Enterprise Security Aspects

Ron Bodkin

New Aspects of Software[*], 216 27th Street, San Francisco, CA 94131
rbodkin@newaspects.com

Abstract. This report surveys common security requirements for enterprise applications and analyzes how aspects can meet these. It analyzes how large-scale applications of aspects to security can work. It is intended to stimulate discussion by complementing research that looks at small-scale security aspects. The paper is to stimulate discussion about design patterns and how AOSD systems can provide better support for security. It examines requirements for effective application of aspects to security with a focus on the kinds of join points that are relevant. It considers common security requirements and discusses some possible aspect-oriented designs. It then evaluates the effectiveness of existing AOP systems support, especially in the area of pointcut definitions, and discusses areas for further investigation.

1 Introduction

Aspect-Oriented Programming (AOP) [KICZ96] in particular and the broader field of Aspect-Oriented Software Development (AOSD) in general offer great promise for improving information security. AOSD is relevant for all the major pillars of security: authentication, access control, integrity, non-repudiation as well as for the supporting administration and monitoring disciplines required for effective security.

There has been valuable pioneering research into applying AOP to security. The work of Shah ([SHAH02]) and Viega et al. ([VIEG01]) apply AOP to enforce secure coding practices. The work of De Win et al. ([DEWI01]) applies AspectJ to enforce access control. There have been similar uses of AOP to separate security policy from applications ([WELC03]). Previous work ([ANCO99, WELC00]) used used meta-programming techniques. Meta-programming provides tremendous power, but can be difficult to use; AOP was invented to provide more concrete and usable techniques to achieve similar ends.

There is also a rich literature on the topic of join points, pointcuts, expressiveness, and effective ways of writing pointcuts, etc. (e.g., [GYBE03, BODK03]).

This paper describes examples of crosscutting security rules that are frequently encountered. It then analyzes the relevant join points and properties thereof that can be used to describe those using current AOP systems. It then identifies areas for future work and summarizes the findings.

2 Security Scenarios

In this report we primarily consider how to implement authentication and access control. However, we want to start by identifying a broad list of common security use cases of enterprise IT applications, based on our experiences. Here is a small set of these

- Web user authentication
- Web service authentication
- Indirect database authentication
- Delegated authentication
- Role-based access control to application functions
- Role-based access control to data
- Instance-level access control to data
- Field-level access control to data
- Filtering operations available (or enabled)
- Filtering information displayed
- Encryption and decryption of data at rest
- Encryption and decryption of data in flight
- Functional access control to data
- Audit trail capture
- Data destruction

3 Authentication

Let now us consider the requirements for a common authentication use case: when a Web user requests a page with sensitive information. There are

Use Case: Web User Authentication extends Service Web User Request
Extension Point: User requests Sensitive Web page

1. System checks for cached credentials (e.g., in an HttpSession)
2. Resume request (authorizing access based on authenticated identity)
3. Return result to user

Alternate Flow:

1B. If user not already authenticated, system forces authentication by redirecting user to authentication page

a. User enters credentials and submits page
b. System validates credentials
c. System stores cached credentials
d. System redirects request to resume with original request

There are three important dimensions to consider in implementing this use case with aspects: identifying when a request is received, when to require authentication, and how to force authentication. We discuss each of these in turn.

Identifying when a request is received is fairly straightforward. Handling user requests maps well into method execution (or call) join points. Moreover, it is fairly natural to capture these using existing pointcut definition approaches. For a system built using Java Servlets or technologies that generate these (such as Java Server Pages), it is easy to write pointcuts that enumerate a small number of method execution join points for service requests (or to use pattern-based pointcuts). For systems built using higher-level frameworks that layer on top of Servlets, such as Jakarta Struts, it is typically preferable to capture requests through platform-specific join points (like method executions of Struts action execute methods). For Servlets and higher-level frameworks, the pointcuts corresponding to a request can be defined once and then reused by any application. Indeed, the ajee aspect library [AJEE03] contains reusable definitions for both Servlets and Struts.

Capturing the requests for a specific application is usually straightforward because well-written systems use different name spaces (e.g., packages in Java) to segregate code for different applications. This can be more problematic for reusable UI components, although our experience is that these tend to be aggregated by application-specific request handlers.

There are some practical challenges in weaving into request-handling components. Most popular frameworks use application-specific classes (often extending framework classes) so these can be woven into without altering shared infrastructure. However, systems like JSP that translate a higher-level language into Java require engineering effort to integrate aspects into request processing components.

A more interesting challenge is identifying when to force authentication. There are three basic strategies that vary according to how late (or eager) they are in forcing authentication:

- During request processing: require authentication only when processing a request that attempts to access a secured operation.
- After receiving request: require authentication upon starting processing of a request that may result in access secured operation.
- Before providing resources: require authentication upon starting processing of a request that will provide resources (links) to users that may result in secured operations.

We favor the third approach because it offers the best usability (only showing users available options) and security (by not providing information about capabilities for which a user is not entitled).[1]

A request *requires* authentication if it will perform any operations on data that might be restricted, or if it will provide links to the same. This is very similar to the better studied case of capturing join points that may result in modifying object states (often desired for transaction management). Proposals such those described in [GYBE03], [HUGU01], and [KICZ03] would allow directly writing a pointcut to capture the right join points. Similarly to transaction management aspects, with current AOP systems security implementers typically use application-specific properties to directly identify which request processing components that are secured. A desirable approach is to define secured requests in terms of other domain-level

[1] In the (rare) event that an operation is secured after processing a request that provided a link to a user, this approach will not have forced authentication for the newly secured operation. In practice, the security policy would likely force authentication or simply not authorize access for the user, which could be remedied by refreshing the page.

definitions (e.g., shopping or personalized requests). Where such a decomposition is mostly right then this approach is preferable. By mostly right, we mean that there may be exceptions to the rule and that it is better to define pointcuts in terms of domain concepts and then modify them for exceptions (e.g., using enumeration, marker interfaces, or attributes).

Many systems use *arranged patterns* [GYBE03] that require structuring the system in certain ways to allow classification. For example, it is common to place operations that access sensitive data in specific namespaces or packages (e.g., "restricted/request.do"). This is also a common practice for URL-based security systems like Java Servlets or Identity Management products (such as Netegrity SiteMinder or Tivoli AccessManager).

Alternative approaches include using naming patterns, enumeration, marker interfaces, or adding metadata through attributes (annotations or tags). Use of naming patterns in this context (e.g., using a prefix for any secured Servlet) seems disadvantageous: it is artificial and results in ugly and fragile code. Use of enumeration is also hard to maintain and fragile because of the large number of request handling components. Use of marker interfaces seems like a reasonable strategy although in many cases it is less natural than using package structures.

The use of annotations (also described as metadata or attributes) to identify secured requests is also promising (when support for attributes in AOP systems becomes the norm). As noted earlier, it is preferable to decompose requests that must be authenticated into domain concepts. This is naturally reflected by using annotations such as `@Shopping` or `@Personalized` instead of ones like `@RequiresAuthentication`. Including tags like the latter in the source code is problematic because it is highly desirable to separate security specifications from business logic; this author would favor explicit enumeration of locations to scattering and tangling security annotations throughout a code base. It's also worth noting that the proposed AspectJ extension for `declare annotation` [COLY03] that would allow adding tags to program elements in a crosscutting manner much as `declare parents` allows adding marker interfaces. This could be used to combine explicit tagging with rules that automatically add tags to items based on rules for deal with package structures, naming patterns, or even enumeration.

An additional challenge for identifying requests that need to be secured is the presence of external metadata in XML or other formats (e.g., deployment descriptors or identity management configuration). This information can be important for picking out what requests should be secured. However, we are not aware of any AOP technologies that allow writing pointcuts that reference such externally defined data (there are technologies that use XML to represent pointcuts, but this is a different problem).

There are also significant variations in the implementation of authentication beyond what we have discussed here. Two examples are support for single sign-on across systems and requiring stronger authentication for more sensitive information. However, we believe that these variations impose only a few new requirements for AOSD solutions and are well modularized by OO implementations.

There are several other important use cases for authentication, some of which we listed in section 2. One notable case is delegating requests on behalf of a principal when passing information between realms (e.g., inter-component calls, message calls, or database access requests). Using aspects is promising in scenarios like this. A client-side aspect allows capturing the identity of the caller and then extend the protocol to include the identity (either through using an extra handshake, using an envelop, or adding parameters). A coordinated server-side aspect can unpack the protocol extension and then set up the proper credentials in the recipient. However, adding parameters to method call is relatively difficult using current AOP systems (e.g., see [BODK03b] for a discussion of how to do this by adding a single operation to a stateless session EJB). Ideally, there would be a mechanism to coordinate information flow among distributed aspects.

4 Authorization

Authorization decisions may depend on a tremendous number of factors:

- The requesting user or other principal (by role or individual).
- The data being accessed (by class, instance, or field).
- The function being performed (by classes such as reading or writing, externalizing, business purpose such as treatment or marketing, or instance)
- The context (e.g., delegated requests, who the subject of the data is, what jurisdiction applies, whether the system is being attacked)
- Other rules (e.g., what contract was used or whether a user has opted-in or opted-out, the relationship between requester and subject).

The tremendous complexity of authorization decisions is one of the features of security decisions that makes AOP so valuable for security implementations. The information required for these decisions often crosscuts a large number of components and often changes independently of these. To explore how to apply AOSD to authorization we look at three common use cases:

- functional authorization
- data authorization
- providing access to user interface elements

4.1 Functional Authorization

Use Case: user accesses restricted function
Precondition: user is authenticated

1. System checks whether has adequate permission for operation
2. Operation proceeds

Alternate Flow:

1B. User does not have permission to perform operation:
2B. System throws exception and aborts processing

Functional authorization implement checks for whether an entire function can be accessed. The join points in question are typically method executions (or calls). It is common for authorization checks to be performed upon entry to a system layer (e.g., the user interface or business domain). As such, existing pointcuts typically capture these quite well. E.g., a pointcut can pick out execution of any public methods in a package or subpackage, or any such that is not in the control flow below another.

It is normal to throw security exceptions when violations are detected. It is easy to throw exceptions from before advice from almost any AOP system. Handling exceptions is also a natural application for aspects (e.g., see [LIPP99]); in our case aspects might redirect users to a common "not authorized" page. However, for requests that are not authorized to perform specific functions the issue can become more complicated. This topic is closely related to the discussion on disabling UI elements in 4.3 below (which includes cases of handling partially authorized functions).

There is another wrinkle for Java-based AOP systems. Java is the only widely programming language one with the concept of *checked* exceptions: any method that throws a `Throwable` that doesn't extend `Error` or `RuntimeException` must declare that it throws that exception type (or some supertype thereof). So in Java AOP systems, throwing checked exceptions from advice poses some challenges; either the developer needs to refactor code that is affected by the advice by scattering throws declarations throughout, or else they need to use the Exception Introduction pattern [LADD03] to wrap the checked exception and unpack it where used. Fortunately, Security exceptions are typically unchecked in Java (e.g., `java.lang.security.AccessControlException`), so this is not a common problem for implementing authorization checks with aspects.

Using AOP for functional access checks represents a significant improvement on traditional approaches of scattering access control checks or even scattering tags or redundant XML descriptions that obscure the underlying access control rules and harm maintainability. It is makes it significantly easier to maintain a role-based access control system, notably because it splitting or changing responsibilities can be localized instead of involving coordinating scattered edits.

4.2 Data Access Authorization

Use Case: user accesses restricted data element
Precondition: user is authenticated

1. System checks whether has adequate permission
2. Data access proceeds

Alternate Flow:

1B. User does not have permission to perform operation:
2B. System throws exception and aborts processing

Data Access authorization is typically more fine-grained than functional authorization. As such, it needs to be enforced wherever sensitive data is accessed. One common pattern is restricting access to classes of data depending on the user's role or their relationship to the subject of data. For example, any user with the role of a doctor might have access to any patient record, whereas other users with the role of patient can have access only to their own records. In real medical systems, additional considerations such as family members, power of attorney arrangements, and possibly restrictions on access by doctors for non-emergency uses further complicate the access control decision.

To control access to data, it is necessary to identify *who* is performing the access, *what* the data is, *how* it is being used, as well as *where* (the context). Let us consider each of these in turn.

Determining *who* is accessing data relies on authentication (as described above). When a user is authenticated, then their identity and credentials are typically exposed as part of a security context. Some security systems provide global access to security context (e.g., if using JAAS [LAI99] Permission objects for access control). In other cases, it is natural to set up an access control context at authentication points and to use control flow mechanisms to allow access to this context from access control points.

Rules that deal with *what* data is being accessed typically deal with data categories. For example, the Platform for Privacy Preferences (P3P [P3P]) defines standard categories including physical, health, demographic, and purchase. Defining access rules in terms of categories is analogous to granting functional access to users based on their roles.

Determining what data is being accessed is an interesting challenge. Most sensitive data is ultimately persisted either to databases or to other systems (e.g., through messaging interfaces). It is often natural to categorize data based on storage location (e.g., database columns or fields in messages). However, the logic for accessing and mapping these sources into Java code is usually opaque to OO programs. For example, if JDBC is used to read a database, identifying what fields are being accessed is complicated. Even when O/R mapping technologies are used (like JDO, Hibernate, or Container Managed Persistence entity beans), the mapping information is typically contained in XML files that are not easily parsed in a running program. Similar issues arise in dealing with messages such as JMS and low-level Web services APIs. In practice, with current AOP technologies, one tends to control data access after it is loaded into memory. However, this approach requires careful monitoring for security holes (e.g., back doors that read sensitive data through JDBC or other unexpected means). In addition, providing data security in an application raises the question of how to protect data from direct and other kinds of database access.

Control access to in-memory data involves mapping either entire types or specific fields (or accessor methods). Handling entire types is easier: it is possible to use marker interfaces (e.g., with `declare parents`), package structures or naming patterns (if natural), annotations (when and where available), or enumeration. A more challenging scenario is to categorize access to specific fields within an object (or collection). Sometimes the fields will themselves have a type that can be unambiguously categorized. But for fields of common data types (such as strings and numbers), the options available are naming patterns, enumeration, or annotations (if available).

There are also other cases where the content of data is important for determining access rules (e.g., for XML or other message structures). In these cases, the most reasonable approach is to have advice check content rules whenever a possibly matching structure is accessed.

Aspects that control access to data need to be triggered based on the operation being performed: how the data is being used. The most basic operations are reading or writing the data. For these cases, expressive AOP languages like AspectJ provide effective pointcuts to capture the appropriate kinds of operations. We can combine data categorization with either method calls (on objects) or field get and sets to pick out join points where a given access rule should execute. Naturally, there are more complicated cases; the earlier discussion on functional access control examined this dimension (albeit without considering specific data).

The access context is a catch-all concept that deals with a variety of factors such as what individual owns data, what jurisdiction applies, whether permission has been given, what policy is in effect, the subject about which data is being requested, the relationship between the principal and subject, explicit access control grants, and the purpose of the access. This context is often available using control-flow pointcuts although there are cases where additional state needs to be tracked (e.g., counters or system state).

For simple cases, it is common to use Java code to evaluate access rules. However, in more complex cases it is desirable to use a rules engine. There already exist sophisticated rules engines, both general purpose ones and ones specialized for policy rules. In any event, we believe evaluating rules is *not* a crosscutting concern. AOP systems can be used to enforce policy in a fine-grained matter while delegating policy decisions to such engines. The unique contribution of AOSD is to allow consistent policy enforcement that captures all the appropriate context information.

There *is* an interesting question of meta-level vs. base-level semantics. Some designs use AOP to provide very broad interception and delegate almost all access control decisions to an external engine, passing the engine a great deal of reflective data about the current join point. We believe that it is preferable to use pointcuts to pick out narrowly defined join points where specific rules might run, and to capture context that is then passed to a rules engine.

Note that if one categorizes data by pointcuts that enumerate or use type patterns, it will only be possible to distinguish categories in rules by having separate advice for each category. In contrast, marker interfaces or annotations (will) allow for reflective access to categories within a single advice. This argues for using annotations to categorize data (possibly using the proposed AspectJ extension for `declare annotation`).

There are other important topics to consider, notably tracking the flow of data in-memory and combining it. This allows restricting use of information copied into local variables or combinations of data (e.g., combining birth date, postal code, and age may be deemed personally identifying information). While basic access can be controlled at the “edge” (when first accessing the in-memory representation), this can be inadequate for more sophisticated constraints that are based on context of use or on the specific operations performed.

As we have seen from this discussion, there are a tremendous number of factors involved in providing data-driven access control. There are significant areas where new AOP constructs that support a more direct mapping of concepts would be very helpful. However, overall aspects still help tremendously. The good news is that they allow us to be much more clear, explicit, and consistent in enforcing policy and in defining data categories than was ever before possible.

4.3 User Interface Element Authorization

Use Case: user accesses page containing a restricted user interface resource
Precondition: user is authenticated

1. System checks whether has adequate permission
2. Resource is included in output page

Alternate Flow:

1B. User does not have permission to perform the associated operation (for links, buttons, etc.) or user does not have permission to see the contained data (for output of sensitive data elements)
2B. System omits resource from display

Our last authorization use case represents a twist on the previous two use cases. In the case of resources like buttons that access functions, we want to omit (or disable) these resources when the user does not have permission for the operation activating them would perform.

In a traditional GUI application, this would be fairly straightforward to enforce, e.g., after returning advice on method calls to show buttons disables or hides them if the user doesn't have permission to execute the command associated with the button. However, in a Web application the use of mark-up languages complicates resource identification. In addition, the mapping from request URL's into secured resources is a challenge.

The use of mark-up languages makes it difficult to identify resources because the output of a page is not naturally available in an object-oriented fashion. This problem is relatively easy to handle with language-specific aspects for applications that are developed using a specialized API for any secured resources (e.g., requiring custom JSP tags such as Struts' `<html:link>` tag). A more general-purpose approach is to post-process the output of rendering a page to parse any URL's, and to check those for security. Finding these in markup is straightforward (e.g., by using an XSLT stylesheet or a parser). However, if a page uses client-side scripting it is a generally unsolvable problem. In practice, this is rarely a problem and a bit of design to establish standards for writing script functions that link to other pages would make it easy to solve in practice. AOP support in client-side scripting languages would offer a more general-purpose solution.

Another challenge in identifying resources within markup is establishing resource boundaries. For example, if removing a hypertext link, should auxiliary graphics also be removed? Should the entire table cell containing it be removed? When should a row be removed? This problem does have an analog in the traditional GUI case, e.g., consider needing to hide auxiliary graphics or labels or a parent panel. This is a case where application-specific policies seem important (along with good modularity either by using tags or CSS styles to make it easy to identify resource units).

Once we have requested URLs, it is often a challenge to map these to secured resources. While it should be simple to design APIs that would perform these translations, in practice such predictive APIs are not available. In particular, there is no standard means of determining what Servlet would receive a request from a given

URL, nor of determining what Struts Action would execute from one.[2] A related problem is integrating AOP access control with coarse-grained external policy descriptions, such as declarative security in a Java Web Container, or Netegrity SiteMinder access rules. These policies are often defined in an external XML configuration file with poor APIs for testing whether access to a URL is allowed.

It would be highly desirable for such APIs to be designed (or extended) to support operations to look up permissions as described herein. It is also possible to implement such APIs externally by parsing configuration files. In practice, many projects will encode duplicate rules (perhaps by using generators that emit XML configuration and Java mapping code). It is interesting to note that good OO modularity would benefit from APIs such as this; presumably in practice non-AOP implementations have so rarely achieved the sophistication required that the APIs haven't been demanded.

Another important case when authorizing access to secured resources is displaying restricted data (fields) rather than a link to a restricted function. In this case, it is more natural to perform access inline while generating a page. Attempts to display unauthorized information will generate exceptions, which aspects can catch and then coordinate with the output mechanism (e.g., Servlet or JSP) to "edit out" the corresponding unit of output and then continue at the right level. Continuing after this kind of partial authorization failure can be challenging (e.g., the underlying logic for subsequent processing must be designed to not rely on work that might be aborted due to lack of access rights).

5 Other Rules

There are a number of other security operations that need to be performed beyond authentication and authorization. Some include encrypting or signing data, capturing audit trails, performing intrusion detection, or capturing obligations (such as deletion of data). In many respects, enforcing rules like these is similar to enforcing various authorization rules, including imposing a need to authenticate. However, it is more common for policies like encryption to require data tracking: sensitive data should be encrypted regardless of what interim data structures it is transferred into. This is a marked contrast from access control, which can often be performed once upon first reading data.

6 Conclusions

This paper has outlined several of the problems and opportunities in applying aspects to security, with an emphasis on applications to enterprise Web application written in Java. We have seen that security is indeed an area that benefits greatly from the use of aspects, in describing policies that were hitherto unavailable, in particular for fine-grained security rules.

[2] Naturally, this API would have to support URLs that are *external* to the current application.

In the future, we expect security aspects to benefit from more expressive pointcuts (e.g., predicting control flow and tracking data flow), the use of annotations that identify more domain concepts, more sophisticated exception handling support, to directly capture predictive rules for authentication, and increased integration of aspects with external libraries, distributed systems, and markup languages.

Using aspects for security already offers major benefits in compact code, increased confidence, and separation of roles between application development and security. As such, it is an application area that will continue to drive important and challenging requirements for AOSD.

7 Acknowledgements

Thanks to Todd Hay, Ben Mackenzie, Steve Schreiber, Jeff Branc, David Tompkins, and Zheng Li for their conceptual contributions to this work. Thanks to Bart De Win for reviewing a draft of this report. Thanks also to my former colleagues on the AspectJ team and my AspectMentor colleagues for their insight and support.

References

KICZ96: Kiczales, G., Lamping, J., Mendhekar, A., Maeda, C., Lopes, C., Loingtier, J., Irwin, J. *Aspect Oriented Programming.* In Proc. of ECOOP '97, LNCS 1241, pp. 220-243, Springer-Verlag, 1997

ANCO99: M. Ancona, W. Cazzola, and E. Fernandez, "Reflective Authorization Systems: Possibilities, Benefits, and Drawbacks." In Secure Internet Programming: Security Issues for Mobile and Distributed Objects, 1999

LAI99: Lai, C., et al., *User Authentication And Authorization In The Java(Tm) Platform*, In Proc. of the 15th Annual Computer Security Applications Conference, Phoenix, AZ, December 1999

LIPP9: Lippert, M., Lopes, C., A Study on Exception Detection and Handling Using Aspect-Oriented Programming, Xerox PARC Technical Report P9910229 CSL-99-1, Dec. 99

WELC00: I. Welch and R. Stroud, "Using Reflection as a Mechanism for Enforcing Security Policies in Mobile Code." In Proc. of the Sixth European Symposium on Research in Computer Security

DEWI01: B. De Win, B. Vanhaute, B. De Decker, "Security through Aspect-Oriented Programming", Advances in Network and Distributed Systems Security (B. De Decker, F. Piessens, J. Smits and E. Van Herreweghen, eds.), Kluwer Academic Publishers, 2001, pp. 125-138.

HUGU01: Hugunin, J., *The Next Steps For Aspect-Oriented Programming Languages*, http://www.isis.vanderbilt.edu/sdp. Software Design and Productivity (SDP) Coordinating Group, 2001

VIEG01: J. Viega, J. Bloch, P. Chandra, "Applying Aspect-Oriented Programming to Security", Cutter IT Journal, Vol. 14, No. 2, Feb. 2001

P3P: http://www.w3.org/P3P/

SHAH02: V. Shah, "Using Aspect-Oriented Programming to Address Security Concerns", International Symposium on Software Reliability Engineering, November 2002, Annapolis, MD.

AJEE03: The AJEE reusable library. Hosted within the aTrack project at https://atrack.dev.java.net/.

BODK03: Bodkin, R., Post on Writing Pointcuts, http://dev.eclipse.org/mhonarc/lists/aspectj-users/msg00802.html. AspectJ-Users mailing list.

BODK03b: Bodkin, R., Volkman, M. Email Discussion on Adding Parameter to Distributed Calls. http://dev.eclipse.org/mhonarc/lists/aspectj-users/msg00700.html and http://dev.eclipse.org/mhonarc/lists/aspectj-users/msg00716.html, AspectJ-Users mailing list.

COLY03: Colyer, A., Email on Declaring Metadata

GYBE03: Gybels, K., Brichau J.. *Arranging Language Features for More Robust Pattern-based Crosscuts.* In Proc. Of AOSD 2003, pp. 60-69, ACM Press, 2003.

KICZ03: Kiczales, G. *the fun has just begun*, Keynote from AOSD 2003, http://www.cs.ubc.ca/~gregor/kiczales-aosd-2003.ppt

LADD03: Laddad. R., *AspectJ in Action*, Manning Press, 2003

WELC03: I. Welch and R. Stroud, "Re-engineering Security as a Crosscutting Concern." Comput. J. 46(5): 578-589 (2003)