Attribute Based Access Control for APIs in Spring Security

Alessandro Armando  
DIBRIS, U. of Genova, Italy  
and  
Security and Trust Unit,  
FBK-irst, Trento, Italy  
armando@fbk.eu

Roberto Carbone  
Security and Trust Unit,  
FBK-irst, Trento, Italy  
carbone@fbk.eu

Eyasu Getahun Chekole  
Security and Trust Unit,  
FBK-irst, Trento, Italy  
chekole@fbk.eu

Silvio Ranise  
Security and Trust Unit,  
FBK-irst, Trento, Italy  
ranine@fbk.eu

ABSTRACT

The widespread adoption of Application Programming Interfaces (APIs) by enterprises is changing the way business is done by permitting the implementation of a multitude of apps, customized to user needs. While supporting a more flexible exploitation of available data, services and applications developed on top of APIs are vulnerable to a variety of attacks, ranging from SQL injection to unauthorized access of sensitive data. Available security solutions must be re-used and/or adapted to work with APIs.

In this paper, we focus on the development of a flexible access control mechanism for APIs. This is an important security mechanism to guarantee the enforcement of authorization constraints on resources while invoking their API functions. We have developed an extension of the Spring Security framework, the standard for securing services and apps built in the popular (open source) Spring framework, for the specification and enforcement of Attribute-Based Access Control (ABAC) policies. We demonstrate our work with scenarios arising in a smart energy eco-system.

Categories and Subject Descriptors
D.4.6 [Security and Protection]: Access controls

Keywords
Attribute-based Access Control; Spring Security; Energy@Home

1. INTRODUCTION

Software is becoming more and more pervasive in our life and permeates almost every industry, ranging from hardware/software companies—offering combinations of hardware devices with software operating systems, applications, and cloud services—to sports clothing companies—setting up sports social networks to support targeted applications to customers. This is the result of the widespread development and adoption of apps, that can be used from a variety of devices and permit the reuse of existing information assets while adding value. For instance, Nike+\textsuperscript{1} aggregates data from hundreds of thousands of users every day, collected from a variety of devices (e.g., sport watches) and apps (e.g., the Running App) that allow for tracking sport activities, comparing performances with friends, etc. The idea is that purchasing a product is not the end but the beginning of the marketing cycle by offering consumers a wide range of products and services tailored to their individual needs.

The flexibility for developing tailored apps is offered by the increasing adoption of Application Programming Interfaces (APIs), which are interfaces allowing for invoking software components over a communication network using standard technologies (e.g., HTTP). APIs are used both internally and externally of an enterprise and turn out to be beneficial in both contexts. Internally, APIs reduce complexity by providing a single set of functionalities that can be invoked from clients on any platform, and simplify change management by identifying a core set of services on top of which several others can be fine tuned according to the evolving business needs. Externally, independent developers can think of innovative (sometimes unexpected) ways of using APIs in value-added applications without the need for the enterprise to invest in app development. Indeed, this is made possible only if the enterprise provides an adequate eco-system in which APIs can be exploited. For instance, the Nike Fuel Lab\textsuperscript{2} aims at fostering interest in leveraging the Nike+ technology by other companies. However, making APIs available to developers is a difficult task that must address a variety of issues such as developer management, adequate documentation, community forums, portals, and a set of well-engineered techniques to securing apps. For this reason, infrastructures to design and develop APIs are being proposed by IT companies such as Layer\textsuperscript{3} and WSO2.\textsuperscript{4} A substantial part of these infrastructures aim to provide means to protect services and apps built on top of APIs.

\textsuperscript{1}http://nikeplus.nike.com  
\textsuperscript{2}http://www.nikefuellab.com  
\textsuperscript{3}http://www.layer7tech.com and http://www.apify.co  
\textsuperscript{4}http://wso2.com/products/api-manager
from known web vulnerabilities (e.g., SQL injection) as well as new exploits deriving from insights obtained by analyzing the available API documentation. Following a basic rule in security, API infrastructures provide means to mitigate these vulnerabilities by re-using and adapting available security solutions for, e.g., confidentiality, integrity, authentication, and authorization at various levels.

In this paper, we consider the problem of realizing a flexible access control mechanism for APIs on top of the popular (open source) Spring Security framework, which is used to secure applications developed in the Spring framework. The latter provides a comprehensive programming and configuration model for modern Java-based enterprise applications and supports the development of (RESTful Web) APIs. We present the Attribute-Based Access Control (ABAC) model and argue why it is suitable for APIs (Section 2). We then present the architecture of the enforcement mechanism for ABAC policies in the Spring Security framework (Section 3). Finally (Section 4), we describe a demonstration for ABAC policies in the Spring Security framework.

We present the Attribute-Based Access Control (ABAC) model and argue why it is suitable for APIs (Section 2). We then present the architecture of the enforcement mechanism for ABAC policies in the Spring Security framework (Section 3). Finally (Section 4), we describe a demonstration for ABAC policies in the Spring Security framework.

In [4], ABAC has been identified as the “right” access control model for applications based on the Service Oriented Architecture (SOA) paradigm. Classical access control models are mostly static and coarsely grained since authorization decisions are taken by considering identities or roles. Such models are thus not well-suited for SOA applications where users are not easily authenticated, security-related information is distributed and frequently evolves over time. In contrast, ABAC bases authorization decisions on security-relevant properties (called attributes in ABAC) that can be easily distributed and updated. Furthermore, by suitably defining attributes, ABAC allows security experts not only to simulate and combine classical access control models but also refine them so as to supplement and combine—rather than supplanting—available access schemas; see [2] for a discussion on these and related issues.

It is natural to choose ABAC as the reference access control model also for APIs, since these share many basic architectural principles with SOA. We believe that the flexibility of ABAC can be even more crucial for securing APIs as these are meant to be more open and easily consumable than SOA applications, thereby posing more stringent requirements in terms of expressiveness to support a wide range of authorization requirements. For instance, ABAC permits to express access conditions on the data (called resources in ABAC) on which the API functionalities operate by means of their attributes. As observed in [3], this is beneficial since expressing access condition directly on resources is often more convenient than indirectly encoding them via the permissions of invoking API functions, as done, e.g., in Java Enterprise Edition (JEE) 2 applications using roles. The obvious precondition to use ABAC for securing APIs is the availability of suitable mechanisms for enforcing policies. In Section 3 below, we describe the architecture of one of such mechanisms for Java APIs, integrated in the popular (open source) Spring Security framework. In the rest of this section, we briefly recall the main notions underlying ABAC.

**ABAC in a nutshell.** The ABAC model allows for granting access rights according to the attributes of the entities involved in access control, namely subjects (e.g., users or applications), actions (e.g., read, write, update), resources (e.g., a file, a document, or a database record), and environments (i.e. contextual information such as location or time of day). For simplicity, in this paper, we will omit environments although the discussion can be easily generalized to handle them.

An ABAC policy \( P(s, a, r) \) is a Boolean-valued function on the sets of attributes of the subjects \( s \), actions \( a \), and resources \( r \). We assume that ABAC policies can be written as Boolean assertions. For instance,

\[
s_{\text{subjectId}} = r_{\text{resourceOwnerId}} \land a_{\text{name}} = \text{view}
\]

expresses the authorization condition that any subject \( s \) can view the content of any resource \( r \) provided that \( s \) is the owner of \( r \). An attribute assignment for a subject, an action, or a resource is a mapping of its attributes to appropriate values. For instance, assume that \( \text{Paolo} \) is a subject and that \( \text{subjectId} \) is one of its attributes: an attribute assignment \( v_{\text{Paolo}} \) can map \( \text{subjectId} \) to the integer 1434234278 and similarly for all its other attributes. Given attribute assignments \( v_s, v_a, \) and \( v_r \), we say that subject \( s \) can execute action \( a \) on resource \( r \) according to the ABAC policy \( P \) iff \( \text{true} \) is the value of \( P \) on the attribute values specified by \( v_s, v_a, \) and \( v_r \), written as \( v_s, v_a, v_r \models P \). For instance, \( v_{\text{Paolo}}, v_{\text{view}}, v_{\text{Doc}} \models (1) \) for \( v_{\text{Paolo}}(\text{subjectId}) = 1434234278, v_{\text{view}}(\text{name}) = \text{view}, \) and \( v_{\text{Doc}}(\text{resourceOwnerId}) = 1434234278 \).

**3. SPRING ENFORCEMENT OF ABAC POLICIES FOR API**

The Spring Security framework provides comprehensive and extensible support for both authentication and authorization to Spring-based applications. Here, we discuss some software techniques used in Spring Security that are relevant to our implementation of the enforcement mechanism for ABAC policies. In particular, we focus on method-level security, i.e. how the invocation of Java methods can be mediated by access control, that we customize to use ABAC policies.

**Dependency injection (DI)** is a software design pattern for which, instead of having an object instantiate its needed dependencies, the dependencies are given to the object by another object (usually called injector). The pattern is particularly useful for locating plugin components to customize the behavior of software components.

**Decorator pattern for proxies.** For method-level security, Spring Security uses proxies, i.e. methods are wrapped so that their behavior is extended without changing their original purpose. This is useful when we want to check authorization conditions before the invocation of a method. Syntactical support for the decorator pattern is given by authorization annotations that are added to a Java method \( m \) in order to evaluate policies and then decide to grant or deny...
the permission to invoke \( m \). We use the annotation `@PreAuthorize`\( (\text{SpEL expression}) \), that should be placed immediately before the declaration of a method whose access is supposed to depend on the fact that the `SpEL expression` passed as argument is evaluated to true. The Spring Expression Language (SpEL) is a powerful language available in Spring with constructs for—among others—accessing properties, method invocation, the usual mathematical operators (e.g., addition and subtraction), relational operators (equality, less-than, etc.), and Boolean connectives (conjunction, disjunction, and negation). The Spring Security framework furtherly extends SpEL with constructs for access control lists and role-based policies. In the rest of this section, we discuss how DI and `@PreAuthorize` annotations are used for generating customized proxies of methods in an API capable of enforcing ABAC policies.

**Enforcing ABAC policies for APIs in Spring Security.** The architecture of our solution is depicted in Figure 1 and can be seen as a refinement of the classical schema in which subjects (e.g., users or applications) make requests to execute certain operations (encapsulated in a method) on an object (i.e. an instance of a Java class to which the method belongs). The enforcement mechanism decides whether the particular request is allowed; this is done by the Security Interceptor, which needs to know who is making the request (the subject), what the request is (the action), what the target of the request is (the resource), and which is the policy to be used to take the decision. In Spring Security, the request is specified in the `@PreAuthorize` annotation of the method whereas to determine the subject and the target of the request, the Security Interceptor uses modules Security Context Handler and Authorization Manager, respectively. The correctness of the enforcement mechanism is ensured by the fact that the Security Interceptor intercepts every request on the resource. In our implementation, this is done by extending—via the decorator pattern for proxies based on the `@PreAuthorize` annotation—every method with code for answering authorization queries and permit or deny method invocation.

By using the DI pattern, suitable instances of the Authn (authentication) and Authz (authorization) Managers are injected into the Security Manager when deploying the application (typically by means of configuration files). Any of the methods for authentication available in Spring Security, (such as user-name and password or authentication protocols, e.g., OpenId) can be used. The Security Context Handler offers an interface to store and retrieve the Security Context, which contains security-relevant information including an Authn (authentication) object storing, among other security metadata, all the attributes of the subject. At every authorization request, the Security Context Handler checks the existence of an Authn object in the Security Context: if this is the case, it is passed to the Security Interceptor; otherwise, the Authn Manager is invoked so that the corresponding Authn object can be created.

After retrieving the Authn object, the Authz (authorization) Manager is invoked to establish if the Authn object has the right to perform the action—specified in the `@PreAuthorize` annotation—on the resource. (Notice that the resource is very easy to identify by establishing to which object is applied the invoked method.) The expression in the `@PreAuthorize` annotation is a conjunction of assertions of the form `can_do('action')`, where `action` is the name of a possible action such as ‘read,’ ‘write,’ or ‘view’ that can be performed on the resource. We allow conjunctions of `can_do` assertions since a method may require the subject to have several permissions on the resource; e.g., when reading from and writing to it. At this point, we have collected the attribute assignments for the subject, resource, and the actions, it is then possible to compute authorization decision (one per action in the `@PreAuthorize` annotation) by invoking the Authz (authorization) Manager. Indeed, the invocation of the method is permitted if all the authorization queries are answered positively.

Spring Security offers some implementations of authorization decision algorithms to be used by the Authz Manager, among which is the customizable evaluator for SpEL expressions. We have written our version of the evaluator to support a syntax for ABAC policies which is quite similar to the one presented in Section 2. This is possible by exploiting the usual ‘dot’ notation of Java to dereference attributes of subjects, actions, and resources. For instance, the ABAC policy (1) can be written as the following SpEL expression:

\[
\text{subject.subjectId == resource.resourceOwnerId and action.name == 'view'}\]

where `subject` and `action` stand for the subject and action variables \( s \) and \( a \), respectively. Our customization of the SpEL evaluator amounts to mapping `subject` to the user identified by the current Authn object, `resource` to the value of the variable `this`, which contains a reference to the object on which the method is invoked, and reading the ABAC policy from a file (specified in a configuration file). The specification of the policy in a separate file simplifies the process of changing the policy even while the API functionalities are invoked (it is sufficient to reload the file) without the need of re-compiling the code.

4. **THE ENERGY@HOME SMART GRID DEMONSTRATION**

Smart grids promise to optimize energy consumption. Key to this is the capability of collecting and sharing fine-grained energy consumption data by means of an advanced (smart) metering infrastructure for re-distributing energy according
to the time-varying consumer needs. Designs and implementing the information infrastructure underlying smart metering is difficult and error prone. To simplify this, APIs are being increasingly adopted for sharing data among the involved stakeholders, namely consumers, utilities, and third-party services. As a result, the security of applications built on top of the APIs is dependent upon their security.

We have used the implementation of the ABAC enforcement mechanism of Section 3 to ameliorate the confidentiality, integrity, and availability of the API for the (open source) smart metering framework JEMMA (Java-based Energy Management Application). This is part of the Energy@Home (E@H) platform, which is being actively developed by a non-profit association of companies (also called E@H) with the mission of developing and promoting techniques for energy efficiency in smart homes. We are contributing to secure the E@H platform in the context of the activity “SecSES Secure Energy Systems” of the action line ASES Smart Energy Systems of the EIT ICT Labs.

The architecture of the E@H platform is depicted in Figure 2. Smart metering data are collected by a Cloud Data Center (CDC) operated by Telecom Italia, which offers the API functionalities to enable customers and third-party providers (such as those in the Operations and Market domains) to exploit consumers data. Access must satisfy several authorization constraints. On the one hand, consumers own their consumption data and should be able to impose their own policies. Additionally, a number of legal constraints on the storing and processing of the data must be taken into account by the CDC, which should also be able to impose suitable policies supporting such constraints. On the other hand, third-party service providers (such as the operators in the Operations and Market domains) can access data according to the policies of consumers and the CDC, while constraining access to the data derived from their elaboration of smart metering data according to the contracts stipulated with users. We have discussed how to specify and combine ABAC policies to satisfy these constraints elsewhere [1]. During the demonstration, we will illustrate the policies for an example scenario and show how enforcement mechanism of Section 3 can support their enforcement.

More details, the demonstration will be structured as follows. First, we discuss selected parts of the API, the authorization requirements of the main stakeholders (consumers, utilities, and third-party service providers), and identify the sets of attributes that characterize their profiles. Second, we show the SpEL expressions of the ABAC policies that ensure the authorization requirements previously identified. The results of evaluating some authorization queries will be shown by considering different stakeholders that use a simple web application built on top of the API functions. We also discuss how policies rules can be changed at run-time (simply by editing a policy file) according to some evolving needs, thus leading to different authorization decisions. Finally, we discuss how policies can be changed by consumers (usually not experts in writing authorization conditions and thereby using the policies provided by the CDC when signing the contract) via a dashboard. To understand this, consider the following policy expression:

\[
\text{s.role} = \text{consumer} \land \text{r.owner} = \text{s.id} \land \text{releasable}(\text{r.id}, \text{p}) \land \text{a.name} = \text{read}
\]

where releasable checks whether the user has marked as releasable to the third-party service `p` a resource `r` which they own. The dashboard permits consumers to change the result returned by releasable for all the resources they own so that authorization decisions for each resource can be tuned while leaving the policies unmodified.

5. ACKNOWLEDGMENTS

This work has partially been supported by the activity “SecSES Secure Energy Systems” of the action line ASES Smart Energy Systems of the EIT ICT Labs, and by the MIUR PRIN 2010-11 project “Security Horizons.” We are grateful to Claudio Petrazzuolo and Andrea Ranalli from Telecom Italia for useful comments and discussions.

6. REFERENCES


