Scientific Journal of Impact Factor (SJIF): 4.14

e-ISSN (O): 2348-4470
p-ISSN (P): 2348-6406

International Journal of Advance Engineering and Research Development

Volume 3, Issue 12, December -2016

Authentication and Authorization Patterns in the Spring Security Framework

Prajapati Keyur P

Asst.Prof, MBICT, New V.V.Nagar – 388120,Gujarat, INDIA

Abstract—*In the development of secure applications, patterns are useful in the design of security functionality. Mature security products or frameworks are usually employed to implement such functionality. Yet, without a deeper comprehension of these products, the implementation of security patterns is difficult, as a non-guided implementation leads to non-deterministic results. In this paper, the Spring Security framework is analyzed with the goal of identifying supported authentication and authorization patterns. With this approach it is possible to overcome the gap between pattern-based security design and implementation to implement high quality security functionality in software systems.*

Keywords *- security patterns;security configuration, security framework; security engineering; authorization; authentication*

I. INTRODUCTION

Security engineering aims for a consecutive secure software development by introducing methods, tools, and activities into a software development process. As such, each phase of the software development needs to consider security aspects: in the analysis phase security requirements are identified, in the design phase security functionality is modeled in conjunction with the main business functionality and finally, security solutions are realized in the implementation phase.

Security patterns are an agreed upon method to describe best practice solutions for common security problems. When designing security functionality for an application such patterns can be instantiated in the design model to cover a certain security requirement.

The reuse of existing security functionality, i.e., in the form of security components, frameworks or products, is considered best practice as well, as they usually cover a great percentage of existing security requirements. Their maturity can usually not be achieved by implementing it completely new, so self-made solutions should extend it as well. By doing so, the quality of the security functionality of the developed application is increased. Also, as the main focus of software development lies upon the implementation of the business functionality, the reuse of existing functionality increases the efficiency of the implementation process.

Implementing security patterns using existing security functionality is complicated. For one, their built-in flexibility to support many different application contexts leads to a high complexity, requiring a deep understanding of the internal workings. This often raises the question, if and how the required security patterns can be implemented with the selected product. In such a case, the security functionality needs to be analyzed by security experts to determine the supported patterns.

Such an analysis is especially useful, if a model-driven approach is used to automatically generate security-related artifacts from design models. The identified and supported patterns of the framework or product can be used to describe the target platform and to generate framework artifacts from design models. Such an approach is part of a reuse-based security engineering approach, which we outlined in earlier works.

The rest of this paper is structured as followed: Section 2 introduces the Spring Security framework and discusses related work. In Section 3, the relationship of the pattern-based framework description to our reuse-based security engineering approach is described. The identified security patterns and their equivalent implementation using Spring Security are covered in Section 4. In Section 5, a real-world case study is presented, which shows the security pattern implementation using Spring Security. A conclusion and outlook on future work closes the body of this paper.

II. BACKGROUND AND RELATED WORK

The following section provides a background on the Spring Security framework and discusses related works.

A. Background on Spring Security

Spring Security is an open source Java framework, providing highly flexible and extensible authentication, authorization, and access control solutions.

The modular framework consists of loosely coupled components, which are connected using dependency injection. The core classes and their dependencies are shown in Figure 1. The *Authentication* class stores user information. It is part of a *SecurityContext* class for every authenticated user in an application. An *AuthenticationManager* loads this data and which verifies the authenticity of users using offered credentials and information from a user store .

Although it can be used for desktop applications, the main purpose of Spring Security is to secure web applications based on the Java Platform Enterprise Edition. The framework integrates with many authentication technologies and standards, e.g., Lightweight Directory Access Protocol (LDAP), Central Authentication System (CAS), OpenID and OAuth. Spring Security also provides support for basic role-based access control. Due to its flexible architecture the framework can easily be adapted and extended to support other forms of authentication and authorization and access control as well.

B. Related Work

Due to the identification of security patterns, the work is based on common security pattern literature. A comprehensive catalog of abstract and context-specific security patterns for, e.g., operating systems, can be found in. Identity management as well access control patterns are discussed in and. Patterns specific to the JEE platform are described in. Authorization patterns for the Extensible Access Control Markup language (XACML) are discussed in . An excerpt from the patterns presented in these works is used in this paper to show their support by the Spring

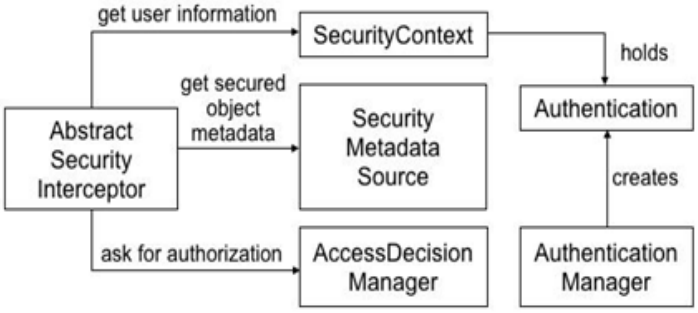


Figure 1 The main classes of the Spring Security Framework

Security framework. Pattern based security engineering processes are discussed in and , yet they do not consider the implementation of patterns using security platforms.

An automated retrieval of security patterns in existing software, such as discussed in and, would be useful in the identification process. Unfortunately, the retrieval rate of the approaches is still to low to be useful for our goals. Applying them would only show the patterns implemented in the software not all possibilities of the security framework. This is why a manual approach was applied.

III. REUSE-BASED SECURITY ENGINEERING

The pattern-based identification and description of security functionality in existing frameworks is part of a reuse-oriented security engineering approach, presented in .We argue for reuse of existing security functionality as well as knowledge throughout the phases of development processes to increase the quality and the development efficiency of the implemented software artifacts. Security problems, which can not be covered by existing models and functionality, can benefit from a reuse approach by extending or adapting them to a new context.

For one, the reuse of knowledge about possible threats and attacks against information resources, as well as appropriate countermeasures, is feasible in the analysis of security requirements of an application.

The topic discussed in this paper covers the design and implementation phase of the engineering process. In the design phase existing security knowledge should be used to determine possible solutions for security problems. Security patterns offer a proven method for describing such best practice solutions and can be integrated with common design patterns. The implementation of security solutions should be based on existing security functionality, e.g., provided by products, frameworks or components. These are more mature and field tested, than a new implementation and usually offer support for existing security standards and technologies.

Yet, to support the security engineering process, there is a need for knowledge of the frameworks used for securing the software product. During the design phase, knowledge about patterns that are supported by a framework is needed in order to avoid incompatibilities between design and implementation. When implementing the design it is beneficial to know how to implement a pattern with a framework. This leads to the need of pattern identification in security frameworks.

IV. AUTHENTICATION AND AUTHORIZATION PATTERN IDENTIFICATION

The following section describes the pattern identification and implementation process using the Spring Security framework. A focus was put on authentication and authorization patterns, as these are the focus of the framework as well. Thereby a distinction is made between the format of security guidelines describing policy patterns, and architectural patterns, describing components using and evaluating the policies.

A. Authentication Patterns Description

The patterns described in this chapter are supporting decisions in the software development process concerning authentication.

1) Authentication Policy Patterns

We have not found an abstract authentication pattern description in the aforementioned literature, which we deem relevant. The *Authentication Information* pattern defines, that a subject has to deliver some sort of information to prove an association to an identity in an application.

2) Authentication Architectural Patterns

Information about known identities needs to be stored for comparison with user input. The abstract *User Store* pattern defines, that user information is stored in some kind of repository. Depending on the type of authentication mechanism different implementations of the *User Store* are required. A LDAP directory or a database, containing usernames and passwords, are examples of User Store pattern implementation.

Enforcing the authentication needs specification of the required components in the software architecture and their interplay. The *Authentication Enforcer* pattern describes these components and their interaction in a web-based application. The pattern abstracts from the applied authentication mechanism, defined through the policies, to enhance reuse. Another aim of the pattern is to centralize authentication functionality and therefore to reduce redundancy.

The main component is the eponymous *Authentication Enforcer*, to which authentication requests of the client are sent to. It takes the information offered by the clients from the request context and compares it to data in the user store. On successful verification, a subject containing information gained from the user store on the subject is created.

B. Authentication Patterns Identification

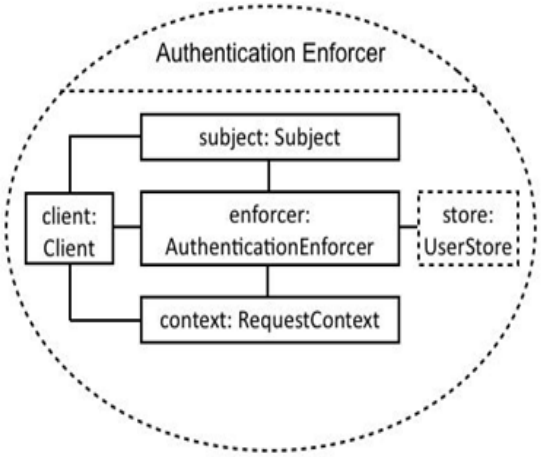
The main interface for implementing the *Authentication Information* pattern is the Spring Security *Authentication* interface, as its implementation offers information depending on the authentication mechanism. The *Authentication* interface is closely coupled to the *AuthenticationProvider* that loads the user information.

Accessing storages with the Spring Security framework, as required by the *User Store* pattern, is achieved through different implementations of the *AuthenticationProvider* interface. Each implementation represents a different *User Store* and uses varying *Authentication* concretions, e.g., the

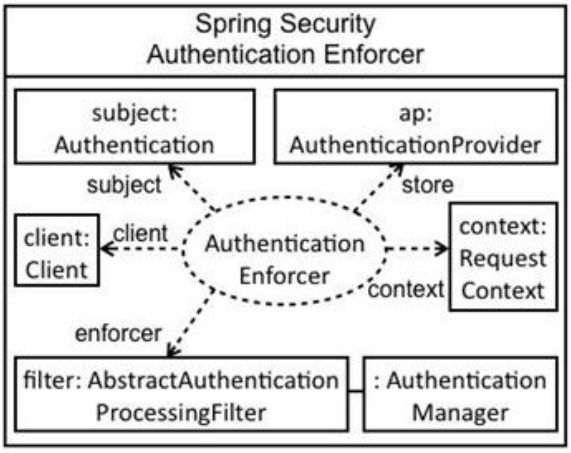
OpenIDAuthenticationProvider offers OpenID authentication by creating an *OpenIDAuthenticationToken* that implements the Authentication interface. The

AuthenticationManager uses the *AuthenticationProvider* to verify authenticity of users. An *AuthenticationManager* and its *AuthenticationProviders* can be configured using XML. An example configuration is shown in Figure 2. The default authentication manager is used and the custom authentication provider class can be inserted.

In Spring Security, the *Authentication Enforcer* pattern is implemented using the filter chain mechanism introduced by



(a) Authentication Enforcer Pattern



(b) Spring Security Implementation of Authentication Enforcer Pattern

Figure 2 Authentication Enforcer Pattern and Implementation with Spring Security

the Java Servlet Specification. The *DefaultLoginPage-GeneratingFilter* is executed if the login URL of the application is called and renders a login page to the client. When the client sends the rendered login form, the

UsernamePasswordAuthenticationFilter tries to authenticate the client using the configured *AuthenticationManager*.

Writing an own filter for supporting, e.g., biometric authentication is possible, too. For each filter specified in the filter chain, there must be a Java class with the same name. The filter chain and authentication provider offers flexibility in adding new authentication mechanisms and user stores needed to support the Authentication Enforcer pattern.

C. Authorization Patterns Description

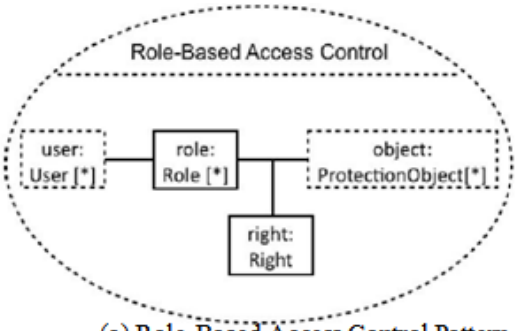
This section introduces patterns that can be used to describe or enforce authorization. Because there is a close relationship between authentication and authorization, some architectural patterns require authentication or even offer it.

1) Authorization Policy Patterns

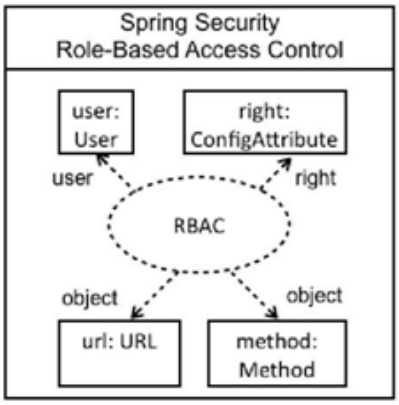
The *Authorization* pattern is used to define access control for resources at a high level of abstraction. A subject is assigned a right for a resource. High level of abstraction means, that subject, right and resource are not specified concretely and can be of any kind.

The direct interpretation of the *Authorization* pattern is called *Identity-Based Access Control*. Due to the structure, the concrete *Subject* gets directly assigned a *Permission* to access a *Resource* in a specific way. Thus a fine-grained definition of access control is established. Usually IBAC is implemented using access control lists (ACL).

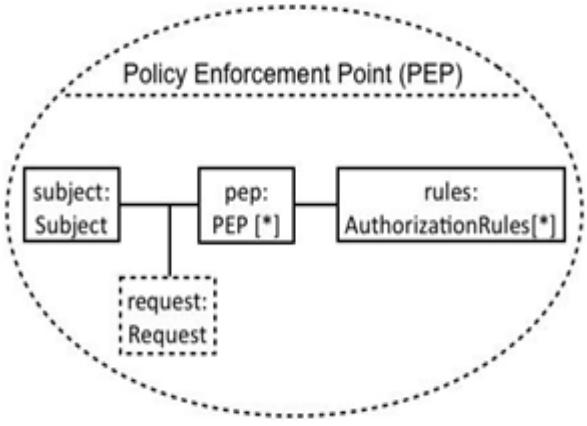
Role-Based Access Control (RBAC), described as a pattern in, is a specialization of the *Authorization* pattern, which refines the right assignment. Instead of directly assigning rights, a *Subject* gets assigned a *Role*, which



(a) Role-Based Access Control Pattern



(b) Implementing Role-Based Access Control with Spring Security



(c) Policy Enforcement Point Pattern

Requests on methods are intercepted using the Spring Aspect-Oriented Programming (AOP) feature. The Spring *AnnotationSecurityAspect* enhances security annotated methods. The advice of the aspect redirects method calls to the *AspectMethodSecurityInterceptor*, which is an implementation of the *AbstractSecurityInterceptor* interface, as well.

Thus, requests to URLs and methods are intercepted by the Spring Security framework and processed to enforce access control. The *AuthorizationRules* are described by the *AuthorizationPolicy* that is used. Method annotation and expressions in configuration for URLs describe the concrete *Authorization* for a *Resource*. The *PEP* pattern is used with Spring Security, if the *Authorization* pattern is set up and the *FilterChain* is configured or method security is activated .

The *Authorization Enforcer* pattern is the concretion of the *PEP* for Java EE applications. Thus, the mentioned protection of methods and URLs is an implementation of the pattern. The Spring Security *AuthenticationManager* takes the role of the *Authentication Provider* and the several authentication filters as well as the *AuthorizationManager* represent the *Authorization Enforcer* role. Thus, the *Authorization Enforcer* pattern can be implemented by using Spring Security access control. The *Intercepting Web Agent* pattern cannot be applied to the method protection, because the pattern defines application execution after access control. Thus the implementation of the pattern is applied through configuration of the *Authentication Enforcer* pattern, the *Authorization* pattern and a configured URL protection.

E. Discussion

The examination of the Spring Security framework revealed support for most known security patterns but failed to offer developers guidance on their implementation. This handicap has been overcome, as the proposed security pattern implementation templates enable the efficient mapping of pattern-based security design in future development processes. Thus, it allows security knowledge reuse as proposed by our security engineering approach described in Section III.

The identification process was thereby laborious as an intensive black box as well as white box examination of the framework was performed. This was only possible due to the excellent documentation and access to the framework's source code, which is not always the case, e.g., with proprietary frameworks, and makes the identification more difficult.

We tried to document the templates as independent of any application context as possible and in the implementation case study, discussed in the next section, we found that the templates are well crafted and suitable. But we do not claim completeness or efficiency. In fact, the templates as well as the pattern to implementation mapping may need to be adjusted to fit a specific context as well as future versions of the framework.

V. IMPLEMENTING CASE STUDY

The knowledge described in the previous sections combined with, e.g., use cases, misuse cases and component

TABLE II. SUPPORTED AUTHORIZATION PATTERNS

Authorization Patterns	Spring Security Implementation
Role-Based Access Control	Hierarchical roles using *GrantedAuthorities*
Identitty-Based Access Control	Access Control Lists
Attribute-Based Access Control	Simplified implementation using Spring Expression Language
Authorization Enforcer	Aspect interceptor for method access
Intercepting Web Agent	Filter mechanism of Java Servlets for URL access

diagrams has been applied to the development of the security functionality of a web application. Spring Security was used as the security platform used to protect the application.

A. KITCampusGuide Scenario Descriptions

The KITCampusGuide application is a navigation tool supporting students, teachers and staff in finding and navigating to points of interest (POI), i.e., any kind of landmark, such as a canteen, an auditorium or offices. Due to restricted areas on the campus and several other requirements, the search for and display of POIs has to be restricted. Users should be able to create private POIs, which can only be seen and modified by themselves. As such, management of POIs is the most relevant to security.

B. Secure Development of a POI Manager Component

A POI Management component was developed by modeling the requirements using UML use cases. Security analysis resulted in a need for user authentication and authorization, when creating private POIs. An architectural decision was made to use a single factor authentication using username-password pairs and RBAC for authorization policies. The security functionality is independent from the

```
<user name="student1"... authorities="ROLE_STUDENT" /> <user name="admin" ... authorities="ROLE_ADMIN" />
```

(a) User definition and role assignment

```
<bean id="rightsToRoles" class="oss.access.hierarchicalroles.RoleHierarchyImpl"> <property name="hierarchy">
    ROLE_ADMIN >ROLE_STUDENT ROLE_STUDENT > PERM_DELETE_POI
    ...
    </property>
</bean>
```

(b) Role definition and permission assignment

```
@RolesAllowed("PERM_DELETE_POI") public void delete(PointOfInterest poi) { ... }
```

(c) Configuring access control on a method using annotations

```
<http use-expressions=”true”> <intercept-url pattern=”/poi/*/delete”
        access=”hasRole(PERM_DELETE_POI)”/>
</http>
```

(d) Configuring access control on URLs

Figure 4 Implementing Role-Based Access Control in Spring Security (unnecessary information is stripped with “…”)

functional logic and supports access control to restrict access using an IWA. The architecture model was enhanced using the appropriate pattern descriptions.

Using the previously acquired knowledge about security patterns supported by the Spring Security framework, the security functionality was implemented by providing appropriate configurations to the framework and applying annotations to relevant methods. Figure 2 shows the necessary configurations to implement RBAC for a delete operation on POIs. Thereby two roles are defined and assigned to two different users. The role "ROLE_ADMIN" inherits the permissions of the role "ROLE_STUDENT", which in the shown example includes the permission to delete a POI. This is controlled using an annotation for the "delete" method as well as an authorization filter for the URL-based "delete" operation.

C. Problems and Experiences

Finding the level of abstraction needed for the application is an important issue during design phase. In the case study the whole development process was traversed by a single person and the application size was manageable. But as the size of the application grows, this could lead to problems. A hierarchy of patterns indicated in the previous chapters would close the gap

between a high level of abstraction and a level close to implementation. This is helpful in concretizing the design step by step.

VI. CONCLUSION AND FUTURE WORK

In this paper, the open source security framework Spring Security was examined in its support for common security patterns for authentication and authorization. Patterns for RBAC and ABAC as well as for username/password-based authentication were identified and appropriate best-practice implementation templates for Spring Security were provided. These templates can be used as a reference to implement the mentioned patterns in other projects. Further, the benefits of a pattern-based security framework description for a model-driven approach were discussed and its role in a reuse-based security engineering process was briefly explained.

In continuation of this work, the possible security design and implementation decisions need to be captured in flexible variation models to provide a decision support. Also, the relationships between the patterns will be determined and specified to identify mandatory or optional dependencies between the design and implementation patterns. In future research, we focus on completing the different parts of our reuse-based security engineering process.

REFERENCES

[1] A. Dikanski and S. Abeck, "Towards a Reuse-oriented Security Engineering for Web-based Applications and Services," Proc. Seventh International Conference on Internet and Web Applications and Services (ICIW 2012), Stuttgart, June 2012, pp. 282–285.

[2]. "Spring Security." SpringSource Community, p. Apache License, Apr. 2008.

[3] .M. Wiesner, "Introduction to Spring Security 3 /3.1," SpringOne 2GX. Chicago, Oct.-2010.

[4]] N. A. Delessy, E. B. Fernandez, and M. M. Larrondo-Petrie, "A Pattern Language for Identity Management," International Multi-Conference on Computing in the Global Information Technology, Guadeloupe City, March 2007, pp. 31–31.

[5]. E. B. Fernandez, G. Pernul, and M. M. Larrondo-Petrie, "Patterns and Pattern Diagrams for Access Control," Proc. Trust, Privacy and Security in Digital Business (TrustBus 2008), Turin, Italy, Sept. 2008,PP. 38–47.

[6] .R. K. Keller, R. Schauer, S. Robitaille, and P. Page, "Pattern-based Reverse-Engineering of Design Components," Proc. International Conference on Software Engineering, Los Angeles, 1999, pp. 226– 235.

