Zás – Aspect-Oriented Authorization Services (first take) – Final draft

Paulo Zenida[1] Manuel Menezes de Sequeira[2] Diogo Henriques[3] Carlos Serrão[4]

July 3, 2006

[1]Paulo.Zenida@iscte.pt, ISCTE – Instituto Superior de Ciências do Trabalho e da Empresa, Av. das Forças Armadas, 1649-026 Lisboa, Portugal
[2]Manuel.Sequeira@iscte.pt
[3]Diogo.Henriques@iscte.pt
[4]Carlos.Serrao@iscte.pt

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.

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

Chapter 1

Introduction

This paper proposes a novel, flexible, and expressive authorization mechanism. Its advantages stem mainly from the fact that an AOP[1] approach is used, allowing it to address some of the problems found in industry standards like JAAS[2] [9, 3, 12]. AspectJ [1] 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 [5], 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 [2], 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 [13], 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[3] [14], 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,[4] which already possessed its own authentication code, our main

[1] Aspect-Oriented Programming
[2] Java Authentication and Authorization Services
[3] Pluggable Authentication Modules
[4] See http://heliopolis.iscte.pt/ (the source code uses Zás and is available in https://svn.ci.iscte.pt/Heliopolis/trunk/).

focus was solely on authorization concerns. Nevertheless, we expect the developed solution to integrate seamlessly with a wide range of authentication mechanisms.

This work was inspired by Laddad's proposal [8] 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 have used as guidelines to develop Zás. Afterwards, Zás implementation is described with some detail. And finally, conclusions will be drawn and some possible directions for further work will be pointed to.

Chapter 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[1] 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 [18] and an integral part of the JDK[2].

JAAS is not as flexible as we would like it to be. Its use requires considerable configuration effort [12]. For example, security policy files have to be used in order to specify the principals[3] 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 in 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 [3], 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[4] approach, thus being prone to the common problems of code scattering and tangling: code must be added to the application 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();
```

[1] Object-Oriented Programming

[2] J2SE (Java 2 Platform Standard Edition) Development Kit

[3] "A name associated with a subject", taken as "any user of a computing service" [9] or "the identity assigned to an entity as a result of authentication" [16].

[4] Object-Oriented

```
                }
            },
            null
        );
    }
}
```

Clearly, the authorization code is entangled with business code, both in the code requesting access to 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 by [8]. Laddad proposes an AO[5] 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 (an interesting study on some of the benefits and problems of annotations can be seen in [7]) 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.

[5] Aspect-Oriented

Chapter 3

Requirements

Originally, the implementation of authorization in Heliópolis was very brittle, being limited to the interface layer, where appropriate menu options were hidden from non-authorized users. Anyone knowing, or guessing, the URL of a page containing private or protected information would be able to gain access. It was a clear case of a simplistic implementation of Yoder's "Limited View" pattern [19]. Our aim was thus to solve the authorization problem not by merely limiting the view of each user to whatever s/he 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 from JAAS.
2. It should be compatible with the simultaneous use of JAAS, especially with its authentication services.
3. 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[1] [4, 15] model, though never dealing directly with roles itself.
4. It should greatly simplify the code of client applications, as compared with alternative solutions.
5. 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 s/he so desires, but also to completely separate access control from the business logic code.
6. It should require less configuration effort than the alternatives.
7. It should allow dynamic changes to the resources' access requirements.
8. It should have a generic logging interface.

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.

[1] Role-Based Access Control

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[2] 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.[3] Hence, the permission required to call `foo()` as defined in

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

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

[3] Using complete signatures as permission names guarantees that overloaded resources.

is `mypackage.MyClass.foo()`.

It should also be possible to annotate attributes, similarly to what happens for methods.[4]:

```
package mypackage;
public class MyClass {
    @AccessControlled protected int bar;
}
```

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

```
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 permission 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()`:

```
import pt.iscte.ci.zas.authorization.*;
public class MyPermission extends Permission {
    @Override
    public String getRequirements(String currentRequirements,
    JoinPoint joinPoint, JoinPoint.StaticPart enclosingStaticPart) {
        ...
    }
}
```

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

Then, in the protected resource, we specify the permission class which must be used to compute the permission specification:

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

Our model saves internally the relationship between the instantiated permission classes and the protected resources being accessed. This improves the application efficiency, because we instantiate each permission class for each protected resource only once and reuse it when necessary, each time that resource is accessed. That is required because it is necessary to compute the permission specification each time a resource is executed.

The current version of Zás does not distinguish between sets and gets, as it should. It also does not support access control for constructors. However, those may be provided in a near future.

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, the access control specification of a class should be inherited by all its non-private members:

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

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

In this case, `foo()` should inherit the access control specification of class `MyClass`, i.e., calling `foo()` requires permission `aPermission` and the access control should be shallowly verified (see Section 3.6).

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). Hence, 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 name ”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.

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 [1] does not allow the capture of package annotations, Zás still does not provide the inheritance mechanism for packages from the source code.[5] Notice, however, that we have created a way to simulate this by using wildcards (see Section 3.5) in outside sources (see Section 3.4).

3.3 Specification locations (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.[6]

It is possible to use AspectJ ITDs[7] to inject annotations in types, methods, attributes, etc. Hence, it is possible to modularize all access control specifications in a single aspect:

```
public aspect AccessSpecifications {
    declare @method:
      void mypackage.MyClass.foo():
        @AccessControlled(requires = "foo");
    ...
}
```

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

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

[7] Inter-Type Declarations

This code should mark method `foo()` in `package.MyClass` as being access controlled and requiring permission `foo`.

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, one using annotations next to the access controlled resource, the other modularizing access control specifications in a single aspect, may be used together without any problem: the compiler will issue an error in case of a collision.

3.4 Dynamic permissions (6)

Permission requirements, as indicated in access control specification annotations are initial permissions, which should be changeable in runtime. That is, permission requirements should be dynamic, but not the access controlled resources.

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, allowing it to use, for example, access control property files:

```
import pt.iscte.ci.zas.authorization.*;
...
public InputStream outsideSource() {
    return ConcreteAC.class.getClassLoader().getResourceAsStream("permissions.properties");
}
```

, thus allowing permissions to be changed dynamically and easily by a system operator. For instance,

```
package mypackage;
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 such as:

```
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 permission specification. This simplifies both the program algorithm and reading the permissions specification.

Symbol	Meaning
*	0 or more characters
+	1 or more characters
?	0 or 1 characters

Table 3.1: Wildcards used in Zás notation.

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 [11, 17] should also be possible in permissions expressions. Table 3.1 shows the currently supported wildcards.

For example, using

```
@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 that is reminiscent of copy depth, access control is applied to method execution either in a `Depth.SHALLOW` or in a `Depth.DEEP` manner, depending on the value of this element. 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:

```
@Privileged
public void foo() {
}
```

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, because we have adopted the template advice idiom [6]. 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) {
        // code to authenticate the
        // user, if necessary.
```

```
    }
}
```

In the case of Heliópolis, authentication was already available. Therefore, no authentication related advice was necessary. However, it was still necessary to get the user trying to access a controlled resource:

```
public aspect MyController extends AccessController {
    private User user;

    public Set<String> currentPrincipalPermissions() {
        // get and return permissions
        // from the roles of "user".
    }

    private pointcut authorizationCallsScope() :
        execution(* *..BaseBean+.new(..)) ||
        execution(* *..BaseBean+.*(..));

    before(AccessControlled specifications, BaseBean baseBean) :
    accessToControlledResources(requirements) &&
    cflow(authorizationCallsScope()) &&
    this(baseBean) {
        // Code to get the current user
    }
}
```

Sub-aspects have precedence relative to the aspect they extended. Hence, the before advice in the code above is executed before Zás' access control related advice, thus making sure the current user is already stored in `user` when `currentPrincipalPermissions()` is called.

Chapter 4

Zás implementation

In our model implementation, there are some important ideas, which will be detailed in the current section. First of all, we have tried to create a reusable library of classes and aspects, which should be easy to use and configure in client projects. We have used AspectJ to implement Zás, whose implementations follow the template advice idiom [6], which entails creating an abstract aspect declaring reusable abstractions, and a concrete aspect tailored to a case-specific code base that defines the case-specific joinpoints to be captured in the logic declared by the abstract aspect.

The concrete aspect, as we showed in Section 3.8, must implement a method for gather the set of permissions associated to principals. We thus delegate all the knowledge related to authentication and how to gather the principals authorization sets to the concrete aspect. This way, we allow the abstract aspect `AccessController` to remain general and independent from the business logic, hence reusable.

Next, we will make a top-bottom list of all the relevant points in the Zás implementation, providing as necessary, an explanation of the code.

The `AccessController` constructor sets the debug mode to false so that no warning/error message is printed to the console (at the moment, we have not provided a generic logging interface, which may be supplied in a future version of Zás). Also, it puts a ”shutdown hook”, which is a thread that executes in the end of the program, just before it finishes. This mechanism is still under development and, therefore, it is not properly tested nor properly implemented. Nevertheless, the idea is for helping the developer about potential errors one might have done when specifying the permission requirements for the access controlled resources. For example, if we have a permission designated by ”foo” but we made a mistake when writing it, setting it as ”fop”, that would be hard to detect. That way, this thread should print, in the end of the program, the packages that have been loaded by the Java class loader and make a comparison with the namespaces provided for the permissions requirements that have been used.

```
public AccessController() {
    setIsDebugActive(false);
    // This makes it possible to preserve the insertion order.
    props = new LinkedHashMap<String,String>();
    permissionsPackages = new HashSet<String>();
    permissionsPerResource = new Hashtable<String,Permission>();
    Runtime.getRuntime().addShutdownHook(new ShutdownHook());
}

private class ShutdownHook extends Thread {
    public void run() {
        ...
    }
}
```

We have adopted the Java way for namespaces: when a class has no package, the default one is used. The same thing happens for permissions requirements:

```
@AccessControlled(requires = "foo")
public void foo() {
}
...
@AccessControlled(requires = "pt.iscte.ci.bar")
public void bar() {
}
```

In the previous example, we assume that the permission ”foo” belongs to the default package and that, for ”bar”, the package `pt.iscte.ci` exists in the project. In the end of the program's execution we would issue a warning if we had not detected the `pt.iscte.ci`. We can only issue a warning because it is not certain that the package does not exist. It simply means that one class in that package might have not been used, therefore, not having been loaded. So, with a warning message provided by Zás, it is possible for the developer to, at least, check for mistakes in the permissions specification. This mechanism should be enabled and disabled as wanted: in development time, it may be useful. However, in deployment time, it may add an undesirable overhead to the application.

One important requirement we have addressed in Zás implementation is related to the source or origin of the permissions requirements specification. We made it possible to load, in runtime, the specifications for each access controlled resource by reading those from a generic input stream, and load them into a map containing the authorizations specification for each joinpoint:

```
protected Map<String,String> getPermissionsFromOutside() throws IOException {
    throw new UnsupportedOperationException();
}
```

By default, it is set to simply throw an `UnsupportedOperationException`, meaning that the client aspect extending Zás needs to override the method, to enable the capability of reading from generic outside streams. An example of a concrete implementation could be (we have not used the *Properties* class from the JDK because it does not care about the order in the file, which is something we are interested in):

```
private java.io.Reader outsideSource() {
    return new java.io.InputStreamReader(ConcreteAccessController.class
    .getClassLoader().getResourceAsStream("permissions.properties"));
}

public Map<String, String> getPermissionsFromOutside() throws IOException {
    if (outsideSource() != null) {
        BufferedReader bufferedReader = new BufferedReader(outsideSource());
        String line;
        while ((line = bufferedReader.readLine()) != null) {
            int firstCommentChar = line.indexOf("#");
            if (firstCommentChar != -1) {
                line = line.substring(0, firstCommentChar);
            }
            if (line.length() != 0) {
                String[] tmp = line.split("=");
                if (tmp.length == 2)
                    addAccessControl(tmp[0].trim(), tmp[1].trim());
            }
        }
    }
    return getPermissionsRequirements();
}
```

Zás verifies the current principal permissions against the permissions specification supplied for a certain protected resource as follows:

```
private final boolean hasPermission(Set<String> authenticatedUserPrivileges,
String permissions) {
```

```
    if(authenticatedUserPrivileges == null)
        return false;
    for (String p : authenticatedUserPrivileges) {
        try {
            if(ExpressionValidator.validate(ExpressionParser.
            convertToBooleanExpression(p, permissions)))
                return true;
        } catch (MalformedBooleanException e) {
            if(isDebugModeActive())
                e.printStackTrace();
        }
    }
    return false;
}
```

The `convertToBooleanExpression()` method will turn a string expression into a boolean one, by matching each permission attached to the principal against the permission requirements provided for the resource under access control, setting true or false according to a successful or not matching. For example, matching ”foo” against the expression ”foo || bar” would result in ”true || false”. Next, the `validate()` method will evaluate the boolean expression result. For the example provided (”true || false”), it would return true.

Notice in the method, when catching the exception that we make a verification by calling the method `isDebugModeActive()`, which is simply a method returning true or false (by default is set to false) that can be overridden in the concrete aspect, in order to enable or disable error messages being shown, when an error occurs.

The requirements related to the propagation of permissions requirements through the members of an access controlled resource such as a class or a package, as we have mentioned previously, needs ITD. That is addressed with the following definition, where we put the `AccessControlled` annotation[1] in all non private methods of annotated types, not having neither a `AccessControlled` nor a `NotAccessControlled` annotations.

```
declare @method :
    !@AccessControlled !@NotAccessControlled !private *
    (@AccessControlled *..*).*(..) :
        @AccessControlled(inherited = true);
```

When we add authorization requirements to types resources like the previous, we set the *inherited()* attribute to true so that the resource will gain the requirements from the type where it is in. The programmer can also make a resource inherit the authorization requirements from the type without ITD by putting the *inherited()* attribute to true. However, if the type is not annotated as *AccessControlled*, Zás will throw a *RuntimeException*. The same happens for attributes. Nevertheless, we were not particularly interested in studying Zás for controlling attributes.

One of the main points of interest of the implementation is related to the quantification[5] mechanism provided by AspectJ, to capture the points of interest in the base code, so that aspects are then able to advice them. We capture all methods calls and sets/gets to attributes which are annotated with `@AccessControlled`. We have used annotations to restrict the applicability of aspect weaving, being sure we are not advising unwanted resources. The following defines that the controlled resources are the sets and gets of attributes, and also the calls to methods annotated with `AccessControlled`. We have chosen the call pointcut instead of execution, because it provides a wider scope, namely both the caller and the callee objects and because, in Access Control, the semantics are granting access or not to the principals’ orders or ”calls” to resources.

```
protected pointcut accessToControlledResources(AccessControlled requirements) :
    (accessToControlledMethods() ||
    accessToControlledFieldsSets() ||
    accessToControlledFieldsGets()) &&
    @annotation(requirements);
```

[1]The `inherited` element is set to true to enable the inheritance of all elements provided in the `AccessControlled` of the type.

The previous pointcut definition is `protected` so that only sub-aspects can ”see” it. Also, the pointcut has not been defined as final to make it possible for aspects extending `AccessController` to override it and, if necessary, further refine the access controlled resources. For example, one could restrict the scope of application of Zás by using the Border Control Design Pattern as defined in [10]:

```
public aspect BorderControl {
...
public pointcut accessControlScope() :
    within(pt.iscte.ci.myproject);
}

import pt.iscte.ci.zas.authorization.*;
public aspect MyAspect extends AccessController {
...
    protected pointcut accessToControlledResources(AccessControlled requirements) :
    AccessController.accessToControlledResources(requirements) &&
    BorderControl.accessControlScope();
}
```

The depth feature requires the definition of different pointcuts to quantify the top level access controlled resources, this is, the resources that have not been called or executed within the control flow of another protected resource, and those that are within the control flow of resources under Zás access control:

```
public final pointcut topLevelAccessToControlledResources(AccessControlled
requirements) :
    accessToControlledResources(requirements) &&
    !cflowbelow(accessToControlledResources(AccessControlled));

public final pointcut nonTopLevelAccessToControlledResources(AccessControlled
requirements, AccessControlled topLevelRequirements) :
    accessToControlledResources(requirements) &&
    cflowbelow(topLevelAccessToControlledResources(topLevelRequirements));
```

Both pointcuts are defined as being final so that they can not be overridden. This way, we prevent the definition for the top level and non top level accesses to be misdefined.

The following method is one of the most important ones in Zás. It computes the permissions or authorizations requirements for a certain `joinPoint`, in a certain `enclosingStaticPart` context, according to the `requirement` annotation and a map of `props` which have been loaded (or not, in the case that the feature has not been implemented in the concrete aspect) from an external source[2]:

```
private synchronized String getPermissionsRequirements(AccessControlled
  requirement, JoinPoint joinPoint, JoinPoint.StaticPart enclosingStaticPart,
  Map<String,String> props) {
    Permission permission = null;
    try {
        if(permissionsPerResource.containsKey(joinPoint.toLongString()))
            permission = permissionsPerResource.get(joinPoint.toLongString());
        else {
            Class permissionClass = requirement.permissionClass();
            permission = (Permission)permissionClass.newInstance();
            permissionsPerResource.put(joinPoint.toLongString(), permission);
        }
        if(requirement.inherited())
            return permission.getRequirements(getPermissionsNamesForInheritedRequires(
            requirement, joinPoint, props), joinPoint, enclosingStaticPart);
        else
            return permission.getRequirements(getPermissionsNamesForNotInheritedRequires(
```

[2]If no external source is specified, props will be null.

```
            requirement, joinPoint, props), joinPoint, enclosingStaticPart);
    } catch(IllegalAccessException e) {
        if(getIsDebugActive())
            System.err.println(e);
    } catch(InstantiationException e) {
        if(getIsDebugActive())
            System.err.println(e);
    }
    // returns null when something wrong
    // occurs. This should never happen
    return null;
}
```

As we have mentioned previously in this paper, we have created a mechanism to store the permission classes used for access controlled resources, so that we can make Zás faster and also to make it possible to add state to the *Permission* classes used. With such a mechanism, we can also add more possibilities to the permission classes, such as make it possible to compute the number of times a certain permission has been used to calculate the access control requirements. The first section of code in the previous method:

```
...
if(permissionsPerResource.containsKey(joinPoint.toLongString()))
    permission = permissionsPerResource.get(joinPoint.toLongString());
else {
    Class permissionClass = requirement.permissionClass();
    permission = (Permission)permissionClass.newInstance();
    permissionsPerResource.put(joinPoint.toLongString(), permission);
}
...
```

is used to search for the stored permissions during the Java project execution in which Zás has been integrated, and reuse the permission if the protected resource has been previously executed. If, however, the protected resource has not been executed so far, the permission class specified for that resource will be instantiated and stored in an internal map of permission classes for join points.

The second part of the method will ”redirect” the computation of the permissions requirements to an appropriate method, according to the `inherited()` attribute of the `AccessControlled` annotation:

```
...
if(requirement.inherited())
    return permission.getRequirements(getPermissionsNamesForInheritedRequires(
    requirement, joinPoint, props), joinPoint, enclosingStaticPart);
else
    return permission.getRequirements(getPermissionsNamesForNotInheritedRequires(
    requirement, joinPoint, props), joinPoint, enclosingStaticPart);
...
```

Both methods `getPermissionsNamesForInheritedRequires()` and `getPermissionsNamesForNotInherited-Requires()` share a similar logic, with particular differences related to the annotation’s `inherited()` element. To simplify this point, we have decided to provide only the description for the most complex of the two (the first one). They are equal except for the logic related to checking for the permissions specification in the types where the resource is contained:

1. Look for permission requirements specified in external sources
   (a) Look for a property matching the resource signature and, if found, return it
   (b) Look for a property matching the type signature where the resource is in and, if found, return it
2. Look for permission requirements specified in the code

(a) Look for the resource permission specification and, if any has been provided (one different than #), return it

(b) Look for the type containing the resource specification and, if there is a permission specification provided (one different than #), return it

3. If all the other steps have failed, simply return the protected resource signature as the permission requirements specification

As we have explained in Section 3.8, concrete aspects need to provide a way for Zás to get the permissions specification. This is supplied by the implementation of a method returning a set of names of permissions or authorizations for an authenticated user, whose implementation is mandatory:

```
public abstract java.util.Set<String> currentPrincipalPermissions();
```

After capturing the important points in the code, it is necessary to check, *before* the protected resources are executed, if the user who is trying to access that resource has access to it. This is designated as advice and next we can see the piece of advice necessary to check for authorization. To capture the resources' annotations elements, we need reflection. That involves searching, in runtime, for the details of a certain object, thus causing a penalty in terms of performance. This subject, however, needs further research.

The first advice requires a more complex logic because it must check if the verification should be executed deeply or shallowly, while the second piece of advice is only for top level resources, thus being much simpler:

```
before(AccessControlled requireHighLevelAuthorization, AccessControlled
requireLocalAuthorization) :
    nonTopLevelAccessToControlledResources(requireLocalAuthorization,
    requireHighLevelAuthorization) &&
    !cflow(privilegedOps(Privileged)) {
        JoinPoint joinPoint = thisJoinPoint;
        JoinPoint.StaticPart enclosingStaticPart = thisEnclosingJoinPointStaticPart;
        if(shouldBeShallowlyVerified(requireLocalAuthorization,
        requireHighLevelAuthorization, joinPoint)) {
            if(requireLocalAuthorization.suspicious()) {
                checkAuthorization(requireHighLevelAuthorization, joinPoint,
                enclosingStaticPart);
            } else {
                Signature enclosingPoint = enclosingStaticPart.getSignature();
                if(joinPoint.getKind().toString().equals(METHOD_CALL)) {
                    for(Method m : enclosingPoint.getDeclaringType().getDeclaredMethods()) {
                        if(m.toString().equals(enclosingPoint.toLongString())) {
                            AccessControlled requireEnclosingPoint = m.getAnnotation(
                            AccessControlled.class);
                            if(requireEnclosingPoint != null)
                                checkAuthorization(requireEnclosingPoint, joinPoint,
                                enclosingStaticPart);
                            else
                                checkAuthorization(requireHighLevelAuthorization,
                                joinPoint, enclosingStaticPart);
                        }
                    break;
                }
            } else { // it is for fields
                for(Field f : enclosingPoint.getDeclaringType().getDeclaredFields()) {
                    if(f.toString().equals(enclosingPoint.toLongString())) {
                        AccessControlled requireEnclosingPoint = f.getAnnotation(
                        AccessControlled.class);
                        if(requireEnclosingPoint != null)
                            checkAuthorization(requireEnclosingPoint, joinPoint,
                            enclosingStaticPart);
                        else
```

```
                            checkAuthorization(requireHighLevelAuthorization,
                            joinPoint, enclosingStaticPart);
                    }
                    break;
                }
            }
        }
    } else
    checkAuthorization(requireLocalAuthorization, joinPoint, enclosingStaticPart);
}

before(AccessControlled requirements) :
    topLevelAccessToControlledResources(requirements) &&
    !cflow(privilegedOps(Privileged)) {
        checkAuthorization(requirements, thisJoinPoint, thisEnclosingJoinPointStaticPart);
}
```

The first piece of advice first verifies if the access controlled is shallowly verified. The method `shouldBeShallowlyVerified()` returns true if the permission requirements for the protected resource have been specified as being shallow, either by inheriting that specification from the type where the resource is defined or in the resource permissions requirements definition itself, or if the *this* object that is accessing the resource is an instance of a class that belongs to the set of trusted classes for that protected resource:

```
private synchronized boolean shouldBeShallowlyVerified(AccessControlled localRequirement,
AccessControlled highLevelRequirement, JoinPoint joinPoint) {
    if(localRequirement.inherited()) {
        AccessControlled classRequirement = getClassRequiresAnnotation(joinPoint);
        checkIfRequiresAnnotationIsNull(classRequirement);
        if(isLevelShallow(classRequirement.depth()))
            return true;
        if(joinPoint.getThis() != null) {
            if(isFriendClass(discardReturnTypeFromSignature(joinPoint.getThis().
            getClass().toString()), classRequirement.trusts()))
                return true;
        }
    } else {
        if(isLevelShallow(highLevelRequirement.depth())) {
            return true;
        }
        if(joinPoint.getThis() != null) {
            if(isFriendClass(discardReturnTypeFromSignature(joinPoint.getThis().
            getClass().toString()), localRequirement.trusts()))
                return true;
        }
    }
    return false;
}
```

If the verification for shallow or deep results in shallow access control, it needs to verify if the protected resource is ”suspicious” or not:

```
if(requireLocalAuthorization.suspicious()) {
...
} else {
...
```

If the protected resource ”A” is suspicious, then Zás will verify if the principal has access to the first protected resource ”B” in the control flow of this access to the protected resource ”A”. Otherwise, it will behave as if it was a deep access control, always checking for the ”closest” protected resource calling ”A”.

If the protected resource needs deep access control, Zás will always check the principal's permissions against each access controlled resource.

It should not be possible to specify a method or attribute as needing and not needing authorization requirements at the same time. Therefore, a compile time error must occur when that happens:

```
declare error:
    (execution(@AccessControlled @NotAccessControlled * *..*.*(..)) ||
     set(@AccessControlled @NotAccessControlled * *..*) ||
     get(@AccessControlled @NotAccessControlled * *..*)) :
        "You cannot specify the same point as requiring and not
        requiring authorization";
```

We have also created an aspect to enforce best practices and policies for Zás. In order for client code to enable it, it simply needs to extend the aspect `PolicyEnforcer`.

This aspect detects private attributes (sets and gets) and private methods annotated as being access controlled (as we said before, those are implementation details, and therefore, should not be access controlled):

```
public pointcut declareWarningScope() :
    pt.zenida.paulo.thesis.common.pointcuts.CommonPointcuts.all();

public final pointcut declareErrorScope() :
    !declareWarningScope();

declare warning :
    declareWarningScope() &&
    (execution(@AccessControlled private * *..*.*(..)) ||
    set(@AccessControlled private * *..*) ||
    get(@AccessControlled private * *..*)) :
        "Private methods and fields should not be annotated as points
         requiring authorization";

declare error :
    declareErrorScope() &&
    (execution(@AccessControlled private * *..*.*(..)) ||
    set(@AccessControlled private * *..*) ||
    get(@AccessControlled private * *..*)) :
        "Private methods and fields should not be annotated as points
         requiring authorization";
```

This aspect has a particular capability, related to the possibility of choosing between compile time warnings and errors, by simply setting the `declareWarningScope()` pointcut. Notice the `declareErrorScope()` is final and defined as being the opposite of the warning declaration. This way, all messages not specified as being shown as warnings will be shown as errors. By default, it uses a commonly used pointcut from a pointcuts library we have created:

```
public final pointcut all() : !none();

public final pointcut none();
```

The previous definitions are quite powerful, because `none()` will not capture anything. However, `all()` will be defined as being its opposite, thus capturing everything else.

However, we could set scopes (packages and classes) where we would like warning messages to be shown, and others where we would prefer errors. For example:

```
public pointcut declareWarningScope() : within(pt.iscte.ci.foo+);
```

The previous would make the compiler show warning messages for each private attribute or method annotated as being access controlled in any class in the package `pt.iscte.ci.foo` and all its subpackages. Every other private resources in different package classes would cause the compile to show error messages.

Chapter 5

Conclusions

A new AO 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 AO 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 [8], 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 [2], 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.[1] 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.

The Zás source code and related projects can be downloaded from https://svn.ci.iscte.pt/zenida. 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.

5.1 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. [2]

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

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

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

and access control? What does this tell us regarding the relation between the runtime permission requirements of a method and the method it overrides?

Chapter 6

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.

Bibliography

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

[2] Curtis Clifton and Gary T. Leavens. Spectators and assistants: Enabling modular aspect-oriented reasoning. Technical Report 02-10, Iowa State University, Department of Computer Science, October 2002.

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

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

[5] Robert E. Filman and Daniel P. Friedman. Aspect-oriented programming is quantification and obliviousness. In *Aspect-Oriented Software Development*, chapter 2, pages 21–35. Addison-Wesley, Boston, Massachusetts, 2005.

[6] Stefan Hanenberg, Rainer Unland, and Arno Schmidmeier. AspectJ idioms for aspect-oriented software construction. In *Proceedings of the 8th European Conference on Pattern Languages of Programs (EuroPLoP'03)*, Irsee, Germany, June 2003.

[7] Wilke Havinga, Istvan Nagy, Lodewijk Bergmans, and Mehmet Aksit. Detecting and resolving ambiguities caused by inter-dependent introductions. In *Proceedings of the 5th International Conference on Aspect-Oriented Software Development*, Bonn, Germany, March 2006.

[8] Ramnivas Laddad. *AspectJ in Action*. Manning, Greenwich, Connecticut, 2003.

[9] Charlie Lai, Li Gong, Larry Koved, Anthony Nadalin, and Roland Schemers. User authentication and authorization in the Java^{TM}platfom. In *Proceedings of the 15th Annual Computer Security Applications Conference*, Phoenix, Arizona, December 1999.

[10] Russell Miles. *AspectJ Cookbook*. O'Reilly, Boston, Massachusetts, 1st edition, January 2005.

[11] Dana Nourie and Mike McCloskey. Regular expressions and the Java programming language, 2001 [2002]. http://java.sun.com/developer/technicalArticles/releases/1.4regex/.

[12] Scott Oaks. *Java Security*. O'Reilly, 2nd edition, 2005.

[13] Elçin A. Recebli. Pure aspects. Master's thesis, University of Oxford, Computing Laboratory, August 2005.

[14] Vipin Samar and Charlie Lai. Making login services independent of authentication technologies. In *Proceedings of the SunSoft Developer's Conference*, 1996. http://java.sun.com/security/jaas/doc/pam.html.

[15] Ravi Sandhu, Edward J. Coyne, Hal L. Feinstein, and Charles E. Youman. Role-based access control models. *IEEE Computer*, 29(2):38–47, 1996.

[16] Sun Microsystems, Inc. Java technology: Glossary, [8th March, 2006]. http://java.sun.com/docs/glossary.html.

[17] Sun Microsystems, Inc. Java 2 platform SE 5.0 API: Pattern class, [April 16th, 2006]. http://java.sun.com/-j2se/1.5.0/docs/api/java/util/regex/Pattern.html.

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

[19] Joseph Yoder and Jason Barcalow. Architectural patterns for enabling application security. In *PLoP'97, Proceedings of the 4th Conference on Patterns Language of Programming*, 1997.