ZÁS - ASPECT-ORIENTED AUTHORIZATION SERVICES

Paulo Zenida, Manuel Menezes de Sequeira, Diogo Henriques, Carlos Serrão
ISCTE - Instituto Superior de Ciências do Trabalho e da Empresa
Av. das Forças Armadas, Lisboa, Portugal
Paulo.Zenida@iscte.pt, Manuel.Sequeira@iscte.pt, Diogo.Henriques@iscte.pt, Carlos.Serrao@iscte.pt

Keywords: JAAS, RBAC, authorization, Java, AspectJ, AOP, Zás

Abstract: This paper proposes Zás, a novel, flexible, and expressive authorization mechanism for Java. Zás has been inspired by Ramnivas Laddad's proposal to modularize Java Authentication and Authorization Services (JAAS) using an Aspect-Oriented Programming (AOP) approach. Zás' aims are to be simultaneously very expressive, reusable, and easy to use and configure. Zás allows authorization services to be non-invasively added to existing code. It also cohabits with a wide range of authentication mechanisms.
Zás uses Java 5 annotations to specify permission requirements to access controlled resources. These requirements may be changed directly during execution. They may also be calculated by client supplied permission classes before each access to the corresponding resource. These features, together with several mechanisms for permission propagation, expression of trust relationships, depth of access control, etc., make Zás, we believe, an interesting starting point for further research on the use of AOP for authorization.

1 INTRODUCTION

This paper proposes a novel, flexible, and expressive authorization mechanism. Its advantages stem mainly from the fact that an AOP approach is used, allowing it to address some of the problems found in industry standards like JAAS (Lai et al., 1999; Coté, 2006; Oaks, 2005). AspectJ (AspectJ Team, 2006) has been used to develop the proposal.

AOP is strong in terms of reduction of code scattering and tangling, provides for the separation of crosscutting concerns from the core code, and nicely integrates with the expressiveness of Java 5 annotations. This, together with our practical interest in authorization services for Java applications, led us to attempt to develop a new, aspect-oriented authorization mechanism, called Zás. Our aims were to make Zás simultaneously very expressive, allowing programmers to state very clearly what they mean, and independent from the business context, though being possible to make it business aware, allowing the separation of particular permission specifications from the authorization concerns embedded into the code by programmers. Another of our goals was to make the application of Zás to already existing code as simple as possible and, if necessary, totaly non-invasive.

Expressiveness requires code which is as clear and simple as possible. We do not want to tangle business code with code related to the checking of user permissions. We also do not want permission checking code scattered all over the application, in every method requiring verification of authorized access. In other words, we want to modularize, into aspects, the crosscutting concerns related to authorization. On the other hand, the programmer should be able, though not required, to guide the application of the authorization concerns to his own code. In this sense, one might say that one of the tenets of AOP, viz. obliviousness (Filman and Friedman, 2005), is violated. However, annotations may be though of as allowing the programmer to express in the code its required semantics, still oblivious of the exact way in which this semantics will actually be implemented.

According to (Clifton and Leavens, 2002), aspects can be divided into two categories: assistants and spectators. They suggest that assistance should be explicitly accepted by a module. Once a module accepts assistance of an aspect, then the aspect is allowed to advise that module. Annotations can be seen as a way to express assistance acceptance. Authorization annotations, as proposed by Zás, will thus acknowledge specific semantics for, say, a method, and implicitly

accept (or rather, require) assistance of a corresponding aspect implementing that semantics.

In (Recebli, 2005), Recebli proposes a different classification related to the role of aspects in software systems from a higher-level engineering point of view. He proposes the division into integral and attachable aspects. According to him, we can say that the aspects within our approach are attachable, since they can be removed from an application, without in any way changing the correctness of its business implementation (except, of course, in what concerns authorization).

The JAAS authentication service, based on PAM[1] (Samar and Lai, 1996), provides an abstraction layer that greatly simplifies changes in the actual authentication method used. However, since this work was motivated by the need to add authorization to the Heliópolis Web application,[2] which already possessed its own authentication code, our main focus was solely on authorization concerns. Nevertheless, the developed solution can be seamlessly integrated with a wide range of authentication mechanisms.

This work was inspired by Laddad's proposal (Laddad, 2003) to use AOP to modularize JAAS-based authentication and authorization. The main interest of his proposal, at least from our point of view, is related to the authorization concerns. Our work is thus based in Laddad's, extending further the modularization of authorization concerns, thus reducing code tangling and scattering, and reducing the configuration effort required from the programmers.

This paper is structured as follows. The next section will review some of the existing Java-based authorization mechanisms. It is followed by a detailed description of the requirements that have been used as guidelines to develop Zás. And finally, conclusions will be drawn and some possible directions for further work will be pointed to.

The Zás source code and related projects can be downloaded from https://svn.ci.iscte.pt/zenida, namely the Zás source code and the Web application used as a case study for this research. Further details about Zás, its implementation and the case study results can be found in (Zenida et al., 2006).

2 AUTHORIZATION SOLUTIONS IN JAVA

Authorization is not a new research topic. There are many different proposals and tools readily available, ranging from ad hoc solutions, where the developer implements everything from scratch, to complete solutions like JAAS, and from OOP approaches to approaches where the power of AOP is leveraged.

Our main interest while studying existing authentication and authorization mechanisms was JAAS, since it is a standard for authentication and authorization services in Java (Sun Microsystems, Inc., 2006) and an integral part of the JDK.

JAAS is not as flexible as we would like it to be. Its use requires considerable configuration effort (Oaks, 2005). For example, security policy files have to be used in order to specify the principals and what they are permitted to do. For example:

```
grant Principal
sample.principal.Principal "user" {
  permission test.Permission "perm";
};
```

Besides, and importantly, the permissions can not be changed at runtime. This is a serious restriction for dynamic applications, where an administrator must be able to add users and their corresponding permissions during the operation of the system. It is possible to use a database for this task, as in the example provided in (Coté, 2006), increasing the flexibility of the system by allowing the privileges of the principals to be specified at runtime. However, such a solution requires the use of a specific database model that, for already existing systems, may not be easy to accomplish.

The original JAAS model is implemented with an OO approach, thus being prone to the common problems of code scattering and tangling: code must be added to the business classes in order to implement authorization.

```
public class MyClass {
  public void businessMethod() {
    AccessController.checkPermission(
      new MyPermission("aPermission")
    );
    // business code
  }
  public static
  void main(String args[]) {
    // authentication code
    MyClass a = new MyClass();
    Subject authenticatedSubject =
      lc.getSubject();
    Subject.doAsPrivileged(
      authenticatedSubject,
      new PrivilegedAction() {
        public Object run() {
          a.businessMethod();
        }
      },
      null
    );
  }
}
```

Clearly, the authorization code is entangled with business code, both in the code requesting access to

[1] Pluggable Authentication Modules

[2] See http://heliopolis.iscte.pt/ (the source code uses Zás and is available in https://svn.ci.iscte.pt/Heliopolis/trunk/).

the resource (the caller code) and in the resource code itself (the callee code). Moreover, authorization code will be scattered through the application, since it must be used wherever access control is required. Both problems can be fixed using AOP, as shown in (Laddad, 2003). Laddad proposes an aspect-oriented approach to the application of JAAS that significantly simplifies the code required for access control, though only at the caller:

```
public class MyClass {
  // as before
  public static
  void main(String args[]) {
    MyClass a = new MyClass();
    a.businessMethod();
  }
}
```

We still need to call the `checkPermission()` method in the business methods. This can be avoided if we use the expressiveness of Java 5 annotations and modularize that call into an aspect responsible for authorization verification:

```
@AccessControlled(
  requires = "aPermission",
  permissionClass = MyPermission.class
)
public void businessMethod() {
  // business code
}
```

The use of annotations clearly improves the quality of the code, augmenting its expressiveness while reducing scattering and entanglement. However, by itself it does not decrease the required configuration effort nor makes access control dynamic. Zás, as will be seen in the next sections, does.

3 REQUIREMENTS

Originally, the implementation of authorization in Heliópolis was a clear case of a simplistic implementation of Yoder's "Limited View" pattern (Yoder and Barcalow, 1997), where appropriate menu options were hidden from non-authorized users. Our aim was thus to solve the authorization problem not by merely limiting the view of each user to whatever she is allowed to view or manipulate, but mainly by making sure, at the business layer itself, that no user can ever gain unpermitted access to any resource that is outside the privileges associated with his roles within the system.

For the reasons stated in the previous section, we were not satisfied with the available solutions to this problem. Zás, as described in this paper, was thus developed as a non-ad hoc solution to the authorization problem fulfilling the following broad requirements:

1. It should be independent and compatible with the simultaneous use of JAAS, especially with its authentication services.
2. Its authorization services should require no more from the application model than access to the current principal's permissions. It should thus support the RBAC (Ferraiolo et al., 2006; Sandhu et al., 1996) model, though never dealing directly with roles itself.
3. It should greatly simplify the code of client applications, as compared with alternative solutions.
4. It should be as non-invasive as possible, allowing business code to concentrate on the business logic, allowing the programmer to specify access requirements within the code, if she so desires, but also to completely separate access control from the business logic code.
5. It should require less configuration effort than the alternatives.
6. It should allow dynamic changes to the resources' access requirements.

Since Zás was meant to be a Java/AspectJ library of classes and aspects for use in Java applications, the requirements above were further refined into the following detailed requirements:

1. The access requirements for each such resource should be specifiable using Java 5 annotations.
2. The resources whose access should be controlled are represented by constructors, methods, and attributes.
3. It should be possible to force the propagation of the access requirements of a resource to all its members. For instance, from a package to all its types and nested packages, and from a class to all its (non-private) methods and attributes.
4. It should not be possible to propagate access control specification to resources explicitly marked as having no access control.
5. It should be possible to define access requirements either next to the corresponding resource definition (invasive usage), centralized in a single or in several access requirement definition aspects (non-invasive), or both.
6. It should be possible to use outside sources of permission requirements, such as property files. Permission requirements should be possible to change dynamically.
7. It should allow the definition of access requirements using boolean expressions involving permission names.
8. It should allow the quantification of the definition of access requirements using wildcards.

9. It should allow the specification of the required depth of access control as either deep or shallow.
10. When shallow access control is required, it should be possible to specify the degree of suspiciousness of a resource.
11. Special cases should be provided to bypass access control, viz. using privileged methods and trusted classes.
12. It should be easy to add authorization features to existing projects.

The next sections go through several of these requirements, exemplifying their impact in the client code, and thus clarifying the importance of the requirements themselves. Notice, however, that at the current state not all requirements have been implemented and some are only partially implemented. The status of development will be stated wherever appropriate.

3.1 ANNOTATIONS (1 AND 2)

Zás should allow the programmer to guide the application of aspects through the annotation of the non-private[3] resources where access control is required:

```
import pt.iscte.ci.zas.authorization.*;
public class MyClass {
  @AccessControlled(
    requires = "aPermission"
  )
  public void foo() {}
}
```

The previous code explicitly states that access to method `foo()`, that is, calling permission, is restricted to principals having permission `aPermission`. When not specified in the annotation, the access requirements correspond to a single permission whose name is the signature of the method without the return type.[4] Hence, the permission required to call `foo()` as defined in

```
package mypackage;
public class MyClass {
  @AccessControlled
  public void foo() {}
}
```

is `mypackage.MyClass.foo()`.

It should also be possible to annotate attributes, similarly to what happens for methods.[5]

Access requirements should always be filtered by method `getRequirements()` of the `Permission` class:

[3]Private "resources´´ are implementation details.

[4]Using complete signatures as permission names guarantees that overloaded resources are distinguished.

[5]The current version of Zás does not distinguish between sets and gets, as it should.

```
package pt.iscte.ci.zas.authorization;
public class Permission {
  public String getRequirements(
    String currentRequirements,
    JoinPoint joinPoint,
    JoinPoint.StaticPart
      enclosingStaticPart) {
    return currentRequirements;
  }
}
```

Before each access to a protected resource, this method shall be passed the current access requirements, which the default implementation will simply return, as well as the execution context of the access, including the caller and callee objects.

It should be possible to provide access control specifications with client classes extending `Permission` and overriding `getRequirements()`:

```
package mypackage;
public class MyClass {
  @AccessControlled(
    permissionClass = MyPermission.class
  )
  protected void foo() {}
}
```

Hence, arbitrary client code may be executed during access control, making it possible to add business specific access control methods to Zás.

3.2 PROPAGATION (3 AND 4)

Zás should provide a mechanism allowing access control specifications to be propagated to members of the corresponding resource, if any. For instance, in

```
@AccessControlled(
  requires = "aPermission",
  depth = Depth.SHALLOW
)
public class MyClass {
  public void foo() {}
  @AccessControlled
  public void bar() {}
  @NotAccessControlled
  public void baz() {}
}
```

`foo()` would inherit its access control specifications from class `MyClass`: the permission `aPermission` and the depth (see Section 3.6) verification as shallow. However, `bar()`, while access controlled, would not inherit required permissions from `MyClass`, and `baz()` would remain free of any access controls.

By default, non-private resources should not be access controlled, except when propagation is being used.

Notice that there should be two different effects in propagation. The first one is static, and leads to all non-private members of an access controlled resource, with the exception of those marked with annotation `@NotAccessControlled`, to also be access

controlled. The second one is dynamic, and leads to all non-private members of an access controlled resource *that have not been explicitly marked as being either access controlled or not access controlled* to dynamically inherit the required permissions from the enclosing resource (see Section 3.4).

The current version of Zás still does not provide the same mechanisms in the case of attributes. This problem will be solved in the near future.

Also, since the current version of AspectJ (AspectJ Team, 2006) does not allow the capture of package annotations, Zás still does not provide the inheritance mechanism for packages from the source code.[6]

3.3 LOCATION (5)

Usually resources requiring access control are directly annotated as such, i.e., their definition is directly annotated. This requires source code invasion and leads to scattering the meta-information related to access control concerns, which in some cases may be considered a bad practice.[7]

It is possible to use AspectJ ITDs[8] to inject annotations in types, methods, attributes, etc. Hence, it is possible to modularize all access control specifications in a single aspect, solving the problem of scattering meta-information.

Just as Java prohibits double annotations, AspectJ prohibits the injection of an annotation already present in the source code, next to the resource definition. Hence, the two approaches may be used together without any problem: the compiler will issue an error in case of a collision.

3.4 DYNAMIC PERMISSIONS (6)

Access control specifications should specify *initial* permissions, changeable at runtime. That is, permission requirements should be dynamic, while the access control character of resources should be static.

In conjunction with the ability to use wildcards (see Section 3.5) both to specify permissions and to specify the resources to which the permissions apply, this requirement makes it possible to dynamically load permissions specification from a generic input stream (connected to, e.g., access control property files), thus allowing permissions to be changed dynamically and easily by a system operator. For instance,

```
package mypackage;
public class MyClass {
  @AccessControlled
  public void foo(String s) {}
}
```

specifies that `foo` is access controlled and initially requires permission `mypackage.MyClass.foo(String)`. It should be possible to change the required permission using a properties file:

```
mypackage.MyClass.foo(String) = foo
```

In this case, after loading the properties file, the required permission for calling `foo()` is no longer `mypackage.MyClass.foo(String)`, but `foo`. Of course, the same effect should be obtained by directly calling a permission changing method of Zás:

```
import pt.iscte.ci.zas.authorization;
...
AccessController.addAccessControl(
  "mypackage.MyClass.foo(String)",
  "foo"
);
```

The use of external sources of permission requirements allows them to be provided at the appropriate granularity level. For example,

```
mypackage.MyClass.foo() = foo || bar
mypackage.MyClass.* = bar
mypackage.*() = foo
```

which might be found in an access control property file, states that all access controlled methods without any parameters within package `mypackage` will require permission `foo`, with the exception of those within class `MyClass`, which require permission `bar`. Again, method `MyClass.foo()` is an exception, since it requires either permission `foo` or permission `bar` (see Section 3.5). The order is relevant because Zás will always look for the first occurrence of a matching signature and load the corresponding permission specification.

3.5 EXPRESSIONS (7 AND 8)

It should be possible to compose Boolean permission expressions, both in-code as initial permission requirements, and dynamically (e.g., inside property files). For instance, in the access control specification

```
@AccessControlled(
  requires =
    "aPermission || !anotherPermission"
)
public void foo() {}
```

the permission expression requires any principal calling `foo()` either to have permission `aPermission` or to lack permission `anotherPermission`.

Currently, Zás supports operators `||` (“or”), `&&` (“and”), and `!` (“not”), as well as the use of parentheses to control evaluation order.

Regular expressions should also be possible in permissions expressions. For example, using

[6] However, we will soon open an AspectJ feature request.

[7] The authors consider this use of annotations to be advisable, however, since it leads to improved source expressiveness without hampering abstraction.

[8] Inter-Type Declarations

```
@AccessControlled(requires = "perm*")
public void foo() {}
```

any call to `foo()` would require a principal having at least one permission whose name starts with `perm` (e.g., `perm` or `permission`). Notice that regular expressions introduce a form of quantification into Zás. In this case they introduce existential quantifiers into permission requirements.

Wildcards should also be possible when dynamically specifying permission requirements, of course. In this case, however, they can also be used to specify multiple resources in a single step, as shown in the last example of Section 3.4. This introduces the notion of universal quantifiers into Zás.

3.6 DEPTH (9 AND 10)

By default, access control should be applied for all accesses to access controlled resources, regardless of the context. Regardless, namely, of the controls which have already been performed in upper levels of the current call stack. This is usually the safest option and thus the most desirable default. However, occasionally it may be necessary to turn off access control in the control flow of a given method execution.

The `@AccessControlled` annotation's element `depth` represents the level of access control. In a way reminiscent of copy depth, access control is applied to method execution either in a `Depth.SHALLOW` or in a `Depth.DEEP` manner. Shallow access control means that if access to a method is granted to a principal, it will also be granted to its complete flow of control, effectively turning off access control during its execution. On the contrary, if access to a method specifying deep access control, which is the safe default, is granted to a principal, it will *not* be automatically granted to all other accesses to resources in the method's control flow.

For example, in

```
public class A {
  @AccessControlled
  public void foo() {
    new B().bar();
  }
}
public class B {
  @AccessControlled
  public void bar() {}
}
```

a call to `foo()` will be possible only if the principal has both permission `A.foo()` and permission `B.bar()`. Changing the depth to `Depth.SHALLOW` in the annotation of `foo()`, access control will *not* be applied during the execution of `foo()`:

```
@AccessControlled(
  depth = Depth.SHALLOW
)
public void foo() {
  new B().bar();
}
```

Using shallow access control should generally be considered dangerous. Hence, a mechanism should be devised to short-circuit the consequences of shallow access control. If a given method declares itself to be suspicious, its access control specification should *not* be turned off in the flow of a shallowly access controlled method. For instance, in

```
public class A {
  @AccessControlled(
    depth = Depth.SHALLOW
  )
  public void foo() {
    new B().bar();
    new C().baz();
  }
}
public class B {
  @AccessControlled
  public void bar() {
    new C().baz();
  }
}
public class C {
  @AccessControlled(suspicious = true)
  public void baz() {}
}
```

a call to `foo()` will fail if the principal does not have permission `C.baz()`: it is not sufficient for him to have permission `A.foo()`, since `baz()` is suspicious. On the other hand, permission `B.bar()` is not necessary when the call is performed in the flow of control of `foo()`, since `bar()` is unsuspecting and `foo()`'s access control is shallow.

3.7 BYPASSES (11)

Zás should provide two methods to bypass access control. The first is more dangerous, and should be used with care: it should be possible to annotate some methods as privileged, i.e., as turning off access control to calls within their control flow. The difference between calling a privileged method and calling a method with shallow access control is that the first call always succeeds, while the success of the second one depends (solely) on the current principal having permission to make the call.

The second required bypassing mechanism, trust, is more disciplined and less dangerous. Instead of being used in a method to bypass access control during its entire execution, regardless of the access control specifications of the intervening resources, trust in specified classes is explicitly acknowledged by the callee method. For example, given

```
public class A {
  @AccessControlled(
    requires = "aPermission",
```

```
    trusts = { B.class }
  )
  public void foo() {}
}
public class B {
  @AccessControlled(
    requires = "anotherPermission"
  )
  public void bar() {
    new A().foo();
  }
}
```

any call to `bar()` will require a principal with permission `anotherPermission`, as usual, but the call to `foo()` from within `bar()` will not be subject to access control, since `foo()` declared its trust in class `B`. Notice, however, that calls from within the flow of control of `foo()` will in general be access controlled, since trust does not propagate. This will improve even further the safety of trust relationships.

3.8 EASE OF USE (12)

Zás should be easily integrated into an existing project. Indeed, if the requirements illustrated in the previous sections are fulfilled, particularly the ability to use ITDs to modularize access control specifications, little or no changes will be required in existing code.

Zás integration shall simply require

1. adding the `zas.jar` Java archive into the class path of the application;
2. defining a concrete aspect that extends the provided abstract aspect `AccessController`; and
3. adding the access control specifications either directly to the resources requiring authorization, or using ITDs concentrated in, e.g., the concrete aspect defined.

These steps are quite straightforward, with the possible exception of the definition of the concrete aspect. Access control is only possible if the current principal's set of permissions is available. However, Zás should be as independent as possible both of the authentication mechanism used in the application, and of the roles existing in the application and their corresponding permissions. How and where to find the permissions associated with (the roles of) the current principal is not Zás' problem. `AccessController` simply declares an abstract method `currentPrincipalPermissions()` which the concrete aspect, defined in the client code, should implement.

For example, the definition of the concrete aspect for a simple desktop application should be as simple as:

```
package pt.iscte.ci.myapp;
import pt.iscte.ci.zas.authorization.*;
public aspect MyController
  extends AccessController {
  private User user;
  public Set<String>
  currentPrincipalPermissions() {
    // get and return permissions
    // from the roles of "user".
  }
  before() :
    accessToControlledResources(
      AccessControlled
    ) {
    // if necessary, authenticate user.
  }
}
```

4 CONCLUSIONS

A new aspect-oriented authorization package, Zás, has been proposed which leverages AspectJ to make it possible to add authorization concerns to existing applications in a simple, non-invasive way. The model used is both independent of the authentication mechanism used, and of the specific way permissions are attached to principals. Hence, while supporting RBAC, Zás is not strictly speaking RBAC-based.

Even though in its early stages of development, Zás has shown the potential of aspect-oriented approaches to authorization concerns, making them simpler to implement, support, and configure. Zás is also dynamic, allowing runtime changes to the permission requirements associated with access controlled resources. The use of Zás, which builds on a previous proposal by Laddad (Laddad, 2003), greatly reduces the scattering of authorization code and its entanglement with business code. The use of Java 5 annotations led to a model where Zás' client code is not explicitly guiding advice introduction (Clifton and Leavens, 2002), but augmenting the expressiveness of the code by annotating it with authorization meta-information that is then taken into account by Zás' aspects. If this is deemed unacceptable, or if it is impossible in practice, then authorization concerns can be concentrated in a single module, thus freeing business code not only from authorization-related code *but also* from scattered meta-information.

Zás is, in certain cases, a good alternative to JAAS: it behaves much like JAAS, though with some important limitations. For instance, unlike JAAS, it can not be used to add access control concerns to resources inside JDK classes, since AspectJ does not allow ITDs to add annotations to code inside JDK's archives.[9]

[9] An interesting extension point for Zás would be the creation of an alternative to annotations to be used in such situations.

However, Zás' aim is not to replace JAAS, since Zás can be used together with JAAS-based authorization and, in a future version, Zás may even leverage JAAS authorization services.

Even though Zás is still in its infancy, we plan to revise and improve it regularly. Some possible next steps to the improvement of Zás are described next.

4.1 FURTHER WORK

In the near future we intend to improve Zás, especially taking into account the insight gained by its use in a large scale Java-based Web application. [10]

Nevertheless, some points requiring further research have already been identified. Should the basic concepts of authorization be extended such that each domain object is considered a principal, with its own set of permissions and its own set of trust relationships with other objects? What is the connection between trust and the composition, aggregation, and association relations? Should a distinction be somehow drawn between query and modifier methods, in the same way we need to distinguish sets and gets in the case of attributes? How do contracts relate to authorization and access control? What does this tell us regarding the relation between the runtime permission requirements of a method and the method it overrides? What if other crosscutting concerns of the application are implemented using aspects? How do we deal with potential conflicts that may arise (including the possibility of overriding authorization controls)? Nakajima and Tamai (Nakajima and Tamai,) proposed an analysis technique to assess the coherence between authorization policies and application code. The proposal, however, assumes the authorization policies are static. How could their analysis technique be applied in the case of dynamic policies, as allowed by Zás?

5 ACKNOWLEDGEMENTS

Special thanks to Professor Dulce Domingos for her important suggestions and for trying to make sure we would not miss the most important authorization references.

REFERENCES

AspectJ Team ([April 16th, 2006]). The AspectJ project at Eclipse.org. http://www.eclipse.org/aspectj/.

Clifton, C. and Leavens, G. T. (2002). Spectators and assistants: Enabling modular aspect-oriented reasoning. Technical Report 02-10, Iowa State University, Department of Computer Science.

Coté, M. ([April 16th, 2006]). JAAS book: Java authentication and authorization. Originally written for publication by Manning, http://www.jaasbook.com/.

Ferraiolo, D. F., Kuhn, D. R., Chandramouli, R., and Barkley, J. ([8th March, 2006]). Role Based Access Control (RBAC). http://csrc.nist.gov/rbac/.

Filman, R. E. and Friedman, D. P. (2005). Aspect-oriented programming is quantification and obliviousness. In *Aspect-Oriented Software Development*, chapter 2, pages 21–35. Addison-Wesley, Boston, Massachusetts.

Laddad, R. (2003). *AspectJ in Action*. Manning, Greenwich, Connecticut.

Lai, C., Gong, L., Koved, L., Nadalin, A., and Schemers, R. (1999). User authentication and authorization in the Java™ platfom. In *Proceedings of the 15th Annual Computer Security Applications Conference*, Phoenix, Arizona.

Nakajima, S. and Tamai, T. Formal specification and analysis of JAAS framework. In *Proceedings of the 2006 International Workshop on Software Engineering for Secure Systems*.

Oaks, S. (2005). *Java Security*. O'Reilly, 2nd edition.

Recebli, E. A. (2005). Pure aspects. Master's thesis, University of Oxford, Computing Laboratory.

Samar, V. and Lai, C. (1996). Making login services independent of authentication technologies. In *Proceedings of the SunSoft Developer's Conference*. http://java.sun.com/security/jaas/doc/pam.html.

Sandhu, R., Coyne, E. J., Feinstein, H. L., and Youman, C. E. (1996). Role-based access control models. *IEEE Computer*, 29(2):38–47.

Sun Microsystems, Inc. ([April 16th, 2006]). Java technology: Security and the Java platform. http://java.sun.com/security/.

Yoder, J. and Barcalow, J. (1997). Architectural patterns for enabling application security. In *PLoP'97, Proceedings of the 4th Conference on Patterns Language of Programming*.

Zenida, P., Menezes de Sequeira, M., Henriques, D., and Serrão, C. (2006). Zás - Aspect-Oriented Authorization Services (first take). Technical Report CI-2006-01, CI, ISCTE, Lisboa, Portugal. http://ci.iscte.pt/publicacoes/relatorios_tecnicos/CI-2006-01.pdf.

[10] Namely FénixEDU®. See `http://fenix-ashes.ist.utl.pt/FrontPage/`.