
Soutenance de Thèse de doctorat de:

Pierre Parrend
Equipe INRIA-Amazones, Laboratoire Citi
INSA-Lyon
Directeurs de Thèse: Stéphane Ubéda (Pr.)
Stéphane Frénot (McF)

Jury

Rapporteurs: Didier Donsez (Pr. Uni. Grenoble I)
Ralf Reusser (Pr. Uni. Karlsruhe)
Examinateurs: Ciaran Bryce (MER Uni Genève)
Pierre-Etienne Moreau (CR INRIA)
Context

- The Evolution of software

| Monolithic Applications | Component-based Applications | Service-oriented Programming (SOP) Applications |

- Challenges
  - Management
  - Integration
  - Security

Motivating example: Dynamic SOP applications

What happens if the WebCamDriver Component is a Malware?
Context

- Service-oriented programming (SOP) platforms
  - EJB 3.0, OSGi, Spring, Google Guice
Context

- Attack vectors against SOP platforms
  - Example: The Java/OSGi platform

![Diagram showing attack vectors against SOP platforms]

This work
Contributions

1. Sign
2. Publish
3a. Load
3b. Check

Malicious Component
Protection Domain

Component Repository

Hardened OSGi
WCA
CBAC
Host

Digital Signature

Research Contributions
Implementation Enhancement
Specification Implementation
Outline

- Security for Java-based Software Systems
- Contributions
- Conclusions
Security for Java-based Software Systems

- Building secure software systems: The software development life-cycle
  - ‘Software security assurance’

- Monolithic view
- Systems are built from several mandatory and optional parts

Goertzel, et al. 
Security for Java-based Software Systems

- Identification of suitable protection mechanisms
  - Benefits/cost trade-off
  - Cost estimation
    - Minimal when flaws are repaired early
    - Grows dramatically latter in the life-cycle
  - Components
    - Reparation only possible if the code is available
    - Detection otherwise

Security for Java-based Software Systems

- Java application security: The principles
  - Type safety
    - Objects only perform actions defined through their type
  - Automated memory management
    - Through garbage collection
  - Bytecode validation
    - Executed code is not trusted
  - Isolation of components through class loaders
    - Prevent naming conflicts between components
- Limitations
  - Security use case: execution of one malicious applets in the JVM
  - Class loaders enforce namespace isolation only
Security for Java-based Software Systems

- The Java Security Manager

- Java policy file

```java
keystore "file:/home/pierre/keystore.ks";
grant signedBy "alice" {
    permission java.io.FilePermission "/opt/secret/secretKeys", "read";
    permission org.osgi.framework.PackagePermission ",", "export";
    permission org.osgi.framework.ServicePermission ",", "register";
};
grant signedBy "bob" {
    permission org.osgi.framework.ServicePermission
        "fr.inria.ares.testservice.MyService", "register";
    permission org.osgi.framework.PackagePermission ",", "export";
};
```

- OSGi: Conditional Permissions
Security for Java-based Software Systems

- Critics of Java permissions
  - High performance overhead
    - 20 to 30% runtime overhead
    - Cause the withdrawal of security in commercial applications
  - Hard-coded definition of sensitive methods
    - New permissions for new code only
  - Permission hell
    - Must be extracted for each configuration
    - Tedious manual process
  - Runtime verification
    - Abort or execute dangerous calls
    - In mobile apps for instance, authorization depends on the user
Outline

- Security for Java-based Software Systems
- Contributions
  - Building a secure Platform: The SPIP Method
  - Enforcing security for components: CBAC, WCA
- Conclusions
Building a secure Platform: The SPIP Method

- The ‘Spiral Process for Intrusion Prevention’
- The problem
  - Identification of security issues in complex systems
    - For each subsystem
    - Comparison of various implementations
  - Evaluation of protection mechanisms
    - Security assessment
    - Comparison
Building a secure Platform: The SPIP Method

- The SPIP Method
Building a secure Platform: The SPIP Method

- Quantification of the security of a system: the ‘Protection Rate’
  - Security level of complex systems
    - Not a binary metric: never free of vulnerabilities
  - ‘Percentage of the known vulnerabilities that are protected’
    - Against a reference system (here: an OSGi implementation with all known vulnerabilities)
  - Based on the ‘Attack Surface‘ metric
    \[
    PR = \left(1 - \frac{\text{Attack Surface of the evaluated System}}{\text{Attack Surface of the Reference System}}\right) \times 100
    \]
  - Enables to
    - Assess individual security mechanisms
    - Compare execution environments

9/12/2008
Software Security Models for SOP Platforms
Building a secure Platform: The SPIP Method

- Implementation for the OSGi platform
  - Iteration 1: The Java/OSGi platform
  - Iteration 2 .. 4: Propositions
    - Hardened OSGi
    - Component-based Access Control - CBAC
    - Weak Component Analysis - WCA
  - Iteration 5: Integration with the JnJVM, a secure JVM implementation for OSGi applications
### Building a secure Platform: The SPIP Method

- **Results:** The vulnerability catalogs – ‘Malicious Bundles’

<table>
<thead>
<tr>
<th>Vulnerability Category</th>
<th>#</th>
</tr>
</thead>
<tbody>
<tr>
<td>Local Access Control Management</td>
<td>-</td>
</tr>
<tr>
<td>Invalid Workflow</td>
<td>1</td>
</tr>
<tr>
<td>No control on service registration</td>
<td>2</td>
</tr>
<tr>
<td>Invalid Metadata</td>
<td>3</td>
</tr>
<tr>
<td>Fragments</td>
<td>3</td>
</tr>
<tr>
<td>Invalid Archive</td>
<td>3</td>
</tr>
<tr>
<td>Invalid Activator</td>
<td>2</td>
</tr>
<tr>
<td>Bundle Management</td>
<td>2</td>
</tr>
<tr>
<td>Proper removal</td>
<td>1</td>
</tr>
<tr>
<td>Native Code execution</td>
<td>1</td>
</tr>
<tr>
<td>File Handling</td>
<td>1</td>
</tr>
<tr>
<td>Reflection</td>
<td>3</td>
</tr>
<tr>
<td>ClassLoader</td>
<td>3</td>
</tr>
<tr>
<td>No algorithm safety</td>
<td>7</td>
</tr>
<tr>
<td>Runtime stopping methods</td>
<td>2</td>
</tr>
<tr>
<td>Thread management</td>
<td>4</td>
</tr>
<tr>
<td>Optimization errors (not considered)</td>
<td>-</td>
</tr>
</tbody>
</table>

Building a secure Platform: The SPIP Method

- Results: The vulnerability catalogs – ‘Vulnerable Bundles’

<table>
<thead>
<tr>
<th>Vulnerability Category</th>
<th>#</th>
</tr>
</thead>
<tbody>
<tr>
<td>Flaws in parameter validation</td>
<td>10</td>
</tr>
<tr>
<td>Exposed Internal Representation</td>
<td>6</td>
</tr>
<tr>
<td>Synchronization</td>
<td>2</td>
</tr>
<tr>
<td>Exposed Internal Representation</td>
<td>4</td>
</tr>
<tr>
<td>Avoidable Calls to the Security Manager</td>
<td>9</td>
</tr>
<tr>
<td>Serialization</td>
<td>1</td>
</tr>
</tbody>
</table>

Results: ‘Protection Rate’ for mainstream OSGi platforms

<table>
<thead>
<tr>
<th>Platform Type</th>
<th># of protected Vulns</th>
<th># of identified Vulns</th>
<th>Protection Rate</th>
</tr>
</thead>
<tbody>
<tr>
<td>Concierge</td>
<td>0</td>
<td>28</td>
<td>0 %</td>
</tr>
<tr>
<td>Felix</td>
<td>1</td>
<td>32</td>
<td>3.1 %</td>
</tr>
<tr>
<td>Knopflerfish</td>
<td>1</td>
<td>31</td>
<td>3.2 %</td>
</tr>
<tr>
<td>Equinox</td>
<td>4</td>
<td>31</td>
<td>13 %</td>
</tr>
<tr>
<td>Java Permissions</td>
<td>13</td>
<td>32</td>
<td>41 %</td>
</tr>
<tr>
<td>Concierge with Permissions</td>
<td>10</td>
<td>28</td>
<td>36 %</td>
</tr>
<tr>
<td>Felix with Permissions</td>
<td>14</td>
<td>32</td>
<td>44 %</td>
</tr>
<tr>
<td>Knopflerfish with Permissions</td>
<td>14</td>
<td>31</td>
<td>44 %</td>
</tr>
<tr>
<td>Equinox with Permissions</td>
<td>17</td>
<td>31</td>
<td>55 %</td>
</tr>
</tbody>
</table>
Building a secure Platform: The SPIP Method

- Results: Hardened OSGi
  - Protection Rate: 25 % for the ‘Malicious Bundles’ catalog entries

**Introduce**
- Check component size before download, and control the cumulated size of loaded components
- Check digital signature at install time
- Launch the component activator in a separate Thread
- Limit the number of registered services

**Systematize**
- Do not reject harmless unnecessary metadata
- Remove all component data from disk at uninstallation

Outline

- Security for Java-based Software Systems
- **Contributions**
  - Building a secure Platform: The *SPIP* Method
  - **Enforcing security for components: CBAC, WCA**
- Conclusions
Enforcing Security for Components: CBAC, WCA

- The problem
  - Security issues with components
    - Maliciousness
    - Vulnerability
  - Installing secure components
    - Bytecode analysis only
Enforcing Security for Components: CBAC, WCA

- Definition of tools in the SPIP method
Enforcing Security for Components: CBAC, WCA

- The CBAC model: Principles
  - Component-based Access Control
  - Goal
    - Prevent issues from the ‘Malicious Bundles’ catalog
  - Principles
    - Install time analysis of the execution rights of components
      - Sensitive calls must be explicitly granted
    - Take composition into account
    - Intends to be an alternative to Java permissions
  - Hypotheses
    - The component platform is not modified
    - Each component contains a valid digital signature
The CBAC model: Definition

- Security Policy
  - Policy(A)=D.d2
  - Policy(A)=B.d1.D.d2

- System Structure

- Install Time
  - Policy(A)=D.d2 => D.d1 not Allowed
  - PSC = D.d1, D.d2

- Runtime

- Method Call
Enforcing Security for Components: CBAC, WCA

- The CBAC model: Performances

![Graph showing performance comparison between CBAC Check and Signature Check](image)

Time (ms)

<table>
<thead>
<tr>
<th>Size (KBytes)</th>
<th>CBAC Check</th>
<th>Signature Check</th>
</tr>
</thead>
<tbody>
<tr>
<td>5.51</td>
<td>6.33</td>
<td>7.62</td>
</tr>
<tr>
<td>6.49</td>
<td>10.32</td>
<td>7.92</td>
</tr>
<tr>
<td>7.92</td>
<td>11.74</td>
<td>12.3</td>
</tr>
<tr>
<td>10.32</td>
<td>13.42</td>
<td>13.72</td>
</tr>
<tr>
<td>12.23</td>
<td>14.28</td>
<td>14.71</td>
</tr>
<tr>
<td>13.72</td>
<td>17.71</td>
<td>17.72</td>
</tr>
<tr>
<td>14.28</td>
<td>24.92</td>
<td>24.92</td>
</tr>
<tr>
<td>17.71</td>
<td>34.06</td>
<td>34.06</td>
</tr>
<tr>
<td>24.92</td>
<td>38.72</td>
<td>38.72</td>
</tr>
<tr>
<td>34.06</td>
<td>52.2</td>
<td>52.2</td>
</tr>
<tr>
<td>38.72</td>
<td>86.37</td>
<td>86.37</td>
</tr>
<tr>
<td>52.2</td>
<td>131.48</td>
<td>131.48</td>
</tr>
<tr>
<td>86.37</td>
<td>356.76</td>
<td>356.76</td>
</tr>
</tbody>
</table>
The CBAC model: Benefits and limitations

- Benefits
  - No runtime overhead, reduced install time overhead
  - No application interruption, at the cost of false positive
  - No misleading pop-up windows
  - Arbitrary methods and meta-data can be set as sensitive
    - Enables to protect against vulnerabilities that are discovered after design
  - Protection Rate: 50 % for the ‘Malicious Bundles’ catalog entries

- Limitations
  - Policies must be defined in advanced
Enforcing Security for Components: CBAC, WCA

- The WCA approach: Principles
  - **Weak Component Analysis**
  - Goal
    - Prevent issues from the ‘Vulnerable Bundles’ catalog
  - Principles
    - Vulnerability identification through static analysis
      - In exposed code only
      - Through the code meta-model
      - Matching with ‘vulnerability patterns’
    - Development and install time use
      - XML version for flexibility
      - Hardcoded version for performance
Enforcing Security for Components: CBAC, WCA

- The WCA approach: Performances
Enforcing Security for Components: CBAC, WCA

- The WCA approach: Benefits and limitations
  - Benefits
    - Identification of exploitable vulnerabilities in Java components
    - According to the exposition of the code
    - Principally easy to extend
    - Development and runtime use
    - Protection Rate: 36% for the ‘Vulnerable Bundles’ catalog entries
  - Limitations of the implementation
    - Hardcoded version is slower
    - Only structural patterns are supported so far
    - Limited flexibility of the definition of patterns
Outline

- Security for Java-based Software Systems
- Contributions
- Conclusions
Conclusions

- Development overview

**Research Contributions**
- Implementation Enhancement
- Specification Implementation

**Tools**

<table>
<thead>
<tr>
<th>Tool</th>
<th>Lines of Code</th>
</tr>
</thead>
<tbody>
<tr>
<td>Digital Signature</td>
<td>330</td>
</tr>
<tr>
<td>SF-JarSigner</td>
<td>557</td>
</tr>
<tr>
<td>Malicious Components</td>
<td>155 bundles</td>
</tr>
<tr>
<td>Hardened OSGi</td>
<td>224</td>
</tr>
<tr>
<td>CBAC</td>
<td>577</td>
</tr>
<tr>
<td>WCA</td>
<td>2026</td>
</tr>
</tbody>
</table>

**Diagram**

- SF-JarSigner
- Component Repository
- Malicious Component
- Protection Domain
- Hardened OSGi
- WCA
- CBAC

**Steps**
1. Sign
2. Publish
3. Load
4. Check
Conclusions

- Evaluation of the proposed solutions
  - SPIP
    - Promising methodology for security analysis
    - Requires
      - Validation on further systems
      - Support for cost estimation
  - Tools for secure component-based applications
    - CBAC
      - Refined static analysis approach
    - WCA
      - Only a subset of best practices are enforced so far
      - Need of actual isolation between the bundles
  - Consider further attack vectors
Conclusions

- Who can benefit from this work?

<table>
<thead>
<tr>
<th>Role</th>
<th>Platform developer</th>
<th>Application architect</th>
<th>Application developer</th>
</tr>
</thead>
<tbody>
<tr>
<td>Focus on</td>
<td>Execution environment</td>
<td>Architecture</td>
<td>Components</td>
</tr>
<tr>
<td>System entity</td>
<td>Platform</td>
<td>Components</td>
<td>Components</td>
</tr>
<tr>
<td>Life-Cycle Activity</td>
<td>Platform design and coding</td>
<td>Application design</td>
<td>Application Coding</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Our propositions</td>
<td>Hardened OSGi</td>
<td>CBAC</td>
<td>WCA</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Integration</td>
</tr>
</tbody>
</table>
Conclusions

- Open challenges
  - Resource isolation
    - First solution: Integration with the JnJVM
  - Development for industrial use of the OSGi platform
    - Specifications
    - Life-cycle support for bundles
    - Management
    - Critical applications: strong isolation between applications
    - Multi-user applications: strong access control mechanism
Questions?

1. sign
2. publish
3. a load
3. b check

Malicious Component
Protection Domain

Component Repository

SF-Jarsigner

Digital Signature

Research Contributions
Implementation Enhancement
Specification Implementation

Hardened OSGi
WCA
CBAC

Host

9/12/2008 Software Security Models for SOP Platforms
References


Selected Publications

- **Journal Article**

- **International Conferences, Industrial Conferences**

- **Research Reports**
  - Java Components Vulnerabilities - An Experimental Classification Targeted at the OSGi Platform, Pierre Parrend, Stéphane Frenot, INRIA Research Report n° 6231, June 2007.”