the evaluated system}}{\text{Attack Surface of the reference system}}\right) * 100 \quad (1)$$

The PR metric enables the comparison between 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 with the set of vulnerabilities that are identified for these specifications. The *PR* metric still has some limitations. It does not only represent the rate of known vulnerabilities protected in a given implementation of the considered system. In particular, it does not reflect the relative importance of the vulnerabilities, e.g. according to the impact of the attacks that are made possible by these vulnerabilities.

The PRs for the various implementations of the OSGi platform are given in Table V, for the set of vulnerabilities identified in the 'Malicious Bundles' catalog. As the consequences of the attacks show important variations, it is difficult to automate the tests. The *PR* metric is therefore extracted by running all sample attack bundles on the platform being benchmarked.

The default Felix implementation is robust against one of the identified attacks, out of 32 ($PR = 3.1\%$). The default Knopflerfish implementation is robust against one attack, out of 31 (Attack Surface is 31, because the digital signature of bundle is not supported; $PR = 3.2\%$). Namely, it cleans bundle-related data properly at uninstallation. The default Equinox implementation is robust against four vulnerabilities out of 31 ($PR = 13\%$), which is slightly better, but cannot be considered as satisfactory. Namely, the Equinox platform does support duplicating package imports, which are rejected by other implementations in compliance with the OSGi specifications: supports oversized manifest files, which cause hanging of the JVM and are not protected by other implementations; properly cleans bundle-related data at uninstallation. The default Concierge implementation is subject to all identified vulnerabilities. Since they do not support Bundle Fragments nor bundle signature, their PR is $0/28 = 0\%$.

DOI: 10.1002/spe

Table V. Protection rate for Mainstream OSGi platforms, relative to the vulnerabilities from the 'Malicious Bundles' catalog.

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
Concierge with Security Manager	10	28	36
Felix with Security Manager	14	32	44
Knopflerfish with Security Manager	14	31	45
Equinox with Security Manager	17	31	55

Java Permissions represent one solution that can be used to improve the security status of the OSGi platform. They enable the protection against 13 attacks (13; $PR = 40\%$). Felix and Knopflerfish with the Security Manager enabled have a PR of, respectively, 43 and 45%. Equinox with the Security Manager enabled has a PR of 53%. Concierge with the Security Manager enabled has a PR of 36%.

These results show that the various implementations are designed with rather limited security in mind. A good practice to enhance the security status of OSGi-based systems is to use Java Permissions. However, this peculiar mechanism is known to induce an important performance overhead during execution. For sensitive applications, a trade-off must as yet be found between security and performance.

3.3.4. Conclusion

A Vulnerability Pattern for benchmarking of OSGi platforms is defined. The vulnerabilities of the OSGi platform are identified and described. Since a good half of them is due to the underlying Java Platform, this catalog can also be of interest for all designers of Java-based component systems. Moreover, widespread open source implementations of the OSGi platform are benchmarked against this set of vulnerabilities.

An immediate requirement when vulnerabilities are identified is to make suitable security mechanisms available. Since this analysis is focussed on the OSGi platforms, OSGi specific protections must be first defined. The creation of Java-specific security protections is mandatory for building secure production OSGi platforms, but they are out of the scope of this study.

4. A HARDENED OSGI PLATFORM

The need for protecting OSGi platforms urges us to define and evaluate a series of recommendations for developing Hardened OSGi platforms.

First recommendations for building *Hardened OSGi* implementations are given, along with precise development requirements. Next an evaluation of the prototype implementation, Hardened Felix, is performed through the PR metric defined in the previous section. Then the evaluation of performances of the prototype implementation is presented to validate the usability of the recommendations.

DOI: 10.1002/spe

4.1. Recommendations for Hardened OSGi implementations

Recommendations for implementing Hardened OSGi platforms are made up of two parts. The first part contains recommendations that intend to complete the OSGi specifications. They are classified according to the layer structure of the OSGi platform: Module Layer, Life-Cycle Layer, Service Layer. The second part contains API and implementation requirements, i.e. modifications that have been necessary to implement the *Hardened OSGi* recommendations in our prototype.

The following modifications are to be taken into account at the implementation level. They are presented according to the OSGi platform element concerned.

Module Layer: The recommendation for a Hardened OSGi Module Layer intends to make bundle dependency management more robust.

- *Bundle Dependency Resolution Process*: Do not reject duplicate imports. Just ignore them; *OSGi R4 par. 3.5.4*. This prevents the attack 'Duplicate Package Import' (mb.osgi.1).

Life-Cycle Layer: Recommendations for a Hardened OSGi Life-Cycle Layer intend to protect the platform from installation of malicious or ill-formed bundles, as well as to guarantee that the uninstallation process is performed in a clean manner.

- *Bundle Download*: When a bundle download facility is available, the total size of the bundles to install should be checked immediately after the dependency resolution process. The bundles should be installed only if the required storage is available. This prevents the attack 'Big Component Installer' (mb.archive.2), without downloading the bundles on the platform. This security mechanism is to be used in conjunction with the maximum storage size mechanism in the context of OBR clients [6] for OSGi.
- *Bundle Installation Process*: A maximum storage size for bundle archives is set. Alternatively, a maximum storage size for all data stored on the local disk is set (*Bundle Archives and files created by the bundles*); *OSGi R4 par. 4.3.3*. This prevents the attack 'Big Component Installer' (mb.archive.2) of the OSGi Vulnerability Catalog.
- *Bundle Signature Validation Process*: The digital signature must be checked at installed time. It must not rely on the Java built-in validation mechanism, since this latter is not compliant with the OSGi R4 Specifications [3]; *OSGi R4*, *Paragraph 2.3*. This prevents the attack 'Invalid Digital Signature Validation' (mb.archive.1).
- *Bundle Start Process*: Launch the Bundle Activator in a separate thread; *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).
- *Bundle Uninstallation Process*: Remove the data on the local bundle filesystem when a bundle is uninstalled (and not when the platform is stopped); *OSGi R4 par. 4.3.8*. This prevents the attack 'Zombie Data' (mb.osgi.8).

Service Layer: The recommendation for a Hardened OSGi Service Layer intends to make the Service-Oriented Programming (SOP) features of the platform more reliable.

- *OSGi Service Registration*: set a platform property that explicitly limits the number of registered services (default could be 50); *OSGi R4 par. 5.2.3*. This prevents the attacks 'Numerous Service Registration' (mb.osgi.10) and 'Freezing Numerous Service Registration' (mb.osgi.11).

DOI: 10.1002/spe

Table VI. Protection rate for Hardened OSGi platforms.

Platform type	# of protected flaws	# of known flaws	Protection rate (%)
Hardened Felix	8	32	25
Hardened Felix with Security Manager	21	32	66
Hardened Equinox with Security Manager (expected)	23	31	74

To support these modifications of the OSGi platform implementations, the following changes have been applied to the API. These modifications are required for storage size control.

- In the Class BundleContext, a method getAvailableStorage() is defined.
- A property osgi.storage.max is defined, which is set in the property configuration file of the OSGi framework.
- In the class org.osgi.service.obr.Resource, a method getSize() is defined. This method relies on the size entry of the bundle Meta-data representation (usually a XML file).

Several attacks relative to the bundle archive, the bundle manifest, the Bundle Activator and OSGi API are prevented through this set of modifications. In particular, memory exhaustion due to the installation of big bundles or unclean uninstallation, or DOS against the Platform Management Utility and the Service broker mechanism are prevented. Installation of maliciously modified bundles is also prevented.

These recommendations can be used in any OSGi implementation, so as to make it more robust in presence of malicious or simply ill-coded bundles. A prototype has been built based on the Felix Apache implementation[‖‖] of the OSGi R.4 Platform to validate their usability.

4.2. Evaluation of Hardened Felix: PR

The benefit of the *Hardened OSGi* Recommendations must now be evaluated. This is first done through the *PR* metric, which is extracted for the following configurations: absolute *PR* value, *PR* value for Hardened Felix, *PR* value for Hardened Felix with the Security Manager enabled. Next, the relative importance of OSGi and Java vulnerabilities that stay unprotected within Hardened OSGi implementations are discussed. These data enable the comparison with other OSGi implementations.

The PRs for Hardened OSGi platforms are given in Table VI. The Hardened OSGi Recommendations prevent eight of the identified vulnerabilities, out of 32. The PR is therefore $PR = 25\%$. Our prototype, Hardened Felix, benefits of all of these protections, but one that is redundant with Felix features. Our prototype used with the Security Manager enabled is robust against 21 vulnerabilities out of 32, i.e. 66%.

The relative importance of the remaining Java and OSGi-specific vulnerabilities are discussed for the same configurations. Hardened Felix without Java Permissions has twenty-four (24) remaining vulnerabilities. Seventeen of them are bound with the underlying JVM, i.e. 71%. Seven (7) of them are bound with OSGi features, i.e. 29%. However, these latter are due to OSGi features

[‖‖] http://felix.apache.org.

DOI: 10.1002/spe

such as bundle management, OSGi services and fragments. They can be protected through OSGi Permissions.

Hardened Felix with the Security Manager enabled has 10 remaining vulnerabilities. Nine of them are bound with the JVM, i.e. 90%. One is bound with the OSGi platform, i.e. 10%.

One single OSGi vulnerability is not protected in Hardened OSGi: *Erroneous values of Manifest Attributes*. The risk of involuntary introduction of errors based on this vulnerability can be limited by automated generation of these attributes, which is often the case when Integrated Development Environments such as Eclipse are used. Moreover, this weakness may not be considered as an actual vulnerability but as a configuration error as any system can experience.

OSGi vulnerabilities can be considered to be fully patched by the Hardened OSGi Recommendations, when suitable Java and OSGi permissions are set. However, Java specific vulnerabilities make OSGi platforms at risk. The remaining major attacks that can occur in a Hardened Felix Platform with the Security Manager enabled 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 manifest size.

Relative to default OSGi platforms, Hardened OSGi Recommendations bring an important step toward building secure systems. When used together with the Security Manager, Hardened Felix enables the development of platforms that have a high PR. Hardened Felix with the Security Manager enabled has a PR of 66%. Implementing Hardened OSGi Recommendations within the Equinox implementation would provide a PR of 74%.

Hardened OSGi Recommendations fulfill their objective, namely hardening the vulnerabilities of the OSGi platform that are introduced by OSGi itself. However, major Java vulnerabilities are still open, such as the 'recursive thread creation', which causes a platform crash, or uncontrolled CPU or memory resource consumption. Hardened OSGi Recommendations are thus necessary to develop secure platforms, but it is far from being sufficient since the underlying virtual machine has severe security drawbacks when used for multi-application systems.

4.3. Evaluation of Hardened Felix: performances

The benefits of *Hardened OSGi* Recommendations must also be evaluated according to the performance impact they have on the system. Conditions of the experiments are first given, and results are provided and commented upon.

The tests are performed on an 1686 architecture, with a Pentium M processor of 1.86 GHz. Only the installation phase will be considered since most of the modifications refer to the bundles' installation phase. Performances during the execution are likely to be impacted by the Java Permissions, and not by the Hardened OSGi architecture, and are thus out of the scope of this study. Five profiles are considered: a void OSGi platform (started without any bundles), a default Felix Platform (with the OSGi shell for local management, and the OBR client for discovering and downloading new bundles), the 'JMX manageable Platform' profile (which supports remote management through a JMX-based Console), the 'UPnP Control Point' profile (that can play the role of UPnP gateway in intelligent Homes) and the applicative profiles 'JMX Console' and 'SF-Jarsigner'*** (which supports the digital signature and publication of OSGi bundles onto OBR repositories).

***http://sf-jarsigner.gforge.inria.fr/.

DOI: 10.1002/spe

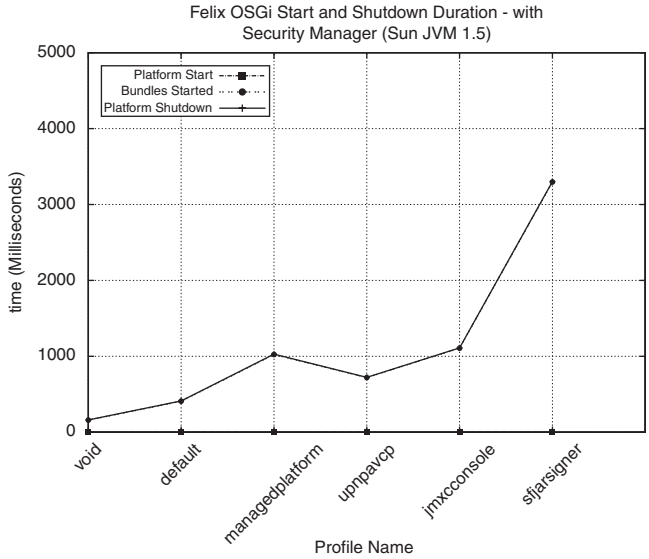

Figure 8. Performances of Felix, with the Security Manager enabled.

Tests are therefore performed for the following OSGi instance configurations: with and without integration of the Hardened OSGi Recommendations, and with and without the Security Manager. Figure 8 shows default Felix implementation performances with the Security Manager enabled. Figure 9 shows Hardened Felix performances with the Security Manager enabled.

Performances of 'Hardened Felix' are weaker than those of Felix, except in the case of the void profile: this is due to the verification of the validity of the bundle digital signature, which is relatively heavyweight—and not available in most OSGi implementations, which are not aware of security threats. Starting an application without Java Permissions is quicker than starting it with these Permissions. This difference is particularly important in the case of execution of complex programs, such as the starting process of the 'SF-Jarsigner' tool.

The other main modification in the platform behavior introduced by the Hardened OSGi Recommendations is the Bundle Activator's launching in a separate thread. Consequently, two phases must be considered: the time point where the platform is available for receiving new commands (dotted lines), and the time point where the launching of the whole bundles is effective (plain line). The second phase is negligible in the case of basic activation processes (such as in the void, default, JMX manageable Platform or UPnP Control Point). The two-phase launch behavior of Hardened OSGi improves the system's reactivity, and is especially appreciated for embedded or handheld devices [39], in the case where the activation is a complex process, for instance when GUIs are started. These modifications thus also imply a better usability (and not only dependability) of the OSGi platforms.

As shown by the test results, other Recommendations only have a negligible impact on the performance of OSGi platforms, since they are made of lightweight controls or actions. The performance measures also show the overhead implied by Java Permissions.

DOI: 10.1002/spe

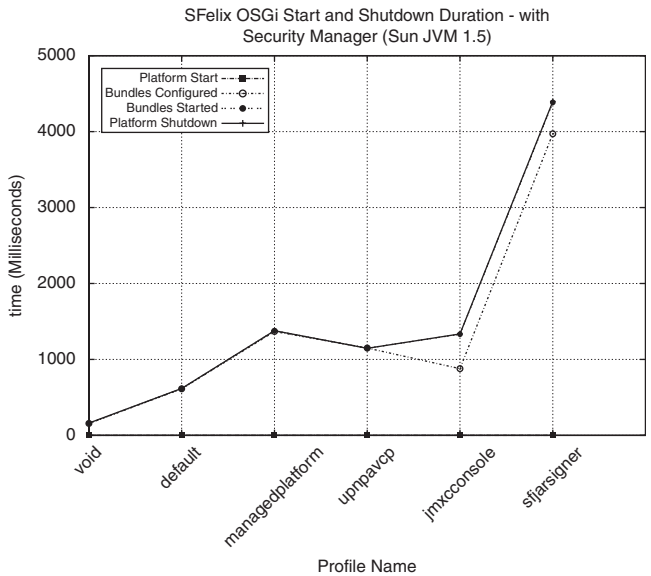

Figure 9. Performances of Hardened Felix, with the Security Manager enabled. [The performances are extracted for following OSGi profiles, i.e. set of bundles for one specific application: void (no bundle), default (with shell and bundle repository client bundles), managedplatform (for JMX support), upnpcp, a control point for the UPnP protocol for home networks, jmxconsole (a client console for JMX management) and sf-jarsigner (a graphical tool for signing and publishing bundles).]

The Hardened OSGi Recommendations enable to build OSGi platforms that are robust against some ill-coded bundles, or bundles that exploit some of the vulnerabilities that are described in this study. The overhead of securing an OSGi platform cannot be considered as negligible. Main performance losses are implied by Bundle Digital Signature and Java Permissions. The proposed Hardened Architecture in itself has no noticeable performance drawbacks. On the contrary, by using separate threads for executing applications, it provides better usability in addition to the dependability performance.

4.3.1. Conclusion

Secure implementations of the OSGi platform are required to prevent the exploitation of the following attack vector specific to Extensible Component Platforms: dynamic and transparent extension of the system at runtime. Recommendations for Hardened OSGi platforms are given and provide a great improvement in these platforms' security status. These recommendations are validated through qualitative evaluations with the PR Metric and through performances evaluations.

Hardened OSGi Recommendations tackle almost all the vulnerabilities that are due to the OSGi specifications themselves. However, Java-specific vulnerabilities are not addressed, and build an obvious requirement for future study.

DOI: 10.1002/spe

5. CONCLUSIONS AND PERSPECTIVES

The contribution of this study is threefold. First, tools are defined to support the security engineering process for such systems: a specific Vulnerability Pattern is introduced, as well as the Protection rate metric, which expresses the efficiency of security mechanisms. Second, the security benchmarking of several open source OSGi platforms is performed, which enables us to provide recommendations for building secure OSGi-based Systems. Third, a Hardened implementation of Apache Felix is provided. It supports the validation of our recommendations and tools.

This study also brings evidence of a strong requirement for further research efforts in order to build secure OSGi-based systems. In particular, we identify Five Security Challenges that are currently not prevented through any available security mechanism for the OSGi platform:

1. *Infinite loop/hanging thread in method call*: any bundle can contain non-returning calls.
2. *Memory load injection*: any bundle can consume memory.
3. *Decompression bomb*: although it is identified by anti-virus software, this vulnerability is not prevented in OSGi-based systems.
4. *Exponential thread number*: any bundle can cause the platform to crash due to memory exhaustion.
5. *Service short circuit*: the numerous and optional service management mechanisms in OSGi do not support security enforcement so far.

The most urgent of these challenges is to patch the *Exponential thread number* attack, which implies the most serious damages to the platform.

An important conclusion of this study is that more than half of the vulnerabilities in an OSGi platform are not bound with OSGi itself, but to the underlying JVM. This means that all Java-based (and probably most VM-based) systems that support multi-application execution have a high probability to be weak in front of the presented attacks.

Through this study, we intend to increase the knowledge of the security implication of Java-based Extensible Component Platforms, in particular the OSGi platform. We also intend to solve most of OSGi-specific security issues by patching identified vulnerabilities. An important research effort is still required to achieve the long-term objective that is made technically possible by the OSGi platform: running secure multi-application systems with mutually untrusted bundles.

NOTE

Another vulnerability has been identified after the submission of the paper: 'Numerous unnecessary imports'. The dependency resolution process in the presence of numerous (e.g. one million) of package imports is likely to imply a non-negligible overhead. However, related experiments have not been performed, which prevents its introduction in the vulnerability catalog.

ACKNOWLEDGEMENTS

This work is partially funded by the ANR-LISE Project ANR-07-SESU_007. We also would like to thank the reviewers for their valuable comments.

DOI: 10.1002/spe

REFERENCES

1. JSR 118 Expert Group, MIDP 2.0', Sun Specification. 2002.
2. OSGI Alliance. OSGi Service Platform, Core Specification Release 4.1, Draft, 2007.
3. Parrend P, Frenot S. Supporting the secure deployment of OSGi Bundles. *First IEEE WoWMoM Workshop on Adaptive and DependAble Mission—and bUsiness—Critical Mobile Systems*, Helsinki, Finland, 2007.
4. Bieber G, Carpenter J. Introduction to Service-Oriented Programming (Rev 2.1), OpenWings Whitepaper, 2001.
5. Howes T. *The String Representation of LDAP Search Filters*. IETF RFC, Network Working Group, Request for Comments: 2254, 1997.
6. OSGi Alliance, Hall RS. 'Bundle Repository', OSGi Alliance RFC 112, 2006.
7. Sun Microsystems, Inc. JAR File Specification, Sun Java Specifications, 2003.
8. Lindholm T, Yellin F. *The Java*(TM) *Virtual Machine Specification* (2nd edn). Prentice-Hall PTR: Englewood Cliffs, NJ, 1999.
9. Saraswat V. Java is not type-safe, AT&T Research Whitepaper, 1997.
10. Drossopoulou S, Eisenbach S. Java is type safe—Probably. *ECOOP'97—Object-Oriented Programming*. Springer: Berlin/Heidelberg, 1997; 389–418.
11. Jensen T, Metayer DL, Thorn T. Security and dynamic class loading in Java: A formalisation. *IEEE International Conference on Computer Languages*, Chicago, IL, U.S.A., 1998; 4–15.
12. Dean D, Felten EW, Wallach DS. Java Security: From HotJava to Netscape and beyond. *SP'96*: *Proceedings of the 1996 IEEE Symposium on Security and Privacy*. IEEE Computer Society: Washington, DC, U.S.A., 1996; 190.
13. Gong L, Ellison G, Dadgeforde M. *Inside Java 2 Platform Security—Architecture*, *API Design*, *and Implementation* (2nd edn). Addison-Wesley: Reading, MA, 2003.
14. Gong L, Mueller M, Prafullchandra H, Schemers R. Going beyond the sandbox: An overview of the new security architecture in the Java development Kit 1.2. *Proceedings of the USENIX Symposium on Internet Technologies and Systems*, Monterey, CA, U.S.A., 1997.
15. Bryce C. Isolates: A new approach to multi-programming in Java Platforms, May 2004, LogOn Technology Transfer OT Land. Available at: http://www.bitser.net/isolate-interest/papers/bryce-05.04.pdf.
16. Czajkowski G, Daynès L, Titzer B. A multi-user Virtual Machine. *Usenix*, 2003; 85–98.
17. Czajkowski G, Hahn S, Skinner G, Soper P, Bryce C. A resource management interface for the Java Platform, SMLI TR-2003-124, May 2003.
18. Aslam T. A taxonomy of security faults in the Unix operating system. *Master's Thesis*, Purdue University, 1995; 120.
19. Bloch J. *Effective Java Programming Language Guide*. Addison-Wesley Professional: Reading, MA, 2001.
20. Bloch J, Gafter N. *Java Puzzlers—Traps*, *Pitfalls and Corner Cases*, *Pearson Education*. Addison-Wesley: Reading, MA, 2005.
21. Cotroneo D, Orlando S, Russo S. Failures classification and analysis of the Java Virtual Machine. *Twenty-sixth IEEE International Conference on Distributed Computing Systems* (*ICDCS'06*), Lisboa, Portugal, 2006.
22. OSGi Alliance. RFC 18: Security Architecture Specification, OSGi Alliance Request for Comment, 2001.
23. Huang C-C, Wang P-L, Hou T-W. Advanced OSGi Security Layer. *AINAW '07*: *Proceedings of the 21st International Conference on Advanced Information Networking and Applications Workshops*, vol. 2, Niagara Falls, Canada, 2007; 518–523.
24. Sun Microsystems, Inc. JavaTM Virtual Machine Profiler Interface (JVMPI). *JavaTM 2 Platform Standard Edition 5.0—Overview*. Available at: http://java.sun.com/j2se/1.5.0/docs/guide/jvmpi/, 2004.
25. Sun Microsystems, Inc. JVMTM Tool Interface (JVM TI). *JavaTM 2 Platform Standard Edition 5.0—Overview*. Available at: http://java.sun.com/j2se/1.5.0/docs/guide/jvmti/, 2004.
26. Gamma E, Helm R, Johnson R, Vlissides J. *Design Patterns*: *Elements of Reusable Object-Oriented Software*. Addison-Wesley: Reading, MA, 1994.
27. Schumacher M. *Security Engineering with Patterns*. Springer: Berlin, 2003.
28. Arvidsson J, Cormack A, Demchenko Y, Meijer J. TERENA's incident object description and exchange format requirements, IETF RFC 3067, 2001.
29. EISPP Consortium. EISPP Common Advisory Format Description, EISPP Whitepaper EISPP-D3-001-TR, 2003.
30. Martin RA. Transformational vulnerability management through standards, CrossTalk. *The Journal of Defense Software Engineering* 2005; 12–15.
31. Mowbray TJ, Malveau RC. *Corba Design Patterns*. Wiley: New York, 1997.
32. Necula GC. Proof-carrying code. *Conference Record of POPL '97*: *The 24th ACM SIGPLAN–SIGACT Symposium on Principles of Programming Languages*, Paris, France, 1997; 106–119.
33. Parrend P, Frenot S. Java Components Vulnerabilities—An experimental classification targeted at the OSGi Platform'(RR-6231). *Technical Report INRIA*, 2007; 84.
34. Black PE. Software assurance metrics and tool evaluation. *Proceedings of the 2005 International Conference on Software Engineering Research and Practice* (*SERP'05*), Las Vegas, NV, U.S.A., 2005.

DOI: 10.1002/spe

35. Neumann PG, Parker DB. A summary of computer misuse techniques. *Proceedings of the 12th National Computer Security Conference*, 1989; 396–406.
36. Lindqvist U, Jonsson E. How to systematically classify computer security intrusions. *IEEE Symposium on Security and Privacy*, Oakland, CA, U.S.A., 1997; 154–163.
37. Arnold TF. The concept of coverage and its effect on the reliability model of a repairable system. *IEEE Transactions on Computers* 1973; **22**:251–254.
38. Howard M, Pincus J, Wing J. Computer Security in the 21st Century. *Measuring Relative Attack Surfaces*. Springer: New York, 2005; 109–137.
39. Mikkonen T. *Programming Mobile Devices*: *An Introduction for Practitioners*. Wiley: New York, 2007; 244.

DOI: 10.1002/spe