Static Vulnerability Detection in Java Service-Oriented Components

François Goichon · Guillaume Salagnac · Pierre Parrend · Stéphane Frénot

Abstract Extensible component-based platforms allow dynamic discovery, installation and execution of components. Such platforms are service-oriented, as components may directly interact with each other via the services they provide. Even robust languages such as Java were not designed to handle safe code interaction between trusted and untrusted parties. Dynamic installation of code provided by different third parties leads to several security issues. The different security layers adopted by Java or component-based platforms cannot fully address the problem of untrusted components trying to tamper with other components via legitimate interactions. A malicious component might even use vulnerable ones to compromise the whole component-based platform.

Our approach identifies vulnerable components in order to prevent them from threatening services security. We use static analysis to remain as exhaustive as possible and to avoid the need for non-standard or intrusive environments. We show that a static analysis through tainted object propagation is well suited to detect vulnerabilities in Java service-oriented components. We present STOP, a Service-oriented Tainted Object Propagation tool, which applies this technique to statically detect those security flaws. Finally, the audit of several trusted Apache Felix bundles shows that nowadays component-based platforms are not prepared for malicious Java interactions.

University of Lyon, INRIA
INSA-Lyon, CITI

F. Goichon, G. Salagnac and S. Frénot
CITI Laboratory, INSA Lyon F-69621 Villeurbanne Tel.: +33 4 7243 6415
Fax: +33 4 7243 6227
E-mail: francois.goichon@insa-lyon.fr · P. Parrend
Universit de Strasbourg
ECAM Strasbourg-Europe

Keywords OSGi Security, Tainted Object Propagation, Vulnerability Detection, Static Analysis, Java

1 Introduction

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. Third-party repositories or official repositories are priority targets for attackers: Google recently assessed that several malicious applications have been detected and removed from their official Android application store [1]. In this paper, we address the detection of vulnerabilities that can be exploited by malicious software to compromise the behavior of a component-based application.

Examples of component-based platforms include OSGi platforms [2], Java MIDP [3], Android[1], or the Microsoft .Net platform[2]. We studied OSGi components, as its specifications provide a framework that crosses existing Java implementations. This allows developers to ignore the differences between underlying frameworks, while providing the dynamics and extensibility of other component-based platforms.

Although components are designed to be independent, they interact with other components via services in order to achieve more complex tasks. For instance, a component designed for data storage management provides facilities for other components to store and re-

[1] http://code.google.com/android/
[2] http://msdn.microsoft.com/netframework

trieve data. Each component stays focused on its main task, requiring others to achieve auxiliary tasks. A service provider first registers its service to the management platform. A consumer may ask the platform for the provider's service. The consumer may then call the public methods of this service with its own parameters. These parameters provided by the consumer may have an influence over the provider's code. On the other hand, an object returned by the provider's service is untrusted for the consumer.

These interactions can lead to several security issues. Each component may be provided by different sources and their installation and execution is highly dynamic: it is possible for malicious components to get installed on open platforms. As a component cannot necessarily trust others, external data emanating from other components can be considered untrusted and should not be able to alter the behavior of the component. For instance, it should continue to provide its services independently from those influences. For the Java language, Herzog *et al.* [4] pointed out the difficulties of securely running untrusted services as Java threads. Our recent work [5,6] classify the underlying vulnerability families for Java components running and interacting with untrusted ones. Those vulnerabilities induce a new range of possible exploits that malicious components can use to leak sensitive data, escalate privileges or cause Denial of Service to the platform or vulnerable components.

Protecting platforms and their components from such malicious behaviors is an important issue, which has not been fully addressed yet. The infinite possibilities to exploit a single flaw considerably toughen up malicious code detection. Our approach is to add an additional component isolation mechanism, which prevents unsafe components from providing services and importing packages. Unsafe components are the potentially vulnerable ones. Only components signed as safe by a trusted static analysis tool would be able to interact with others.

Statically analyzing Java components brings up new challenges, such as the fact that external code is not known before deployment time. State of the art analyses are not able to work on an isolated component as they require a complete application with an unique entry point to build analysis metadata. In this paper, we show that those challenges can be systematically handled for a security analysis with an adapted tainted object propagation modelization. We implement this technique in STOP, a Service-oriented Tainted Object Propagation tool designed to track typical components vulnerabilities. Our experimental results show that STOP is able to successfully detect exploitable vulnerabilities in real-life OSGi components in a matter of minutes.

Section 2 introduces our attacking model and vulnerabilities existing in Java components. Section 3 reviews past work on vulnerability detection with a focus on Java code interaction. Section 4 proposes an adapted tainted object propagation technique to statically detect components vulnerabilities. It takes advantage of the Java programming language and the OSGi service-oriented architecture to ease the static analysis. Section 5 presents STOP, an implementation of the proposed technique to detect component-specific vulnerabilities and shows experimental results evaluating STOP's performance and vulnerability detection efficiency.

2 Context

Java components are prone to a large set of vulnerabilities. Those can be used by malicious components to alter other components or the behavior of the platform. We introduce in this section the context of our vulnerability analysis: the attacker's goals and Java code interaction problems. We also provide a vulnerability example we use throughout the paper.

2.1 Attacking model

In this work, we focus on a realistic attacking model where the attacker has full control of one or more components on the otherwise trusted platform. They can export classes and import interfaces from vulnerable components to use their services. Google Mobile Team's statement on the effective corruption of several applications in their Android store [1] is a good example of such a scenario. In this case, a specific attack surface arises, as the attacker can provide whole malicious objects to its targets, which contain data as well as code. In this paper, we focus on this attacking model, where the attacker provides the targeted components with malicious objects to alter or take advantage of their behavior.

2.2 Java Code Interactions Flaws

Java provides implicit mechanisms for memory management and bytecode verification that make applications more robust against classical application-level vulnerabilities. However, like most programs, Java applications are prone to classical language-independent flaws, such as SQL injections [7] or information disclosure. Most work related to security analysis in Java applications focuses on those vulnerabilities [8–11], which are not specific to the Java programming language.

There exist however other Java-specific vulnerabilities, critical when considering Java code interaction. Many basic Java interactions can threaten encapsulation and execution safety. Reflection, subclassing, serialization, synchronization or access to system APIs allow malicious code to alter trusted objects or modify the expected execution flow [6]. Throughout this paper, we use the synchronized denial of service, to illustrate how such vulnerabilities work in general.

Vulnerability Example: Synchronized Denial of Service. 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 the code. 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 class. If such a deadlock occurs during a service call, any further access to the service will be denied [5].

```
1 public class DataStorage{
2  public void synchronized storeList(ArrayList list){
3   Iterator it = list.iterator();
4   while (it.hasNext()){ store(it.next()); }
5  }
6 }
```

Alg. 1: DataStorage class containing a vulnerable service prone to Denial of Service

The listing in Alg. 1, inspired by Parrend *et al.*'s vulnerability examples [5], is an illustration of such vulnerabilities. It contains the service class *DataStorage*. The main service method is *storeList*, which takes an *ArrayList* as its only parameter (line 2). It gets its *Iterator* (line 3) and takes each object in the array to *store* it. The sample exploit in Alg. 2 shows a crafted extension of *ArrayList*. When the *ArrayList iterator* method is called, the thread indefinitely loops, denying any further call to the *DataStorage* service. Multiple exploitations are possible, as another crafted *ArrayList* extension could provide a malicious *Iterator* extension that blocks within the *hasNext* and *next* methods.

```
1 public class MaliciousArrayList extends ArrayList{
2  public Iterator iterator(){
3   while(true);
4   return null;
5  }
6 }
```

Alg. 2: Malicious ArrayList implementation

2.3 Mitigation based on Security Layers

To mitigate possibilities to compromise other objects, the Java Standard Edition [12] and the OSGi specifications [2] provide additional security layers. They allow an administrator to control access to sensible operations such as reflection or object replacement during serialization. However, several flaws in its access control design can allow untrusted code to exploit vulnerabilities in trusted code and bypass those restrictions [6].

Those vulnerabilities have a huge impact on Java Applets or application servers but are less common in the OSGi world, where security layers are rarely used at all. Security layers actually induce an average overhead of 100% for Java applications [13] and bring a lot of constraints opposed to the fundamental concepts of generic and dynamic programming. Even when there is a Security Manager, it may be hard to tell which components may be trusted and which ones may not. It is even harder to tell which permissions should be granted to trusted or untrusted components. In this case, components have an uncontrolled access to the platform anyway.

Basic Java interactions induce theoretical vulnerabilities in Java applications interacting with untrusted code. Open component-based platforms turn those vulnerabilities into real threats and even bring new vulnerabilities to the fore [5]. With those new vulnerability exposures emerging comes the necessity to either detect those vulnerabilities or prevent their exploitation. The next section goes through software vulnerability detection techniques and their application on Java components.

3 Related Work

This section reviews past work on vulnerability detection for Java applications. We focus on static analysis to ensure a better detection coverage for such complex vulnerabilities. We present the different techniques and expose the lack of past work on advanced static analysis to detect Java code interaction vulnerabilities and component-based vulnerabilities. We also give further

details on the tainted object propagation technique that can be used to follow untrusted data in an application.

3.1 Static vs. Dynamic Vulnerability Detection.

In computer security and penetration testing, vulnerabilities can be detected and/or inhibited either statically or dynamically. On the first hand, a static vulnerability analysis tries to find vulnerabilities within a program without having to run it. The analysis is often performed from source files or compiled bytecode. On the other hand, a dynamic analysis runs the program to either find vulnerabilities or prevent their exploitation.

Static analyses can access all necessary data from source code or bytecode. However, they often need large amounts of memory and time to analyze a program, especially on large programs. Dynamic detection addresses this problem, by analyzing an application in a black-box fashion, only observing the system's reaction to specific queries and inputs. Dynamic detection cannot spot most vulnerabilities triggered by uncommon combinations of internals and is therefore non-exhaustive. Moreover, they often use non-standard, intrusive execution environments to monitor the target's execution. If the application is to be migrated back in a standard environment, it is not protected anymore. We focused on static analysis techniques, to remain non-intrusive, more precise and with a better coverage.

3.2 Static Analysis Techniques for Vulnerability Prevention.

Syntactic and lexical analyses. The simplest way to statically detect vulnerabilities is to perform syntactic and lexical analyses. A lexical analysis uses simple matches against simple signatures. For instance, searching for synchronized methods or public fields of a class. Simple pattern matching is highly efficient but cannot handle more elaborated signatures depending on instruction sequences. However, syntax trees can handle simple instruction sequences. The analysis builds a tree containing ordered operations and instructions. This tree is then matched against some predefined vulnerability signature tree.

Code transformations. Another way to prevent vulnerabilities is to transform the vulnerable source code into a safe version without changing the semantics. An example of such techniques is the cookie check introduced in the Visual Studio compiler [14] to prevent buffer overflow exploitation on exposed methods. There is also the possibility to simply annotate the code and let the compiler or runtime perform necessary checks. For example, Haldar *et al.* [15] integrate static analysis in SQL Injection detection by transforming the class files to perform security checks at runtime.

Tainted Object Propagation. Tainted object propagation is a kind of dataflow analysis [16]. Dataflow analysis is designed to handle complex application behaviors, including interprocedural behaviors or determining the range of possible values. Tainted object propagation is the main dataflow analysis applied to security: the analysis starts from an entry point where a remarkable value is assigned to a variable. It then builds a control-flow graph, which denotes all the possible paths in the studied part of the application. Following those different paths, the analysis follows the special variables to study the propagation of their values throughout the program. Tainted object propagation addresses two main issues: integrity and confidentiality [16]. In the case of integrity checks, the aim of the analysis is to verify that untrusted data cannot influence important variables that should not get corrupted. Confidentiality analysis, on the other hand, studies the reverse flow, by following important data that should not be leaked and verifying whether it exists a path from this data to known untrusted disclosure APIs, like serialization or unencrypted writes.

	Generic	Java Interactions	Security Layers
Syntactic / Lexical	AMNESIA [11]	WCA [17]	No
Transformation	Haldar *et al.* [15]	No	No
Tainted Object Propagation	Livshits *et al.* [8] TAJ [9] Liu *et al.* [10] Avvenuti *et al.* [18]	No	No

Table 1: Java Application Vulnerabilities and Applied Static Vulnerability Detection Techniques

Table 1 summarizes the most relevant past efforts on static vulnerability detection for Java applications. It illustrates that past work mostly focuses on common software vulnerabilities and data leakage. Livshits *et al.* [8] perform a static dataflow analysis on Java Web applications to detect SQL injections. They follow untrusted inputs, such as parameters of a Java servlet, and track their possible propagation to Java SQL APIs. Liu *et al.* [10] and Avvenuti *et al.* [18] apply reverse tainted object propagation to detect whether inputs with a high security level can be disclosed to lower security level outputs. The runtime monitoring tool of

Haldar *et al.* [15] annotates and transforms class files to perform checks at runtime or remove vulnerabilities.

Table 1 clearly highlights the lack of work on vulnerabilities specific to Java code interactions. Only Parrend *et al.* [17]'s tool, WCA, is designed to statically find potential security vulnerabilities in OSGi components. It uses simple pattern matching to analyze reachable code from OSGi service calls. For example, WCA issues a warning if a synchronized method is accessible, but cannot assess whether the code can indeed lead to a denial of service. WCA illustrates well the limitations of syntactic and lexical analyses when applied to components vulnerabilities. Transforming class files to prevent such vulnerabilities seems a hard task, as even for simple examples such as the code in Alg. 1, it is not possible to decide what instructions should or should not stay synchronized without altering the service semantics.

Tainted object propagation is a very powerful technique that has not yet been explored in the context of Java components. Moreover, Livshits *et al.* work on SQL injection detection [8] can easily be projected to the case of component interaction. They consider untrusted data flowing into an application via a servlet HTTP parameters. A servlet is no less than a service instance providing HTTP methods. We further explored the possibilities to apply tainted object propagation to the detection of component vulnerabilities. In the next paragraph, we detail a key to perform such analysis: context sensitivity.

3.3 Tainted Object Propagation and Context Sensitivity.

Tainted object propagation detects possible spreading of untrusted data, coming from so-called taint sources, to critical parts of the programs, the sinks. Tainted data has propagation vectors, the taint propagators. Sinks and taint sources are variables of the program and taint propagators are software instructions. Therefore, the first step is to perform a points-to analysis, to extract variables relationships. A points-to analysis computes a points-to set for each relevant variable, which contains all other variables connected by taint propagators.

The points-to analysis has to track relevant variables for the propagation throughout all execution paths, as untrusted data can influence critical variables through numerous subcalls. A single local variable in a method should not reference the same variable when the method is called in different contexts. An analysis that differentiates same variables used in different calling contexts is called context-sensitive. Representing explicitly all possible calling contexts requires lots of time and memory and is not practicable at the time being with medium-sized Java programs, even when skipping recursive cycles [19]. However, context-sensitivity is the key to precise points-to information computation for programs written in Java [20]. Several techniques have been developed to simplify those analyses. We present the three most important techniques in the next paragraph, which focus on different key points: the amount of code to analyze and the amount of data to keep for the analysis.

Summary-based analysis is the most traditional way to perform an interprocedural context-sensitive analysis. The first pass computes a summary for each method in the possible call paths. It contains all relevant information for the requested analysis. The second pass processes the call paths and computes information from the summaries for each method encountered. When summaries cannot be made compact enough, the analysis needs huge amounts of memory and fails to scale to medium-size programs [21]. However, summary-based analyses can be very efficient when summaries are compact enough [22].

Refinement-based analysis, proposed by Sridharan *et al.* [23], is designed to eliminate as much irrelevant code to analyze as possible. They perform successive refinements of points-to sets, to determine at each step if an assignment path actually exists between two variables, until either the path proves to be broken or the whole points-to set path proves to be right. Refinement-based analyses are demand-driven: they are specific to the client analysis needs, since they must know beforehand what are the objectives in terms of points-to computation to make the successive refinements towards those objectives. Sridharan *et al.* results show that refinement-based techniques are very precise and use significantly less memory than a classical context-sensitive analysis for large program.

BDD Cloning-based analysis, proposed by Whaley *et al.* [24], is an efficient way to process context-sensitive analyses that leverages the fact that different contexts used for a single method have a lot of redundant data. It uses Binary Decision Diagrams to efficiently store this data in a database-like structure. Their technique scales to complex programs, up to 10^{14} different contexts. However, the analysis still has to process the whole program and the amount of data to store is still huge for recent JDK versions.

Even with recent improvements in context-sensitive analysis such as BDD cloning- or refinement-based analyses, we decided to use a summary-based approach for our tainted object propagation. Indeed, we want our analysis to be compatible with common hardware configurations, whereas our experiments show that cloning-

based techniques require several gigabytes for small programs with recent versions of the Java library. A refinement-based analysis is not applicable to our vulnerabilities in the general case, as we do not know the whole list of taint sources and sinks beforehand and still have to analyze irrelevant code to search for them.

Past work did not focus yet on applying tainted object propagation to Java code interactions vulnerabilities. We propose in the next section a service-oriented tainted object propagation technique that uses the specificities of the Java language and Service-Oriented Programming platforms to identify specific vulnerabilities that could damage a component or its platform.

4 Service-oriented Tainted Object Propagation for Java

Historically, tainted object propagation has been mostly used in dynamic monitoring. However, some past work bring up the idea to use tainted object propagation in static software analysis for Java, such as Livshits *et al.*'s [8] web-based vulnerability detection on Java applications.

Statically analyzing Java components to find vulnerabilities related to code interaction, such as the simple fact that external code is not available at analysis time, brings up new challenges that have never been studied to the best of our knowledge. We describe how those specificities and paradigms from service-oriented components and Java can be handled to provide an easy and efficient static analysis through tainted object propagation. As stated in section 1, we studied the particular case of OSGi platforms, as OSGi provides a service-oriented platform whose behavior is independent from the virtual machine's.

4.1 Static Security Analysis for Service-oriented Components

In Java Service-oriented components, there are several ways for untrusted components to corrupt objects used by targeted components or to get code executed in the context of the targeted component. Components are designed to be independent but several means of interaction exist: registered services allow a component to execute code from another component with its own parameters, whereas package importation and exportation mechanisms in OSGi allow components to dynamically provide exported classes to other components and to retrieve classes imported by others. Service calls are executed in any order and there may be distributed use and modification of critical objects. Those specificities of service-oriented architectures constitute unexplored challenges for static security audits that we discuss in this section.

4.1.1 Static Analysis of a Standalone Component

The static analysis of a standalone component without its actual dependencies brings new challenges. We show that specificities from standalone component analysis allow to easily distinguish potentially malicious data from harmless data, and that external data does not have to be analyzed.

Some objects inside components are exposed to external modifications. A component-based application has to be viewed as a single stand-alone application with multiple modules that can be plugged in and out. The modifications made by a single component can have repercussions on others. For instance, shared static fields can be modified by any component. Even if the field is private, it is possible to get an object serialized with crafted fields. It is even possible to modify the fields by subclassing when the targeted class is not *final*. In component-based platforms, each component uses a different ClassLoader that knows only classes from the JDK, from the component and from imported packages. Therefore, a malicious component can only modify the static fields from packages exported by other components. From a static analysis point of view, all static fields from exported packages should be considered as corruptible.

All objects from third-party components are untrusted. As other components can be malicious, objects provided by external sources are untrusted. All imported packages but JDK and framework packages are provided by other components and are therefore untrusted. Moreover, parameters coming from registered services are forged and instantiated by others and can be crafted to alter the behavior of the component. Any of those objects that can be injected by external components has to be considered as untrusted.

Moreover, the component-based programming model approach brings a significant difference with past work on vulnerability detection for Java applications, as it is possible to execute unknown code, provided by untrusted or unknown components. For example, when Livshits *et al.* [8] detect potential SQL injections, they follow the propagation of corrupted Strings throughout a sample program. In Java, Strings are *final* objects, therefore their methods cannot get modified. At the opposite, we are considering potentially non-*final* objects forged by malicious components on the same platform.

When analyzing a single component, those references to third-party objects and code cannot be an-

alyzed. This does not compromise static analysis but rather simplifies it. Indeed, if untrusted code gets executed, it can only influence its parameters and the return value. As Java passes parameters by value and as any object is a pointer, modifying an entire parameter does not have any repercussion in the calling context. However, there are several ways to modify the fields of a parameter within a method: subclassing, cast, or deserialization [6]. The return value and the fields of the parameters can therefore always be considered as untrusted when calling an untrusted method. Therefore, while analyzing a component, any untrusted reference can be treated the same way, and a worst case assumed.

4.1.2 Security Analysis Concerns

The security analysis of a service-oriented application brings new challenges to be addressed: the order of service calls and the existence of mitigation techniques.

Service calls are subject to race conditions. A particular problem of service-oriented architectures is the existence of race conditions. In our case, race conditions happen when an untrusted object is introduced in a particular service call, stored as a field in any object and used as a sink in another service call. Race conditions can therefore miss tainted sinks if service calls are analyzed independently. Our approach to solve race conditions is to keep track of which method has been analyzed in which particular taint context. The taint context means the effective taint of the parameters and any related field accessed within the method. Each service call is analyzed independently, while keeping track of the taint context for each method call. Then, each service call is reanalyzed if the taint context is different from the original context in the first analysis.

Object-oriented sanitization is irrelevant. With any security analysis comes the problem of sanitization. Sanitizations are runtime checks within the components performed to ensure the safety of a particular object. If the object is unsafe, it is sanitized and gets a default, harmless value to protect the execution. Security analyses have to detect such code to avoid reporting false positives. Most sanitization cases do not differ from classical range value analyses. However, our particular analysis needs to keep track of whole objects controlled by the attacker. Therefore, a relevant real-time sanitization would suppose that the programmer checks for the actual class of the object and perform special actions for irrelevant classes.

We label such sanitizations as object-oriented and give an example in Alg. 3. In this algorithm, the method *doCheck* returns true if the list passed as a parameter is an instance of java.util.ArrayList, or false otherwise.

```
public boolean doCheck(ArrayList list){
 String toCheck = list.getClass();
 String expect = "java.util.ArrayList";

 return toCheck.equals(expect);
}
```

Alg. 3: Sanitization of an ArrayList Object

If this list is a crafted extension of *ArrayList*, the check fails. However, this kind of code is clearly not in line with Java ideology. A good Java programmer would provide its own *final* extension of the *ArrayList* class and use only instances of this new class whose methods can be trusted. Therefore, object-oriented sanitization seems an irrelevant effort and does not need particular attention.

In a service-oriented architecture, untrusted code and objects are easily identified. Moreover, missing code and objects do not need to be analyzed, as they are always untrusted and therefore the worst case can always be assumed in terms of security. It is also possible to handle race conditions between service calls by keeping and updating the actual trust level of shared objects. Finally, one may note that vulnerability mitigation by object sanitization is very unlikely and not conform to Java principles.

4.2 Tainted Object Propagation for Service-oriented Components

Specificities from the Java programming language and service-oriented architectures reduce the attack surface and limit the ways corrupted data propagates throughout a program. In terms of tainted object propagation, they provide generic and precise taint sources and taint propagators for Java service-oriented components. Only sinks have to be specified differently for each vulnerability family.

4.2.1 A Generic Taint Sources Definition

As stated in 4.1.1, the component isolation in OSGi does not allow access to the private, non-exported classes of other components. Exported classed are however at risk. If a component exports a package, public static fields can be directly modified by any component importing this package. Moreover, private static fields can also be modified via malicious subclassing, deserialization or cast. Objects declared or controlled by untrusted imported packages constitute a threat, as their internal state and the associated methods are controlled by

the untrusted package provider. With the addition of service parameters that can be provided by untrusted components, we have the list of taint sources for one specific component, which are all objects that can be directly modified by untrusted third-party code:

- registered services parameters
- objects from untrusted imported packages
- static fields from exported packages

For most OSGi components, we consider every package of the framework and classical JDK packages as trusted, which is the case with Java application servers or Java applets. We do not consider the additional taint sources represented by the trusted system resources that can be manipulated by malicious components, such as files. While those third-party resources can actually inject tainted values in code, they cannot inject tainted objects with arbitrary methods. However, if a weak component uses tainted data such as a filename to manipulate third-party resources, the resulting object is accordingly tainted.

4.2.2 A Generic Taint Propagators Specification

The Java language greatly reduces the taint propagation vectors, as it does not allow explicit memory management. Only three operations can propagate the taint from one object to another. Those propagators are illustrated in Alg. 4.

- execution of untrusted code (lines 4 and 9)
- assignments (lines 7 and 8)
- arithmetics (lines 12 and 13)

```
1  import untrustedpackage.Untrusted;
2
3  class A(){
4   void method(Trusted trusted1, int iUntrusted){
5    Untrusted untrusted = new Untrusted();
6    Trusted trusted2 = new Trusted();
7    trusted1 = untrusted;
8    trusted1 = untrusted.field;
9    trusted1 = untrusted.uMethod(trusted2);
10
11   int iTrusted;
12   iTrusted = 1 + iUntrusted;
13   iTrusted = -iUntrusted;
14  }
15 }
```

Alg. 4: Taint Propagators examples

In instructions at lines 4, 7, 8, 9, 12 and 13, a tainted value is propagated. If a class from an imported package is instantiated, then its fields and associated methods are controlled by untrusted code. Of course, assignment instructions propagate the taint of a value and the fields of an untrusted object are untrusted. The execution of untrusted code also propagates the taint. If an untrusted method is executed, the fields of all parameters may be replaced and are therefore tainted. Moreover, the returned object is also controlled by the attacker.

4.2.3 A Specific Sink Specification

Therefore, a service-oriented tainted object propagation on Java components just have to specify the sinks for each single vulnerability class. Finding the right set of sinks is a human expert task and we therefore did not aim at providing the best sinks specifications for the targeted vulnerabilities. The most important part for us is to have reliable sinks that can assess the vulnerability detection of a specific vulnerability. Nevertheless, we describe in this section the method we used to find sinks for a sample vulnerability: the synchronized denial of service.

General Method. First of all, the expert has to identify the specificities that distinguish vulnerable and regular code. All possible ways for a developer to bring these specificities into its code are the exploitation vectors. Those exploitation vectors can come from different layers in the OSGi model:

- The Java language, restricted to its regular set of bytecode instructions.
- The standard Java library, which includes features common to all code running on the platform. This set is restricted to the packages loaded by the initial ClassLoader such as packages from *java.lang*.
- The OSGi Framework, which manages components and provides facilities to register and use services.
- Third-party code, which includes other components, or packages exported by these components. This third-party code is likely to be untrusted and to constitute a threat source.

The analyst has to look whether one of these layers can bring one or more exploitation vectors that can be used by untrusted code to exploit the vulnerability. When these particular methods or instructions are found, the last step is to determine which objects linked to each of those exploitation vectors must be corrupted for the exploitation to be possible. Those objects will constitute the sinks set.

Application to the Synchronized Denial of Service Case. In the case of the synchronized denial of service vulnerability, the specificity of vulnerable code is simple to determine: it is the fact that any code called

in synchronized context never returns to the caller. We then go through the different layers of code that can be used in the component, to find which exploitation vectors could never return to the caller:

- The Java language. Loops can never return back to the caller. In Alg. 1, if the *hasNext* method always returns true, the while condition is always true and indefinitely loops.
- The standard Java library. A method that throws *InterruptedException* can block on I/O operations and never return back to the caller. Moreover, the Java standard library brings reflection, that can be used to call any method. This method can be a blocking one, or an ill-coded method that never return to the caller.
- The OSGi Framework. The OSGi framework does not bring any exploitation vector.
- Third-party code. Any untrusted method call from third-party code can block or never return to the caller, as shown in Alg. 2.

We then list how an attacker could influence the objects associated with each of the exploitation vectors to find the sinks list. The loops can block if their condition variable is controlled by the attacker. Therefore, the goto conditions in Java Bytecode should not get corrupted and are sinks. Methods that throw *InterruptedException* are known to block - most I/O operations throw this exception. A precise sink specification would have to determine, for each of these methods, which parameters can influence the blocking operations. We considered here that each parameter of these methods are dangerous and are therefore sinks. Corrupted reflection calls are always an issue in Java. If the attacker controls the parameters of a call to a reflection API, it can execute any method and hijack the execution flow. Executing untrusted code in synchronized context is a risk, as untrusted code may block itself. Therefore, any object executing an untrusted method is a sink. Untrusted methods are either calls to external untrusted packages, or calls to methods from trusted packages that can be overridden.

Here is the list of sinks to follow for this particular vulnerability:

- The Java language. *goto* conditions when the code can loop.
- The standard Java library. Parameters from methods that throw *InterruptedException* and parameters from reflection calls.
- The OSGi Framework. The OSGi framework does not bring any sink.
- Third-party code. Any object executing a method from an untrusted package or a non-*final* method from a non-*final* class from a trusted package.

We defined sinks for each vulnerability to detect in the same fashion. Those specifications represent simple predicates on variables and objects and can easily be identified by a static analysis tool. We provide more details on the underlying implementation of sink specification in section 5.1.

5 STOP

STOP is a Service-oriented Tainted Object Propagation tool, which implements the analysis technique described in the previous section. STOP performs a summary-based points-to analysis to extract points-to relations and propagates the taint on-the-fly. It has been evaluated on several proof of concept components and components from the open-source OSGi project Apache Felix[3].

5.1 STOP Overview

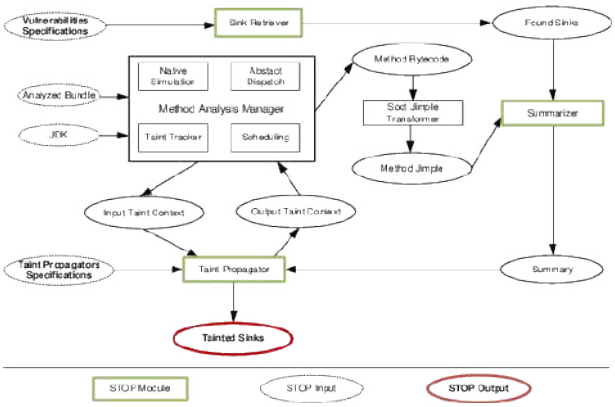


Fig. 5: STOP Overview

STOP is a vulnerability detection tool of roughly 4,000 lines of code that searches an OSGi bundle for one or several vulnerability types. STOP is implemented on top of Soot, a popular extensible framework to perform and implement various analyses and optimizations on Java code. It transforms Java source files or Java Bytecode to much simpler representations that keep the semantics while restricting the set of instructions. The most used representation within Soot is Jimple [25], a

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

simple 3-adresses register-based ISA which uses only 15 instructions, instead of the more than 200 in normal Java Bytecode. In STOP, Soot transforms class files to Jimple and performs a class hierarchy analysis.

Fig. 5 provides an overview of the application architecture. The Method Analysis Manager is STOP's core as it keeps track of context sensitivity and control flow, and orchestrates the other modules. STOP injects a crafted main class containing subcalls to each of the public service method in the component. This dummy main class serves as an entry point for the analysis. In the next paragraphs, we explain in further depth two modules, specific to our summary-based, service-oriented tainted object propagation.

The SinkRetriever. The SinkRetriever maintains a list of sink analyzers. Each sink analyzer is the implementation of a particular sink specification. Alg. 6 is the formal definition of sink analyzers.

```
1 public interface ISinkAnalyzer{
2  Set<Sink> analyzeStatement(Unit statement);
3  void enterMethod(SootMethod method);
4  void exitMethod(SootMethod method);
5 }
```

Alg. 6: Sink analyzer formal definition

The callbacks *enterMethod* and *exitMethod* provide relevant information about the method and the current stack trace. The actual sink detection happens in the method *analyzeStatement*, which returns any sink found in a particular statement. Simple checks can be performed to detect whether a variable in the statement matches a sink definition. Sink analyzers are currently hardcoded for each vulnerability family but could be described in a specific descriptive language like PQL [26]. Sink analyzers can be activated or deactivated depending on the targeted vulnerabilities.

The Summarizer. STOP provides a summary-based context-sensitivity. The summary-based technique adapts well to analyses needing to keep only a small amount of data from each different method. In STOP, the Summarizer reduces the information stored for one method to the minimum needed for further evaluation of the actual objects taint. The summary produced keeps track of relationships between input variables, from which the taint may flow into the method, to output variables, such as sinks and variables used in other methods. Therefore, STOP easily deduces the actual taint of a method's output variables from its summary and the initial taint of input variables.

Fortunately, relevant input and output variables are quite limited. For any method, the input variables are the fields of the instance if the method is not static, the output variables of subcalls and the internal taint sources if any. Output variables are the fields of the instance object if the method is not static, the parameters' fields, the returned values and the internal sinks if any.

Keeping only relationships between those variables considerably reduces the amount of data needed in memory compared to analyses needing the whole pointer assignment graph. To optimize its analysis, STOP skips methods that don't have any tainted value within their input variables. This particular heuristic can generate false negatives as a very deep taint source, such as a corrupted static field, could be missed. However, as STOP is highly configurable, this mode can be disabled and one can force a full analysis. Our practical experiments showed that disabling this heuristic divides the analysis time by a coefficient between 3 and 5. On the several real life components we tested, no such false positives were found. Therefore, this heuristic allows a fast analysis while ensuring a good threat coverage.

5.2 Experimental Results

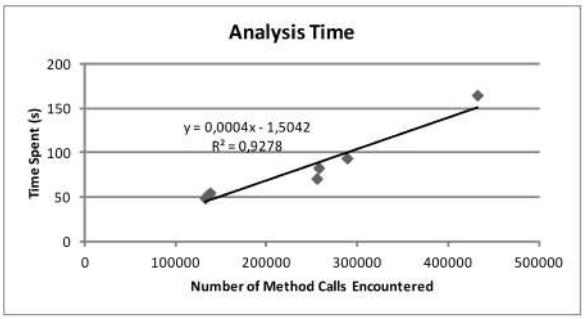
5.2.1 Benchmark Evaluation

The benchmark suite used to validate STOP contains four vulnerable proof of concept bundles, as well as 3 different OSGi bundles coming from the Apache Felix project.

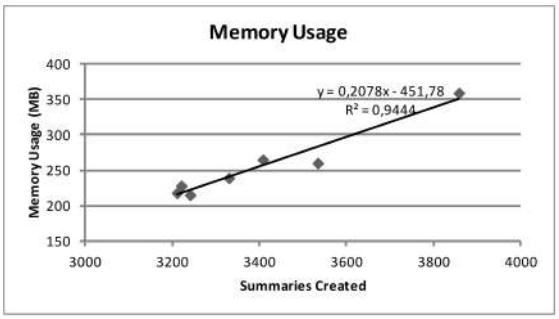
The proof of concept bundles provide basic vulnerabilities implementations. Those bundles are lightweight but can assess that simple vulnerability implementations are indeed detected. The lifecycle violation a very common vulnerability which happens when trusted code allows internal untrusted code execution. Synchronized vulnerabilities detailed in section 2.2 are a subset of lifecycle violations. Restricted APIs gateway happens when a privileged bundle elevates its privileges to access restricted APIs with untrusted data. Privileged Deserialization is an uncommon yet critical vulnerability that happens when untrusted objects are deserialized in privileged context. Those vulnerabilities allow untrusted code to respectively prevent vulnerable components uninstallation, deny vulnerable services, try to exploit access to native APIs or escalate privileges, disregarding the actual security policy [6]. We also pick three interesting bundles from the Apache Felix project, since each of them has a special characteristic that can make it complex to analyze. The Apache Felix Bundle

	Bundle	Lines of Code	Attack Surface - Public Service Methods	Complexity - Internal Method Calls
1	Life Cycle Bundle	70	1	126,655
2	Restricted APIs Bundle	145	3	123,105
3	Synchronized DoS Bundle	109	3	108,925
4	Privileged Deserializer	201	1	104,351
5	Apache Felix Web Console 3.1.8	10,482	14	375,981
6	Apache Felix Bundle Repository 1.6.4	5,735	25	493,786
7	Apache Felix Shell 1.4.2	2,782	87	234,786

Table 2: Benchmark information



(a) Analysis Time and Subcalls Correlation



(b) Memory Usage and Summaries Creation Correlation

Fig. 7: Relations between Analysis Time and Memory Usage with Components Complexity

Repository relies on a lot of external libraries which implies a high complexity, as shown by the number of calls encountered by a sample STOP analysis. The Apache Felix Web Console has more than 11,000 lines of code, which is a lot for a single component. The Apache Felix Shell has a lot of registered services and therefore a high attack surface to cover.

Table 2 provides more information on those bundles. STOP has been tested on a computer having a 2.10 GHz processor and 4 GB of memory, with Soot 2.4.0 and Sun JDK 1.6.0_23.

5.2.2 Performance

Memory usage and time spent while analyzing are the keys to evaluate a static analysis tool. We evaluated STOP's performance on the different bundles presented in table 2. Fig. 7 contains the results of this study. Each bundle has been analyzed for all 4 vulnerabilities, with the analyzer considering that a Security Manager is used in the application environment and therefore tracking for each call whether the context is privileged or not.

The time needed to analyze components is a matter of minutes, even for complex components such as the Apache Felix Bundle Repository which has been analyzed in 160 seconds. We can also observe a linear relationship between the number of method calls encountered during the analysis and the analysis duration. This observation is supported by Fig. 7a that shows an acceptable linear regression with a correlation coefficient of 0.96. We therefore assume that the time spent in the analysis does not explode with the complexity of the component, as it seems to follow a linear relationship of the number of subcalls to analyze.

We observe in Fig. 7b that the number of summaries created does not vary heavily from one component to another, even if the size and complexity of the studied components are very different. This is mainly because the classes of the Java Standard Library are strongly connected. When analyzing one method, lots of other classes and methods have to be processed as well. Therefore, summarizing methods seems a good option as it levels the memory usage needed for the different components.

We see that the maximum memory usage for the most complex component is approximately 360 MB, which is quite low compared to the nowadays common hardware configurations. From our experimentations, the cloning-based context-sensitive technique developed by Whaley *et al.* [24] requires several GB of memory with recent JDKs. The refinement-based technique [23] requires around ten times less memory than

Bundle	Lifecycle Violations	Restricted APIs Gateway	Synchronized DoS	Privileged Deserialization
1-4	1	1	1	1
5	5	0	2	0
6	13	1	10	0
7	41	0	18	0

Table 3: Vulnerabilities Found

our approach but does not seem a valid option as we do not know where sinks are located before the analysis starts. Fig. 7b supports that the memory usage evolves roughly with the number of summaries needed, which seems normal as summaries are the biggest data structure maintained in STOP.

5.2.3 Vulnerability Detection

STOP implements a default vulnerability specification for the 4 vulnerabilities of the proof of concept bundles. Table 3 shows the results of the vulnerability detection for each studied bundle. The proof of concept bundles have only been audited for their related vulnerability. For each pair studied bundle/detected vulnerability, we report the number of different instances found.

We first observe that the vulnerability detection for the proof of concept bundles works, as each time the sample vulnerability injected is indeed detected. We successfully verified that the highlighted sinks were indeed signatures of the vulnerabilities introduced on purpose. Vulnerabilities can also be found in all Apache Felix bundles. We manually assessed the correctness of the detection and found no false positives. We were however not able to assess whether the analysis missed vulnerabilities, as the whole JDK should be reviewed. As expected, table 3 demonstrates that vulnerabilities allowing to bypass security layers, such as the privileged deserialization and the uncontrolled access, do not seem to represent a common exposure of real life components.

Unfortunately, the lack of past work on component-related vulnerabilities does not allow us to compare the accuracy of our vulnerability detection with others techniques, in terms of false negatives essentially. The only relevant tool, WCA [17], is not able to find our targeted vulnerabilities, as it uses simple pattern matching.

We noted that, even if the vulnerabilities signatures are different, most vulnerabilities break the same set of high-level best practices for components security. For example, all components execute methods from untrusted objects that can alter their life cycle and quality of service. It is not surprising to find instances of such vulnerabilities, as safe code interaction is not a prerequisite in Java and service-oriented ideologies. For example, any component can always call *System.exit(0)* to exit the whole platform.

From those vulnerability reports, we developed proof of concept bundles that can successfully exploit vulnerabilities in these Apache Felix bundles. For further details, the reader may refer to our recent technical report [6] containing exploit examples in each Apache Felix bundle analyzed.

6 Conclusion

The emergence of extensible component-based platforms brings up new security challenges, mainly due to components interactions via services. These issues that already existed in classical Java applications are emphasized by the platforms dynamics and the fact that untrusted code can be automatically installed. Recent acknowledgment from the Google Mobile Team that malicious software could be downloaded from their official Android Market [1] proves that in such environments, components have to protect themselves from an untrusted environment. To protect those components and automatically detect components vulnerabilities, a static analysis through tainted object propagation seems a promising approach.

We proposed a service-oriented tainted object propagation mechanism, that takes advantage of the design of service-oriented platforms and the specificities of the Java programming language to detect those vulnerabilities. We also evaluated the most promising techniques to perform a points-to analysis as a basis of our tainted object propagation. We found that a summary-based implementation is a good basis to build a service-oriented tainted object propagation.

We specified some core vulnerabilities in terms of tainted object propagation and were able to detect vulnerabilities on several Apache Felix components. The fact that we found vulnerabilities in those components is not surprising but rather illustrates the tradeoff between tight control and dynamics in programs. Java service-oriented components try to use generic and dynamic code at its maximum potential and necessarily give up essential controls in terms of security. Writing a secure Java component is still feasible, but brings too many constraints for most developers.

We were not able to study the existence of missed vulnerabilities, as manually auditing the whole JDK is very time-consuming. This would allow to complete our observation regarding the accuracy and the limits of such a vulnerability specification. Further work

on STOP should also focus on currently missing features such as reflection calls, uncommon native methods and untrusted data from the filesystem and databases. We also observed that most vulnerabilities have a lot of sinks in common. Therefore, studying programming best practices that prevent component vulnerabilities, and performing tainted object propagation to spot bad practices, can be a promising approach.

Acknowledgments

We would like to thank the anonymous reviewers for their detailed discussions, Yvan Royon from Alcatel Lucent for his knowledge and accurate criticism, Cédric Lauradoux for his constructive and complete reviews and the whole Amazones team for providing us a convivial and productive working environment. This article is granted by the LISE (Liability Issues in Software Engineering) project, funded by the ANR (Agence Nationale de la Recherche) under the SeSur 2007 program (ANR-07-SESU-007).

References

1. Google Mobile Team. An update on Android Market security.
2. O.S.G.i. Alliance. *OSGi service platform core specifications*.
3. JSR 118 Expert Group. *MIDP 2.0, Sun specification*, 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. Pierre Parrend and Stéphane Frénot. More vulnerabilities in the Java/OSGi platform: a focus on bundle interactions. Research Report RR-6649, INRIA, 2008.
6. François Goichon and Stéphane Frénot. Exploiting Java code interactions. Technical Report RT-0419, INRIA, 2011.
7. Rain Forest Puppy. NT web technology vulnerabilities. *Phrack*, 54, 1998.
8. 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.
9. Omer Tripp, Marco Pistoia, Stephen J. Fink, Manu Sridharan, and Omri Weisman. TAJ: effective taint analysis of web applications. In *PLDI '09: Proceedings of the 2009 ACM SIGPLAN conference on Programming language design and implementation*, pages 87–97, New York, NY, USA, 2009. ACM.
10. Liu Yin and Milanova Ana. Static information flow analysis for Java. Technical report, Rensselaer Polytechnic Institute, 2008.
11. William G. J. Halfond and Alessandro Orso. AMNESIA: analysis and monitoring for neutralizing SQL-injection attacks. In *ASE '05: Proceedings of the 20th IEEE/ACM international Conference on Automated software engineering*, pages 174–183, New York, NY, USA, 2005. ACM.
12. Sun Microsystems Inc. *Java Security Architecture Specifications*, 2002.
13. Almut Herzog. Performance of the Java security manager. *Computers & Security*, 24(3):192–207, 2005.
14. Ollie Whitehouse. Analysis of GS protections in Microsoft Windows Vista. Technical report, Symantec Advanced Threat Research, 2006.
15. Vivek Haldar, Deepak Chandra, and Michael Franz. Dynamic taint propagation for Java. In *ACSAC '05: Proceedings of the 21st Annual Computer Security Applications Conference*, pages 303–311, Washington, DC, USA, 2005. IEEE Computer Society.
16. M. Pistoia, S. Chandra, S. J. Fink, and E. Yahav. A survey of static analysis methods for identifying security vulnerabilities in software systems. *IBM Syst. J.*, 46(2):265–288, 2007.
17. Pierre Parrend. Enhancing automated detection of vulnerabilities in Java components. In *AReS '09: Fourth International Conference on Availability, Reliability and Security*, Fukuoka, Japan, 2009.
18. Marco Avvenuti, Cinzia Bernardeschi, and Nicoletta De Francesco. Java bytecode verification for secure information flow. *SIGPLAN Not.*, 38(12):20–27, 2003.
19. Monica S. Lam, John Whaley, V. Benjamin Livshits, Michael C. Martin, Dzintars Avots, Michael Carbin, and Christopher Unkel. Context-sensitive program analysis as database queries. In *PODS '05: Proceedings of the twenty-fourth ACM SIGMOD-SIGACT-SIGART symposium on Principles of database systems*, pages 1–12, New York, NY, USA, 2005. ACM.
20. Ondřej Lhoták and Laurie Hendren. Context-sensitive points-to analysis: is it worth it? Technical report, McGill University, Sable Research Group, 2005.
21. John Whaley and Martin Rinard. Compositional pointer and escape analysis for Java programs. *SIGPLAN Not.*, 34(10):187–206, 1999.
22. Thomas Reps, Susan Horwitz, and Mooly Sagiv. Precise interprocedural dataflow analysis via graph reachability. In *POPL '95: Proceedings of the 22nd ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 49–61, New York, NY, USA, 1995. ACM.
23. Manu Sridharan and Rastislav Bodík. Refinement-based context-sensitive points-to analysis for Java. *SIGPLAN Not.*, 41(6):387–400, 2006.
24. John Whaley and Monica S. Lam. Cloning-based context-sensitive pointer alias analysis using binary decision diagrams. *SIGPLAN Not.*, 39(6):131–144, 2004.
25. Ondřej Lhoták and Laurie Hendren. Scaling Java points-to analysis using Spark. In *CC '03: Proceedings of the 12th International Conference on Compiler Construction*, volume 2622 of *LNCS*, pages 153–169, Warsaw, Poland, April 2003. Springer.
26. Michael Martin, Benjamin Livshits, and Monica S. Lam. Finding application errors and security flaws using PQL: a program query language. In *OOPSLA '05: Proceedings of the 20th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, pages 365–383, New York, NY, USA, 2005. ACM.