More Vulnerabilities in the Java/OSGi Platform: A Focus on Bundle Interactions

Pierre Parrend, Stéphane Frénot

INSTITUT NATIONAL DE RECHERCHE EN INFORMATIQUE ET EN AUTOMATIQUE

More Vulnerabilities in the Java/OSGi Platform: A Focus on Bundle Interactions

Pierre Parrend — Stéphane Frénot

N° 6649

09 2008

Thème COM

ISSN 0249-6399 ISRN INRIA/RR--6649--FR+ENG

More Vulnerabilities in the Java/OSGi Platform: A Focus on Bundle Interactions *

Pierre Parrend, Stéphane Frénot

Thème COM — Systèmes communicants
Projet ARES

Rapport de recherche n° 6649 — 09 2008 — 78 pages

Abstract: *Extensible Component Platforms* can discover and install code during runtime. Although this feature introduces flexibility, it also brings new security threats: malicious components can quite easily be installed and exploit the rich programming environment and interactions with other components to perform attacks against the system. One example of such environments is the Java/OSGi Platform, which widespreads in the industrial world.

Attacks from one component against another can not be prevented through conventional security mechanisms, since they exploit the lack of proper isolation between them: components often share classes and objects. This reports intends to list the vulnerabilities that a component can contain, both from the literature and from our own experience. The *Vulnerable Bundle catalog* gathers this knowledge. It provides informations related to the characteristics of the vulnerabilities, their consequence, the security mechanisms that would help prevent their exploitation, as well as to the implementation state of the proof-of-concept bundles that are developed to prove that the vulnerability is actually exploitable.

The objective of vulnerability classification is of course to provide tools for identifying and preventing them. A first assessment is performed with existing tools, such as Java Permission and FindBugs, and a specific prototype we develop, WBA (*Weak Bundle Analysis*), and manual code review.

Key-words: Software Security, Vulnerability Benchmarking, Code Static Analysis, Java Language, Component Platforms, OSGi

* This Work is partially founded by the ANR-07-SESU_007 LISE Project.

Unité de recherche INRIA Rhône-Alpes
655, avenue de l'Europe, 38334 Montbonnot Saint Ismier (France)
Téléphone : +33 4 76 61 52 00 — Télécopie +33 4 76 61 52 52

De Nouvelles Vulnérabilités dans la Plate-forme Java/OSGi: Etude des Interactions entre les Bundles

Résumé : *Les Plates-formes Extensibles à composants* peuvent découvrir et installer des programmes pendant leur exécution. Bien que cette possibilité introduise de la flexibilité, elle apporte également de nouvelles menaces de sécurité: des composants malveillants peuvent être aisément installés, et exploiter l'environment de programmation de la plate-forme, ainsi que les interactions avec les autres composants, pour attaquer le système. Un exemple d'environnement de ce type est Java/OSGi, qui devient de plus en plus utilisé dans l'industrie.

Les attaques d'un composant contre un autre ne peuvent pas être évitées par des mécanismes de sécurité conventionnels, dans la mesure où elles exploitent le manque d'isolation entre composants, qui partagent souvent des classes et des objets. Ce rapport liste les vulnérabilités possibles d'un composant à partir de la litérature mais également de notre propre expérience. Le catalogue *Bundles Vulnerables* rassemble ces données. Il contient des informations concernant les caractéristiques des vulnérabilités, leur conséquence, les mécanismes de sécurité qui permettent d'éviter leur exploitation, de même que des informations concernant la mise en œuvre des bundles que nous développons pour démontrer que ces vulnérabilités sont aisément exploitables.

L'objectif d'une telle classification de vulnérabilités est bien sûr de fournir des outils afin de les identifier et de les prévenir. Une première évaluation est réalisée, avec des outils existants comme les Permissions Java et FindBugs, un outil ad hoc que nous avons développé, WBA (*Weak Bundle Analysis*), ainsi que l'examen manuelle de code.

Mots-clés : Sécurité Logicielle, Evaluation de Vulnérabilités, Analyse statique de Code, Langage Java, Plates-formes à Composants OSGi

Contents

List of Figures

List of Tables

1 Introduction

Java Component Platforms increasingly become dynamic, *i.e.* they support the installation of new components at runtime. These components can be downloaded from the environment, and are often not provided by the owner of the platform itself. These advanced features imply that the installed code is not controlled before being installed. Even though security mechanisms such as digital signature are set up, it is relatively easy for attackers to publish malicious components and get them installed.

So as to enable the use of such *Extensible Java Component Platforms* in production environments, security tools must be available. This in turn requires that the security implications of such platforms are well known, which is currently not the case. Malicious components that are installed on a Java platform can exploit two different entities: the platform itself, and other components. Attacks against the Java/OSGi Platform are the subject of an earlier work we did [PF07]. Attacks against other components can be limitated through the enforcement of development best practices [HP04, Lai08], but the risks that they involve are not known precisely. Other attacks could be performed against Java component platform, but they are not specific to component support, and thus are beyond the scope of this work: attacks from the local host against the Java Component Platform, or attacks that are performed through remoting technologies (RMI, web services, etc.).

For these reasons, we intend to identify and classify all attacks that can be performed from a malicious component against a naive one. To provide data that can be used by application developers to build secure systems, the catalog lists the vulnerabilities that make these attack possible, rather than the attack themselves. Our experiments are conducted with the OSGi Platform, which is a prototypical example of *Extensible Java Component Platforms*. This platform is more likely than others to be plagued by attacks through malicious Java Components, since components (or Bundles in the OSGi language) from unknown providers can be dynamically discovered and installed, which is usually not the case in static component platforms such as EJB or Spring Platforms. The vulnerabilities are presented as a catalog. For each entry, we provide data that characterize them, as well as informations related to existing or potential protections, and to the development status of the related proof-of-concept bundles.

This Research Report is organized as follows. Section 2 presents related works that highlight the possible abuse of Java component-based applications. Section 3 presents specific attacks against java components, at the example of OSGi bundles, taxonomies that enable to classify these attacks, and assessment of these attacks. Section 4 concludes this work.

2 Abusing Java Component-based Applications

Attacks against Java Component-based Systems can be performed in two ways: by exploiting the vulnerabilities of the platform itself, or by exploiting the vulnerabilities of the Java components. Exploits that are based on the latter necessarily abuse the code that is made available to third party components: the *public code*. Even though no systematic work exists that reference such exploits, several recommendations have been published and are presented here.

2.1 Attack Vectors in Java Component-based Applications

Abusing a Java Component Platform can be done in three ways.

- First, available remoting mechanisms can be exploited. This is not specific to Java-based systems, and abuses are highly protocol dependent. One typical example is provided by Web Services [BK07].
- Secondly, the Java Component Platform can be attacked through access from the local Operating System. This can be done in particular through the JVMTI (JVM Tool Interface) tool [Sun], and in not bound with component properties.
- Thirdly, the Java Component Platform can be attacked by malicious components that are installed inside it. Both platform and other components can be abused, since only limited controls are available to prevent such misuse.

Attacking component platforms through malicious components is likely to become more widespread, since dynamic platforms such as OSGi [OSG07] and MIDP [JSR02] are becoming more successfull: they enable the discovery and installation of components from the environment, *i.e.* from potentially unknown and thus uncontrolled sources. Attacks that exploit vulnerabilities of the Java/OSGi Platform itself have been described in previous work [PF07]. The exploits are referenced in the *Malicious Bundles* catalog. The current study focuses on attacks where malicious bundles exploit vulnerabilities in other installed bundles and aims at providing a systematic overview of these vulnerabilities. They are presented in the *Vulnerable Bundles* (VB) catalog. They actually build a complement of the first catalog, as shown in Figure 1. The full catalog is presented in Appendix C.

Attacks against Java Component Platforms are due to two main types of vulnerabilities: those originating in the intra-component structure, and those originating in inter-component interactions.

Figure 1 shows the relationship between the topics covered in the *Vulnerable Bundle* catalog which focuses on inter-component interactions (in bold red line) and the component structure, using the example of Java/OSGi Platforms. This taxonomy is given for this platform specifically to support a precise description of the actual structure of the considered components.

Intra-component structure can be parted in three distinct entities: the archive, the manifest which contains the meta-data, and the application code. This latter is compound of the activator, native code, Java standard API calls, Java language constructs, and OSGi

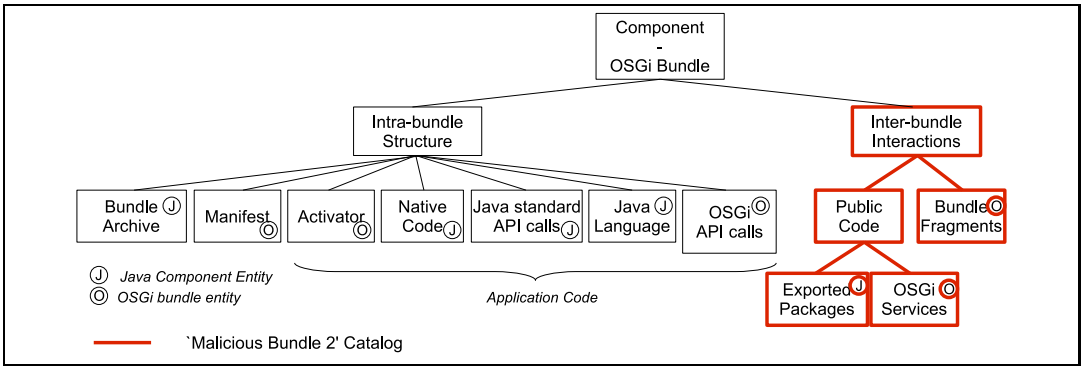

Figure 1: Topics covered in the 'Vulnerable Bundle' Catalog
[Their relationship with the taxonomy for describing OSGi bundles [PF07].]

API calls. Inter-bundle interactions are either performed through Fragments or public code, *i.e.* exported packages and registered services.

Our previous work [PF07] presents in detail attacks that exploit the platform itself. However, building secure component platforms also requires that the components interactions are protected.

2.2 Known Attacks exploiting Public Code

Known exploits against Java code are referenced by the FindBugs project, and by Sun in its 'Secure Coding Security Guidelines'.

The first set of candidate vulnerabilities that flaw Java components is provided by the Findbugs tools Vulnerability List [1] [HP04], and namely in the *Malicious Code* Vulnerability category. Findbugs entries list the vulnerabilities that are related to code exposition by a given code element (*e.g.* an OSGi bundle) to another potentially untrusted code element.

Findbugs entries for the *Malicious Code* Vulnerability category are given in Appendix B.1.

The second set of candidates vulnerabilities that flaw Java components is provided by the 'Sun Java Security Coding Guidelines' [Sun07]. Each guideline is a good practice that matches a code flaw that can be exploited by untrusted code to perform malicious actions.

Sun Java Security Coding Guidelines are given in Appendix B.2.

These entries are often too complex and context-dependent to be subject to Static Code Analysis. However, flaws and solutions are well documented which makes possible to prevent these vulnerabilities through careful manual code review.

Sun Java Security Coding Guidelines are completed by Charlie Lai's Java Insecurity Subtleties [Lai08].

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

We do not pretend to cover all existing vulnerabilities that can occur in bundle interactions in a Java Component Platform, nor in the specific Java/OSGi platform. In particular, more flaws are presented and corrected in the SafeJava Language [Boy04]. However, this work proposes to solve the problems it addresses by modifying the type system of Java. This is clearly beyond the scope of this study which intends to provide support for building secure systems in standard production environments.

3 A Classification of Attacks against Java Components

So as to properly write component public code, potential attacks are to be listed systematically and in a structured way, and solutions for their prevention are to be identified and assessed. The first step is to document attacks that are so far not referenced. The second step is to propose a taxonomy of attacks that can be conducted against component public code. The last step is to describe and assess the properties of these attacks. Documentation for the attacks *Malicious Inversion of Control through overridden Parameters* and *Synchronized Code*, which are to the best of our knowledge not documented precisely elsewhere, is provided in Section 3.2.

3.1 Taxonomy of Attacks exploiting Inter-Bundle Interactions

Writing secure public code requires two complementary informations. First, a taxonomy of attacks must be available for training developers. Secondly, an extensive catalog of vulnerabilities must be published as a reference work. A list of the vulnerabilities is presented here. The catalog is given in Appendix C.

Figure 2 represents the taxonomy for exploiting the public code Attack Vector.

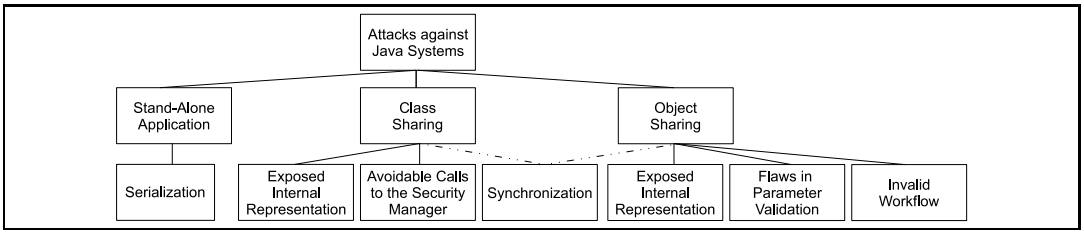

Figure 2: The Taxonomy for exploiting the Public Code Attack Vector

The main categories in this taxonomy are the *Stand-Alone* applications, *Class-sharing* mechanisms, and *Object Sharing* mechanisms. Stand-Alone applications concerns leak of data that do not require access to the application code. Class-sharing mechanisms induce vulnerabilities that can be exploited by access to libraries or packages. In the case of the OSGi Platform, this concerns packages that are exported by bundles. Object-sharing mechanisms induce vulnerabilities that can be exploited only when access to shared objects is provided. In the case of the OSGi Platform, this concerns objects that are registered as SOP singleton services, which is the default configuration. In specific cases, vulnerabilities can be exploited either through Class-sharing or through Object-sharing. This is the case of synchronization of code blocks, which can lead to the freezing of subsequent calls if the synchronized block does not return. This vulnerability usually concerns Object-sharing mechanisms, but can also be exploited in the case of Class-sharing if static access to the synchronized block is provided.

The 32 identified vulnerabilities in Java Component code are classified according to their category in the taxonomy.

Stand-Alone Applications Vulnerabilities

Expose Internal Representation - Serialized Sensitive Data (vb.java.1) All data in a serialized object can be read. In particular, security checks that may exist in the code are no longer enforced. No sensitive data must be stored in serializable objects.

Class Sharing Vulnerabilities - Exposed Internal Representation

Stores Mutable Object in static Variable (vb.java.class.1) A method stores a reference to a mutable object in a static variable. Internal Data of the victim object can be read and/or modified.

Stores Array in Static Variable (vb.java.class.2) A method stores a reference to a array in a static variable. Process. Internal Data of the victim object can be read and/or modified.

Non Final Static Variable (vb.java.3) A method keeps a reference to a static non final static object. Internal Data of the victim object can be read and/or modified.

Shutdown Hook (vb.java.class.4) Shutdown Hooks enable to execute code when the platform is stopped. In particular, this implies that components can execute code after they have been uninstalled.

Private nested Classes and Attributes made protected (vb.java.class.5) Private nested classes and attributes are made protected at compilation. Consequently, OSGi Bundle Fragments can be exploited to access the target package through the 'Split Package' vulnerability, and access the private Class or Attribute as a protected one.

Class Sharing Vulnerabilities - Avoidable Calls to the Security Manager

Override Method (vb.java.class.6) Security
Checks that are performed in overridable methods can be by-passed by rewriting the methods.

Privileged Execution of Code provided by the Caller (vb.java.class.7) Privileged Code Execution must be restricted to code provided by the privileged bundle. If code origin is not properly controlled, less trusted bundles can provide their own code for Privileged Execution.

Privileged Execution of Code provided by the Caller - Class Loader Privileges (vb.java.class.8) Privileged Code Execution must be restricted to code provided by the privileged bundle. If code origin is not properly controlled, less trusted bundles can provide their own code for Privileged Execution. Privileged Execution is granted for several calls according to the current ClassLoader (Reflection, Library Loading).

Cloning (vb.java.class.9) Calls to the 'clone' method enable to create a new instance of a class without calling the constructor, which often contains security checks such as calls to the Security Manager.

Deserialization (vb.java.class.10) Deserialization enables to create a new instance of a class without calling the constructor, which often contains security checks such as calls to the Security Manager.

Call Overrideable Method in Constructor (vb.java.class.11) Calling non-final methods in constructor enable sub-classes to access to partially initialized instances of the objects, and break security and configuration assumptions that are made in the superclass.

Call Overrideable Method in Clone Method (vb.java.class.12) Calling non-final methods in clone method enable sub-classes to access to partially initialized instances of the objects, and break security and configuration assumptions that are made in the superclass.

Finalize Method (vb.java.class.13) Methods on a Class that is protected through a Security Manager can be called by creating a subclass that, after creation abortion, performs calls on the partially initialized object during finalization.

Shutdown Hook (vb.java.class.14) Shutdown Hooks enable to execute code when the platform is stopped. In particular, this implies that components can execute code after they have been uninstalled. Moreover, if a security check is performed in the constructor after static global variable have been initialized, their value can be accessed.

Class or Object Sharing - Synchronization

Synchronized Method (vb.java.15) A method is synchronized, to as to avoid the execution of the same method by two different clients (used in particular in case of access to resources). If the method call is blocked for any reason (infinite loop during execution, or delay due to an unavailable remote resource), all subsequent clients that call this method are freezed.

Synchronized Code (vb.java.16) A
method contains a synchronized block, so as to avoid the execution of the same method by two different clients (used in particular in case of access to resources). If the method call is blocked for any reason (infinite loop during execution, or delay due to an unavailable remote resource), all subsequent clients that call this method are freezed.

Object Sharing Vulnerabilities - Exposed Internal Representation

Returns Reference to Mutable Object (vb.java.object.1) A method returns a reference to a mutable object. Internal Data of the victim object can be read and/or modified.

Returns Reference to Array (vb.java.object.2) A method returns a reference to a array. Internal Data of the victim object can be read and/or modified.

Visibility (vb.java.object.3) A method keeps a reference to a variable with too much visibility. Internal Data of the victim object can be read and/or modified.

non Final non Private Field (vb.java.object.4) A method keeps a reference to a static non final non private object. Internal Data of the victim object can be read and/or modified.

No Wrapper (vb.java.object.5) Variables can be accessed on the object without wrapper methods, which prevent the execution of security or parameter checks. Variables can be accessed directly on the object.

Information Leak through Exceptions (vb.java.object.6) Exception Messages often contain data that describes the configuration of the system. These data should not be propagated to external callers, unless it directly concerns caller input.

Object Sharing Vulnerabilities - Flaws in Parameter Validation

Unchecked Parameters - Malicious Program Abuse - Java Code (vb.java.object.7) Un- checked parameters in bundle public code (OSGi Services or Exported Packages) can be exploited to execute malicious code.

Unchecked Parameters - Malicious Program Abuse - Native Code (vb.java.object.8) Un- checked parameters in bundle public code (OSGi Services or Exported Packages) can be exploited to execute malicious code, especially native code that does not provide any security guarantees.

Unchecked Parameters - Accidentally unsupported Value (vb.java.object.9) Unchecked parameters in bundle public code (OSGi Services or Exported Packages) can lead to unexpected program behavior if constraints on their values are not enforced.

Parameters Checked without Copy (vb.java.object.10) A parameter that is checked without being copied beforehand can be modified after validation and lead a TOCTOU (Time of Check To Time of Use) attack.

Copied and Checked Parameters - Fake Clone Method (vb.java.object.11) Copying parameters before their validation can be worthless if the copy is done through an overridden 'clone' method that is implemented partially or with a malicious objective.

Copied and Checked Parameters - Fake Copy Constructor (vb.java.object.12)Copying parameters before their validation can be worthless if the copy is done through a fake copy constructor that is implemented partially or with a malicious objective.

Copied and Checked Parameters - Uncomplete Copy - State Omission (vb.java.object.13) Copying parameters before their validation can be unsufficient if some states are omitted.

Copied and Checked Parameters - Uncomplete Copy - Mutable States (vb.java.object.14) Copying parameters before their validation can be unsufficient if some states are mutable.

Non Final Parameters - Malicious Implementation (vb.java.object.15) Non-final parameters in bundle public code (SOP Services or Exported Packages) can be exploited to execute malicious code, possibly exploiting internal data of the victim bundle.

Non Final Parameters - Inversion of Control (vb.java.object.16) Non-final parameters in bundle public code (SOP Services or Exported Packages) can be exploited to execute malicious code through inversion of control, ie. actions in the malicious bundle can

be triggered through the victim bundle, possibly leaking data. Data from the victim bundle can be exploited. Certain type of access control mechanisms can be by-passed.

3.2 Attack Examples

Two vulnerabilities from the catalog are presented. To the best of our knowledge, they are so far not documented. The first vulnerability is *Malicious Inversion of Control through overridden Parameters*. The second one is *Synchronized Code*.

The *Malicious Inversion of Control through overridden Parameters* vulnerability is referenced as vulnerability **vb.java.object.16**. It occurs when a *Servant* bundle has public code that exposes methods with non-final parameters [2]. 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 3 as an (extended) UML Component Diagram.

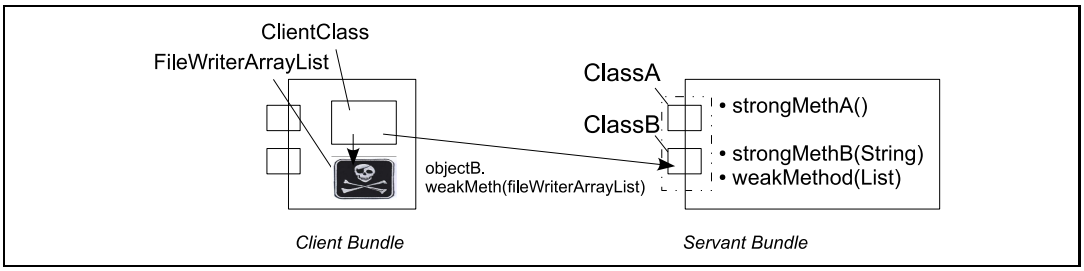

Figure 3: An Example Implementation of malicious Inversion of Control: Component Diagram

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 `FileWriterArrayList`, 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.

[2] A *Servant* bundle is a bundle that provides code to other, as opposed to a *Client*. Each bundle can be simultaneously servant and client

```
package fr.inria.ares.exporterbundle;

public class ClassA
{
  public void strongMethodA()
  {
    System.out.println("method ClassA.strongMethodA");
  }
}

public class ClassB
{
  //print name
  public void strongMethodB(String name)
  {
    System.out.println("method ClassB.strongMethodB");
    System.out.println("Be polite. Say \" Hello "+ name+" \"");
  }

  //print name list
  public void weakMethod(List names)
  {
      Iterator it=names.iterator();
      String name;
      while(it.hasNext())
      {
         name=(String)it.next();
         System.out.println(name);
      }
  }
}
```

Table 1: Code of a Servant Bundle

The following listings give the implementation of the example of abuse of a servant bundle, which provides the public code, by a Client Bundle. Listing 1 shows the classes that are exported by the servant bundle. Listing 2 shows the code that is used by a malicious Client to abuse the latter servant.

The second vulnerability we document is *Synchronized Code*, which is referenced as vulnerability **vb.java.15** (synchronized method) and **vb.java.46** (synchronized code). It occurs when code in a public class is tagged as `synchronized`, which means that one single

```
package fr.inria.ares.controlinverterbundle.activator;

import fr.inria.ares.exporterbundle.ClassA;
import fr.inria.ares.exporterbundle.ClassB;
import fr.inria.ares.controlinverterbundle.FileWriterArrayList;
public class Activator implements BundleActivator
{
   public void start(BundleContext ctx)
   {
       System.out.println("controlinverterbundle activator started");
       (new ClassA()).strongMethodA();
       ClassB bibi = new ClassB();
       bibi.strongMethodB(new String("Malory"));
       ArrayList crackers = new FileWriterArrayList();
       crackers.add("eve");
       crackers.add("malory");
       bibi.weakMethod(crackers);
   }

   public void stop(BundleContext ctx)
   {
       System.out.println("controlinverterbundle activator stopped");
   }
}

---
package fr.inria.ares.controlinverterbundle;

public class FileWriterArrayList extends ArrayList
{
   public Iterator iterator()
   {
     System.out.println("Beware, I am Malory !!");
     return super.iterator();
   }
}
```

Table 2: Code of a Client Bundle that abuses its Servant

client bundle can access it at a time. Synchronization is used in particular to protect transactions. Abuse occurs when the synchronized method is forced to hang, which causes all subsequent calls to the method to freeze.

Figure 4 shows the UML Sequence Diagram for an example of this attack.

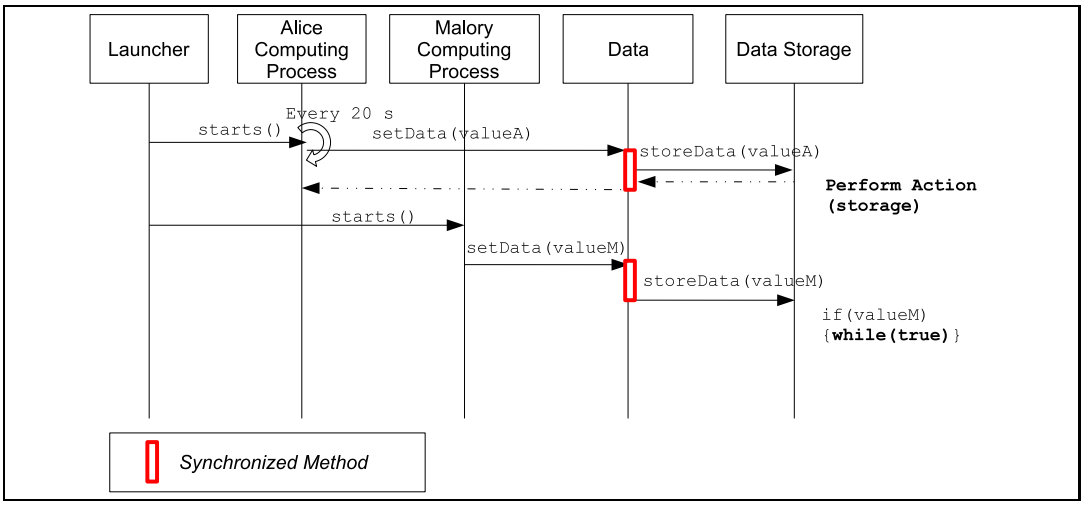

Figure 4: An Example Scenario for an Attack against a Synchronized Method

The detail of the implementation of this example in now given.

The Implementation of the Service 'Data', with a synchronized method, is given in the Listing 3.

The implementation of the malicious 'DataStorage' service is given in the Listing 5.

The implementation of the malicious 'DataStorage' service is given in the Listing 5.

These two example of Java component vulnerability show that strong restrictions have to be put on Java components if secure systems are to be built using code from heterogeneous providers. First, tight access control is to be enforced. Secondly, programming constraints are to be enforced on public code: synchronized methods should not be tolerated, and all method parameters should be final.

3.3 Assessment of the considered Attacks

The assessment of the considered attacks is performed in two steps. First, these attacks must be described in a systematic manner. Secondly, related security protections must be identified, and their efficiency is to be quantified.

The description of the attacks is performed through the Descriptive Vulnerability Pattern, which enable to gather informations related the vulnerabilities that are exploited to perform each attack. This Descriptive Vulnerability Pattern is defined in [PF07], where it is

```
package fr.inria.ares.data;

import fr.inria.ares.datastorage.DataStorage;

public class DataImpl
  implements Data
{
   private DataStorage storage;

   public DataImpl(DataStorage distantStorage)
   {
       this.storage=distantStorage;
   }

   public synchronized void setData(String value)
   {
     if(storage!=null)
     {
        storage.storeData(value);
     }
   }
}
```

Table 3: Implementation of the Service 'Data', with a synchronized method

```
package fr.inria.ares.datastorage;

import java.util.ArrayList;

public final class DataStorageImpl
  implements DataStorage
{
   ArrayList myArray;

   public void storeData(String value)
   {
      System.out.println("store data: "+value);
      if("valueM".equals(value))
      {
         while(1==1);
      }
      else
      {
         if(myArray==null){myArray=new ArrayList();}
         myArray.add(value);
      }
   }
}
```

Table 4: Implementation of the Service 'Data', with a synchronized block

```
package fr.inria.ares.datastorage;

import java.util.ArrayList;

public final class DataStorageImpl
  implements DataStorage
{
   ArrayList myArray;

   public void storeData(String value)
   {
      System.out.println("store data: "+value);
      if("valueM".equals(value))
      {
         while(1==1);
      }
      else
      {
         if(myArray==null){myArray=new ArrayList();}
         myArray.add(value);
      }
   }
}
```

Table 5: Implementation of the malicious 'DataStorage' Service

simply referred to as the 'Vulnerability Pattern'. It must be extended so as to support the specific properties that appear in the *Vulnerable Bundles* vulnerability catalog. An updated version of the pattern is given in Appendix A.

Two mains updates are done, related to the taxonomy for interactions between bundles and to the potential attack targets.

Figure 5 shows the updated taxonomy for the interactions between bundles.

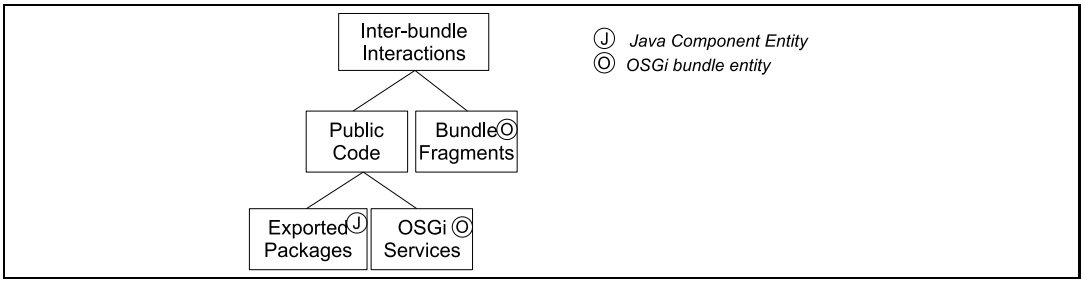

Figure 5: The updated Taxonomy for Inter-Bundle Interactions

The taxonomy now details the public code entity, which is compound of exported packages and SOP services.

Figure 6 shows the updated taxonomy for Attack Targets in OSGi-Based Systems.

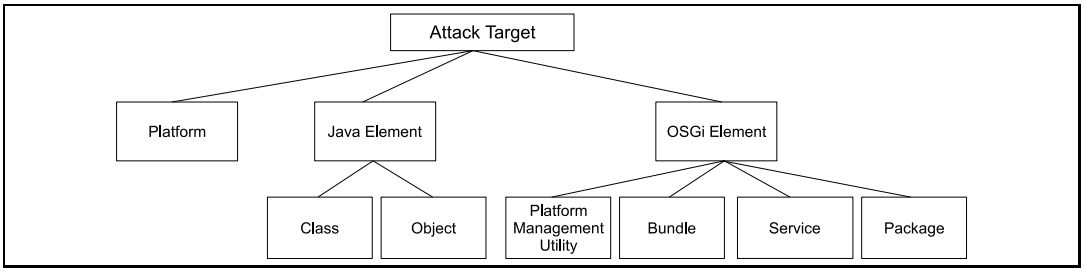

Figure 6: The updated Taxonomy for Attack Targets in OSGi-Based Systems

The attack target taxonomy is extended with Java elements, *i.e.* Classes and Objects, which where not explicitly mentioned in the original version.

Figure 7 shows the type of intrusion techniques that can be exploited against OSGi Bundles, according to Neumann and Parker's classification [NP89]. This classification encompasses all technological and non technological techniques that can be exploited to abuse a system: external misuse (NP1), hardware misuse (NP2), masquerading (NP3), setting up subsequent misuse (NP4), by-passing intended control (NP5), active misuse of resources such as write or execution access to the system (NP6), passive misuse of resources such as reading (NP7) and misuse resulting from inaction (NP8). In the case of attacks that exploit

vulnerable bundles, only technological and direct techniques can be exploited: by-passing intended control, active misuse and passive misuse of resources.

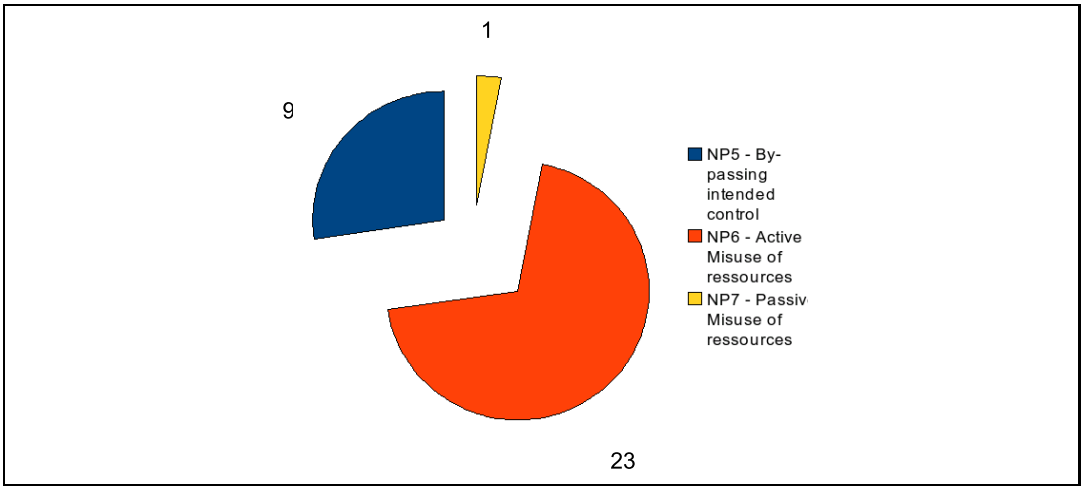

Figure 7: Type of Intrusion Techniques that can be exploited against OSGi Bundles

The technique that is used in most cases is *Active Misuse of Resources*, which tantamounts to 23 occurrences, out of the 33 considered vulnerabilities. The related vulnerabilities enable to modify data that should be kept internal to the bundles, or more marginally to freeze the bundle calls through Denial-of-Service attacks. The second technique is *By-passing intended Control*, which tantamounts to 9 occurrences. These vulnerabilities enable to by-pass checks such as calls to the `SecurityManager` or verification of parameters. The last technique is *Passive Misuse of Resources*, which concerns 1 occurrence, namely Data leak in `Exceptions` (vb.java.object.6).

Figure 8 provides an overview of the type of consequence of attacks against the OSGi Bundles, according to the taxonomy we developed for the *Vulnerability Pattern* for security benchmarking of component platforms [PF07], and according to Lindqvist classification [LJ97].

Almost all attacks that can be performed on OSGi bundles lead to *Undue Access*, being either simple exposure (1 occurrence, 3 %), or erroneous output, *i.e.* modification of the data (30 occurrences, 91 %). Two vulnerabilities (6 %) can be exploited to force the unavailability of specific method calls. This graphics also shows that, contrary to the attacks against the OSGi Platform, partial Denial-of-Service attacks such as performance breakdown can be be directly be performed by attacking the bundles.

Three security mechanisms are identified in the vulnerability catalog to protect from attacks on vulnerable bundles: Java Permission, Code Review and Static Analysis. Security Assessment is performed to compare them.

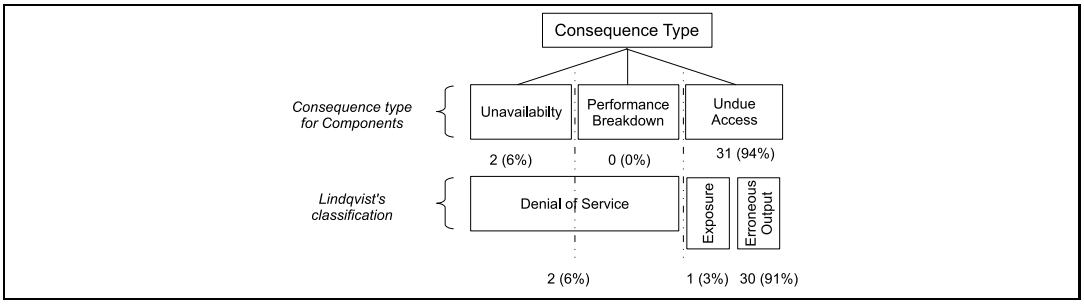

Figure 8: Type of Consequences of attacks against OSGi Bundles

The first assessment tool is the attack protection table, which highlights the efficiency of each security mechanism for all considered attacks. This table is given in Table 6. The `X` sign indicates that the vulnerability is prevented by the related security mechanism. The question mark indicates that the vulnerability should be relatively easy to prevent, but that current state of our prototype does not implement it in a satisfactory manner.

Compared tools are Java Permissions, manual code review, the Findbugs tool, and our WBA (*Weak Bundle Analysis*) prototype. The Findbugs tool provides an important set of vulnerabilities that are referenced, but do not seem to identify them all in the test bundles. This is due to the fact that it intends to provide programming best practices for all classes in a given system. Our test consisted in identifying all constructs that enable one bundle to get access to information of another one, which is a more aggressive scenario. The protection of bundles against others imply to put more drastic constraints on their code than the constraints that can reasonably be set on all classes of an application, as intended by the FindBugs tool. The WBA tool is a static analysis tool which identifies code patterns that correspond to the given vulnerabilities. A first prototype has been developed to show that static analysis is a promising method for identifying the given vulnerabilities, bu it does not constitute the subject of this report.

Vulnerability	**Permission**	**Code Review**	**Findbugs**	**WBA**
Expose Internal Rep - Serialize Data (vb.java.1)		X		CBAC
Expose Internal Rep - Mutable static Variable (vb.java.class.1)		X	-	X
Expose Internal Rep - Array static Variable (vb.java.class.2)		X	-	X
Expose Internal Rep - Non-final static Variable (vb.java.class.3)		X	-	X
Expose Internal Rep - Shutdown Hook (vb.java.class.4)	X	X		CBAC
Private Nexted Class (vb.java.class.5)	in OSGi	X	-	X
By-pass Security Checks - override method (vb.java.class.6)		X		?
By-pass Security Checks - Privileged Execution (vb.java.class.7)		X		
By-pass Security Checks - Privileged Execution through Class Loader Structure (vb.java.class.8)		X		-
By-pass Security Checks - Clone (vb.java.class.9)		X		?
By-pass Security Checks - Deserialization (vb.java.class.10)		X		CBAC
By-pass Security Checks - call overridable method in constructor (vb.java.class.11)		X	-	?
By-pass Security Checks - call overridable method in clone() (vb.java.class.12)		X	-	?
By-pass Security Checks - malicious finalization (vb.java.class.13)		X	-	X
By-pass Security Checks - Shutdown Hook (vb.java.class.14)	X	X		CBAC
Synchronized method in public code (vb.java.15)		X	-	X
Synchronized block in public code (vb.java.16)		X	-	X
Expose Internal Rep - Returns Mutable (vb.java.object.1)		X	-	X
Expose Internal Rep - Returns Array (vb.java.object.2)		X	-	X
Expose Internal Rep - Too much Visibility (vb.java.object.3)		X		-
Expose Internal Rep - Non-final Non-Private Variable (vb.java.object.4)		X	-	X
Expose Internal Rep - No Wrapper (vb.java.object.5)		X		-

Vulnerability	**Permission**	**Code Review**	**Findbugs**	**WBA**
Expose Internal Rep - Data in Exception (vb.java.object.6)		X		-
Unchecked Malicious Parameters - Java (vb.java.object.7)		X		-
Unchecked Malicious Parameters - Native (vb.java.object.8)		X		-
Unchecked Unsupported Parameters (vb.java.object.9)		X		-
Checked Parameters without Copy (vb.java.object.10)		X		-
Fake Clone Method (vb.java.object.11)		X		-
Fake Copy Constructor (vb.java.object.12)		X		-
Uncomplete Copy - Omission (vb.java.object.13)		X		-
Uncomplete Copy - Mutable States (vb.java.object.14)		X		-
Non-final Malicious Parameter (vb.java.object.15)		X	-	X
Non-final Parameter - Inversion of Control (vb.java.object.16)		X	-	X
Total	**3**	**33**	**0**	**12+5(+3)**

Table 6: Prevention against Component Vulnerabilities

The second assessment tool used is the *Protection Rate* metric, that has proved to be a simple and efficient measure of the security coverage that is provided by a given security mechanism. The Protection Rate is based on the Attack Surface metric [HPW05]. It is defined as follows:

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

The Protection Rate for each security mechanism is the following. Java Permissions enable to protect from 3 attacks out of 32, *i.e.* 9 %. Static Analysis enables to protect from 17 attacks out of 32, according to our first tests, *i.e.* 53 %. Reasonable development efforts should extend this result to 20 protected attacks, *i.e.* 62,5 %. Code review by experienced reviewers should be able to identify all vulnerabilities, *i.e.* 100 %. However, since it is not automated, this top quality can not be considered as guaranteed. The limit of both code review and static analysis is that a lot of features that are considered as vulnerabilities

are common in Java development. Consequently, code analysis must be performed in the development phase to adapt the code to identified requirement. It is best used as warning generator rather than rejection oracle, unless the code is especially developed to cope with these recommendations. Otherwise, it is very likely that most if not all legacy components will be rejected.

This security assessment analysis highlights two main requirements. First, developers need to be trained in order to develop safe bundles, and to be able to perform code review on their own code. Secondly, suitable static code analysis tools must be developed and integrated together, so as to exploit this technique as a full-fledged security mechanism.

4 Conclusions and Perspectives

Our contribution in this report is the *Vulnerable Bundle* catalog, which identifies 33 occurrences of vulnerabilities that enable a malicious Java component, in our case an OSGi bundle, to perform attacks against other components that are installed on the same platform.

The constitution of this catalog implies to adapt the *Vulnerability Pattern for security benchmarking of component Platforms* [PF07], which is used to document the vulnerabilities, so as to support properties of vulnerabilities that are specific to the component themselves. The lightweight extensions that are required show that the original proposition can be considered as a stable one.

A first assessment of security mechanisms that would help prevent the exploitation of these vulnerabilities is provided. Java Permissions, the FindBugs tool, our WBA (Weak Bundle Analysis) static analysis prototype, as well as manual code review are considered. Java Permissions are not of great help to protect components against each others, except in marginal cases such as the exploitation of Shutdown hooks which enable to execute code just before the OSGi Platform is shut down. The FindBugs tool proves to be unsufficient to support proper access isolation of bundles. It is actually meant to improve the overall quality of code by applying best practices, but is not suitable to solve the specific security problem of vulnerabilities in interactions between untrusted Java components. The WBA tool is a promising approach: the vulnerabilities are represented under the form of a formal vulnerability pattern, and their presence in the public code of bundles is detected through pattern matching. However, some complex vulnerabilities can only be identified through manual code review. This approach theoretically supports the identification of all vulnerability times, but it is error prone and very time consuming,

The availability of the *Vulnerable Bundle Catalog* opens several perspectives. First, the identified vulnerabilities, which have been identified in the context of the Java/OSGi Platform, are very likely to be found on other Java component platforms such as Spring, J2EE and the EJBs, and so on. More experiments are to be conducted to check whether the identified vulnerabilities are also exploitable in these environments, and whether other vulnerabilities appears in each specific platform. Secondly, tools that can be used in production environments are to be developed so as to identify and patch the vulnerabilities. The WBA tool is a first attempt in this direction. The current prototype should be completed to more completely integrate vulnerabilities that are identified by other tools such as FindBugs, and to support realistic use during the development process of OSGi bundles - for instance as Eclipse or Maven plugins.

This work provides important knowledge to enforce proper access isolation between mutually untrusted components that are installed on the same platform. It should be completed with the second type of isolation: resource isolation, since so far it is not possible to prevent one bundle to consume an important part of resources such as CPU or memory that are shared across all components.

References

[BK07] Nishall Bhalla and Sahba Kazerooni. Web services vulnerabilities. In *BlackHat Europe, Amsterdam*, 2007.

[Boy04] C. Boyapati. *SafeJava: A Unified Type System for Safe Programming*. PhD thesis, Massachusetts Institute of Technology, 2004.

[CO05] D. Crocker and P. Overell. Augmented bnf for syntax specifications: Abnf. IETF RfC 4234, October 2005.

[HP04] David Hovemeyer and William Pugh. Finding bugs is easy. In *ACM SIGPLAN Notices*, volume 39, pages 92 – 106, 2004. COLUMN: OOPSLA onward.

[HPW05] M. Howard, J. Pincus, and J.M. Wing. *Computer Security in the 21st Century*, chapter Measuring Relative Attack Surfaces, pages 109–137. Springer, March 2005.

[JSR02] JSR 118 Expert Group. Midp 2.0. Sun Specification, November 2002.

[Lai08] Charlie Lai. Java insecurity: Accounting for subtleties that can compromise code. *IEEE Software*, 25(1):13–19, 2008.

[LJ97] Ulf Lindqvist and Erland Jonsson. How to systematically classify computer security intrusions. In *IEEE Symposium on Security and Privacy*, pages 154–163, May 1997.

[NP89] P. G. Neumann and D. B. Parker. A summary of computer misuse techniques. In *Proceedings of the 12th National Computer Security Coifererice*, page 3961107, Baltimore, Maryland, USA, October 1989.

[OSG07] OSGI Alliance. Osgi service platform, core specification release 4.1. Draft, 05 2007.

[PF07] Pierre Parrend and Stéphane Frénot. Java components vulnerabilities - an experimental classification targeted at the osgi platform. Research Report 6231, INRIA, 06 2007.

[Sun] Sun Microsystems, Inc. Jvmtm tool interface (jvm ti).

[Sun07] Sun Microsystems Inc. Secure coding guidelines for the java programming language, version 2.0. Sun Whitepaper, 2007. http://java.sun.com/security/seccodeguide.html.

Appendix

A The Descriptive Vulnerability Pattern

This section presents the Vulnerability Pattern in the Augmented Backus Naur Form (BNF) [CO05].

The current grammar is not meant to be closed: it reflects the knowledge relative to the considered vulnerabilities at a given time. It can be extended with additional attribute values.

The catalog of the OSGi Malicious Bundles is referred as the 'mb' catalog.

Vulnerability Reference

- CATEGORY ::= text
- VULNERABILITY_NAME ::= text
- IDENTIFIER ::= CATALOG_ID.SRC_REF.ID
  with:
  CATALOG_ID ::= mb
  SRC_REF ::= archive|java(|.class|.object)|native|osgi
  ID ::= (0-9)*
- ORIGIN ::= text
- LOCATION ::= Bundle (Archive | Manifest | Activator | Fragment) | Application Code - (Native Code | Java (Code | Bytecode | API) | OSGi API)
- SOURCE ::= (ENTITY (FUNCTIONNALITY | FLAW ;)+;)+
  with ENTITY ::= OS | JVM - (Runtime API | APIs)| OSGi Platform - ((Module | Life-Cycle | Service) Layer | Bundle Repository Client)| Application Code | Compiler
  FUNCTIONNALITY ::= Kill utility | Value of Method Parameters | (System.exit | Runtime.halt | Runtime.addShutdownHook) method | Native Code Execution | Garbage Collection | Thread API | Reflection API | ClassLoader API | File API | Java Archive | Bundle Management | Bundle Fragments | No constructor call in 'clone' method | Serialization | No constructor call during object deserialization | Exceptions | Synchronization and:
  FLAW ::= No Algorithm Safety - (Java | Native Code)| Non OSGi R4-compliant Digital Signature Validation in the JVM | No Verification of Bundle Archive Validity | No Check of Size of Loaded Bundles | No Check of Size of stored Data | No safe Bundle Start | No Removal of Uninstalled Bundle Data | Bundle Meta-data Handling - No Safe-Default | Uncontrolled Service Registration | Architecture of the Application - No Validation of Service Dependency | Private Nested Class and Attributes are made Protected at Compilation | No Parameter Check in Bundle Public Code | Parameter Check without Copy in Public Code | Parameter Type is not final | Fake Clone Method | Fake Copy Constructor | Uncomplete Manual Copy | Expose Internal Representation | Non-final method (call in Constructor | with Security Checks) | Privileged Execution of Code provided by the Caller

- TARGET ::= Platform | Java Element - (Class | Object) | OSGi Element - (Platform Management Utility | Bundle | Service | Package) | Configuration Data
- CONSEQUENCE_TYPE ::= (Unavailability | Performance Breakdown | Undue Access | Unvalid Type or Value)(- (Platform | Service | Package | Class | Object | Native Code Execution | Configuration Data) (, (Platform | Service | Package | Class | Object | Native Code Execution | Configuration Data))*)?
- INTRODUCTION_TIME ::= Platform Design or Implementation | Development | Compilation | Bundle Meta-data Generation | Bundle Digital Signature | Installation | Service Publication or Resolution
- EXPLOIT_TIME ::= Download | Installation | Bundle Start | Execution

Vulnerability Description

- DESCRIPTION ::= text
- PRECONDITIONS ::= text
- ATTACK_PROCESS ::= text
- CONSEQUENCE_DESCRIPTION ::= text
- SEE_ALSO ::= VULNERABILITY_NAME (, VULNERABILITY_NAME)*

Vulnerability Implementation

- CODE_REFERENCE ::= FILE_NAME
  with FILE_NAME the name of a file, as defined by Unix File Names
- OSGI_PROFILE ::= - | CDC-1.0/Foundation-1.0 | OSGi/Minimum-1.1 | JRE-1.1 | J2SE-1.2 | J2SE-1.3 | J2SE-1.4 | J2SE-1.5 | J2SE-1.6 | PersonalJava-1.1 | PersonalJava-1.2 | CDC-1.0/PersonalBasis-1.0 | CDC-1.0/PersonalJava-1.0
- DATE ::= MONTH.DAY.YEAR
  with MONTH ::= (1-12), DAY ::= (1-31), YEAR ::= (0-3000)
- TEST_COVERAGE ::= (0-100) %
- TESTED_ON ::= Oscar | Felix | Knopflerfish | Equinox

Protection

- EXISTING_MECHANISMS ::= Java Permissions | OSGi AdminPermission | SFelix OSGi Security Layer | Check Internal Object State in Code - Initialization | Copy and Check Object State in Code - Parameters | Use final Types for Parameters| Deep Copy of Data - (Before making it Public | Method Parameters) | Avoid ((non Private | Static) non Final Variables | Synchronization) in Public Code | Use Wrapper Methods to Access Variables | Reduce Variable Visibility as much as Possible | Add manual Call to the Security Manager | Purge Exceptions from Sensitive Data before Propagation | Make all methods that (are called in instanciation methods - Constructor, clone or deserialization - | contain security checks) final | FindBugs static analysis | -
- ENFORCEMENT_POINT ::= Platform startup | Bundle Installation | Execution |-

- POTENTIAL_MECHANISMS ::= (POTENTIAL_MECHANISM_NAME (POTENTIAL_MECHANISM_DESCR)?)+ with POTENTIAL_MECHANISM_NAME ::= Code static Analysis | Code Rewriting - Insert Initialization Checks | OSGi Platform Modification - (Bundle Startup Process | Installation Meta-data Handling | Service Publication | Bundle Uninstall Process) | Bundle size control before download | Service-level dependency validation | Resource Control and Isolation - (CPU | Memory | Disk Space)| Access Control - FileSystem | Miscellaneous | - and POTENTIAL_MECHANISM_DESCR ::= text
- ATTACK_PREVENTION ::= Code Reviewing | Stop a ill-behaving thread | Avoid using private nested classes | -
- REACTION ::= Correct the flawed bundle | Uninstall the malicious bundle | Erase files | Stop the system process | Restart the platform | -

B Reference Vulnerability Lists

B.1 FindBugs Malicious Code Vulnerability

FindBugs entries for the *Malicious Code Vulnerability* category are the following:

- May expose internal representation by returning reference to mutable object
- May expose internal representation by incorporating reference to mutable object
- Finalizer should be protected, not public
- May expose internal static state by storing a mutable object into a static field
- Field isn't final and can't be protected from malicious code
- Public static method may expose internal representation by returning array
- Field should be both final and package protected
- Field is a mutable array
- Field is a mutable Hash table
- Field should be moved out of an interface and made package protected
- Field should be package protected
- Field isn't final but should be

B.2 Sun Guidelines

Sun Java Security Coding Guidelines are the following:

- 1-1 Limit the accessibility of classes, interfaces, methods, and fields
- 1-2 Limit the extensibility of classes and methods
- 1-3 Understand how a superclass can affect subclass behavior
- 2-1 Create a copy of mutable inputs and outputs
- 2-2 Support copy functionality for a mutable class
- 3-1 Treat public static fields as constants
- 3-2 Define wrapper methods around modifiable internal state
- 3-3 Define wrappers around native methods
- 3-4 Purge sensitive information from exceptions
- 4-1 Prevent the unauthorized construction of sensitive classes
- 4-2 Defend against partially initialized instances of non-final classes
- 4-3 Prevent constructors from calling methods that can be overridden
- 5-1 Guard sensitive data during serialization
- 5-2 View de-serialization the same as object construction
- 5-3 Duplicate the Security Manager checks enforced in a class during serialization and de-serialization
- 6-1 Safely invoke java.security.AccessController.doPrivileged
- 6-2 Safely invoke standard APIs that bypass Security Manager checks depending on the immediate caller's class loader
- 6-3 Safely invoke standard APIs that perform tasks using the immediate caller's class loader instance

- 6-4 Be aware of standard APIs that perform Java language access checks against the immediate caller

B.3 WBA Vulnerabilities

Vulnerabilities that are indentified in the WBA catalog are the following:

1. Avoid Private Nested classes and variables
2. Avoid Static non Final Variables in Public Code
3. Reduce Variable Visibility as much as possible
4. Use Wrapper methods to Access Variables
5. Copy and Check Object State in Code - Parameters
6. Use final Types for Parameters
7. Deep Copy of Data - Method Parameters or before making it public
8. Prevent by-passing of security checks at instantiation (Constructor, clone, de-serialization), by making those method final when possible and adding additional Calls to the Security Manager.
9. Make all methods that are called in instantiation methods - Constructor, clone or de-serialization - final
10. Make all methods that contain security checks final
11. Ensure that object initialization is completed before executing the code of any method
12. Forbid the use of finalizers (finalize, shutdown hooks)
13. Ensures that no Privileged Code is executed on behalf on another bundle
14. Never serialize sensitive data
15. Purge Exceptions from Sensitive Data before Propagation

C Catalog

This section provides the Catalog of new vulnerabilities in OSGi Platforms.

C.1 Stand-Alone Applications Vulnerabilities

C.1.1 Serialized Sensitive Data

Vulnerability Reference

- **Vulnerability Name:** Serialized Sensitive Data
- **Category:** Exposed Internal Representation
- **Identifier:** Vb.java.14
- **Origin:** Sun Secure Coding Guidelines for the Java Programming Language
- **Location of Exploit Code:** Application Code - Java Code
- **Source:** JVM - Runtime API (Serialization)
- **Target:** Java Element - Object
- **Consequence Type:** Undue Access - Object
- **Introduction Time:** Development
- **Exploit Time:** Execution

Vulnerability Description

- **Description:** All data in a serialized object can be read. In particular, security checks that may exist in the code are no longer enforced. No sensitive data must be stored in serializable objects.
- **Preconditions:** No technical precondition. Ill-coded program.
- **Attack Process:** Data that represents the serialized object is analyzed to extract the value of variables.
- **Consequence Description:** All data can be read.
- **See Also:** Deserialization

Protection

- **Existing Mechanisms:** -
- **Enforcement Point:** -
- **Potential Mechanisms:** Code static Analysis
- **Attack Prevention:** Code Reviewing
- **Reaction:** Correct the flawed bundle

Vulnerability Implementation

- **Code Reference:** Serializesensitivedata-0.1.jar
- **OSGi Profile:** J2SE-1.6

- **Date:** 2008-02-18
- **Test Coverage:** 10%
- **Known Vulnerable Platforms:** Oscar; Felix; Equinox; Knopflerfish

C.2 Class Sharing - Exposed Internal Representation

C.2.1 Stores Mutable Object in Static Variable

Vulnerability Reference

- **Vulnerability Name:** Stores Mutable Object in Static Variable
- **Category:** Exposed Internal Representation
- **Identifier:** Vb.java.class.1
- **Origin:** Findbugs Bug Patterns
- **Location of Exploit Code:** Application Code - Java Code
- **Source:** Application Code (Expose Internal Representation)
- **Target:** Java Element - Object
- **Consequence Type:** Undue Access - Object
- **Introduction Time:** Development
- **Exploit Time:** Execution

Vulnerability Description

- **Description:** A method stores a reference to a mutable object in a static variable.
- **Preconditions:** This vulnerability can be exploited in two ways: dynamic discovery of accessible fields with blind modifications, or on purpose modifications which require knowledge of code behavior (e.g. access to source code).
- **Attack Process:** Internal Data of the victim object is read and/or modified
- **Consequence Description:** -
- **See Also:** Stores Mutable Element in Static Variable - Array

Protection

- **Existing Mechanisms:** Avoid Static non Final Variables in Public Code
- **Enforcement Point:** Execution
- **Potential Mechanisms:** Code static Analysis
- **Attack Prevention:** Code Reviewing
- **Reaction:** Correct the flawed bundle

Vulnerability Implementation

- **Code Reference:** Mutableobjectinstaticvariable-0.1.jar
- **OSGi Profile:** J2SE-1.6
- **Date:** 2008-02-17
- **Test Coverage:** 50%
- **Known Vulnerable Platforms:** Oscar; Felix; Equinox; Knopflerfish
- **Known Robust Platforms:** SFelix

C.2.2 Stores Array in Static Variable

Vulnerability Reference

- **Vulnerability Name:** Stores Array in Static Variable
- **Category:** Exposed Internal Representation
- **Identifier:** Vb.java.class.2
- **Origin:** Findbugs Bug Patterns
- **Location of Exploit Code:** Application Code - Java Code
- **Source:** Application Code (Expose Internal Representation)
- **Target:** Java Element - Object
- **Consequence Type:** Undue Access - Object
- **Introduction Time:** Development
- **Exploit Time:** Execution

Vulnerability Description

- **Description:** A method stores a reference to a array in a static variable.
- **Preconditions:** No technical precondition. This vulnerability can be exploited in two ways: dynamic discovery of accessible fields with blind modifications, or on purpose modifications which require knowledge of code behavior (e.g. access to source code).
- **Attack Process:** Internal Data of the victim object is read and/or modified.
- **Consequence Description:** -
- **See Also:** Stores Mutable Element in Static Variable - Mutable Object

Protection

- **Existing Mechanisms:** Avoid Static non Final Variables in Public Code
- **Enforcement Point:** Execution
- **Potential Mechanisms:** Code static Analysis
- **Attack Prevention:** Code Reviewing
- **Reaction:** Correct the flawed bundle

Vulnerability Implementation

- **Code Reference:** Arrayinstaticvariable-0.1.jar
- **OSGi Profile:** J2SE-1.6
- **Date:** 2008-02-17
- **Test Coverage:** 100%
- **Known Vulnerable Platforms:** Oscar; Felix; Equinox; Knopflerfish
- **Known Robust Platforms:** SFelix

C.2.3 Non Final Static Field

Vulnerability Reference

- **Vulnerability Name:** Non Final Static Field
- **Category:** Exposed Internal Representation
- **Extends:** No Wrapper
- **Identifier:** Vb.java.class.3
- **Origin:** Findbugs Bug Patterns
- **Location of Exploit Code:** Application Code - Java Code
- **Source:** Application Code (Expose Internal Representation)
- **Target:** Java Element - Object
- **Consequence Type:** Undue Access - Object
- **Introduction Time:** Development
- **Exploit Time:** Execution

Vulnerability Description

- **Description:** A method keeps a reference to a static non final static object.
- **Preconditions:** No technical precondition. This vulnerability can be exploited in two ways: dynamic discovery of accessible fields with blind modifications, or on purpose modifications which require knowledge of code behavior (e.g. access to source code)
- **Attack Process:** Internal Data of the victim object is read and/or modified.
- **Consequence Description:** -
- **See Also:** Unsufficient Access Restriction - Field with too much Visibility

Protection

- **Existing Mechanisms:** Avoid Static non Final Variables in Public Code
- **Enforcement Point:** Execution
- **Potential Mechanisms:** Code static Analysis
- **Attack Prevention:** Code Reviewing
- **Reaction:** Correct the flawed bundle

Vulnerability Implementation

- **Code Reference:** Nonfinalstaticvariable-0.1.jar
- **OSGi Profile:** J2SE-1.6
- **Date:** 2008-02-17
- **Test Coverage:** 100%
- **Known Vulnerable Platforms:** Oscar; Felix; Equinox; Knopflerfish
- **Known Robust Platforms:** SFelix

C.2.4 Shutdown Hook

Vulnerability Reference

- **Vulnerability Name:** Shutdown Hook
- **Category:** Exposed Internal Representation
- **Identifier:** Vb.java.class.4
- **Origin:** Charlie Lai's Java Insecurity Subtleties
- **Location of Exploit Code:** Application Code - Java Code
- **Source:** JVM - Runtime API (Runtime.addShutdownHook method)
- **Target:** Platform
- **Consequence Type:** Undue Access - Platform
- **Introduction Time:** Development
- **Exploit Time:** Execution

Vulnerability Description

- **Description:** Shutdown Hooks enable to execute code when the platform is stopped. In particular, this implies that components can execute code after they have been uninstalled.
- **Preconditions:** ShutdownHooks Permission set. A malicious implementation is provided waiting for execution. Since this attack is based on inheritance, behavior of the mother class is assumed to be known during development.
- **Attack Process:** Shutdown Hooks Threads are set through the Runtime.addShutdownHook method.
- **Consequence Description:** Execution of code after the removal of the shutdown hook setter.
- **See Also:** -

Protection

- **Existing Mechanisms:** Java Permissions
- **Enforcement Point:** Bundle Installation
- **Potential Mechanisms:** Code static Analysis
- **Attack Prevention:** -
- **Reaction:** -

Vulnerability Implementation

- **Code Reference:** Bypass_shutdownhook.exporter-0.1.jar,bypass_shutdownhook.abuser-0.1.jar
- **OSGi Profile:** J2SE-1.6
- **Date:** 2008-02-16
- **Test Coverage:** 100%
- **Known Vulnerable Platforms:** Oscar; Felix; Equinox; Knopflerfish

C.2.5 Private Nested Class and Attributes made Protected

Vulnerability Reference

- **Vulnerability Name:** Private Nested Class and Attributes made Protected
- **Category:** Exposed Internal Representation
- **Extends:** Access Protected Code Through Split Package
- **Identifier:** Wb.java.class.5
- **Origin:** Sun Secure Coding Guidelines for the Java Programming Language
- **Location of Exploit Code:** Application Code - Java Bytecode
- **Source:** Compiler (Private Nested Class and Attributes are made Protected at Compilation)
- **Target:** Java Element - Class
- **Consequence Type:** Undue Access - Class
- **Introduction Time:** Compilation
- **Exploit Time:** Execution

Vulnerability Description

- **Description:** Private nested classes and attributes are made protected at compilation.
- **Preconditions:** Nested class is static. This vulnerability can be exploited in two ways: dynamic discovery of accessible fields with blind modifications, or on purpose modifications which require knowledge of code behavior (e.g. access to source code).
- **Attack Process:** Exploit Fragments to Access the target Package through the 'Split Package' vulnerability, and access to the private Class or Attribute as a protected one.
- **Consequence Description:** Execute code that is contained in the private class.
- **See Also:** -

Protection

- **Existing Mechanisms:** OSGi AdminPermission
- **Enforcement Point:** Bundle Installation
- **Potential Mechanisms:** Code static Analysis
- **Attack Prevention:** Avoid using private nested classes
- **Reaction:** Uninstall the malicious bundle

Vulnerability Implementation

- **Code Reference:** Privatenestedclass-0.1.jar;privatenestedclass.abuser-0.1.jar;private nestedclass.public-0.1.jar
- **OSGi Profile:** J2SE-1.6
- **Date:** 2008-02-14
- **Test Coverage:** 100%
- **Known Vulnerable Platforms:** Oscar; Felix; Equinox; Knopflerfish
- **Known Robust Platforms:** SFelix

C.3 Class Sharing Vulnerabilities - Avoidable Calls to the Security Manager

C.3.1 Override Method

Vulnerability Reference

- **Vulnerability Name:** Override Method
- **Category:** Avoidable Calls to the Security Manager - Method Call
- **Identifier:** Vb.java.class.6
- **Origin:** Sun Secure Coding Guidelines for the Java Programming Language
- **Location of Exploit Code:** Application Code - Java Code
- **Source:** Application Code (Non-final method with Security Checks)
- **Target:** Java Element - Object
- **Consequence Type:** Undue Access - Object
- **Introduction Time:** Development
- **Exploit Time:** Execution

Vulnerability Description

- **Description:** Security Checks that are performed in overridable methods can be by-passed by rewriting the methods.
- **Preconditions:** Overriden method must not be private nor final. Knowledge of code behavior (e.g. access to source code) is a requisite for developping malicious code since expoit is class specific.
- **Attack Process:** Override non-final method that contains security checks.
- **Consequence Description:** -
- **See Also:** Finalize Method, Privileged Execution of Code provided by the Caller

Protection

- **Existing Mechanisms:** Make all methods that contain security checks final
- **Enforcement Point:** Execution
- **Potential Mechanisms:** Code static Analysis
- **Attack Prevention:** Code Reviewing
- **Reaction:** Correct the flawed bundle

Vulnerability Implementation

- **Code Reference:** Bypass_override.exporter-0.1.jar,bypass_override.abuser-0.1.jar
- **OSGi Profile:** J2SE-1.6
- **Date:** 2008-02-18
- **Test Coverage:** 100%
- **Known Vulnerable Platforms:** Oscar; Felix; Equinox; Knopflerfish

C.3.2 Privileged Execution of Caller provided Code

Vulnerability Reference

- **Vulnerability Name:** Privileged Execution of Caller provided Code
- **Category:** Avoidable Calls to the Security Manager - Method Call
- **Identifier:** Vb.java.class.7
- **Origin:** Sun Secure Coding Guidelines for the Java Programming Language
- **Location of Exploit Code:** Application Code - Java Code
- **Source:** Application Code (Privileged Execution of Code provided by the Caller)
- **Target:** Java Element - Class
- **Consequence Type:** Undue Access - Class
- **Introduction Time:** Development
- **Exploit Time:** Execution

Vulnerability Description

- **Description:** Privileged Code Execution must be restricted to code provided by the privileged bundle. If code origin is not properly controlled, less trusted bundles can provide their own code for Privileged Execution.
- **Preconditions:** No technical precondition. Knowledge of code behavior (e.g. access to source code or poorly encapsulated privileged code calls) is a requisite for developing the malicious code.
- **Attack Process:** A bundle A provides malicious code for privileged execution to bundle B.
- **Consequence Description:** -
- **See Also:** Finalize Method, Override Method

Protection

- **Existing Mechanisms:** -
- **Enforcement Point:** -
- **Potential Mechanisms:** -
- **Attack Prevention:** Code Reviewing
- **Reaction:** Correct the flawed bundle

Vulnerability Implementation

- **Code Reference:** Bypass_privilegedexecution.exporter-0.1.jar,bypass_privilegedexecution.abuser-0.1.jar
- **OSGi Profile:** J2SE-1.6
- **Date:** 2008-02-18
- **Test Coverage:** 10%
- **Known Vulnerable Platforms:** Oscar; Felix; Equinox; Knopflerfish

C.3.3 Privileged Execution of Caller Code - ClassLoader Privileges

Vulnerability Reference

- **Vulnerability Name:** Privileged Execution of Caller Code - ClassLoader Privileges
- **Category:** Avoidable Calls to the Security Manager - Method Call
- **Extends:** Privileged Execution of Code provided by the Caller
- **Identifier:** Vb.java.class.8
- **Origin:** Sun Secure Coding Guidelines for the Java Programming Language
- **Location of Exploit Code:** Application Code - Java Code
- **Source:** Application Code (Privileged Execution of Code provided by the Caller)
- **Target:** Java Element - Class
- **Consequence Type:** Undue Access - Class
- **Introduction Time:** Development
- **Exploit Time:** Execution

Vulnerability Description

- **Description:** Privileged Code Execution must be restricted to code provided by the privileged bundle. If code origin is not properly controlled, less trusted bundles can provide their own code for Privileged Execution. Privileged Execution is granted for several calls according to the current ClassLoader (Reflection, Library Loading).
- **Preconditions:** No technical precondition. Knowledge of code behavior (e.g. access to source code or poorly encapsulated privileged code calls) is a requisite for developing the malicious code.
- **Attack Process:** A bundle A provides malicious code for privileged execution to bundle B, and ensures that ClassLoader of caller and ClassLoader of manipulated object is compatible with performed checks.
- **Consequence Description:** -
- **See Also:** Finalize Method, Override Method

Protection

- **Existing Mechanisms:** -
- **Enforcement Point:** -
- **Potential Mechanisms:** -
- **Attack Prevention:** Code Reviewing
- **Reaction:** Correct the flawed bundle

Vulnerability Implementation

- **Code Reference:** -
- **OSGi Profile:** -
- **Date:** 2008-02-18

- **Test Coverage:** 0%
- **Known Vulnerable Platforms:** Oscar; Felix; Equinox; Knopflerfish

C.3.4 Cloning

Vulnerability Reference

- **Vulnerability Name:** Cloning
- **Category:** Avoidable Calls to the Security Manage - At Instantiation
- **Identifier:** Vb.java.class.9
- **Origin:** Findbugs Bug Patterns
- **Location of Exploit Code:** Application Code - Java Code
- **Source:** JVM - Runtime API (No constructor call in 'clone' method)
- **Target:** Java Element - Class
- **Consequence Type:** Undue Access - Class
- **Introduction Time:** Development
- **Exploit Time:** Execution

Vulnerability Description

- **Description:** Calls to the 'clone' method enable to create a new instance of a class without calling the constructor, which often contains security checks such as calls to the Security Manager.
- **Preconditions:** Have a reference to an instance of the target class. Ill-coded program.
- **Attack Process:** -
- **Consequence Description:** -
- **See Also:** Deserialization, Call Overridable Methods in Constructor

Protection

- **Existing Mechanisms:** Add manual Call to the Security Manager
- **Enforcement Point:** Execution
- **Potential Mechanisms:** Code static Analysis ; Code Rewriting - Insert Initialization Checks
- **Attack Prevention:** Code Reviewing
- **Reaction:** Correct the flawed bundle

Vulnerability Implementation

- **Code Reference:** Bypass_cloning.exporter-0.1.jar,bypass_cloning.abuser-0.1.jar
- **OSGi Profile:** J2SE-1.6
- **Date:** 2008-02-17
- **Test Coverage:** 100%
- **Known Vulnerable Platforms:** Oscar; Felix; Equinox; Knopflerfish

C.3.5 Deserialization

Vulnerability Reference

- **Vulnerability Name:** Deserialization
- **Category:** Avoidable Calls to the Security Manager - At Instantiation
- **Identifier:** Vb.java.class.10
- **Origin:** Findbugs Bug Patterns
- **Location of Exploit Code:** Application Code - Java Code
- **Source:** JVM - Runtime API (No constructor call during object deserialization)
- **Target:** Java Element - Class
- **Consequence Type:** Undue Access - Class
- **Introduction Time:** Development
- **Exploit Time:** Execution

Vulnerability Description

- **Description:** Deserialization enables to create a new instance of a class without calling the constructor, which often contains security checks such as calls to the Security Manager
- **Preconditions:** Have a serialized instance of the target object. Ill-coded program.
- **Attack Process:** See description.
- **Consequence Description:** -
- **See Also:** Cloning, Call Overridable Methods in Constructor

Protection

- **Existing Mechanisms:** Add manual Call to the Security Manager
- **Enforcement Point:** Execution
- **Potential Mechanisms:** Code static Analysis ; Code Rewriting - Insert Initialization Checks
- **Attack Prevention:** Code Reviewing
- **Reaction:** Correct the flawed bundle

Vulnerability Implementation

- **Code Reference:** Bypass_deserialize.exporter-0.1.jar,bypass_deserialize.abuser-0.1.jar
- **OSGi Profile:** J2SE-1.6
- **Date:** 2008-02-17
- **Test Coverage:** 100%
- **Known Vulnerable Platforms:** Oscar; Felix; Equinox; Knopflerfish

C.3.6 Call Overridable Methods in Constructor

Vulnerability Reference

- **Vulnerability Name:** Call Overridable Methods in Constructor
- **Category:** Avoidable Calls to the Security Manager - At Instantiation
- **Extends:** Override Method
- **Identifier:** Vb.java.class.11
- **Origin:** Sun Secure Coding Guidelines for the Java Programming Language
- **Location of Exploit Code:** Application Code - Java Code
- **Source:** Application Code (Non-final method call in Constructor)
- **Target:** Java Element - Object
- **Consequence Type:** Undue Access - Object
- **Introduction Time:** Development
- **Exploit Time:** Execution

Vulnerability Description

- **Description:** Calling non-final methods in constructor enable sub-classes to access to partially initialized instances of the objects, and break security and configuration assumptions that are made in the superclass.
- **Preconditions:** No technical precondition. Knowledge of code behavior (e.g. access to source code) is a requisite for developing malicious code since exploit is class specific.
- **Attack Process:** Override non-final method that is called in the constructor.
- **Consequence Description:** -
- **See Also:** Call Overridable Methods in Clone method

Protection

- **Existing Mechanisms:** Make all methods that are called in instantiation methods - Constructor, clone or deserialization - final
- **Enforcement Point:** Execution
- **Potential Mechanisms:** Code static Analysis
- **Attack Prevention:** Code Reviewing
- **Reaction:** Correct the flawed bundle

Vulnerability Implementation

- **Code Reference:** Calloveridablemethodsinconstructor-0.1.jar
- **OSGi Profile:** J2SE-1.6
- **Date:** 2008-02-18
- **Test Coverage:** 100%
- **Known Vulnerable Platforms:** Oscar; Felix; Equinox; Knopflerfish
- **Known Robust Platforms:** SFelix

C.3.7 Call Overridable Methods in Clone method

Vulnerability Reference

- **Vulnerability Name:** Call Overridable Methods in Clone method
- **Category:** Avoidable Calls to the Security Manager - At Instantiation
- **Extends:** Override Method
- **Identifier:** Vb.java.class.12
- **Origin:** Sun Secure Coding Guidelines for the Java Programming Language
- **Location of Exploit Code:** Application Code - Java Code
- **Source:** Application Code (Non-final method call in Constructor)
- **Target:** Java Element - Object
- **Consequence Type:** Undue Access - Object
- **Introduction Time:** Development
- **Exploit Time:** Execution

Vulnerability Description

- **Description:** Calling non-final methods in clone method enable sub-classes to access to partially initialized instances of the objects, and break security and configuration assumptions that are made in the superclass.
- **Preconditions:** No technical precondition. Knowledge of code behavior (e.g. access to source code) is a requisite for developing malicious code since exploit is class specific.
- **Attack Process:** Override non-final method that is called in the constructor.
- **Consequence Description:** -
- **See Also:** Call Overridable Methods in Constructor

Protection

- **Existing Mechanisms:** Make all methods that are called in instantiation methods - Constructor, clone or deserialization - final
- **Enforcement Point:** Execution
- **Potential Mechanisms:** Code static Analysis
- **Attack Prevention:** Code Reviewing
- **Reaction:** Correct the flawed bundle

Vulnerability Implementation

- **Code Reference:** Calloveridablemethodsinclone-0.1.jar
- **OSGi Profile:** J2SE-1.6
- **Date:** 2008-02-18
- **Test Coverage:** 100%
- **Known Vulnerable Platforms:** Oscar; Felix; Equinox; Knopflerfish
- **Known Robust Platforms:** SFelix

C.3.8 Finalize Method

Vulnerability Reference

- **Vulnerability Name:** Finalize Method
- **Category:** Avoidable Calls to the Security Manager - At Instantiation
- **Identifier:** Vb.java.class.13
- **Origin:** Sun Secure Coding Guidelines for the Java Programming Language
- **Location of Exploit Code:** Application Code - Java Code
- **Source:** JVM - APIs (Garbage Collection)
- **Target:** Java Element - Class
- **Consequence Type:** Undue Access - Class
- **Introduction Time:** Development
- **Exploit Time:** Execution

Vulnerability Description

- **Description:** Methods on a Class that is protected through a Security Manager can be called by creating a subclass that, after creation abortion, performs calls on the partially initialized object during finalization.
- **Preconditions:** Available malicious implementation of a subclass of the target class. Knowledge of code behavior (e.g. access to source code) is a requisite for developing malicious code since exploit is class specific.
- **Attack Process:** Create an instance of a class wit a malicious implementation of the 'finalize' method. Instantiation is aborted due to a Security Manager check, but code from finalize is executed when it is called by the Garbage Collector.
- **Consequence Description:** Data can be leaked, especially if object is partially initialized
- **See Also:** Override Method, Privileged Execution of Code provided by the Caller

Protection

- **Existing Mechanisms:** Check Internal Object State in Code - Initialization
- **Enforcement Point:** Execution
- **Potential Mechanisms:** Code Rewriting - Insert Initialization Checks (Checking whether the Object has been correctly initialized makes calls in the finalize method useless if the object has not been created properly.); Code static Analysis
- **Attack Prevention:** Code Reviewing
- **Reaction:** Correct the flawed bundle

Vulnerability Implementation

- **Code Reference:** Finalizer-0.1.jar
- **OSGi Profile:** J2SE-1.6

- **Date:** 2008-02-18
- **Test Coverage:** 100%
- **Known Vulnerable Platforms:** Oscar; Felix; Equinox; Knopflerfish
- **Known Robust Platforms:** SFelix

C.3.9 Shutdown Hook

Vulnerability Reference

- **Vulnerability Name:** Shutdown Hook
- **Category:** Avoidable Calls to the Security Manager - At Instantiation
- **Identifier:** Vb.java.class.14
- **Origin:** Charlie Lai's Java Insecurity Subtleties
- **Location of Exploit Code:** Application Code - Java Code
- **Source:** JVM - Runtime API (Runtime.addShutdownHook method)
- **Target:** Platform
- **Consequence Type:** Undue Access - Platform
- **Introduction Time:** Development
- **Exploit Time:** Execution

Vulnerability Description

- **Description:** Shutdown Hooks enable to execute code when the platform is stopped. In particular, this implies that components can execute code after they have been uninstalled. Moreover, if a security check is performed in the constructor after static global variable have been initialized, their value can be accessed
- **Preconditions:** ShutdownHooks Permission set, access variables or method is static, and bundle must be properly installed inspite of the constructor abortion (exception is typically catched in the activator). Knowledge of code behavior (e.g. access to source code) is a requisite for developing malicious code since exploit is class specific.
- **Attack Process:** Shutdown Hooks Threads are set through the Runtime.addShutdownHook method.
- **Consequence Description:** Execution of code after the removal of the shutdown hook setter.
- **See Also:** -

Protection

- **Existing Mechanisms:** Java Permissions
- **Enforcement Point:** Bundle Installation
- **Potential Mechanisms:** Code static Analysis
- **Attack Prevention:** -
- **Reaction:** -

Vulnerability Implementation

- **Code Reference:** Bypass_shutdownhook.exporter-0.1.jar,bypass_shutdownhook.abuser-0.1.jar
- **OSGi Profile:** J2SE-1.6

- **Date:** 2008-02-16
- **Test Coverage:** 100%
- **Known Vulnerable Platforms:** Oscar; Felix; Equinox; Knopflerfish

C.4 Class or Object Sharing - Synchronization

C.4.1 Synchronized Method

Vulnerability Reference

- **Vulnerability Name:** Synchronized Method
- **Category:** Synchronization
- **Identifier:** Vb.java.publicclasses.1
- **Origin:** Ares research project 'malicious-bundle' (thanks to N. Geoffray and G. Thomas, LIP6, Paris, F.)
- **Location of Exploit Code:** Application Code - Java Code
- **Source:** Application Code (Synchronization)
- **Target:** Java Element - Object
- **Consequence Type:** Unavailability - Object
- **Introduction Time:** Development
- **Exploit Time:** Execution

Vulnerability Description

- **Description:** A method is synchronized, to as to avoid the execution of the same method by two different clients (used in particular in case of access to resources).
- **Preconditions:** No technical precondition. Knowledge of code behavior (e.g. access to source code) is a requisite for developing the malicious code, since exploiting the vulnerability requires either access to the dependencies of the weak code, or knowledge of its blocking conditions.
- **Attack Process:** If the method call is blocked for any reason (infinite loop during execution, or delay due to an unavailable remote resource), all subsequent clients that call this method are freezed.
- **Consequence Description:** -
- **See Also:** Synchronized Code

Protection

- **Existing Mechanisms:** Avoid Synchronization in Public Code
- **Enforcement Point:** Execution
- **Potential Mechanisms:** Code static Analysis
- **Attack Prevention:** Code Reviewing
- **Reaction:** Correct the flawed bundle

Vulnerability Implementation

- **Code Reference:** Synchronizedmethod-0.1.jar
- **OSGi Profile:** J2SE-1.6

- **Date:** 2008-03-13
- **Test Coverage:** 100%
- **Known Vulnerable Platforms:** Oscar; Felix; Equinox; Knopflerfish
- **Known Robust Platforms:** SFelix

C.4.2 Synchronized Code

Vulnerability Reference

- **Vulnerability Name:** Synchronized Code
- **Category:** Synchronization
- **Identifier:** Vb.java.publicclasses.2
- **Origin:** Ares research project 'malicious-bundle' (thanks to N. Geoffray and G. Thomas, LIP6, Paris, F.)
- **Location of Exploit Code:** Application Code - Java Code
- **Source:** Application Code (Synchronization)
- **Target:** Java Element - Object
- **Consequence Type:** Unavailability - Object
- **Introduction Time:** Development
- **Exploit Time:** Execution

Vulnerability Description

- **Description:** A method contains a synchronized block, so as to avoid the execution of the same method by two different clients (used in particular in case of access to resources).
- **Preconditions:** No technical precondition. Knowledge of code behavior (e.g. access to source code) is a requisite for developing the malicious code, since exploiting the vulnerability requires either access to the dependencies of the weak code, or knowledge of its blocking conditions.
- **Attack Process:** If the method call is blocked for any reason (infinite loop during execution, or delay due to an unavailable remote resource), all subsequent clients that call this method are freezed.
- **Consequence Description:** -
- **See Also:** Synchronized Code

Protection

- **Existing Mechanisms:** Avoid Synchronization in Public Code
- **Enforcement Point:** Execution
- **Potential Mechanisms:** Code static Analysis
- **Attack Prevention:** Code Reviewing
- **Reaction:** Correct the flawed bundle

Vulnerability Implementation

- **Code Reference:** Synchronizedcode-0.1.jar
- **OSGi Profile:** J2SE-1.6
- **Date:** 2008-03-13

- **Test Coverage:** 100%
- **Known Vulnerable Platforms:** Oscar; Felix; Equinox; Knopflerfish
- **Known Robust Platforms:** SFelix

C.5 Object Sharing Vulnerabilities - Exposed Internal Representation

C.5.1 Returns Reference to Mutable Object

Vulnerability Reference

- **Vulnerability Name:** Returns Reference to Mutable Object
- **Category:** Exposed Internal Representation
- **Identifier:** Vb.java.object.1
- **Origin:** Findbugs Bug Patterns
- **Location of Exploit Code:** Application Code - Java Code
- **Source:** Application Code (Expose Internal Representation)
- **Target:** Java Element - Object
- **Consequence Type:** Undue Access - Object
- **Introduction Time:** Development
- **Exploit Time:** Execution

Vulnerability Description

- **Description:** A method returns a reference to a mutable object.
- **Preconditions:** Ill-coded Class is Public Code (registered Service or exported Package). This vulnerability can be exploited in two ways: intentional (through knowledge of target code behavior) or accidental (through non malicious modification of the object value).
- **Attack Process:** Internal Data of the victim object is read and/or modified.
- **Consequence Description:** -
- **See Also:** Returns Reference to Mutable Data - Array

Protection

- **Existing Mechanisms:** Deep Copy of Data - Before making it Public
- **Enforcement Point:** Execution
- **Potential Mechanisms:** Code static Analysis
- **Attack Prevention:** Code Reviewing
- **Reaction:** Correct the flawed bundle

Vulnerability Implementation

- **Code Reference:** Returnref2mutableobject-0.1.jar
- **OSGi Profile:** J2SE-1.6
- **Date:** 2008-02-17
- **Test Coverage:** 100%
- **Known Vulnerable Platforms:** Oscar; Felix; Equinox; Knopflerfish
- **Known Robust Platforms:** SFelix

C.5.2 Returns Reference to Array

Vulnerability Reference

- **Vulnerability Name:** Returns Reference to Array
- **Category:** Exposed Internal Representation
- **Identifier:** Vb.java.object.2
- **Origin:** Findbugs Bug Patterns
- **Location of Exploit Code:** Application Code - Java Code
- **Source:** Application Code (Expose Internal Representation)
- **Target:** Java Element - Object
- **Consequence Type:** Undue Access - Object
- **Introduction Time:** Development
- **Exploit Time:** Execution

Vulnerability Description

- **Description:** A method returns a reference to a array.
- **Preconditions:** Ill-coded Class is Public Code (registered Service or exported Package). This vulnerability can be exploited in two ways: intentional (through knowledge of target code behavior) or accidental (through non malicious modification of the array value).
- **Attack Process:** Internal Data of the victim object is read and/or modified.
- **Consequence Description:** -
- **See Also:** Returns Reference to Mutable Object

Protection

- **Existing Mechanisms:** Deep Copy of Data - Before making it Public
- **Enforcement Point:** Execution
- **Potential Mechanisms:** Code static Analysis
- **Attack Prevention:** Code Reviewing
- **Reaction:** Correct the flawed bundle

Vulnerability Implementation

- **Code Reference:** Returnref2array-0.1.jar
- **OSGi Profile:** J2SE-1.6
- **Date:** 2008-02-17
- **Test Coverage:** 100%
- **Known Vulnerable Platforms:** Oscar; Felix; Equinox; Knopflerfish
- **Known Robust Platforms:** SFelix

C.5.3 Field with too much Visibility

Vulnerability Reference

- **Vulnerability Name:** Field with too much Visibility
- **Category:** Exposed Internal Representation
- **Identifier:** Vb.java.object.3
- **Origin:** Findbugs Bug Patterns
- **Location of Exploit Code:** Application Code - Java Code
- **Source:** Application Code (Expose Internal Representation)
- **Target:** Java Element - Object
- **Consequence Type:** Undue Access - Object
- **Introduction Time:** Development
- **Exploit Time:** Execution

Vulnerability Description

- **Description:** A method keeps a reference to a variable with too much visibility.
- **Preconditions:** No technical precondition. This vulnerability can be exploited in two ways: dynamic discovery of accessible fields with blind modifications, or on purpose modifications which require knowledge of code behavior (e.g. access to source code)
- **Attack Process:** Internal Data of the victim object is read and/or modified.
- **Consequence Description:** Data that should be kept internal is made available.
- **See Also:** Unsufficient Access Restriction - non Final Field

Protection

- **Existing Mechanisms:** Reduce Variable Visibility as much as Possible
- **Enforcement Point:** Execution
- **Potential Mechanisms:** -
- **Attack Prevention:** Code Reviewing
- **Reaction:** Correct the flawed bundle

Vulnerability Implementation

- **Code Reference:** -
- **OSGi Profile:** -
- **Date:** 2008-02-17
- **Test Coverage:** 0%
- **Known Vulnerable Platforms:** Oscar; Felix; Equinox; Knopflerfish

C.5.4 Non Final non Private Field

Vulnerability Reference

- **Vulnerability Name:** Non Final non Private Field
- **Category:** Exposed Internal Representation
- **Extends:** Field with too much Visibility; No Wrapper
- **Identifier:** Vb.java.object.4
- **Origin:** Findbugs Bug Patterns
- **Location of Exploit Code:** Application Code - Java Code
- **Source:** Application Code (Expose Internal Representation)
- **Target:** Java Element - Object
- **Consequence Type:** Undue Access - Object
- **Introduction Time:** Development
- **Exploit Time:** Execution

Vulnerability Description

- **Description:** A method keeps a reference to a static non final non private object.
- **Preconditions:** No technical precondition. This vulnerability can be exploited in two ways: dynamic discovery of accessible fields with blind modifications, or on purpose modifications which require knowledge of code behavior (e.g. access to source code)
- **Attack Process:** Internal Data of the victim object is read and/or modified.
- **Consequence Description:** -
- **See Also:** Unsufficient Access Restriction - Field with too much Visibility

Protection

- **Existing Mechanisms:** Avoid non Private non Final Variables in Public Code
- **Enforcement Point:** Execution
- **Potential Mechanisms:** Code static Analysis
- **Attack Prevention:** Code Reviewing
- **Reaction:** Correct the flawed bundle

Vulnerability Implementation

- **Code Reference:** Nonfinalnonprivatevariable-0.1.jar
- **OSGi Profile:** J2SE-1.6
- **Date:** 2008-02-17
- **Test Coverage:** 100%
- **Known Vulnerable Platforms:** Oscar; Felix; Equinox; Knopflerfish
- **Known Robust Platforms:** SFelix

C.5.5 No Wrapper

Vulnerability Reference

- **Vulnerability Name:** No Wrapper
- **Category:** Exposed Internal Representation
- **Identifier:** Vb.java.object.5
- **Origin:** Sun Secure Coding Guidelines for the Java Programming Language
- **Location of Exploit Code:** Application Code - Java Code
- **Source:** Application Code (Expose Internal Representation)
- **Target:** Java Element - Object
- **Consequence Type:** Undue Access - Object
- **Introduction Time:** Development
- **Exploit Time:** Execution

Vulnerability Description

- **Description:** Variables can be accessed on the object without wrapper methods, which prevent the execution of security or parameter checks.
- **Preconditions:** No technical precondition. This vulnerability can be exploited in two ways: dynamic discovery of accessible fields with blind modifications, or on purpose modifications which require knowledge of code behavior (e.g. access to source code)
- **Attack Process:** Variables are accessed directly on the object
- **Consequence Description:** -
- **See Also:** -

Protection

- **Existing Mechanisms:** Use Wrapper Methods to Access Variables
- **Enforcement Point:** Execution
- **Potential Mechanisms:** -
- **Attack Prevention:** Code Reviewing
- **Reaction:** Correct the flawed bundle

Vulnerability Implementation

- **Code Reference:** Nonfinalnonprivatevariable-0.1.jar
- **OSGi Profile:** J2SE-1.6
- **Date:** 2008-02-18
- **Test Coverage:** 0%
- **Known Vulnerable Platforms:** Oscar; Felix; Equinox; Knopflerfish

C.5.6 Information Leak through Exceptions

Vulnerability Reference

- **Vulnerability Name:** Information Leak through Exceptions
- **Category:** Exposed Internal Representation
- **Identifier:** Vb.java.object.6
- **Origin:** Sun Secure Coding Guidelines for the Java Programming Language
- **Location of Exploit Code:** Application Code - Java Code
- **Source:** JVM - Runtime API (Exceptions)
- **Target:** Configuration Data
- **Consequence Type:** Undue Access - Configuration Data
- **Introduction Time:** Development
- **Exploit Time:** Execution

Vulnerability Description

- **Description:** Exception Messages often contain data that describes the configuration of the system. These data should not be propagated to external callers, unless it directly concerns caller input.
- **Preconditions:** No technical precondition. Ill-coded program.
- **Attack Process:** Configuration Data is read from the Exception messages
- **Consequence Description:** See description
- **See Also:** -

Protection

- **Existing Mechanisms:** Purge Exceptions from Sensitive Data before Propagation
- **Enforcement Point:** Execution
- **Potential Mechanisms:** -
- **Attack Prevention:** Code Reviewing
- **Reaction:** Correct the flawed bundle

Vulnerability Implementation

- **Code Reference:** Exceptiondataleak.service-0.1.jar
- **OSGi Profile:** J2SE-1.6
- **Date:** 2008-02-18
- **Test Coverage:** 100%
- **Known Vulnerable Platforms:** Oscar; Felix; Equinox; Knopflerfish

C.6 Object Sharing Vulnerabilities - Flaws in Parameter Validation

C.6.1 Unchecked Parameters - Malicious Program Abuse - Java Code

Vulnerability Reference

- **Vulnerability Name:** Unchecked Parameters - Malicious Program Abuse - Java Code
- **Category:** Flaws in Parameter Validation
- **Identifier:** Vb.java.object.7
- **Origin:** Sun Secure Coding Guidelines for the Java Programming Language
- **Location of Exploit Code:** Application Code - Java Code
- **Source:** Application Code (No Parameter Check in Bundle Public Code)
- **Target:** Java Element - Object
- **Consequence Type:** Unvalid Type or Value - Object
- **Introduction Time:** Development
- **Exploit Time:** Execution

Vulnerability Description

- **Description:** Unchecked parameters in bundle public code (OSGi Services or Exported Packages) can be exploited to execute malicious code
- **Preconditions:** Absence of copy and check of method parameters. Knowledge of code behavior (e.g. access to source code) is a requisite for developing the malicious code.
- **Attack Process:** Malicious code can be introduced for instance by using fake classes that inherit from the parameter class type.
- **Consequence Description:** Application dependent
- **See Also:** Unchecked Parameters - Malicious Program Abuse - Native Code, Unchecked Parameters - Accidentally unsupported values, Parameter Checked without Copy

Protection

- **Existing Mechanisms:** Copy and Check Object State in Code - Parameters
- **Enforcement Point:** Execution
- **Potential Mechanisms:** -
- **Attack Prevention:** Code Reviewing
- **Reaction:** Uninstall the malicious bundle

Vulnerability Implementation

- **Code Reference:** Controlinverterbundle-0.1.jar, exporterbundle-0.1.jar
- **OSGi Profile:** J2SE-1.6
- **Date:** 2008-02-16

- **Test Coverage:** 100%
- **Known Vulnerable Platforms:** Oscar; Felix; Equinox; Knopflerfish

C.6.2 Unchecked Parameters - Malicious Program Abuse - Native Code

Vulnerability Reference

- **Vulnerability Name:** Unchecked Parameters - Malicious Program Abuse - Native Code
- **Category:** Flaws in Parameter Validation
- **Identifier:** Vb.java.object.8
- **Origin:** Sun Secure Coding Guidelines for the Java Programming Language
- **Location of Exploit Code:** Application Code - Java Code
- **Source:** Application Code (No Parameter Check in Bundle Public Code)
- **Target:** Java Element - Object
- **Consequence Type:** Undue Access - Native Code Execution
- **Introduction Time:** Development
- **Exploit Time:** Execution

Vulnerability Description

- **Description:** Unchecked parameters in bundle public code (OSGi Services or Exported Packages) can be exploited to execute malicious code, especially native code that does not provide any security guarantees.
- **Preconditions:** Absence of copy and check of method parameters. Knowledge of code behavior (e.g. access to source code or poorly encapsulated native code calls) is a requisite for developing the malicious code.
- **Attack Process:** Malicious code can be introduced for instance by using fake classes that inherit from the parameter class type.
- **Consequence Description:** Application dependent
- **See Also:** Unchecked Parameters - Malicious Program Abuse - Java Code, Unchecked Parameters - Accidentally unsupported values, Parameter Checked without Copy

Protection

- **Existing Mechanisms:** Copy and Check Object State in Code - Parameters
- **Enforcement Point:** Execution
- **Potential Mechanisms:** -
- **Attack Prevention:** Code Reviewing
- **Reaction:** Uninstall the malicious bundle

Vulnerability Implementation

- **Code Reference:** Nativecodeexecution.service-0.1.jar,nativecodeexecution.client-0.1.jar
- **OSGi Profile:** J2SE-1.6
- **Date:** 2008-02-18
- **Test Coverage:** 100%
- **Known Vulnerable Platforms:** Oscar; Felix; Equinox; Knopflerfish

C.6.3 Unchecked Parameters - Accidentally unsupported values

Vulnerability Reference

- **Vulnerability Name:** Unchecked Parameters - Accidentally unsupported values
- **Category:** Flaws in Parameter Validation
- **Identifier:** Vb.java.object.9
- **Origin:** Charlie Lai's Java Insecurity Subtleties
- **Location of Exploit Code:** Application Code - Java Code
- **Source:** Application Code (No Parameter Check in Bundle Public Code)
- **Target:** Java Element - Object
- **Consequence Type:** Unvalid Type or Value - Object
- **Introduction Time:** Development
- **Exploit Time:** Execution

Vulnerability Description

- **Description:** Unchecked parameters in bundle public code (OSGi Services or Exported Packages) can lead to unexpected program behaviour if constraints on their values are not enforced.
- **Preconditions:** Absence of copy and check of method parameters. This vulnerability can be exploited in two ways: intentional (through knowledge of target code behavior) or accidental (through a simple call with unusual parameter values).
- **Attack Process:** A parameter that has an unsupported value is passed to a public code method.
- **Consequence Description:** Program instability.
- **See Also:** Unchecked Parameters - Malicious Program Abuse - Java Code, Unchecked Parameters - Malicious Program Abuse - Native Code, Parameter Checked without Copy

Protection

- **Existing Mechanisms:** Copy and Check Object State in Code - Parameters
- **Enforcement Point:** Execution
- **Potential Mechanisms:** -
- **Attack Prevention:** Code Reviewing
- **Reaction:** Correct the flawed bundle

Vulnerability Implementation

- **Code Reference:** Unsupportedparametervalue.service-0.1.jar,unsupportedparametervalue.client-0.1.jar
- **OSGi Profile:** J2SE-1.6
- **Date:** 2008-02-17

- **Test Coverage:** 100%
- **Known Vulnerable Platforms:** Oscar; Felix; Equinox; Knopflerfish

C.6.4 Parameter Checked without Copy

Vulnerability Reference

- **Vulnerability Name:** Parameter Checked without Copy
- **Category:** Flaws in Parameter Validation
- **Identifier:** Vb.java.object.10
- **Origin:** Sun Secure Coding Guidelines for the Java Programming Language
- **Location of Exploit Code:** Application Code - Java Code
- **Source:** Application Code (Parameter Check without Copy in Public Code)
- **Target:** Java Element - Object
- **Consequence Type:** Unvalid Type or Value - Object
- **Introduction Time:** Development
- **Exploit Time:** Execution

Vulnerability Description

- **Description:** A parameter that is checked without being copied beforehand can be modified after validation and lead a TOCTOU (Time of Check To Time of Use) attack.
- **Preconditions:** No internal copy of parameter before performing check. This vulnerability can be exploited in two ways: intentional (through knowledge of target code behavior) or accidental (through a simple call and subsequent modification of the value of an object that is passed as parameter).
- **Attack Process:** The object that is passed as parameter is modified by the attacker after its validation.
- **Consequence Description:** Program instability, uncoherence, or undue execution of code.
- **See Also:** Unchecked Parameters - Malicious Program Abuse - Java Code, Unchecked Parameters - Malicious Program Abuse - Native Code, Unchecked Parameters - Accidentally unsupported values

Protection

- **Existing Mechanisms:** Copy and Check Object State in Code - Parameters
- **Enforcement Point:** Execution
- **Potential Mechanisms:** -
- **Attack Prevention:** Code Reviewing
- **Reaction:** Correct the flawed bundle

Vulnerability Implementation

- **Code Reference:** Parametervalidationerror.service-0.1.jar,parametervalidationerror.client-0.1.jar,parametervalidationerror.scenario-0.1.jar
- **OSGi Profile:** -

- **Date:** 2008-02-17
- **Test Coverage:** 100%
- **Known Vulnerable Platforms:** Oscar; Felix; Equinox; Knopflerfish

C.6.5 Copied and Checked Parameters - Fake Clone Method

Vulnerability Reference

- **Vulnerability Name:** Copied and Checked Parameters - Fake Clone Method
- **Category:** Flaws in Parameter Validation
- **Identifier:** Vb.java.object.11
- **Origin:** Sun Secure Coding Guidelines for the Java Programming Language
- **Location of Exploit Code:** Application Code - Java Code
- **Source:** Application Code (Fake Clone Method)
- **Target:** Java Element - Object
- **Consequence Type:** Unvalid Type or Value - Object
- **Introduction Time:** Development
- **Exploit Time:** Execution

Vulnerability Description

- **Description:** Copying parameters before their validation can be worthless if the copy is done through an overridden 'clone' method that is implemented partially or with a malicious objective.
- **Preconditions:** Use of non final parameter. A malicious implementation is provided waiting for execution. Knowledge of code behavior (e.g. access to source code) is required for attacks against a specific code excerpt.
- **Attack Process:** The 'clone' method does not perform as expected.
- **Consequence Description:** Program instability, uncoherence, or undue execution of code.
- **See Also:** Copied and Checked Parameters - Fake Copy Constructor

Protection

- **Existing Mechanisms:** Use final Types for Parameters
- **Enforcement Point:** Execution
- **Potential Mechanisms:** -
- **Attack Prevention:** Code Reviewing
- **Reaction:** Correct the flawed bundle

Vulnerability Implementation

- **Code Reference:** Parametervalidationerror.service-0.1.jar,parametervalidationerror.client-0.1.jar,parametervalidationerror.scenario-0.1.jar
- **OSGi Profile:** J2SE-1.6
- **Date:** 2008-02-17
- **Test Coverage:** 100%
- **Known Vulnerable Platforms:** Oscar; Felix; Equinox; Knopflerfish

C.6.6 Copied and Checked Parameters - Fake Copy Constructor

Vulnerability Reference

- **Vulnerability Name:** Copied and Checked Parameters - Fake Copy Constructor
- **Category:** Flaws in Parameter Validation
- **Identifier:** Vb.java.object.12
- **Origin:** Sun Secure Coding Guidelines for the Java Programming Language
- **Location of Exploit Code:** Application Code - Java Code
- **Source:** Application Code (Fake Copy Constructor)
- **Target:** Java Element - Object
- **Consequence Type:** Unvalid Type or Value - Object
- **Introduction Time:** Development
- **Exploit Time:** Execution

Vulnerability Description

- **Description:** Copying parameters before their validation can be worthless if the copy is done through a fake copy constructor that is implemented partially or with a malicious objective.
- **Preconditions:** Use of non final parameter. A malicious implementation is provided waiting for execution. Knowledge of code behavior (e.g. access to source code) is required for attacks against a specific code excerpt.
- **Attack Process:** The copy constructor does not perform as expected.
- **Consequence Description:** Program instability, uncoherence, or undue execution of code.
- **See Also:** Copied and Checked Parameters - Fake Clone Method

Protection

- **Existing Mechanisms:** Use final Types for Parameters
- **Enforcement Point:** Execution
- **Potential Mechanisms:** -
- **Attack Prevention:** Code Reviewing
- **Reaction:** Correct the flawed bundle

Vulnerability Implementation

- **Code Reference:** Parametervalidationerror.service-0.1.jar,parametervalidationerror.client-0.1.jar,parametervalidationerror.scenario-0.1.jar
- **OSGi Profile:** J2SE-1.6
- **Date:** 2008-02-17
- **Test Coverage:** 100%
- **Known Vulnerable Platforms:** Oscar; Felix; Equinox; Knopflerfish

C.6.7 Copied and Checked Parameters - Uncomplete Copy - State Omission

Vulnerability Reference

- **Vulnerability Name:** Copied and Checked Parameters - Uncomplete Copy - State Omission
- **Category:** Flaws in Parameter Validation
- **Identifier:** Vb.java.object.13
- **Origin:** Sun Secure Coding Guidelines for the Java Programming Language
- **Location of Exploit Code:** Application Code - Java Code
- **Source:** Application Code (Uncomplete Manual Copy)
- **Target:** Java Element - Object
- **Consequence Type:** Unvalid Type or Value - Object
- **Introduction Time:** Development
- **Exploit Time:** Execution

Vulnerability Description

- **Description:** Copying parameters before their validation can be unsufficient if some states are omitted
- **Preconditions:** -
- **Attack Process:** All states of the object are not copied
- **Consequence Description:** Program instability or uncoherence
- **See Also:** Copied and Checked Parameters - Uncomplete Manual Copy - Mutable States

Protection

- **Existing Mechanisms:** Deep Copy of Data - Method Parameters
- **Enforcement Point:** Execution
- **Potential Mechanisms:** -
- **Attack Prevention:** Code Reviewing
- **Reaction:** Correct the flawed bundle

Vulnerability Implementation

- **Code Reference:** Parametervalidationerror.service-0.1.jar,parametervalidationerror.client-0.1.jar,parametervalidationerror.scenario-0.1.jar
- **OSGi Profile:** -
- **Date:** 2008-02-17
- **Test Coverage:** 100%
- **Known Vulnerable Platforms:** Oscar; Felix; Equinox; Knopflerfish

C.6.8 Copied and Checked Parameters - Uncomplete Copy - Mutable States

Vulnerability Reference

- **Vulnerability Name:** Copied and Checked Parameters - Uncomplete Copy - Mutable States
- **Category:** Flaws in Parameter Validation
- **Identifier:** Vb.java.object.14
- **Origin:** Sun Secure Coding Guidelines for the Java Programming Language
- **Location of Exploit Code:** Application Code - Java Code
- **Source:** Application Code (Uncomplete Manual Copy)
- **Target:** Java Element - Object
- **Consequence Type:** Unvalid Type or Value - Object
- **Introduction Time:** Development
- **Exploit Time:** Execution

Vulnerability Description

- **Description:** Copying parameters before their validation can be unsufficient if some states are mutable.
- **Preconditions:** Mutable states of the object are not deep copied.
- **Attack Process:** Mutable states are modified from other classes.
- **Consequence Description:** Program instability, uncoherence, or execution of malicious code.
- **See Also:** Copied and Checked Parameters - Uncomplete Manual Copy - Omission of states

Protection

- **Existing Mechanisms:** Deep Copy of Data - Method Parameters
- **Enforcement Point:** Execution
- **Potential Mechanisms:** -
- **Attack Prevention:** Code Reviewing
- **Reaction:** Correct the flawed bundle

Vulnerability Implementation

- **Code Reference:** Parametervalidationerror.service-0.1.jar,parametervalidationerror.client-0.1.jar,parametervalidationerror.scenario-0.1.jar
- **OSGi Profile:** -
- **Date:** 2008-02-17
- **Test Coverage:** 50%
- **Known Vulnerable Platforms:** Oscar; Felix; Equinox; Knopflerfish

C.6.9 Non-final Parameters - Malicious Implementation

Vulnerability Reference

- **Vulnerability Name:** Non-final Parameters - Malicious Implementation
- **Category:** Flaws in Parameter Validation
- **Identifier:** Vb.java.object.15
- **Origin:** Sun Secure Coding Guidelines for the Java Programming Language
- **Location of Exploit Code:** Application Code - Java Code
- **Source:** Application Code (Parameter Type is not final)
- **Target:** Java Element - Object
- **Consequence Type:** Undue Access - Object
- **Introduction Time:** Development
- **Exploit Time:** Execution

Vulnerability Description

- **Description:** Non-final parameters in bundle public code (SOP Services or Exported Packages) can be exploited to execute malicious code, possibly exploiting internal data of the victim bundle
- **Preconditions:** Non-final method parameter(s). A malicious implementation is provided waiting for execution. Knowledge of code behavior (e.g. access to source code) is required for attacks against a specific code excerpt.
- **Attack Process:** Malicious code can be introduced by using fake classes that inherit from the parameter class type.
- **Consequence Description:** Application dependent
- **See Also:** Unchecked Parameters - Malicious Program Abuse - Java Code, Unchecked Parameters - Malicious Program Abuse - Native Code, Unchecked Parameters - Accidentally unsupported values, Parameter Checked without Copy, Non-final Parameters - Inversion of Control

Protection

- **Existing Mechanisms:** Use final Types for Parameters
- **Enforcement Point:** Execution
- **Potential Mechanisms:** Code static Analysis
- **Attack Prevention:** Code Reviewing
- **Reaction:** Uninstall the malicious bundle

Vulnerability Implementation

- **Code Reference:** Controlinverterbundle-0.1.jar, exporterbundle-0.1.jar
- **OSGi Profile:** J2SE-1.6
- **Date:** 2008-02-17

- **Test Coverage:** 100%
- **Known Vulnerable Platforms:** Oscar; Felix; Equinox; Knopflerfish
- **Known Robust Platforms:** SFelix

C.6.10 Non-final Parameters - Inversion of Control

Vulnerability Reference

- **Vulnerability Name:** Non-final Parameters - Inversion of Control
- **Category:** Flaws in Parameter Validation
- **Extends:** Non-final Parameters - Malicious Implementation
- **Identifier:** Vb.java.object.16
- **Origin:** Ares research project 'malicious-bundle' (thanks to Emmanuel Coquery)
- **Location of Exploit Code:** Application Code - Java Code
- **Source:** Application Code (Parameter Type is not final)
- **Target:** Java Element - Object
- **Consequence Type:** Undue Access - Object
- **Introduction Time:** Development
- **Exploit Time:** Execution

Vulnerability Description

- **Description:** Non-final parameters in bundle public code (SOP Services or Exported Packages) can be exploited to execute malicious code through inversion of control, ie. actions in the malicious bundle can be triggered through the victim bundle, possibly leaking data. Data from the victim bundle can be exploited. Certain type of access control mechanisms can be by-passed.
- **Preconditions:** Non-final method parameter(s). A malicious implementation is provided waiting for execution. Knowledge of code behavior (e.g. access to source code) is required for attacks against a specific code excerpt.
- **Attack Process:** Malicious code can be introduced by using fake classes that inherit from the parameter class type.
- **Consequence Description:** Application dependent
- **See Also:** Unchecked Parameters - Malicious Program Abuse - Java Code, Unchecked Parameters - Malicious Program Abuse - Native Code, Unchecked Parameters - Accidentally unsupported values, Parameter Checked without Copy

Protection

- **Existing Mechanisms:** Use final Types for Parameters
- **Enforcement Point:** Execution
- **Potential Mechanisms:** Code static Analysis
- **Attack Prevention:** Code Reviewing
- **Reaction:** Uninstall the malicious bundle

Vulnerability Implementation

- **Code Reference:** Controlinverterbundle-0.1.jar, exporterbundle-0.1.jar

- **OSGi Profile:** J2SE-1.6
- **Date:** 2008-02-17
- **Test Coverage:** 100%
- **Known Vulnerable Platforms:** Oscar; Felix; Equinox; Knopflerfish
- **Known Robust Platforms:** SFelix

Unité de recherche INRIA Rhône-Alpes
655, avenue de l'Europe - 38334 Montbonnot Saint-Ismier (France)

Unité de recherche INRIA Futurs : Parc Club Orsay Université - ZAC des Vignes
4, rue Jacques Monod - 91893 ORSAY Cedex (France)
Unité de recherche INRIA Lorraine : LORIA, Technopôle de Nancy-Brabois - Campus scientifique
615, rue du Jardin Botanique - BP 101 - 54602 Villers-lès-Nancy Cedex (France)
Unité de recherche INRIA Rennes : IRISA, Campus universitaire de Beaulieu - 35042 Rennes Cedex (France)
Unité de recherche INRIA Rocquencourt : Domaine de Voluceau - Rocquencourt - BP 105 - 78153 Le Chesnay Cedex (France)
Unité de recherche INRIA Sophia Antipolis : 2004, route des Lucioles - BP 93 - 06902 Sophia Antipolis Cedex (France)

Éditeur
INRIA - Domaine de Voluceau - Rocquencourt, BP 105 - 78153 Le Chesnay Cedex (France)
http://www.inria.fr
ISSN 0249-6399