How Secure is AOP and What can we Do about it?

Bart De Win  
Katholieke Universiteit Leuven  
DistriNet  
Celestijnenlaan 200A  
B-3001 Leuven, Belgium  
bartd@cs.kuleuven.be

Frank Piessens  
Katholieke Universiteit Leuven  
DistriNet  
Celestijnenlaan 200A  
B-3001 Leuven, Belgium  
frank@cs.kuleuven.be

Wouter Joosen  
Katholieke Universiteit Leuven  
DistriNet  
Celestijnenlaan 200A  
B-3001 Leuven, Belgium  
wouter@cs.kuleuven.be

ABSTRACT
From a software engineering perspective, using Aspect-Oriented Programming (AOP) to build secure software has clear advantages. Until recently, the security perspective of this approach has been given less attention, however. This paper analyses the security risks in using AOP to develop secure software and discusses one particular solution to some of the identified risks, an aspect permission system. This permission system is one part of an overall AOP-based development platform for secure software.

Categories and Subject Descriptors
D.2.0 [Software Engineering]: General Protection mechanisms; K.6.5 [Management of Computing and Information Systems]: Security and Protection

General Terms
design, security

Keywords
AOP, security, risks, permission system

1. INTRODUCTION
It is well known that software vulnerabilities introduced during the development activity are one of the main causes for the current lack of security in software systems. The inherent complexity of implementing security in software, both from a security and from a software engineering perspective, lies at the heart of this problem. From a software engineering perspective in particular, the pervasive character of security is not well supported by mainstream software development methods (such as object-orientation).

Aspect-Oriented Programming (AOP), a recent software engineering paradigm, promises to facilitate the modularisation of cross-cutting problems, including security. Several studies and testimonials have already discussed the benefits of using AOP to implement security solutions [3, 4, 9], but most of these results have highlighted the software engineering perspective. The benefits of using AOP from a security perspective are less clear, however. The security impact of using AOP to build secure systems has never been reported on in detail, and given the fact that AOP enables powerful interferences between modules in an application, it will be clear that the security properties of a system could be seriously jeopardised. This is particularly true for weaving-based AOP tools since they rewrite application logic (whether it be source code or class binaries). In order for AOP to get accepted as a development method for secure software, it is crucial that the security perspective is focused on as well, and that possible issues are resolved. The goal of this paper is to start filling this gap.

The contributions of this paper are twofold. Firstly, for one specific weaving-based AOP tool, AspectJ 5, the security characteristics of woven applications are analysed and several issues are discussed in more detail. Rather than being complete, we focus on the most striking issues, both from a development as well as from a runtime perspective. Secondly, possible solutions that can tackle these issues are discussed and one particular solution, an aspect permission system, is motivated and elaborated upon in more detail. Note that the material in this paper is work in progress; a full discussion and implementation of the ideas presented here would require an considerable amount of extra work. However, we consider the ideas presented in this paper sufficiently mature to serve as the basis for discussions during the workshop.

The structure of the rest of this paper is as follows. In Section 2, an example is introduced to motivate the security issues that can result from using AOP to develop security-sensitive software. Section 3 discusses the security risks of AOP in more detail and focuses on three particular types of risks. A solution to these problems in the form of a permission system for AOP-based applications is presented and discussed in Section 4. Finally, Section 5 concludes.

2. A MOTIVATING EXAMPLE
In this section we first demonstrate a concrete security problem related to the use of AOP. Of course, the size of the examples is intentionally kept small for demonstration purposes. The next section will analyse the problem from a more general perspective.

Consider a trivial application that consists of a class and an aspect. The class, SensitiveData has an internal data member secret that for demonstration purposes is chosen to be security-sensitive.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

ACM ’06, May 20–21, 2006, Shanghai, China.
Copyright 2006 ACM 1-59593-085-X/06/0005 ...$5.00.
package mypackage;
public class SensitiveData{
    private String secret;
    public SensitiveData(String s){
        secret = s;
    }
    String getSecret(){
        return secret;
    }
    public static void main(String[] args) {
        SensitiveData sd = new SensitiveData( "My_first_secret" );
        sd.setSecret("My_second_secret");
        System.out.println(sd.getSecret());
    }
}

In addition to this class, an aspect is introduced to seal the secret data of the class. It does so by (i) defining a pointcut that matches the execution of the getSecret() method and (ii) by defining an advice that intercepts invocations to this method and checks for every invocation whether it conforms to the security policy (represented as a separate class). If it does, the invocation is allowed to continue, if not a RuntimeException is thrown. Since the secret member was declared private in the class and since the only way to get external access to the secret is via the getSecret() method, it is expected that the security policy regarding the access of the secret member will be correctly enforced.

package security ;
aspect Authorization{
    private static Policy pol;
    pointcut accessrestriction (): execution (SensitiveData . getSecret ());
    void around ( ): accessrestriction () {
        if ( ! pol . isAllowed ( . . . ) )
            throw new RuntimeException ( "Denied!" );
        else proceed ( );
    }
}

class Policy{
    Policy(){
        //install policy into hashtable
    }
    public boolean isAllowed (Subject s , String method , Object o) {
        boolean result = false;
        //lookup in policy hashtable
        return result;
    }
}

Now suppose the SniffingAspect aspect (see code fragment below) is added to the system as well. The aspect affects all events where the value of the secret within SensitiveData is modified by printing the new value of the secret member to the standard output stream. Access to the private datamember secret is possible since the aspect is declared privileged. Consequently, the aforementioned security policy, assumed to be correctly implemented by SensitiveData and Authorization, is actually broken when inserting this aspect.

package unsecure ;
privileged aspect SniffingAspect{
    after (SensitiveData sd): {
        if (sd.getSecret().equals("My_first_secret")) {
            System.out.println("The old secret is ", sd.getSecret());
        } elseif (sd.getSecret().equals("My_second_secret")) {
            System.out.println("The new secret is ", sd.getSecret());
        }
    }
}

3. DETAILED PROBLEM STUDY
An extensive study on the applicability of AOP for security [2] has, among others, shown that the use of AOP can result in several security issues and, hence, AOP must be used with precaution to develop secure software. The origin of these security issues varies: some are related to the aspect-specific language extensions, while others are specific to a particular aspect compiler implementation. In this section, a selection of these security issues will be discussed with a focus on the risks that appear within the boundaries of a system, including programming language features as well as compiler results. It is, however, not the goal of the paper to present a complete security analysis.

In the following discussion we distinguish between the concepts of class and aspect where relevant. We use the term module to denote a development artefact that may be a class or an aspect.

3.1 Invocation interception
A first class of problems is related to the interception capabilities of AspectJ (and most other AOP tools). In brief, AspectJ offers the possibility to intercept a particular method invocation and influence the subsequent processing of this invocation. This can impact the security properties of a system in two ways.

Parameter alteration
A first issue involves the possibility to alter invocation parameters (or the return value) of a particular method invocation. This scenario is applicable to method invocations that originate from regular classes as well as from other aspects in the system. Apart from influencing the correctness of the system, the security properties can clearly be jeopardised as well. More specifically, for the example discussed in the previous section an extra aspect could easily be constructed that simply modifies the return value of the method invocation within the Authorization aspect to the Policy class such that all policy queries evaluate to true. This scenario is illustrated in the code below.

package unsecure ;
aspect PolicyMod{
    pointcut polcheck ( ): execution ( boolean Policy . isAllowed ( . . . ) );
    //invoke the policy and always
    //return true
    boolean around ( ): polcheck () {
        boolean res = proceed ( );
        return true;
    }
}

Another example in the context of Java security consists in modifying the permission presented to the AccessController (via the checkPermission() method) into a permission that
is supported by the active policy. This scenario can be easily implemented when the AccessController is invoked from the application code (e.g., for application-specific permission checks). Note that it will be a lot harder to employ this technique to interfere in the basic Java security system: this would require weaving the Java language runtime libraries as well. This is not common practice and it should be avoided at all cost from a security perspective!

**Invocation hijacking**

Apart from altering the parameters of an invocation, the invocation can be redirected or even discarded entirely using around advice. This is a construct explicitly designed to fully control the processing of an invocation by providing control over the exact moment and circumstances of the invocation execution.

From a security perspective, several dangerous scenarios are possible when this technique is used. In the example of Section 2, one could for instance redirect the invocation within the Authorization aspect (to the Policy class) to a less restrictive Policy class. In this way, more actions will be granted. One could also just discard this invocation entirely and always return true to a policy request. This can be achieved by just removing line 9 in the aspect described in the previous section. Compared with parameter alteration, this can make a difference in the case of side effects. Also, the effect of other aspects will be discarded as well. Finally, one could even imagine denial-of-service situations where invocations to particular (security) services are just blocked by some aspect (possibly in favour of other, less secure services).

The `@precedence` construct is one particular mechanism that could help to protect against these scenarios. By means of this construct, an aspect developer can indicate that particular aspects should have priority for advising joinpoints. The straightforward way to employ this for security would be to declare a security aspect to have precedence over all other aspects. Unfortunately, this scenario does not work very well in general: a security aspect will *not* always have precedence over all other aspects (e.g., in case of encrypted communication a logging aspect could want to log before or after encryption/decryption). Furthermore, for a considerably sized system containing a lot of aspects it is difficult to determine the exact order for all the aspects and in some cases the specification will even not be feasible (since precedence is specified at the level of aspects, rather then advices or even join points).\(^1\)

As a sidenote, it is interesting to note that it used to be possible to circumvent the application of advice in a very similar way. One could for instance, by crafting a specific around advice, make sure that invocations to the SensitiveData class were never checked by the Authorization aspect as a result of weaving. With the most recent versions of the AspectJ compiler (i.e., AspectJ 5), this scenario seems no longer possible.

### 3.2 Privileged aspects

A second class of problems is related to the possibility to declare aspects `privileged`. This enables aspect developers to access internals of the classes they are operating on, even the private parts of these classes. It is clear that, from a security perspective, there are considerable risks associated with this mechanism. We elaborate on several scenarios.

**Access to security-related information**

By accessing (private) internals of some modules in a system, the security properties of an application can be negatively influenced. A straightforward scenario is the alteration of a security module by some other aspect. Applied to the example of Section 2, a new aspect could be written that modifies the access control policy that is represented as a private member within the Authorization aspect. Similarly, other sensitive information such as private keys stored in a security engine could be accessed/modified by an aspect. It is clear that such modifications will most probably violate the intended security properties of the security module.

In a way, the above scenario seems to violate the encapsulation properties of Java: private class members can be accessed from other modules. In practice, the code generated by the weave will correctly adhere to the type safety properties. However, from the perspective of the affected application module, the encapsulation is broken since it can no longer guarantee that private internals will not be read or altered.

A somewhat more complicated, and tricky, scenario is the violation of a security policy by modifying a module that is not directly related to security. Suppose, for instance, that some security aspect enforces username/password authentication. Suppose also that this mechanism is implemented using a call-back handler implemented by a regular class – as is the case in typical authentication framework such as JAAS. When a third aspect would then implement logging by writing the details of all I/O events to file, clear pass-words passing through the call-back handler will be written to the log file, which will probably violate the security policy of the authentication module.

The previous scenario is related to the general problem of information flow. Information flow verifiers are based on the type-safetyness of Java. Since one cannot guarantee that private internals will not be read or altered by other aspects, it also becomes a lot harder to make information flow guarantees for an application. It is certain that AOP will influence the information flow guarantees one can make about a system, and without proper care, these guarantees can even be broken by means of AOP.\(^2\) In general it will be hard to predict whether privileged accesses to some module will have an impact on the security properties of a system.

As a sidenote, note that the use of wild-cards in aspects (i.e. in the definition of pointcuts) involves a certain risk as well, since it is hard to predict in which places the aspect will interfere.\(^3\) Given the invasive power of privileged aspects, this risk is further magnified in this context. Therefore, one should at all times try to avoid the use of wildcards in privileged aspects for the sake of security.

---

\(^1\)In essence, one must make sure to perform the infoflow verification step after the aspect weaving process. For compile-time weaving, this will normally not be a major issue in practice. However, it will be a lot harder to support using load-time or even run-time weaving.

\(^2\)Tool support is available to inspect the impact of a certain aspect. However, these tools only work at compile-time and, hence, are of little use in load-time weaving scenarios.
Is privileged necessary?

One could question whether support for privileged aspects is required at all. After all, the above discussions clearly demonstrate that the use of privileged aspects involves a considerable security risk, which seems contrary to the idea of using AOP to improve the security posture of a system in the first place. Unfortunately, there is no black-and-white answer. Separation of concerns for security is a goal worthwhile pursuing since it offers considerable advantages both from a software engineering perspective (specialisation and maintainability) as well as from a security perspective (easier to verify the implementation of the security policy). However, for complex security policies it will not always be possible to completely separate the security logic from other modules in a system when obeying the established rules of object-oriented software engineering (i.e. encapsulation). For instance, a security policy could dictate that the outcome of a rule is dependent on the internal state of some module. Or, a particular execution flow within a module could depend on a security rule. In these cases, discarding privileged aspects is no option: either we have to restrict the set of security policies that can be implemented, or we have to accept privileged aspects.

The authors’ experience is that the availability of privileged aspects is necessary in many cases, especially when the software, be it just some particular module of the software, is not designed to support the security policy. Therefore, it would be unwise to fully discard the support for privileged aspects. Rather, until better abstractions become available that provide adequate (and secure) support in these situations, we feel that privileged aspects should be offered, but at the same time that, in order to support the construction of secure software, the use of privileged aspects should be controlled. We further elaborate on this topic in Section 4.

3.3 Tool specific problems

The aspect compilers that are used to weave aspects into an application contain particular security problems as well. Experiments with the most recent compiler (AspectJ, compiler version 1.5.0) have brought forward two particularly dangerous transformations. A first case relates to the use of privileged aspects to access private class members. This is achieved by introducing into the target class a public method that executes the particular access: in case of a read the value of the member is returned, in case of write a new value is written into the private member. From a security perspective this is problematic: a program that is known to be secure without aspects, even when loading extra untrusted code, can become insecure after weaving in aspects because security-specific values become publicly readable and writable. The security model of the application can be broken in this way. An access control policy could, for instance, be modified by an untrusted class.

A second case relates to the use of inter-type member declarations. An aspect can introduce extra members and methods in classes and as such modify the types of these classes. Aspects can also declare these members/methods as private, and semantically this means that only the aspect can access these members/methods. When inspecting the compiler output, however, we have noticed that these members/methods are actually declared public in the target class. This means that security-sensitive members/methods cannot be safely introduced into classes.

In general, there is a clear requirement for a weaver to respect the safety and security properties of an application. Scenarios in which a secure application, once given to the aspect weaver, is turned into an insecure version should be avoided at all cost. Therefore, we feel that a thorough security-specific audit of the strategies used by a compiler is highly recommended.

3.4 Problem synthesis

Figure 1 shows the security risks that exist in AOP-based, or more specifically AspectJ-based systems. In summary, (i) the logic of a module (business as well as security) can be modified and (ii) the interaction or composition of modules can be influenced in different ways. A major problem for these risks is that security problems can be introduced intentionally (e.g., untrusted code) as well as unintentionally. While adequate measures can be implemented to shield against the former (e.g., do not allow the loading of untrusted code), it is a lot harder to address the latter. Every aspect developer could, by not knowing the exact overall impact of his aspects on the system, introduce security holes without even being aware of this. Furthermore, while the design of a system might be secure, the security properties are not necessarily correctly realized or maintained by the aspect weaver. It is clear that these problems seriously jeopardise the development of secure software using AOP altogether.

The origin of this unfortunate situation is to a large extent due to the level of control that aspects have over other modules in a system. In order to tackle this situation, support could be provided to impose extra restrictions on this control. However, as discussed before disabling these interactions altogether is no solution. Rather, selectively controlling these interactions seems the key to an adequate solution.

Controlling the invasive power of aspects can be realized in different ways. Several results have already been published that propose modifications or extensions of the aspect programming language, including [1, 8, 6, 5]. Most of these results offer support for modules to describe what parts can be accessed by other modules. An important issue in this context is that in order to correctly (and securely) enforce such mechanisms, modifying only the compiler (or
aspect weaver) does not suffice. After all, the extra code instrumentations are translated away at compile-time, unless adequate load and/or run-time enforcement mechanisms are installed as well (e.g., load-time verification, type-safe run-time abstractions). This is one of the major problems for current weaving-based AOP approaches: programming language abstractions are translated into standard object-oriented abstractions that no longer exhibit AOP-like characteristics.

An alternative approach to this problem is to make use of a run-time permission system to enforce these restrictions, which is similar to the standard Java permission system. This approach is attractive for several reasons. Firstly, an aspect permission system is a logical extension of the standard Java permission system to aspects. With an aspect permission system, aspects as well as classes could be checked for particular permissions. For instance, it would become possible to grant a MyFilePermission depending on the requestor being a particular aspect or not. Secondly, an aspect permission system could allow controlling dynamic actions that are specific to AOP such as the occurrence of a cflow, or dynamic advice activation (supported among others in CaesarJ [7]). Without an aspect permission system, it would be unsafe to impose control upon these actions based on the identity of the aspect. Thirdly, until better abstractions become available that are enforced securely, a permission system is an effective way to securely implement the restrictions discussed before. In fact, a permission system is more secure than a programming language solution that is only enforced at compile-time without load-time verification (as is the case for many of the solutions mentioned above).

In the rest of this paper we elaborate on the concrete characteristics of a permission system for aspects and on how this could be set up in current AOP tools.

4. TOWARDS AN ASPECT-SPECIFIC PERMISSION SYSTEM

The idea of an aspect permission system is quite simple. Based on the underlying permission system (in our case Java), extra permission checks can be inserted in the executing code such that dangerous actions, or interactions, that involve aspects are checked by the Java security system (which is known to be safe) against a particular aspect security policy. When the checks are inserted in the correct way, the result will be a more secure version of the same program.

While a permission system is a general purpose approach, the adversary model that is targeted is specific. A “regular” development environment is considered in which the software that is built is being deployed within the same company. An adversary cannot directly impact the developed code (all developers are trusted), although he could provide class/aspects libraries that are integrated within the software product. Furthermore, the adversary has no direct control over the development environment, e.g., to alter the software binaries. In some cases, he can however contact the software product remotely.

We elaborate on some open issues in the design of an aspect permission system in such context.

4.1 Policy types

A first issue is the types of policies one wants to enforce using aspect permissions. Is it necessary to support cross-cutting actions in the system, or can one focus on fundamental object-oriented events? Can we identify particular types of interactions that are more security-sensitive than others and, hence, must be focused on in particular? Three specific approaches seem worthwhile to consider.

1. A low-level approach is to focus on policies for restricting basic object-oriented interferences between aspects and other modules. For instance, an aspect can read or write class members and it can execute class methods. As discussed above, these interferences are particularly dangerous in the case of privileged aspects (otherwise, the standard Java type-system rules are adhered to) and, hence, it suffices to focus on these specific interferences.

2. A more higher-level approach is to base policies on security-sensitive language features that are specific to AOP. Examples of this include accessing an aspect’s pointcut, advising a particular joinpoint or declaring an inter-type member. Permissions in this case map very closely to programming language concepts, but are perhaps harder to link to run-time concepts.

3. The most general approach is to support arbitrary permissions rather than trying to focus on a particular set of actions. This approach is comparable to the Java permission system which enables developers to enforce arbitrary permissions.

All approaches have their own merits. The first approach supports less complex checks, but is easier to implement compared to the second and vice versa. The third approach is based on a different philosophy compared to the other two, but it can actually be used to implement the other approaches as well. Whichever approach is preferred, it is important to find the right balance between the amount of restrictions one can impose and the resulting performance of the system.

Given the nature of the most stringent security risks discussed in the previous section, the rest of this section will focus on the first type of policies (imposing restrictions on basic interferences). The following discussions should be reconsidered for implementing other types of policies.

4.2 Compiler support

The Java permission system is based on code invoking AccessController.checkPermission() in order to enforce the possession of a particular permission. The situation is different for aspects: particular statements get translated into object-oriented code and there is not always a way in which one could insert these invocations into the actual woven code. For instance, it would not be possible for a developer to insert a permission check for an inter-type member declaration. Furthermore, for those situations where explicit insertion is possible, it might violate the separation of concerns principle to expect developers to insert these statements themselves (and they will easily be forgotten as well!).

This is primarily the case for aspect to class interactions. It is also relevant, though, for aspect to aspect interactions.
To address this problem, the compiler could automatically instrument woven code such that permissions checks are inserted at relevant places. In particular, the compiler should add invocations to the AccessController for all dangerous interferences. For basic interferences it suffices to work with a single permission type that needs to be checked. We propose the use of an AspectPermission that matches for a particular target class (a string) and a particular action (read, write or execute). For more advanced interferences, it might be necessary to introduce different types of permissions. A permission hierarchy could be used for this purpose, similar to the Java permission hierarchy.

There is one major problem in this approach: in order for the run-time system to decide whether a particular aspect is granted the right to execute a dangerous action, the identity of the aspect must be known. After all, the Java permission system is fundamentally based on identities (classes, or subjects executing code). Unfortunately, during weaving some of the aspect logic is inserted into classes and the identity of these aspects is actually lost (at least from the perspective of the Java security subsystem). Code that in source code was part of an aspect can become part of another class and the security subsystem cannot distinguish between these two identities. In general, there are three different (run-time) scenarios in which a security-sensitive access attempt triggered from an (source-level) aspect can occur:

- the original class is modified by the aspect and accesses the sensitive logic
- a class that represents the aspect accesses the sensitive logic
- some other module that is affected by the same aspect accesses the sensitive logic

For the second case, the problem can be addressed in a quite straightforward manner: since the identity of the aspect is identical to the identity of a class, the Java permission system can be applied directly to control the access attempt. For the other cases, the standard Java permission system can be applied. A particular aspect weaver, has been discussed. A significant security problem is the level of control that aspects have over other modules in a system. Several examples have illustrated security problems in this context. In a second part of the paper, an aspect permission system has been proposed to address a number of these issues. Apart from discussing the general setup, a number of specific problems in this context have been elaborated on in more detail. A permission system could serve as a basis for a sandbox of untrusted aspects that operate with a very limited permission set, given that run-time loading of aspects is supported.

This paper has addressed only a part of the secure AOP puzzle. In order to get AOP ready to develop secure software, a number of steps have to be taken, including:

- define appropriate language extensions that support placing restrictions on security-sensitive parts of an application
- analyse the weaver to inventorise all weaving techniques and identify dangerous scenarios
- invent techniques to seal woven code
- integrate AOP in the Java security architecture, among others by supporting the permission system, the stack walk and bytecode verification for aspects.

5This assumes that an aspect can not be advised by another aspect—which is the case for AspectJ—, since this is required to guarantee that code in the aspect class actually originates from the original aspect.

6Experiments have shown that the AspectJ compiler actually does this, at least for our code in particular.
public class SensitiveData {

    // method generated to access the private secret datamember
    public static String ajc$privFieldGet$unsecure_SniffingAspect$mypackage_\SensitiveData$secret(SensitiveData sensitivedata) {
        // inserted permission check
        AccessController.checkPermission("SensitiveData.secret", "read");
        // original code
        return sensitivedata.secret;
    }

    <snip>
}

public class SniffingAspect {

    SniffingAspect() {
    }

    // method representing after advice
    public void ajc$after$unsecure_SniffingAspect$1$1718ef54(SensitiveData sd) {
        Authorization.aspectOf().ajc$before$security_Authorization$2$cf3a20ae();
        System.out.println(new StringBuilder("The new secret value is:")
            .append(SensitiveData.ajc$privFieldGet$unsecure_SniffingAspect$mypackage_\SensitiveData$secret(sd))
            .toString());
    }

    <snip>
}

Figure 2: A permission check inserted to secure private datamember access

6. REFERENCES


