SECURITY THROUGH ASPECT-ORIENTED PROGRAMMING

Bart De Win, Bart Vanhaute and Bart De Decker

K.U.Leuven, Departement of Computer Science

Celestijnenlaan 200A, B-3001 Heverlee, Belgium

bart.dewin@cs.kuleuven.ac.be

Abstract Since many applications are too complex to be solved ad hoc, mechanisms are being developed to deal with different concerns separately. An interesting case of this separation is security. The implementation of security mechanisms often interacts or even interferes with the core functionality of the application. This results in tangled, unmanageable code with a higher risk of security bugs.

Aspect-oriented programming promises to tackle this problem by offering several abstractions that help to reason about and specify the concerns one at a time. In this paper we make use of this approach to introduce security into an application. By means of the example of access control, we investigate how well the state of the art in aspect-oriented programming can deal with the separation of security from an application. We also discuss the benefits and drawbacks of this approach, and how it relates to similar techniques.

Keywords: aspect-oriented programming, security, separation of concerns

1. Introduction

In the open world of the Internet it is very important to use secure applications, servers and operating systems in order to avoid losing valuable assets. According to different sources (e.g. CERT [cer, 2001]) updating and patching these systems to fix security holes is necessary frequently. The fact is that writing a *secure* application in an open, distributed environment is a far from straightforward task. There are several reasons why this is so hard to achieve.

First, securing an application is a very complex matter and requires a thorough understanding of what can go wrong and might be exploited. An average application programmer has not enough expertise in this area to know the exact requirements for his specific case. Moreover,

compared to the core functionality of an application, security is often considered as less important and as such it is only added afterwards. This results in overlooking some details which might introduce security holes. Furthermore, in the supposition that the security requirements are clearly understood and known in advance, it is necessary to implement them in the correct way. Several examples exist (e.g. the Netscape random generator bug [Demailly, 1996]) where a small weakness or bug in security related code could bring down the security of the whole application.

A second important reason why applications are difficult to secure is the structural difference between an application and the required security solution. Confidentiality for instance requires sealing and unsealing of sensitive information. Although they are logically joint and in fact very similar, they are typically separated in the implementation. Contrary to the application logic, security deals with principals that use particular services of the application and by doing so exchange sensitive information. Clearly, this information is scattered throughout the functionality of the application. This structural mismatch often leads to duplication of security code over different locations. Management of software is complicated considerably in this way, which unfortunately often introduces security problems.

To solve these problems, this paper uses aspect-oriented programming to implement security. The separation of concerns offered by this technique allows a programmer to only focus on the core functionality of his application. At the same time, a security engineer can analyze the security requirements and add these to the application without difficulty. As an extra advantage, the security requirement implementations can be reused for other applications when properly designed. Moreover, since the security related aspects are separated from the actual application, there is no reason to fear losing the overall security picture and as such forgetting important issues.

The structure of this paper is as follows. We will first give a short introduction to AspectJ, an aspect-oriented programming language for Java, and explain how this can be used to secure an application by means of a concrete example. This mechanism will be generalized in order to construct a framework of security aspects, after which the advantages/disadvantages of the approach will be discussed. We end this paper with a section on related work where we compare the aspect-oriented approach with other existing techniques.

2. Security as an aspect

For many problems, the modularity as offered by Object-Oriented programming is not powerful enough to clearly capture some important design decisions, which results in code that is scattered throughout the program. As a short introduction to aspect-oriented programming, we will briefly discuss one particular system, AspectJ [asp, 2001]. This is a Java language extension to support the separate definition of cross-cutting concerns. In AspectJ, *pointcuts* define specific points in the dynamic execution of a java program. Pointcut definitions are specified using primitive pointcuts such as the reception of a method call, the creation of a specific type of object, etc. Primitive pointcuts can be combined using logical operators.

On pointcuts, *advice* can then be defined in order to execute certain code. AspectJ supports before and after advice, depending on the time the code is executed. In addition, around advice enables the combination of the former and the latter into one advice. The use of the pointcut and advice constructs will become clearer when we discuss a concrete example.

The definition of pointcuts and the specification of advice on these pointcuts together form an aspect definition. Besides these special constructs¹, an aspect is similar to a class and can as such contain data members, methods, etc. To deploy the aspects in a concrete application, AspectJ provides a special compiler that parses all application and aspect code and transforms them into normal Java code, which is then compiled using a standard Java compiler.

The technique of aspect-oriented programming helps us considerably with the problems described in the introduction. On the one hand, it provides a mechanism to combine separate pieces of code easily, which encourages the separate implementation of non-functional issues like security. Using this divide and conquer strategy, the overall complexity of the problem is reduced considerably. Moreover, it allows different specialists (e.g. an application engineer, a security engineer, ...) to work simultaneously and to concentrate on their field only.

On the other hand, security concerns are often interwoven throughout the application. Aspect-oriented programming is particularly aimed at these crosscutting concerns. It enables interweaving these concerns into the application based on particular rules and automates as such a difficult task that is normally performed by the programmer manually. Hence, it eases the reflection about logically joint, but physically distinct parts.

2.1. An example: Access Control

The example presented in this section discusses how to perform access control in an application. We have chosen this problem because it very clearly shows that security related code can be separated from the functionality of the application in an elegant way.

Basically, access control can be described as follows: at a certain point, the application asks the user to authenticate himself, after which it can allow/deny access based on his identity. However, this abstract view hides several difficult details. The key to turn the above description into an aspect-oriented application is the identification of the important domain concepts and their mutual dependencies.

First, what is the exact entity that has to be authenticated? From a user-oriented view² the user of the global application might be a reasonable decision here. In this case the user has to login once, after which this identity is used during the rest of the application. However, the granularity of this approach will clearly not suffice for some applications, e.g. a multi-user or a multi-agent system. A second approach consists of linking the identity to a certain object in the application. Here, login information will be reused as long as the actions are initiated by the same object. However, the identity of the user might change over time. It is then necessary to associate the identity with the initiator of a certain action. In this case, an authentication procedure is required every time the specific action is initiated.

Next, for what resources do we want to enforce access control? Again, one can think of different scenario's. An identity might require access to one resource instance (e.g. a printer). When more instances are available, one could have access to the whole group or to only a particular subgroup. In case of different resource types the identity could require access to a specific combination of these resources. In general, this will often correspond to a combination of (some parts of) application objects.

A last but not less important consideration deals with specifying where the two previous concepts meet each other. This path from the authenticated entity to the resources is necessary to pass login information to the access control mechanism. In a distributed system for instance, authentication and access control might be performed on different hosts. In that case, authentication information must evidently be passed to the access control mechanism in order to ensure correct execution. One obvious example of such access path is the invocation of a specific service of a resource.

Each of the above concepts (identity, resource and access path) is actually a crosscutting entity to the application and maps closely to an

aspect. In fact, the three concepts capture the conceptual model of access control and they can as such be used for every access control problem. Note however that we did not discuss any issues concerning concrete mechanisms for authentication and access control. Although certainly relevant, it is important to realize that these are implementation decisions and they will as such depend on the underlying security architecture.

Figure 1 shows the details of one particular case of access control, where each object is authenticated once and where access is checked for each invocation of a particular service. The implementation of the other discussed access control mechanisms would be fairly similar. The observant reader will notice that the aspect code is written for a minimal application that consists of a *Server* implementing a *ServerInterface* with a method *service* and a *Client* invoking this *service*.

The **Identification** aspect is used to tag the entities that must be authenticated. In this case, every object of the class *Client* is considered as a possible candidate. Furthermore, the aspect implementation contains a field *Subject* that is used to store the identity information. As such, this information will be available as if it were glued to the particular *Client* object.

The authenticationcall pointcut of the **Authentication** aspect specifies all places where the service method of the *ServerInterface* is invoked. Through the use of cflowroot³ the Authentication aspect travels along with the invocation. Before the method is actually invoked, the identity information from the Identification aspect is copied to a local field of this aspect. As such, it is able to pass the authentication information to the access control mechanism. If the *Client* was not yet authenticated, this is the right place to do this.

Finally, the **Authorization** aspect checks access based on the identity information received through the Authentication aspect. This check is performed for every execution of the service method (*checkedMethods* pointcut). In this example, as you can see, the login and access control phase are written in pseudo code. The actual code will depend on the underlying security architecture as discussed before. In our implementation, we have used the Java Authentication and Authorization Service [Lai et al., 1999] for this purpose.

Weaving the above aspects into the application will result in a new, more secure version of the application. In the latter, the access controlling code defined in the Authorization aspect will be executed before every invocation of *service()*. At this point, the application will continue its normal execution if access is granted, however an exception will be

```

aspect Identification of eachobject(instanceof(Client)) {
    public Subject subject null ;
}

aspect Authentication of eachcflowroot(authenticationCall()) {
    private Subject subject;

    pointcut serviceRequest() : calls(ServerInterface, * service(..)) ;

    pointcut authenticationCall() :
        hasaspect (Identification) && serviceRequest() ;

    before (Object caller) : instanceof (caller) && authenticationCall () {
        final Identification id =IdentificationaspectOf(caller)
        if(id.subject == null) {
            <login> ;
            subject = id.subject ;
        }
    }

    public Subject getSubject () {
        return subject ;
    }
}

aspect Authorization {
    pointcut checkedMethods () : execution(* service(..)) ;

    before() returns Object : checkedMethods() {
        Authentication au = Authentication.aspectOf() ;
        Subject subject = au.getSubject() ;
        boolean allowed = <check access control> ;
        if (allowed){
            return proceed();
        }
        else{
            throw new Exception("Access denied") ;
        }
    }
}

```

Figure 1. Aspect code for object-based access control

thrown if the (un)authenticated entity is not allowed to do so. As such, conventional⁴ use of the method *service()* will be restricted to certain users depending on the security policy, just as would have been the case by coding the access control mechanism directly into the applition code.

2.2. Generalization of the example

The deployment of each of the crosscutting entities described in the previous section depends heavily on the actual type and implementation of the particular application. For example, an email client will work on behalf of one user, while a multi-user agenda system will want to distinguish his users. Also, objects representing a user will clearly differ in structure and behaviour between separate applications. In general, it is impossible to define one set of aspects that will be applicable to all possible applications. Therefore, a more generic mechanism is desirable that separates the implementation of security mechanisms from these choices.

Given the previous example you might notice that the deployment decisions are actually contained in the pointcut definitions, which define where and when an advice or an aspect has to be applied. For this purpose, AspectJ provides us with the possibility to declare pointcuts abstract and afterwards extend them to define the actual join points. Using this mechanism, it is possible to build a general authorization aspect and redefine the included abstract pointcuts depending on a specific application. To illustrate this technique, we have applied it to the experiment of the previous section. The result is shown in figure 2. In order to use these generic aspects in a concrete situation, one has to extend the abstract aspects and fill in the necessary pointcuts depending on the specific security requirements of the application.

A major advantage of this generalization phase is the ability to reuse the core structure of the security requirement. Since this will be similar for every situation, it is not necessary to reinvent the wheel for every case. It should be properly designed by a qualified person only once, after which aspect inheritance enables easy reuse.

2.3. Towards a framework of security aspects

For a secure distributed application, other security requirements besides authentication and authorization must be considered, such as confidentiality, non-repudiation etc. We will now briefly describe how they could be implemented using aspects.

Encryption of objects is required for *confidentiality* and *integrity*. This is a quite straightforward task using the Java JCA/JCE [Gong, 1998]. Two issues have to be considered. First, one has to decide where and how to insert this into the application. One possibility is to encrypt objects while they are written to a specific stream. For this case, the stream can be wrapped by a specific encryption stream. Another possibility is to encrypt objects whenever they are serialized. Therefore, the readOb-

```

abstract aspect Identification of eachobject(entities()) {
    abstract pointcut entities() ;

    public Subject subject null ;
}

abstract aspect Authentication of eachcflowroot(authenticationCall()) {
    private Subject subject;

    abstract pointcut serviceRequest() ;

    ...
}

abstract aspect Authorization {
    abstract pointcut checkedMethods() ;

    ...
}

```

Figure 2. Generalized aspect code for access control

ject() and writeObject() methods of the object should be overridden to include encryption here. Second, there is the issue of how to get or store the cryptographic keys. Similar to the identity in the previous section, one has to find some entity in the application with which the keys will be associated. The implementation will vary according to how the keys are to be acquired.

Non-repudiation requires the generation of proof for certain events in the system, e.g. the invocation of a specific method. This is quite similar to the problem of access control described above. One crosscutting entity defines the identity that wants to generate the proof. Another entity stores and manages these proofs. And finally, a third entity defines where and how proofs should be generated and passed along.

In the end, a combination of all the security aspects could form the basis of an aspect framework for application security. This framework will consist of generalized aspects for each of the security requirements. Note that several aspect implementations, depending on different underlying security mechanisms, may be included for the same security requirement. The deployment of the framework for a concrete application will then come down to choosing the necessary aspects and defining concrete pointcut designators for them. This technique actually suggests three distinct tasks to develop a secure application⁵ : build the application, develop a generic security aspect architecture and specify

the aspect deployment pointcuts. A more elaborate discussion on this security framework can be found in [Vanhaute et al., 2001].

3. Discussion

Being able to specify security concerns in a separate way, and still having them applied throughout the whole application is a noble goal. Aspect oriented programming techniques hold a promise of achieving this goal.

Although the technology has not yet fully matured, the current possibilities of AspectJ already give us a number of interesting advantages:

- Security should be applied at all times, if it is to be applied correctly. By looking at the definition of the pointcuts in the aspect that implements that particular security concern, a security engineer immediately knows all the places where this concern will be used, given that the AspectJ compiler does its job correctly.
- The implementation of the security mechanisms does not have to be copied several times. All the implementation code can be gathered within a small number of advices, perhaps all within one source file. As a result, when changes have to be made, the programmer can focus on that one part.
- Another result of the separate specification of the aspect code is that management of the different packages of an application is easier. There can be separate packages for pure application functionality, one for pure security code and a package that defines the points where the security code is to be applied.
- Although it could be argued that simple text substitution tools would also be able to insert code in a generic way into a program, the aspect oriented approach has much less chance of introducing bugs. The constructs aspect oriented transformers work on, are not mere text elements, but language constructs. These map more naturally onto the entities a security policy would speak about.
- By making the aspects more generic with respect to an application, we obtain a good combination of both application independent implementations of security concerns and the use of these implementations within the context of a specific application. The support for abstract pointcuts within AspectJ makes it possible to specify the two in separate files.

For our work, the use of the current version of AspectJ (0.7) has also some drawbacks (see further). On the one hand there are some technical

issues regarding the current implementation of AspectJ. It is expected that these will be solved in later releases of the tool. On the other hand there are problems that are more fundamental in nature. For this, at least a redesign of the aspect-oriented tool is needed.

- For each method call that has some security concern, the AspectJ compiler will insert one or more extra calls. Therefore, the generated code is less efficient, and introduces more overhead than a direct implementation would have. Unfortunately, this is the price to pay for the genericity of our approach. However, it is certainly not, worse than some other systems discussed in the next section. Building a more complex, but less general aspect combination tool could solve this.
- If not all code in the application can be trusted, one has to be very certain the generated code does not add any security holes. For instance in the case of authorization: it should not be possible for a client to call the end-functionality of a server through some other, by the aspect tool generated method, in order to circumvent the authorization checks. This means the security implementer has to have a very clear idea of what and how exactly the aspect tool produces. At this moment, the output of the AspectJ compiler cannot be trusted yet, because the original functionality, without the new aspect code, is only moved into a new method with a special name. However, this is only a problem if not all source code is under the control of the AspectJ compiler. The fact that AspectJ is not a formal proven language only increases this problem. This might become one of our topics for future work.
- Another issue related to the generated code comes up when debugging. As the runtime code does not directly correspond to the code the programmer wrote, it can be harder to figure out what is going on. The aspect-oriented research community is at the moment trying to build better support tools that would help the programmer in relating runtime events with the source code it stems from.
- The implementer of the security code still has to have very detailed knowledge of security mechanisms, their strong and weak points, how to implement them. As AspectJ is a generic tool, it does not help the programmer here, apart from providing a better modularization of the problem. However, this is not a particular problem of AspectJ, but rather of our approach to the problem.

4. Related work

There are already a large number of security architectures proposed or implemented in Java, e.g. [De Win et al., 2000]. Sun for one has for instance released JAAS [Lai et al., 1999] for authentication and authorization, SSE for secured network communication, and there are proposals for a secure RMI implementation. These will often already realize the intended result, and can therefore be used in the implementation of the security aspects. The combination of existing technologies with aspect-oriented programming is not expected to pose severe problems. The added value of aspects in this case is the possibility to have a much more flexible security policy, and this at a granularity that corresponds better with the application, i.e. at the level of method calls and objects. Some of the proposed architectures also have a fine granularity, but the configuration and mapping onto what happens inside an application can be fairly difficult.

By using a number of object-oriented design patterns [Gamma et al., 1994], the existing security architectures also try to be independent of an application structure, and they all succeed in this to some degree. The drawback of this design is that the structure of the solution becomes more complex and harder to understand. With an aspect-oriented approach these implementations can be designed in a more natural way.

Transformations in AspectJ happen on the level of source code. Other tools are available that work on the level of byte code [Cohen et al., 1998, Keller and Holzle, 1998]. This has the advantage that you can add your own aspects even when no source code is available for the application. The disadvantage is that on the level of byte code, a lot of the application logic is already lost. Reconstructing this is often hard, and giving correct descriptions of how a series of byte codes has to be changed to for instance implement authentication will be even harder. Checking and debugging the result will also be difficult.

There is also research into a more declarative description of security properties for an application [Evans and Twyman, 1999, Hagimont and Ismail, 1997]. This corresponds to an aspect-oriented language of the first type of section 2. The real challenge here is to think of the right abstractions the description will consist of. This is not at all an evident matter, certainly if a goal is to be generic. We think it is better to first experiment with a generic aspect-oriented language as described in this paper. From these experiments, we would hope to distill the important abstractions.

Meta level architectures [Chiba, 1995, Robben et al., 1999, Stroud and Wue, 1996] also make it possible to separate application from security

implementation [Ancona et al., 1999, Welch and Stroud, 2000]. They offer a complete reification of what is going on in the application: the events of sending a message, starting the execution, creating an object all get reified into a first class object. Because the meta-program has control over these reified entities, it can intervene in the execution of the base application. In comparison to aspect-oriented programming this mechanism is much more powerful, but it is also heavier. Moreover, the development of meta-programs for security is more complex, because the programmer is forced to think in terms of meta-elements, which is only indirectly related to the application.

Other approaches [Fraser et al., 1999] also use the basic idea of introducing an interceptor between clients and services, for instance to do extra access control checks. They are similar to meta-level architectures in that they also intervene in the communication between client and service, but the intervention is less generic (and heavy): the interceptors are mere decorators around the services. In simple situations, they can be specified fairly easy, perhaps through some declarative description. However, when more and more application state need to be taken into account, writing decorators becomes very hard, or even impossible due to the bounded possibilities of the declarative language.

5. Summary

This paper presented the use of aspect-oriented programming to add security to an application. By means of the example of access control, we first demonstrated the feasibility of this approach. In order to construct a more generic solution, we suggested to abstract relevant pointcuts out of the aspect implementation. This enabled us to separate the security mechanisms from the actual policy, which promotes the reuse of the mechanism implementations. After briefly discussing some other security requirements, we touched upon the feasibility to build a security aspect framework. Finally, we discussed the advantages and disadvantages of our approach.

The most important advantage of this approach is the separation of the application and the security related code. This considerably simplifies the job of the application programmer. Moreover, the security policies are gathered in one place, which makes it easier to check whether all the requirements are met. Still, we think that the deployment of these generalized aspects remains quite difficult. We would like to focus our research in the future on this issue, for example by automating the generation of concrete pointcuts based on a simplified high level description.

Notes

1. AspectJ also supports other constructs like Introduction. Since they are not used directly in this paper, we will not discuss them here.
2. From another point of view, application code might be the subject of authentication. While the mechanism to establish the correct identity of the code originator might be different, the overall authorization mechanism described in this paper will still be applicable.
3. cflowroot is a predefined keyword in AspectJ that denotes every control flow leading to that particular pointcut. Using this keyword, it is possible to 'follow' the invocation stack and pass as such information from the caller to the callee.
4. By predicting the output of the aspect weaver, one might be able to circumvent this access control mechanism under certain circumstances. We discuss this problem in detail in section 3.
5. A similar separation of these tasks has been described in [Robben et al., 1999].

References

- [asp, 2001] (2001). AspectJ Website. <http://www.aspectj.org/>.
- [cer, 2001] (2001). CERT Website. <http://www.cert.org/>.
- [Ancona et al., 1999] Ancona, M., Cazzola, W., and Fernandez, E. (1999). Reflective Authorization Systems: Possibilities, Benefits anti Drawbacks. In *Secure Internet Programming: Security Issues for Mobile and Distributed Objects*.
- [Chiba, 1995] Chiba, S. (1995). A MetaObject Protocol for C++. In *Proceedings of the 1995 Conference on Object-Oriented Programming*.
- [Cohen et al., 1998] Cohen, S., Chase, J., and Kaminsky, D. (1998). Automatic Program Transformation with JOIE. In *Proceedings of the 1998 USENIX Annual Technical Symposium*.
- [De Win et al., 2000] De Win, B., Van den Bergh, J., Matthijs, F., De Decker, B., and Joosen, W. (2000). A Security Architecture for Electronic Commerce Applications. In *Information Security for Global Information Infrastructures*, pages 491–500. IFIP TC11, Kluwer Academic Publishers.
- [Demailly, 1996] Demailly, L. (1996). Netscape Security (problems). <http://www.demailly.com/dl/netscapesec/>.
- [Evans and Twyman, 1999] Evans, D. and Twyman, A. (1999). Flexible Policy-Directed Code Safety. In *Proceedings of the 1999 IEEE Symposium on Security and Privacy*.
- [Fraser et al., 1999] Fraser, T., Badger, L., and Feldman, M. (1999). Hardening COTS Software with Generic Software Wrappers. In *Proceedings of the 1999 IEEE Symposium on Security and Privacy*.
- [Gamma et al., 1994] Gamma, E., Helm, R., Johnson, R., and Vlissides, J. (1994). *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison Wesley Longman.
- [Gong, 1998] Gong, L. (1998). Java Security Architecture. <http://java.sun.com/security>.
- [Hagimont and Ismail, 1997] Hagimont, D. and Ismail, L. (1997). A Protection Scheme for Mobile Agents on Java. In *Proceedings of the International Conference on Mobile Computing and Networking*.

- [Keller and Holzle, 1998] Keller, R. and Holzle, U. (1998). Binary Code Adaptation. In *Proceeding of the 1998 European Conference on Object-Oriented Programming*.
- [Lai et al., 1999] Lai, C., Gong, L., Nadalin, A., and Schemers, R. (1999). User Authentication and Authorization in the Java Platform. In *Proceedings of the 15th Annual Computer Security Applications Conference*.
- [Robben et al., 1999] Robben, B., Vanhaute, B., Joosen, W., and Verbaeten, P. (1999). Non-functional Policies. In Cointe, P., editor, *Meta-Level Architectures and Reflection*, volume 1616 of *Lecture Notes in Computer Science*, pages 74–92. Springer-Verlag.
- [Stroud and Wue, 1996] Stroud, R. and Wue, Z. (1996). Using Metaobject Protocols to Satisfy Non-functional Requirements. In *Advances in Object-Oriented Metalevel Architectures and Reflection*.
- [Vanhaute et al., 2001] Vanhaute, B., De Win, B., and De Decker, B. (2001). Building Frameworks in AspectJ. ECOOP2001 Workshop on Advanced Separation of Concerns.
- [Welch and Stroud, 2000] Welch, I. and Stroud, R. (2000). Using Reflection as a Mechanism for Enforcing Security Policies in Mobile Code. In *Proceedings of the Sixth European Symposium on Research in Computer Security*.