

Security Crosscutting Concerns and AspectJ*

Dima AlHadidi , Nadia Belblidia, Mourad Debbabi
CSL-CIISE, Concordia University
Quebec, Canada
dm_alhad@cs.concordia.ca, {na_bel,debbabi}@ciise.concordia.ca

Abstract

This paper presents a brief description for the mostly used AOP approaches and analyzes them from a security point of view. AspectJ is then considered the most appropriate language to enforce security issues but at the same time it is not complete. This paper shows that some security crosscutting concerns need more means than those that are currently exist in AspectJ.

1 Motivations and Background

Application security becomes one of the fastest growing fields in IT market today. Security precautions built inside applications minimize the probability that hackers will be able to manipulate applications and access critical data.

Aspect Oriented Programming (AOP) is a new paradigm that complements the Object Oriented Programming (OOP) paradigm by supporting a better separation for crosscutting concerns. Crosscutting concerns such as security are concerns that are tangled and scattered across more than one module. AOP languages such as AspectJ, HyperJ, and DJ have adopted pointcut-advice model, multi-dimensional separation of concerns model, and adaptive programming model respectively. An analysis is done for these models from a security perspective. As a result of this analysis, AspectJ, which supports the pointcut-advice model, is considered the most appropriate language to enforce security in Java Applications.

AspectJ extends Java programming language. AspectJ aspects contain new parts that do not exist in an ordinary Java class such as: join points, pointcuts, and advices. AspectJ is the right choice to enforce security but it needs more means than those that are currently exist to do this job successfully. This issue is the one that we will talk about it

*This research is funded by NSERC(Natural Sciences and Engineering Research Council of Canada) DND(Department of National Defence) grant in collaboration with Bell Canada and DRDC(Defense Research and Development Canada) at Valcartier.

extensively in this paper.

This paper contains two basic parts. The first part (section 2) discusses briefly AOP approaches and gives an analysis to these approaches from a security point of view. The second part (section 3) discusses the lacks in AspectJ that are needed to enforce security issues successfully. Finally, a few remarks and discussion of future research are ultimately sketched as a conclusion in Section 4.

2 AOP Security Appropriateness

The mostly used AOP approaches will be discussed in this section followed by an appropriateness analysis for these approaches from a security perspective.

2.1 Pointcut-Advice Model

The fundamental concepts of the pointcut-advice approach are: join points, pointcuts, and advices. A join point is a point in the control flow graph of an application. A pointcut is a constructor that designates a set of join points. Advices are pieces of code attached to pointcuts. An advice is executed when join points satisfying its pointcut are reached. AspectJ [5] is probably the most known representative of the pointcut-advice model.

2.2 Multi-Dimensional Separation of Concerns

Multi-dimensional separation of concerns (MDSOC) [10] allows developers to partition overlapping concerns in software along multiple dimensions of composition and decomposition. MDSOC treats all concerns as first-class and co-equal, including components and aspects, allowing them to be encapsulated and composed at will. As a result, the approach is symmetric, as opposed to the pointcut-advice approach where aspects are composed (woven) into the base application. Hyperspaces are an approach to achieve MDSOC where multiple decompositions of the program are modeled as a set of units called hyperslices (concerns). HyperJ [10] supports hyperspaces in Java. In HyperJ, a set

of hyperslices can be combined into a hypermodule using composition rules.

2.3 Adaptive Programming

Adaptive programming (AP) (proposed by Demeter group [2]) has used the ideas of AOP several years before the name aspect oriented programming was coined. Following the Demeter law, a programming style rule for loose coupling between the structure and behavior concerns can result in a large number of methods scattered throughout the program. Adaptive programming with traversal strategies and adaptive visitors avoids this problem [9].

DJ [9] is a Java library for adaptive programming that allows traversal strategies to be constructed and interpreted dynamically at run time. DJ allows traversing a graph object according to the traversal strategy and allows specifying a visitor to be executed before or after specific nodes.

2.4 Appropriateness Analysis

All the above AOP approaches are candidates to separate crosscutting concerns in general. The multi-dimensional separation of concerns has a serious limitation from a security perspective. It is not possible to add functionality before, after, or around a field access. Access authentication to a given field in a given class is a simple security example that we can not handle with HyperJ which is a representative for MDSOC model. MDSOC approach works at the method granularity and consequently it can not operate within a method body. HyperJ does not support pulling apart of code within method bodies. Picking out multiple concerns within method bodies is required in many situations to enforce security.

The adaptive programming is concerned with the loose coupling between structure and behavior and focuses on certain kinds of concerns. For example, DJ is unable to change a method by a more secure one.

The pointcut-advice model is the most popular model. It offers a better granularity than MDSOC approach and considers more general kinds of concerns than the adaptive programming. Furthermore, the pointcut-advice model adapts extensively the pull approach. It allows tracking subtle points in the control flow of the application. For example points where methods are invoked and fields are set. Hence, we choose AspectJ as the candidate to enforce security issues in Java applications.

3 AspectJ Shortcomings

This section suggests some possible extensions to AspectJ explained by examples in order to handle security issues in applications successfully.

```
pointcut* displayState():
pcflow(execution(void SecurityElement+.draw())
&& get(* SecurityElement+.*);
after set(<displayState()>) (): {
Display.update();
//Take an action according to the type of the change
}
```

Table 1. New Version of Kiczales's Example

3.1 Predicted Control Flow Pointcut

Kiczales [4] has proposed the predicted control flow pointcut (`pcflow`) but this pointcut has not been implemented yet. A pointcut `pcflow(p)` matches at a join point if there may exist a path to another join point where `p` matches. Kiczales has discussed this new pointcut with his known drawing example. In his example, he has used predicted control flow pointcut to select points in the execution that modify variables previously read within the control flow of a method called `FigureElement+.draw()`.

We can get benefit from this example to harden security of applications. In [11], authors spoke about detecting intruders with visual data analysis. Based on this idea, we can draw some charts for security important parameters such as file activity, registry activity, or network traffic. These charts can be analyzed to discover if something wrong happens. By using the same concept in Kiczales's example, any changes in these charts by setting these parameters in a way or another will not only be reflected in the display but also some necessary steps must be taken in response to such changes to protect the system and application. So Kiczales's example can be rewritten as in Table 1.

3.2 Dataflow Pointcut

Masuhara and Kawauchi [6] have defined a dataflow pointcut for security purposes but this pointcut has not been implemented yet. The pointcut identifies join points based on the origins of values. Cross-site scripting (XSS) problem in web-applications is an example presented by them to clarify the need for such a pointcut. A Web site might be vulnerable to XSS attacks if it reflects input back to the user such as search engines and shopping sites. Attacker crafts a link containing malicious code and let the victim click on it by different ways. The victim's browser transmits the attacker's code to the Web site as part of the URL. The Web site reflects the input to the victim's browser. The malicious code runs on the victim's browser because it thinks the code comes from the vulnerable Web site. They define a dflow pointcut to solve this problem. The pointcut intercepts any joint point that prints a string created from one of the client's input parameters. Sanitizing is used to replace this string with quoted characters.

Here is another example that clarifies the need for such a pointcut. If a program opens a confidential file, reads data from this file, and then sends data over the net, this will be considered dangerous from a security point of view. A data-flow analysis using `dflow` pointcut can tell whether the data sent over the net actually depends on the information read from the confidential file.

3.3 Loop Pointcut

Harbulot and Gurd present in [3] a loop join point model which demonstrates the need for a more complex join point in AspectJ. This research lacks the analysis of infinite loops and loops that contain Boolean conditions. Through pointcuts that pick out such loops, an excessive security problems can be solved easily.

Malicious-code writers exploit infinite loops to do their nefarious jobs by launching denial-of-service attacks. Halting the web browser is an example of a denial-of-service attack by running a code that opens a dialog window infinite number of times.

There is no general methods to specify whether a code will ever halt or run forever but AspectJ must include mechanisms to predict the existence of such infinite loops and then notifies the user if she wants to continue with this work or not. Urgent needs to pointcuts that are related to loops must push the research in this area.

3.4 Pattern Matching Wildcard

There is a need for a new wildcard in AspectJ to perform pattern matching. Although pattern matching can be done by plain AspectJ, it is however better to do it in a declarative manner to simplify the code. We illustrate this point with an example related to security. Viruses always inject themselves inside executable files. So, it is essential to control opening and writing files that have an “exe” extension. For example, let us write a pointcut that picks out all constructor call join points of the form `FileWriter(x,y)` where the parameter `x` is a string whose value ends with the word “exe”. Using plain AspectJ, the pointcut will have the following form:

```
pointcut p: call (FileWriter.new(String,String))
&& args(x,*) && if (isExtension(x));
```

Where `isExtension` is a Boolean method to test if its argument value ends with the word “exe”.

Although we were able to write the pointcut using plain AspectJ, this has been done with an extra methods like `isExtension`. We suggest another way that uses the same notations used in SQL such as `like` keyword and `%` character to ease the burden on the user and simplify the code. The previous pointcut definition can be rewritten according to our suggestion as:

```
public class Sensitive {
private String sensitiveInfo;
public void f(){...
System.out.println(sensitiveInfo)
... }}
```

Table 2. Type Pattern Modifiers.

```
pointcut p: call (FileWriter.new(String like
"%exe%", String))
```

Obviously, using such wildcards states directly the programmer’s intent and makes the program clear and crisp.

3.5 Type Pattern Modifiers

AspectJ uses four kinds of patterns in the pointcut syntax: Method pattern , constructor pattern , field pattern , and type pattern. Patterns are used inside primitive pointcut designators to match signatures and consequently to determine the required join points. The syntax of all patterns contains the modifiers keyword except the type pattern syntax. This section discusses the need for such a keyword in the type pattern syntax to enrich the matching process.

A Java class declaration may include the following modifier patterns: `public`, `abstract`, or `final`. A public class is a class that can be accessed from other packages. Any class, method, object, or variable that is not private is a potential entry point for an attack. Hence, using modifiers in the type pattern syntax should be very useful from a security point of view. The example in Table 2 describes a case where the public method `f()` inside the public class `Sensitive` delivers sensitive information. In this case, it is essential to add a security mechanism that authenticates the clients of such public classes that are exposed by the application to the outside world. Hence, we would like to be able to use a `public` modifier pattern in type pattern syntax to pick out public classes only.

3.6 Local Variables Set and Get

AspectJ allows to pick out join points where attributes are referenced or assigned through `get` and `set` designators but it does not provide similar pointcuts to local variables defined inside methods. New pointcut designators that do such a thing will increase the efficiency of AspectJ especially from a security point of view. For example, security debuggers may need to track the values of local variables inside methods. With such new pointcuts, it will be easy to write advices before or after the use of these variables to expose their values. Confidential data can be protected using these kinds of pointcuts by preventing them from being used improperly. A promising approach for protecting privacy and integrity of sensitive data is to statically check information flow within programs This approach is discussed

```

class Test {
private String sensitiveInfo;
public String publicInfo;
private void f(){
String localstr;
sensitiveInfo=/* Some Calculation*/;
localstr=sensitiveInfo; ....
publicInfo=localstr;}}

```

Table 3. Local variables Get and Set.

in [8]. Instead of doing static analysis, we will use AOP to insert checks before or after getting or setting fields or local variables. The following example in Table 3 clarifies the idea.

We can see that the sensitive information stored in the private field `sensitiveInfo` has been exposed by transferring its value to the local variable `localstr` defined inside the method `f()`. Then the value of `localstr` is stored inside the public field `publicInfo` which made the information accessible from outside the class. Using pointcuts that track fields as well as local variables can help us to find such a case and prevent it.

3.7 Synchronized Block Joinpoint

The synchronized block has not been treated yet in AspectJ or in any other AOP framework. Borner [1] has presented an article on these issues and has discussed the usefulness of capturing synchronized blocks such as calculating the time acquired by a lock and thread management. In this paper, we do care about the importance of such pointcuts for security issues. Suppose we have a synchronized block that launches a denial-of-service attack by containing a code that eats the CPU cycles like the code that implements Ackerman function in [7]. It is essential to have a joint point at the beginning of the synchronized block. Through this join point, we can write a before-advice that limits the CPU usage or limit the number of instructions that can run.

Let us take another example that is shown in Table 4. We need to insert advices before synchronized blocks because the same thread can acquire the lock twice. This behavior can cause a denial-of-service attack. To clarify more, if the thread who owns the lock manipulates with files, this will block users from accessing files to which they have access to. A before-advice can use Java assertions to check that you have not got a lock before entering a synchronized block.

4 Conclusion and Future Work

This paper discusses the use of (AOP) paradigm for security hardening in Java applications. In this paper we get

```

public class A {
public void f() {
Before-Advice: assert !Thread.holdsLock(this);
synchronized(this){...
/* access files*/ ... }}

```

Table 4. Synchronized Block

two birds in one stone. First, the paper provides an overview of the three mostly known AOP approaches, analyzes them from a security point of view, and retains the pointcut-advice approach as the most appropriate one. Second, it discusses the shortcomings of AspectJ to enforce security issues successfully. We have shown that security aspects must get benefit from concepts related to pointcut definitions in order to express some security hardening practices. In the future, we plan to give implementation solutions to all the above suggestions and come up with a comprehensive AOP security language.

References

- [1] J. Borner. Semantics for a Synchronized Block Join Point. <http://jonasborner.com/2005/07/18/semantics-for-a-synchronized-block-join-point/>, July 2005.
- [2] Demeter_Group. Demeter: Aspect-Oriented Software Development. <http://www.ccs.neu.edu/research/demeter/>.
- [3] B. Harbulot and J. R. Gurd. A Join Point for Loops in AspectJ. In *Proceedings of the 4th workshop on Foundations of Aspect-Oriented Languages (FOAL 2005)*, Mar. 2005.
- [4] G. Kiczales. The Fun has Just Begun, Keynote talk at AOSD 2003. <http://www.cs.ubc.ca/~gregor/>, 2003.
- [5] G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm, and W. Griswold. An Overview of AspectJ. Budapest, 2001. Springer Verlag.
- [6] H. Masuhara and K. Kawachi. Dataflow Pointcut in Aspect-Oriented Programming. In *APLAS*, pages 105–121, 2003.
- [7] G. McGraw and E. Felten. *Securing Java Getting Down to Business with Mobile Code*. John Wiley & Sons, 1999.
- [8] A. Myers. JFlow: Practical Mostly-Static Information Flow Control. In *Symposium on Principles of Programming Languages*, pages 228–241, 1999.
- [9] D. Orleans and K. Lieberherr. DJ: Dynamic Adaptive Programming in Java. Technical Report NU-CCS-2001-02, Northeastern University, Boston, MA 02115, USA, 2001.
- [10] H. Ossher and P. Tarr. Multi-Dimensional Separation of Concerns and the Hyperspace Approach. In *Proceedings of the Symposium on Software Architectures and Component Technology*. Kluwer, 2000.
- [11] S. Teoh, T. Jankun-Kelly, K. Ma, and F. Wu. Visual data analysis for detecting flaws and intruders in computer network systems. *IEEE Computer Graphics and Applications, special issue on Visual Analytics*, 2004.