Classification of Component Vulnerabilities in Java Service Oriented Programming (SOP) Platforms*

Pierre Parrend and Stéphane Frénot

INRIA Amazones / CITI, INSA-Lyon, F-69621, France
Tel.: +334 72 43 71 29, Fax. +334 72 43 62 27
{pierre.parrend,stephane.frenot}@insa-lyon.fr

Abstract. Java-based systems have evolved from stand-alone applications to multi-component to Service Oriented Programming (SOP) platforms. Each step of this evolution makes a set of Java vulnerabilities directly exploitable by malicious code: access to classes in multi-component platforms, and access to object in SOP, is granted to them with often no control.

This paper defines two taxonomies that characterize vulnerabilities in Java components: the vulnerability categories, and the goals of the attacks that are based on these vulnerabilities. The 'vulnerability category' taxonomy is based on three application types: stand-alone, class sharing, and SOP. Entries express the absence of proper security features at places they are required to build secure component-based systems. The 'goal' taxonomy is based on the distinction between undue access, which encompasses the traditional integrity and confidentiality security properties, and denial-of-service. It provides a matching between the vulnerability categories and their consequences. The exploitability of each vulnerability is validated through the development of a pair of malicious and vulnerable components. Experiments are conducted in the context of the OSGi Platform. Based on the vulnerability taxonomies, recommendations for writing hardened component code are issued.

1 Introduction

Java execution environments evolve from stand alone applications to component-based systems to Service Oriented Programming (SOP) Platforms [1]. Component-based systems introduce multi-application execution. Service Oriented Programming (SOP) Platforms add a strong runtime dynamicity of component linkage, thus supporting more customizable applications. While new features are added, each of these evolutions turns potential vulnerabilities into directly exploitable flaws. Access to component class represents a first important threat. It makes class vulnerabilities directly exploitable by other components. SOP broaden this threat by enabling direct access to objects provided by these

* This work is partially funded by the ANR-07-SESU_007 LISE Project.

components, with often no restriction. This makes object vulnerabilities exploitable. We present a classification of vulnerabilities of components, so as to help developers identify and mitigate this threat and build secure component-based systems.

Experiments are conducted on the Java/OSGi SOP Platform [13]. They aim at identifying vulnerabilities that can actually be exploited by malicious components that are installed on a system, as well as additional preconditions. Is considered as an exploitable vulnerability any feature which use leads to a behavior that break explicit or implicit security policies for the system [8].

In most cases, the condition of exploitation is that vulnerabilities must be present in the code that is made available to other components. This is what we call *Public Code*. This concept is introduced in the Parnas and Wang component model [14], cited by [4].

The following of the paper is organized as follows. Section 2 presents related works. Section 3 describes the vulnerability categories, and provides the related taxonomies. The rationale and experiment of this study is provided in Section 4. Section 5 concludes this work.

2 Related Works

As a part of the Java ecosystem, the Java language itself has been designed with a strong emphasis on security. However, as no system is entirely secure, pitfalls and behaviors exist that turn out to be actual vulnerabilities, in particular in the context of multi-component systems and Service Oriented Programming.

2.1 Attack Vectors against the Java/OSGi SOP Platform

Hackers can attack Java/OSGi applications by exploiting two main attack vectors: platform vulnerabilities, and component vulnerabilities. Platform vulnerabilities can be exploited to indirectly attack other components. The only requirement for exploiting them in a default, non secure Java/ OSGi platform implementation, is to install a bundle that calls dangerous or faulty platform code. This often implies that it is published in a known bundle repository, and sometimes that it is signed. A security analysis of the Java/OSGi Platform is given in [15]. 32 vulnerabilities are identified. They lead to Denial-of-Service (through platform crash or performance breakdown) and to undue access to code. Most vulnerabilities (18 out of 32) are bound with the JVM, such as the lack of CPU and memory isolation between components, the Runtime API, the presence of dangerous functionalities such as native code execution, thread creation, reflection. Others (14 out of 32) are bound with the OSGi Platform itself, such as bundle fragments, bundle management, and lack of control on Service-Oriented-Programming. All of these vulnerabilities lead to attacks against the Platform that can be exploited to harm other components. Other vulnerabilities are specific to given implementations of the JVM [2], or to specific embedded platforms such as the CLDC [3].

Component vulnerabilities can be exploited to directly attack other components. They are due to java language properties [7] that can be misused to achieve a malicious goal.

2.2 Known Vulnerabilities in Java/OSGi Components

After platform vulnerabilities, the second kind of vulnerabilities that can plague Java-based component systems is the presence of flaws in the components themselves. They can be exploited as soon a malicious component can be installed in a Platform where components share code with each other.

Several works provide hints related to some attacks against Java-based systems, without taking a systematic approach. Java features that can lead to vulnerabilities are presented by Long [12]: type safety limitations, public fields, inner classes, serialization, reflection, JVM Tool Interface, debugging and management tools can be exploited to abuse Java-based applications. More weaknesses are mentioned by the Last Stage of Delirium Research Group [18], such as unsafe type conversion, class loader attacks, bad implementation of system classes. Another specific attack consists in executing arbitrary code through forced type mismatch [5]. It is based on memory errors that can mainly be forced through physical access to the machine. These vulnerabilities form the first set of occurences on which our experiments are based.

The first systematic set of candidate vulnerabilities that flaw Java Extensible Component Platforms is provided by the Findbugs tools Vulnerability List [1] [6]. The *Malicious Code Vulnerability* category identifies 12 code patterns that can lead to exposition and modification of object internal data to another potentially untrusted code element, such as returning references to mutable objects or array or storing data in class variable that are not properly encapsulated.

The second systematic set of candidates vulnerabilities that flaw Java Extensible Component Platforms is provided by the 'Sun Java Security Coding Guidelines' [17]. Each guideline matches a code flaw that can be exploited by untrusted code to perform malicious actions. For instance, abuse of inheritance, faulty validation and copy of method parameters or returned objects, security checks by-passing and serialization/de-serialization of sensitive data are referenced. Sun Java Security Coding Guidelines are completed by Charlie Lai's Java Insecurity Subtleties [9]. These two lists of vulnerabilities form the second set of occurences on which our experiments are based. More are detailed in the Appendix A.1.

These references provide useful support both to train developers and for supporting vulnerability identification through static analysis. However, several criticisms can be issued. First, none of these works provides a classification that is structured or complete. Secondly, they do not provide information relative to the exploitability of these vulnerabilities: are they present but harmless, or is any installed component able to exploit them all with little to no additional effort?

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

3 Vulnerabilities in SOP Platforms

Vulnerabilities in Java/OSGi components pertain to three categories: Stand-Alone components, Class Sharing, and Object Sharing. The last category is made exploitable by the Service Oriented Programming (SOP) paradigm. Two taxonomies characterize at best their properties: Categories of the vulnerabilities, and goals of attacks that exploit them. These taxonomies are obtained by classifying the 39 distinct vulnerabilities that we identified through bibliographical review and through our own experience. Two examples that highlight abuse risks are given in the Appendix A.1: *Malicious Inversion of Control* and *Synchronized Code*.

3.1 Vulnerability Classes

Classes of vulnerabilities are defined according to the preconditions that must be enforced to exploit them. These preconditions are: No access to the code (*Stand-Alone component*), access to classes (*Shared Classes*), access to objects (*Shared Objects* or *SOP*). These component vulnerabilities are referenced in two vulnerability catalogs: the *Malicious Bundles* catalog [15], which identifies vulnerabilities that can be exploited through malicious components and are implied by platform features, and the *Vulnerable Bundles* catalog [16], which identifies vulnerabilities that are implied by component features, mostly based on Java language properties [7]. Following features of the Java/OSGi Platform lead to component vulnerabilities: the reflection API, SOP services, and fragments. Other entries of the *Malicious Bundles* catalog are not considered here, since they concern the implementation of the platform and the isolation mechanisms it enforces, and not the way components are coded.

So as to provide an overview of the relative importance of each vulnerability category, their cardinality is extracted.

The total number N of vulnerabilities that we identify in Java/OSGi components is: 6 vulnerabilities from the *Malicious Bundles* catalog, and all 33 vulnerabilities from the *Vulnerable Bundles* catalog.

$$N = 6 + 33 = 39$$

The number N_{SA} of vulnerability in stand alone components is 1, which matches the use of serialization. When not properly protected, it provides access to any entity that is able to read the serialized data, for instance through the network or the file system. This vulnerability may not be restricted to component platforms.

$$N_{SA} = 1$$

Vulnerabilities that pertain to the *Shared Classes* vulnerability category can be exploited provided that two conditions are met. First, victim code must be loaded by the same ClassLoader as the attack CODE, OR be shared among ClassLoaders. In the Java/OSGi case, this concerns exported packages as well as bundle fragments and their hosts. Secondly, the code must be launched by the application. In OSGi, this is for instance done through the bundle activator, or when methods are called. These vulnerabilities occur mainly when static fields exists in the code, and when reflection, inheritance and fragments can be exploited.

$$N_{CL} = 18$$

Some vulnerabilities require the execution of a given method, which can be achieved either through static access or through SOP, depending on the implementation. This is the case *e.g.* for synchronization problems. They can therefore not be classified in one or the other category, though for simplicity one can consider them to be SOP vulnerabilities, because they are much more likely to be exploitable in this case.

$$N_S = 2$$

The vulnerabilities that pertain to the *Object Sharing* category can be executed provided that a malicious component can be installed, and that access to objects is granted. This is typically the case in SOP Platforms. For instance, in OSGi, it is possible to access all objects that are registered as services. The number N_{SOP} of vulnerabilities in the *Object Sharing* category is:

$$N_{SOP} = 18$$

Figure 1 provides an overview of the vulnerability categories in a SOP Platforms.

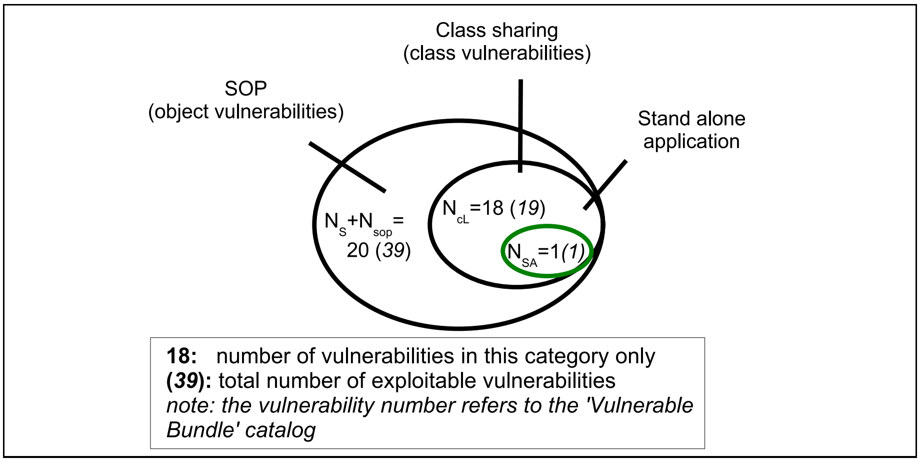
Class sharing
(class vulnerabilities)
SOP
(object vulnerabilities)
Stand alone application
N_S+N_sop = 20 (39)
N_cL=18 (19)
N_SA=1(1)
18: number of vulnerabilities in this category only
(39): total number of exploitable vulnerabilities
note: the vulnerability number refers to the 'Vulnerable Bundle' catalog

Fig. 1. Vulnerability types in a SOP Platform

3.2 Vulnerability Implementations

Vulnerability lists are usually given without regard to their actual likeliness. The following taxonomy provides for each system configuration the set of vulnerabilities that can be exploited without further effort. Each vulnerability category extends the others: Stand Alone Application vulnerabilities can also be exploited in case of Class Sharing system, and Class Sharing vulnerabilities can be exploited in the context of SOP.

Table 1 present the taxonomy for vulnerability categories according to the vulnerability category they pertain to.

Table 1. Taxonomy: Implementations of the Component Vulnerabilities in Java/OSGi SOP Platform

Attack Vector	**Implementation**			**Occurences**
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 instanciation	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

Stand Alone Applications. Stand alone applications do not enable to run third party code. The only code-level vulnerability that can be exploited in this case is the access to internal data that is made available through serialization. This flaw can be prevented by avoiding serialization, or by properly protecting, for instance through cryptography, the serialized data.

Other vulnerabilities may of course also exist, but they are related with the application behavior itself, not with the code properties, and are therefore not of interest here.

Class Sharing. Platforms that support Class Sharing are typically component-based systems. Each component can make classes available, and have dependencies to others. In the OSGi Platform, for instance, this feature is supported by the Module Layer. Class Sharing makes two main category of vulnerabilities open for exploits: *Exposed Internal Representation* and *Avoidable Calls to the Security Manager*. In some specific cases, the *Synchronization* vulnerability can also be exploited.

Exposed Internal Representation enables malicious components to access data inside victim components. In the Class Sharing case, it enables to execute code that should remain hidden, and to access static class members[2]. Four sub-categories exist: *Mutable element in static variable*, *Reflection*, *Fragments* and *No suitable control*.

- *Mutable element in static variable* vulnerabilities consist in giving access to third party components to fields that are both static and final, but which content can nonetheless be modified. This occurs when the fields either contain arrays, or mutable classes. Mutable classes are any classes that store data that the client object can modify. For instance, implementations of the `Set` and `Collection` interfaces are mutable. The only way to prevent this vulnerability to be exploited is to ban such constructs from public variables of public code classes.
- The *Reflection* sub-category consists in exploiting the Reflection API to access and exploit the content of the victim component. It encompasses code observation, component data modification when this data is static, and launching hidden method . The protection against these vulnerabilities are of two types. First, clean encapsulation can prevent unwanted access, since reflection does not allow to access fields and execute methods when visibility modifiers (public, protected, default, and private) forbid it. Secondly, Java Permissions can be set to prevent untrusted components from using the Reflection API.
- *Fragments* vulnerabilities exploit the OSGi-specific fragments. Fragments are used to provide configuration data and code to OSGi bundles, *e.g.* for supporting context specific behaviors such as internationalization. Fragment code is executed in the same ClassLoader as its Host bundle. This enable them to have full access to the code, and to share this access with other components by exporting it. Three implementations exist for this vulnerability category. First, a fragment can access the classes inside its host bundle. It can call classes that do not pertain to public code. Next, the *split package* feature enable to gain access to package protected classes, fields and methods, if the fragment contains a package with the same name as the targeted package in the host. Lastly, private inner classes, which are made package protected at compilation, are thus available from the fragment. Protection against fragments consists in setting `BundlePermission:HOST` and `BundlePermission:FRAGMENT` to trusted components only.
- *No suitable control* vulnerabilities enable to influence the behavior of the application through class access. In particular, shutdown hooks[3] can be exploited to keep a handle on an object after all references have been destroyed in the application. This enables in particular the execution of code after components have been uninstalled. The protection against the shutdown hook attack can be obtained by preventing untrusted components to set such hooks, *e.g.* through Java Permissions.

[2] Class members are fields and methods.

[3] Shutdown hooks are methods that are executed during the shutdown process of the virtual machine. They can be set at any moment.

Avoidable Calls to the Security Manager enables malicious components to by-pass security checks that occur in the code. These vulnerabilities are either exploited by overriding the code that contains the check, or by taking advantage of methods that are executed in spite of the presence of a Security Manager (or any similar check).

Two types of vulnerabilities are identified: avoidable checks 'at instantiation', which allows to create protected objects, and avoidable checks 'in method call', which allows to perform protected actions.

– The avoidable checks *at instantiation* category consists either in not using a constructor that contains security checks to create objects, or in overriding it in a sub-class. Object creation without constructor can be achieved either through the `clone()` method, or through de-serialization, when these two mechanisms are not protected. The protection consists in performing the same security checks in all constructors, in the `clone()` method if the class is `cloneable`, and in the `readObject()` method if the class is `serializable`. Avoiding security checks through overriding simply consist in re-writing the methods that contain the checks. This is possible either if a constructor exists that does not contain checks, or if the checks are performed in other methods. Consequently, these methods should always be final to prevent exploitation.
– The avoidable checks *in method call* category consists in performing actions that should be prevented by the security policy. The simplest way to achieve this is to override a method that contains a security check by a self-defined one. Executing methods of objects which creation has aborted due to security reasons is also possible: the `finalize()` method is always executed, even through the constructor could not be properly executed. Calls on the object, which is in a such case often only partially initialized, can typically reveal internal data. The protection here is to perform security checks at the very beginning of the creator method (constructor or other), to prevent data to be set before the security check. The last vulnerability that avoids security checks consists in executing sensible operations on behalf on untrusted components. This is done through `doPrivileged()` calls. A specific case can occur with security checks that depends only on the local ClassLoader, such as `java.lang.Class.forName` and `java.lang.Class.newInstance`. The protection against these two vulnerabilities is to never execute sensitive operations on behalf of others.

Synchronization vulnerabilities threaten Java/OSGi Platforms with freezing: if a `synchronized` method call does not return, all subsequent calls keep waiting for the lock to be released. Exploiting these vulnerabilities requires either that the `synchronized` call freezes by itself, or that the malicious component is able to interfere with its execution, for instance by providing a malicious service on which the victim method relies. So as to make attack through Shared Classes possible, these methods must be launched through a static method call, either directly (the `synchronized` method is also static) or indirectly (the `synchronized` method is called by a static method). Attack is triggered when this malicious service freezes and thus blocks the `synchronized` call. Two implementations exist: either a full method is synchronized, or a code block inside a method. This vulnerability occurs without regard to the location of the `synchronized` keyword inside the component: they are not restricted to public code. The ways to prevent them is to ban `synchronized` code from components, or to ensure that only trusted and non-freezing components are called by `synchronized` statements.

Service Oriented Programming (SOP). Service Oriented Programming Platforms support the dynamic registration and discovery of local services, *i.e.* objects that are characterized by the interface they implement. In the OSGi Platform, for instance, this feature is supported by the Service Layer. Service Oriented Programming provides full access to the service objects, which means that both read and write access is granted. The vulnerabilities that plague SOP are the following: **Exposed Internal Representation**, **Flaws in Parameter Validation**, and **Invalid Workflow**. Moreover, the exploitation of **Synchronization** vulnerabilities is much easier, since synchronized methods can be targeted without requiring a static access.

Exposed Internal Representation enables, as in the Class Sharing case, malicious components to access data inside victim components. In the SOP case, these vulnerabilities enable malicious code to access and thus modify data that should be kept internal to the object. Two vulnerability categories exist: *Returns reference to mutable element* and *No suitable control*.

– The *Returns reference to mutable element* category occurs when a method returns these very mutable elements. If a proper copy is not performed before giving a reference of a mutable object to a third party component, this latter is able to modify it. Malicious or accidental conflicts can then occur between the modifications that take place inside the vulnerable component, and the modifications that are performed by the caller. The protection consists in copying the mutable element before returning it. This can only be achieved if the considered mutable element does not itself contain mutable elements. Otherwise, the copy process would be overly complex and error prone.
– The *No suitable control* category in the *Object Sharing* vulnerability class enables information leak from one component to another. It encompasses the absence of wrapper (no encapsulation), an excessive visibility for the members[4] or classifier[5]. The protection against ill-coded public classes vulnerabilities consists in a proper encapsulation of all variables. Another vulnerability is the leak of configuration, system, or application sensitive data through exceptions. Exception handing should therefore either be performed internal to the component, or only provide generic data that contain at most references to user input to keep the message informative without revealing the internal component state.

[4] Class members are fields and methods.
[5] Classifiers are classes and interfaces.

Flaws in Parameter Validation enables malicious or ill-coded components to call methods from other ones while passing objects as parameter that are either not supported or lead to unexpected code behavior.

Four sub-categories exist: 'Unchecked parameter', 'Checked parameter without copy', 'Checked and copied parameter', and 'Non final parameter'.

– The *Unchecked parameter* category occurs when the method parameters are not checked before use. It contains three vulnerabilities: accidentally unsupported values that cause the program to behave in an erratic manner, malicious Java code, and malicious native code. In this latter case, the caller can forge and provide arbitrary malicious code. In particular, parameters that are defined as interfaces or as non final classes are vulnerable. The protection against such abuses consists in checking both the value and the actual type of the parameters. Public class methods should only accept parameters which types are final classes, so as to prevent malicious inheritance. Lastly, no native code should be executed on behalf of other components.
– The *Checked parameter without copy* vulnerability consists in performing the validation of the parameter, but without previously copying it to a local variable. If the object is modified in the caller component after the validation occurs, it can take arbitrary values, including those which are rejected by the validation process. The absence of parameter copy makes parameter validation useless because of TOCTOU (Time of Check to Time of Use) attacks. The suitable protection consists of course in copying the parameter object before its validation.
– The *Checked and copied parameter* category highlights the restriction of the parameter copy process: unless an object is serializable and thus explicitly states which fields are `transient` and are thus not required during copy, copying it is not necessarily straightforward. Two types of vulnerabilities exist. The first one is the presence of fake clone methods or copy constructor, which are provided by the malicious parameter itself: a copy statement is present in the code, but does not perform as expected. The protection against this problem is to use trustworthy copy methods only, such as those provided by the Java API, or manual copy. The second type of vulnerabilities is related to the manual copy process, which can be uncomplete. This occurs either when some states are omitted during the copy process, or when the given object contains references to other objects. This later problem implies that parameter objects should have a limited depth of mutable objects so as to prevent copy faults and omissions.
– The *Non-final parameter* category consists in exploiting the extensibility of classes or the possibility of providing self-defined implementation of interfaces to execute arbitrary code. This can also lead to more complex scenario: a malicious parameter can be used to trigger execution of code in the caller bundle, possibly passing back data from the victim bundle. This actually builds a case of malicious inversion of control (see Appendix A.1). As we already mentioned, the protection against this vulnerability is to allow only basic and final types as method parameters in public classes. A copy

mechanism designed to avoid cited flaws can also prevent this vulnerability category from being exploited, as the object passed as parameter is no longer used during the method execution.

Invalid Workflow (SOP) vulnerabilities are bound with invalid configuration of the service dependencies. In Java/OSGi platforms, services are discovered and retrieved through the `BundleContext`, which plays the role of local service repository. Service lookup is performed according to a given Java interface, with possibly additional provider-set properties. Consequently, very little control is enforced, in particular when components from several mutually untrusted providers coexist in a SOP platform. This lack of control have two main consequences. First, there is no guarantee that found services actually provide a valid implementation of the advertised interface. They could either provide arbitrary code, or gather data that is passed to them as parameters. Secondly, there is no guarantee that the service call does not abort. Such abortion can be generated either directly, for instance by systematically throwing exceptions, or indirectly, for instance by creating loops between services that lead to `StackOverflowError`s. To date, most SOP frameworks assume that provided services are benevolent. The identified risks show that a full SOP security framework should be designed if this should not be the case. This is a requirement for future work.

3.3 Goals of the Attacks That Exploit These Vulnerabilities

The goals of the attacks that exploit vulnerabilities in Java/OSGi component interactions are described below. The main goals are *Undue Access* and *Denial of Service*. Undue access is either *Access to internal Data* or *By-pass Security Checks*. Denial of Service (DoS) is restricted to method unavailability, because it is achieved through method calls on the public code. More serious DoS attacks can be performed in Java/OSGi platforms by attacking the platform directly [15].

The taxonomy of the goals of the attacks that can be performed by taking advantage of vulnerabilities in Java/OSGi component interactions is shown in Table 2, along with related vulnerability categories.

Undue Access - Access to internal Data. Hackers can gain *Access to internal Data* through the *Exposed Internal Representation* and the *Fragments* vulnerabilities.

Table 2. Taxonomy: Goals of the Attacks that exploit Vulnerabilities in Java/OSGi Component Interactions

Attack Goal	**Sub-goal**	**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

The first vulnerability provides access to internal data of the component that provides ill-coded Shared Classes of Shared Objects. The second one provides access to all the code of the target component, but without access to the actual objects. Attacks are performed by malicious client components.

Undue Access - By-pass Security Check. Hackers can *By-pass Security Check* through the *Avoidable Calls to the security manager* and the *Flaws in parameter validation* vulnerabilities. The first vulnerability enables to execute code that is not properly protected by security checks. It is specific to the Shared Classes vulnerability category. The second one enables to pass unvalid or malicious code as method parameters. It is specific to the Shared Object vulnerability category. Attacks that exploit both weaknesses are performed by malicious client components.

Denial of Service - Method Unavailability. Hackers can force *Method Unavailability* through the *Synchronization* and the *Invalid SOP Workflow* vulnerabilities. Both vulnerabilities enable to block the normal execution of programs, by freezing them of by forcing them to abort. In most cases they are bound with the Shared Object vulnerability category, but synchronization can also be exploited through Shared Classes. Attacks that exploit both weaknesses are performed by malicious servant components, *i.e.* malicious components which are dependencies of the victim code.

4 Experiments

A vulnerability is any feature that forces a program to behave so that it breaks the implicit or explicit security policy of the considered system [8]. They are generated by errors in the program development or by assumptions that are not valid in the execution context. In the case of vulnerabilities in Java/OSGi component interactions, the second case holds: the Java language has not been designed to support the execution of mutually untrusted components in the same virtual machine.

4.1 Rationale

The rationale for identifying, validating and classifying these vulnerabilities is the following. First, we gather knowledge about Java behaviors that are considered as the expression of vulnerabilities. Sources are the computer science literature as well as our own experience. Secondly, the Java Language Specification is analyzed to identify further vulnerabilities, and to check that no language construct has been neglected [7]. Next, the suspected vulnerabilities are validated through proof of concept implementation of the attack scenarios. Lastly, taxonomies are created to classify both vulnerability type - their implementation - and the goal of the attacks based on the experiment results.

This rationale is strongly inspired by similar studies that focus on Operating System vulnerabilities, such as those by Landwehr [10] and Lindqvist [11] for Unix.

4.2 Implementation of Malicious and Vulnerable SOP Components

Each identified vulnerability must be validated by implementing it so as to confirm that it actually breaks security requirements.

The experiment environment is the Java/OSGi Platform. Tests are conducted on the Sun JVM 1.6, with the Apache Felix[6] open source implementation of the OSGi Platform. Felix 1.0.0 is compliant with the OSGi Release 4 Specifications.

An implementation of a vulnerability validates this vulnerability if the malicious component actually performs an operation on the vulnerable one that breaks the implicit or explicit security policy, *i.e.* that either is able to perform more operations that calling provided methods, or enforces a denial of service.

For each of the 39 vulnerability occurrences that we identify, a malicious/ vulnerable component pair is implemented. Providing an implementation for each attack has a twofold goal. It enables to validate the feasibility of the attack, and provide a sound basis for our documentation effort. And it makes sample code available for subsequent effort toward automated vulnerability identification.

5 Conclusions and Perspectives

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,

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

- 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 contribution of this paper is twofold. First, taxonomies that describes the categories of exploitable vulnerabilities and their goals for Java systems are defined. The three main system types are stand alone applications, multi-component systems, and Service Oriented Programming (SOP) Platforms, which each make a specific set of vulnerabilities directly exploitable. Secondly, recommendations are issued to help software developers build more secure code.These recommendations can be used for training, or to enrich the flaw sets that are identified by static analysis tools. Our approach is validated through a systematic implementation of each vulnerability through a proof-of-concept malicious / vulnerable pair of OSGi bundles. Experiments show that given vulnerabilities are actually directly exposed to malicious components in standard platforms. The only condition is that the malicious component can be installed and executed to perform its abuses.

The perspective of this work is first to disseminate the knowledge gathered through the development of plug-ins for static analysis tools such as FindBugs or PMD.

Further requirements are also identified. A security framework should be defined and developed to enforce security at the SOP level. Current tools, such as SCR for the OSGi Platform, do not take security into account. This mechanism should be made mandatory and support for instance dynamic proxies that would prevent the exploitation of identified vulnerabilities by isolating service implementation and service client. Such a feature could prove to provide a big improvement in the quest after the dynamic discovery of unknown components from the environment, while ensuring that the system security is not at risk.

Aknowledgement

This work has been made possible by valuable discussions with Emmanuel Coquery (LIRIS, Lyon 1 University) and Nicolas Geoffray (LIP6, Paris VI University).

References

1. Bieber, G., Carpenter, J.: Introduction to service-oriented programming (rev 2.1). OpenWings Whitepaper (April 2001)
2. Cotroneo, D., Orlando, S., Russo, S.: Failures classification and analysis of the java virtual machine. In: 26th IEEE International Conference on Distributed Computing Systems (ICDCS 2006) (2006)
3. Debbabi, M., Saleh, M., Talhi, C., Zhioua, S.: Security evaluation of j2me cldc embedded java platform. Journal of Object Technology 5(2), 125–154 (2005)
4. Dolbec, J., Shepard, T.: A component based software reliability model. In: CASCON 1995: Proceedings of the 1995 conference of the Centre for Advanced Studies on Collaborative research, p. 19. IBM Press (1995)

5. Govindavajhala, S., Appel, A.W.: Using memory errors to attack a virtual machine. In: Symposium on Security and Privacy (2003)
6. Hovemeyer, D., Pugh, W.: Finding bugs is easy. In: ACM SIGPLAN Notices, vol. 39, p. 92–106 (2004); COLUMN: OOPSLA onward
7. Steele, G., Bracha, G., Gosling, J., Joy, B.: Java Language Specification, 3rd edn. Addison-Wesley Professional, Reading (2005)
8. Krsul, I.V.: Software Vulnerability Analysis. PhD thesis, Purdue University (May 1998)
9. Lai, C.: Java insecurity: Accounting for subtleties that can compromise code. IEEE Software 25(1), 13–19 (2008)
10. Landwehr, C.E., Bull, A.R., McDermott, J.P., Choi, W.S.: A taxonomy of computer program security flaws, with examples. In: ACM Computing Surveys, September 1994, vol. 26, pp. 211–254 (1994)
11. Lindqvist, U., Jonsson, E.: How to systematically classify computer security intrusions. In: IEEE Symposium on Security and Privacy, pp. 154–163 (May 1997)
12. Long, F.: Software vulnerabilities in java. Technical Report CMU/SEI-2005-TN-044, Carnegie Mellon University (October 2005)
13. OSGI Alliance. Osgi service platform, core specification release 4.1. Draft, 05 2007
14. Parnas, D.L., Wang, Y.: The trace assertion method of module interface specification. Technical Report 89-261, Dept. of Computing and Information Science, Queen's Univ. at Kingston, Ontario, Canada (October 1989)
15. Parrend, P., Frenot, S.: Java components vulnerabilities - an experimental classification targeted at the osgi platform. Research Report RR-6231, INRIA, 06 (2007)
16. Parrend, P., Frenot, S.: More vulnerabilities in the java/osgi platform: A focus on bundle interactions. Technical report, INRIA (to be released, 2008)
17. Sun Microsystems Inc. Secure coding guidelines for the java programming language, version 2.0. Sun Whitepaper (2007), `http://java.sun.com/security/seccodeguide.html`
18. The Last Stage of Delirium. Research Group. Java and java virtual machine. security vulnerabilities and their exploitation techniques. In: Black Hat Briefings (2002)

A Appendix

The Appendix presents additional informations related to vulnerabilities in Java/ OSGi component interactions. Subsection A.1 gives a detailed documentation for two vulnerabilities that exist in component-based applications: *Malicious Inversion of Control through overridden Parameters*, and *Synchronized Code.*

A.1 New Attacks Exploiting Interactions between Java Components

We now present two behaviors that enable malicious components to exploit weak ones in order to achieve security breaks inside component-based applications: *Malicious Inversion of Control through overridden Parameters*, and *Synchronized Code*. The first vulnerability enables an attack that performs undue access to code. The second one enables an attack that performs denial of service. To the best of our knowledge, these behaviors of Java components have not yet been identified and documented as vulnerabilities.

The *Malicious Inversion of Control through overridden Parameters* vulnerability occurs when public code expose methods with non-final parameters. This is the case for all parameters that are defined as interfaces, and most classes with the exception of basic type wrappers (`Integer`, *etc*) and `String`. Abuse occurs when called methods are overwritten, and trigger actions that are not supposed to take place such as spying the behavior of the servant bundle or getting undue access to internal data. An example of an attack that exploits this vulnerability is given in Figure 2 as an UML Component Diagram.

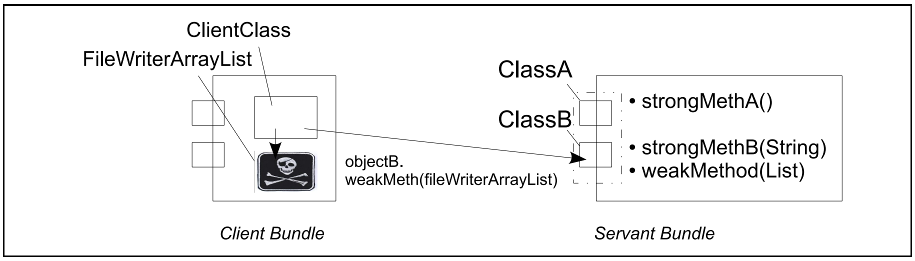

Fig. 2. An Example Scenario of malicious Inversion of Control: Component Diagram

The weak method, named `weakMethod(List)`, is provided by the class `ClassB` of the servant bundle. In our example, it simply manipulates the `List` parameter. The attack is performed as follows. First, the client bundle defines a malicious `FileWriter- ArrayList`, whose `iterator()` method is overwritten and triggers action that it should not. In our case, this is a single text print for demonstration. The client bundle creates a `FileWriterArrayList` object, and passes it as parameter to the `ClassB.weakMethod(List)` method. When code in `ClassB.weakMethod(List)` is executed, malicious code is executed seamlessly. Again, the example does not go further than the demonstration, but shows how a naive servant can execute unrequired code from its caller.

This vulnerability has one main consequence: public code that is intended to be executed by not fully trusted code should never provide methods with non final parameters.

The *Synchronized Code* vulnerability occurs when code in a public class is tagged as `synchronized`, which means that one single client bundle can access it at a time. Synchronization is used in particular to protect transactions or access to system resources. Abuse occurs when the synchronized method is forced to hang, which causes all subsequent calls to the method to freeze. An example of an attack that exploits this vulnerability is given in Figure 3 as an UML Sequence Diagram.

The synchronized method, `setData()`, is provided by the `Data` class. This service relies on another one, `DataStorage`. A default valid scenario is executed by Alice, which is a benevolent component that stores data every 20 seconds. The attack is performed as follows. First, the `DataStorage` service must be replaced

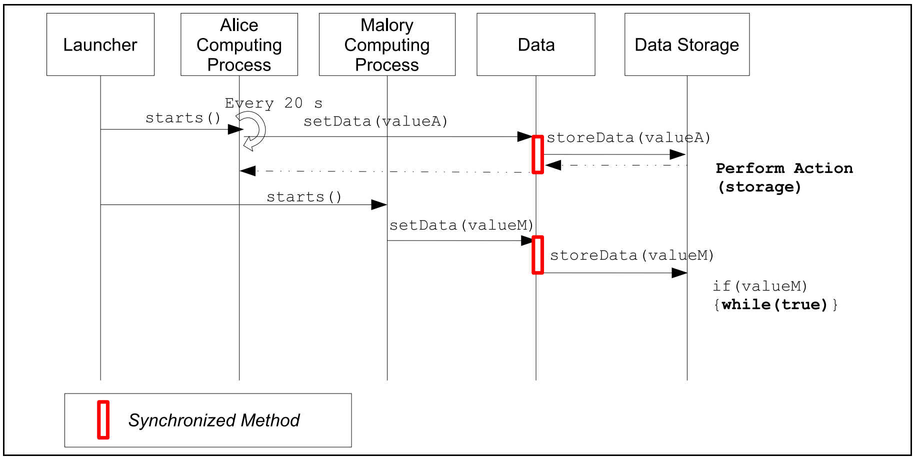

Fig. 3. An Example Scenario for an Attack against a Synchronized Method: Sequence Diagram

by a malicious one, which hangs under certain circumstances (here, a specific `valueM` value of the transmitted data is the signal for hanging). This substitution can be replaced by a Denial-Of-Service Attack against a valid implementation of the `DataStorage` service. The Mallory component is the accomplice of the malicious `DataStorage` service, and therefore knows how to trigger its freezing (transmit data with 'valueM' value). It performs the malicious call to the `Data` service, which in turn calls the `DataStorage` service, which hangs. As a consequence, Alice as well as any other client of the `Data` service will hang.

The *Synchronized Code* vulnerability exists under two flavours: Synchronized method and Synchronized code block. This vulnerability has two consequences. First, access to synchronized methods MUST be granted to trusted components only. Secondly, services on which synchronized methods rely MUST be guaranteed to be trustworthy.