Identification and implementation of authentication and authorization patterns in the spring security framework

Article · January 2012

CITATIONS
6

READS
1,073

3 authors, including:

Aleksander Dikanski
Karlsruhe Institute of Technology
9 PUBLICATIONS **24** CITATIONS

SEE PROFILE

Roland Steinegger
Engineering Steinegger GmbH
9 PUBLICATIONS **27** CITATIONS

SEE PROFILE

Identification and Implementation of Authentication and Authorization Patterns in the Spring Security Framework

Aleksander Dikanski, Roland Steinegger, Sebastian Abeck
Research Group Cooperation & Management (C&M)
Karlsruhe Institute of Technology (KIT)
Karlsruhe, Germany
{ a.dikanski, abeck }@kit.edu, roland.steinegger@student.kit.edu

***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. Additionally, a best practice guide on implementing the identified patterns using the framework is presented. A real world case study is presented, in which the findings are employed to implement security requirements in a web application. 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 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 [1]. 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 [2]. 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 [3].

In this paper, the capabilities of the popular open source authentication and authorization framework Spring Security [4] are examined. The goal thereby is to identify support for common pattern by Spring Security and provide a reusable catalog of best practice advice on how to implement them in a high quality fashion. Theses informal description can be used by developers in the need to evaluate security frameworks as well as a guide to implementation. Also, they can be used to describe formal transformation rules for a model-driven approach.

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 [5][6].

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 [5].

To intercept secured resource access, classes extend the *AbstractSecurityInterceptor* class, which is the central class in terms of authorization. Thereby, the *SecurityContext* and *SecurityMetadataSource* classes offer information about the current user and the secured object respectively. Access decisions are performed by the *AccessDecisionManager*, which is also called by the *AbstractSecurityInterceptor*. The *AccessDecisionManager* calls voters, which decide whether access is granted or not and which can be added dynamically to the application. Thus, the voter system abstracts from an access control mechanism.

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 (JEE, [26]). 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 [6]. 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 [2]. Identity management as well access control patterns are discussed in [7] and [8]. Patterns specific to the JEE platform are described in [9]. Authorization patterns for the Extensible Access Control Markup language (XACML) are discussed in [10]. An excerpt from the patterns presented in these works is used in this paper to show their support by the Spring Security framework. Pattern based security engineering processes are discussed in [11] and [12], yet they do not consider the implementation of patterns using security platforms.

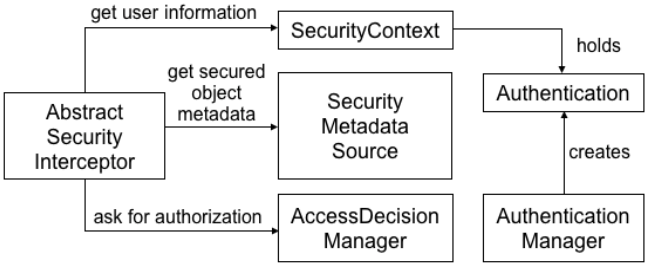


Figure 1 The main classes of the Spring Security Framework

An automated retrieval of security patterns in existing software, such as discussed in [13] and [14], 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.

The pattern-based platform description presented here is a feasible enhancement to model-driven security approaches, which is not considered by other such approaches, e.g., [15][16][17]. We aim at describing the target security platform using security patterns, to simplify the transformations and easily adapt them to new platforms.

Background information on the Spring Security framework, its inner relations and concepts as well as its usage can be found in the community documentation as well as in [5] and [6]. These descriptions are not based on security patterns and do not show all possible applications of the framework.

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 [3]. 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 [2]. 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.

The patterns were identified manually by using practical experience on securing applications with the framework, its openly accessible source code and reference documentation as well as a book about the framework [5]. The selected patterns to identify in the framework cover several areas within authentication and authorization. Another reason in favor for the selection is their publicity. Commonly known patterns were selected from [2] and [9].

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.

In [9], several mechanisms to authenticate a subject are specified, which offer three specializations of the pattern. The distinction is made on what kind of prove has to be presented, i.e., the subject deliver information it knows, e.g., a username and password, it owns, e.g., from a smartcard, or intrinsically has or is, e.g., finger prints. Lastly, the fourth specialization of the *Authentication Information* pattern is the combination of any two or more of these three concretions, which is called multi factor authentication.

2) Authentication Architectural Patterns

Information about known identities needs to be stored for comparison with user input. The abstract *User Store* pattern [9] 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 [9] 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.

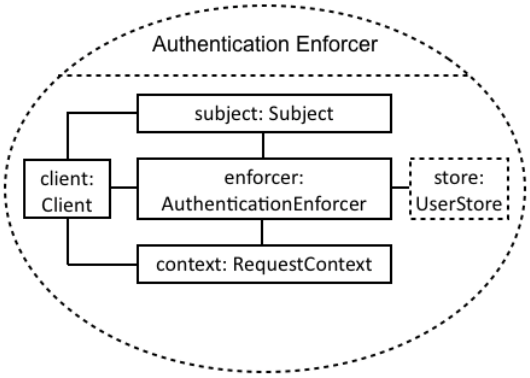
The main component is the eponymous *Authentication Enforcer*, to which authentication requests of the client are sent to. It takes the information offered by the clients from the request context and compares it to data in the user store. On successful verification, a subject containing information gained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

B. Authentication Patterns Identification

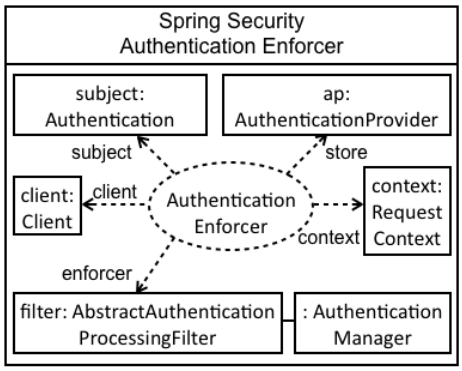
The main interface for implementing the *Authentication Information* pattern is the Spring Security *Authentication* interface, as its implementation offers information depending on the authentication mechanism. The *Authentication* interface is closely coupled to the *AuthenticationProvider* that loads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

TABLE I. SUPPORTED AUTHENTICATION PATTERNS

Authentication Patterns	Spring Security Implementation
Authentication Information	• Single and multi-factor authentication using username-password, OpenID, X.509 certificates, HTTP Basic and Digest authentication (native) • adaptable to other authentication methods using 3rd party frameworks
User Store	• XML configuration, LDAP, Database, properties file (native) • adaptable to 3rd party user store
Authentication Enforcer	• Authentication filters for Java Servlet filter mechanisms

the Java Servlet Specification [18]. 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*. Another example is the *BasicAuthenticationFilter*, which gets the username and password from the request according to RFC 1945 [19] and verifies authenticity. There are also filters for CAS or OpenID authentication, because they depend on an external user store and therefore must be treated differently. Due to the statelessness of HTTP, the *SecurityContextPersistenceFilter* is needed, which persists the security context including the authentication in the HTTP session before responding to a request and recovers the security context at the beginning of the next request.

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

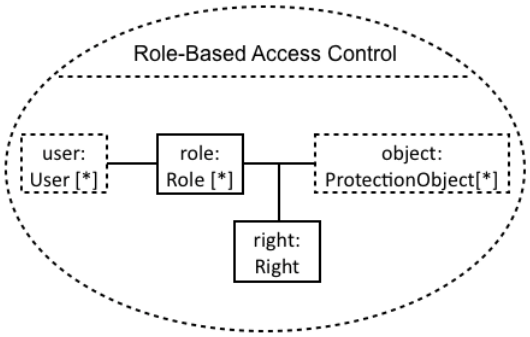
This section introduces patterns that can be used to describe or enforce authorization. Because there is a close relationship between authentication and authorization, some architectural patterns require authentication or even offer it.

1) Authorization Policy Patterns

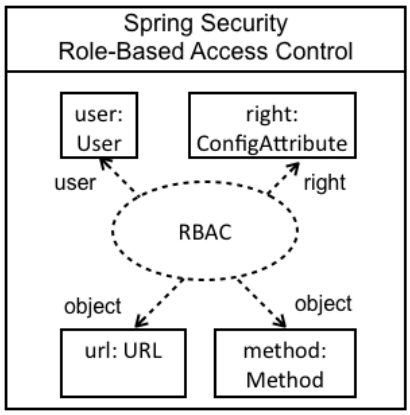
The *Authorization* pattern [2] is used to define access control for resources at a high level of abstraction. A subject is assigned a right for a resource. High level of abstraction means, that subject, right and resource are not specified concretely and can be of any kind.

The direct interpretation of the *Authorization* pattern is called *Identity-Based Access Control* (IBAC [2]). 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).

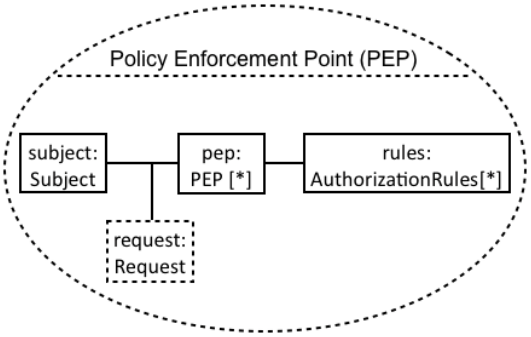
Role-Based Access Control (RBAC), described as a pattern in [2], is a specialization of the *Authorization* pattern, which refines the right assignment. Instead of directly assigning rights, a *Subject* gets assigned a *Role*, which



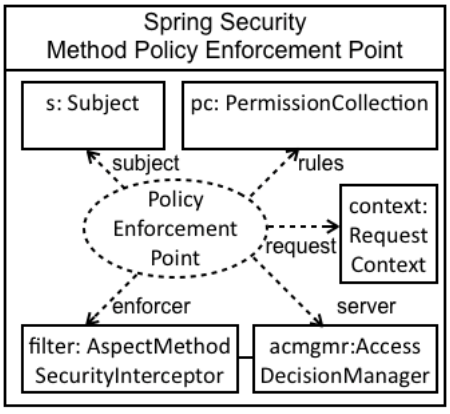
(a) Role-Based Access Control Pattern



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



(c) Policy Enforcement Point Pattern



(d) Spring Security Implementation of Policy Enforcement Point for Method Access

Figure 3 Authorization Patterns in Spring Security

represents a set of *Permissions* to access a *Resource*. Thus, it is possible to assign *Subjects* with the same access rights using *Roles* among a system reducing the complexity of rights management.

Another concretion of the *Authorization* pattern is *Attribute-Based Access Control* (*ABAC* [20]). In contrast to RBAC, *Permission* can be defined through expressions using all available *Attributes* of *Subject*, *Resource* or *Environment*.

2) Authorization Architectural Patterns

Besides defining authorization policies, there are patterns describing their enforcement, i.e., access control. An abstract example for enforcement of access control is the *Policy Enforcement Point (PEP)* pattern, also known as *Reference Monitor* [2][21]. The PEP defines components and flows needed to control access to a resource in an abstract way. Requests to a *Protected Resource* shall be intercepted by the *PEP*. According to *Authorization Rules*, which consist of *Authorization* items, access is granted or denied.

Another concretion of the *PEP* is the *Authorization Enforcer* pattern [9]. The purpose of the pattern is to control access in a JEE application. Due to this circumstance, there are several variations of the pattern using different Java specifications. Requests from a *Client* are intercepted and redirected to the *Authorization Enforcer*, which uses the *Authentication Provider* to set the *Permissions* to the already loaded *Subject*. Thus the pattern needs an authenticated *Subject*, e.g., set by the *Authentication Enforcer* pattern. With the *Permissions* of the *Subject*, the *Authorization Enforcer* decides, whether the access is granted or rejected.

The *Intercepting Web Agent (IWA)* pattern [9] helps in separating application logic from authorization and authentication logic. It can also be used to add access control and authentication after the development of an application. The name already suggests that the patterns operational area is web application development. *Client* requests are intercepted by the eponymous *IWA*. Either the *Client* authenticates itself and its authentication information is persisted through a cookie or the *Client* tries to access a *Resource* directly, in which case the *IWA* loads the previously persisted information of the *Subject*. The request is forwarded by the *IWA*, if the *Subject* is authorized.

D. Implementation of Authorization Patterns

The following section discusses implementing the authorization patterns with the Spring Security framework.

1) Policy Pattern

Due to the voter mechanism used for access decisions, the framework can be enhanced to support several access control patterns, thus it supports the *Authorization* pattern. The sections about ABAC, RBAC, and IBAC show different voters supported by the framework and indicate the flexibility. By implementing an *AccessDecisionVoter*, it is possible to access external frameworks or software and to gain extra information needed for the decision or to ask for the decision from external software.

RBAC raises the need for defining *Roles* of *Users*. In Spring Security *Roles* are called (*Granted-*) *Authorities* [5]. *Authorities* can be assigned to Users *via* configuration or loaded from a *User Store* [5]. A documented best practice is the arrangement of *Authorities* into hierarchies [6]. *Roles* are assigned to *Users* and *Rights* are assigned to *Roles*. Thus a hierarchy is built and *Users* are assigned several *Rights* through their *Role*.

Rights are assigned to *Roles* to access a *Resource*. Spring Security supports the protection of methods and URLs as *Resources* [5]. In the configuration or annotation the corresponding *Right* is used, as can be seen in Figure 2 (c) and (d). Thus only *Users* with a *Role* having the *Right* to modify a resource are allowed to access it. When implementing a web application based on the REST (Representational State Transfer) paradigm [22], the approach of protecting URLs is preferred. Otherwise, method security and the use of annotations according to the Java Specification Request (JSR) 250 should be used. Thus, the flexibility in changing the security framework is saved.

ABAC is not directly supported by Spring Security, but can be easily implemented as shown next. Spring Security offers the Spring Expression Language (SpEL [5]) to describe access control. Instead of annotating a right to methods or to a URL, expressions can be used. When evaluating to *True*, access is granted. In SpEL expressions, *Attributes* of the *Subject* or the *Resource* can be used and compared, e.g., "*authentication.id=#resource.ownerId*", which evaluates to *True*, if the users owns the resource.

These expressions can be combined with "and" and "or". In general, the SpEL fulfills the requirements of the application. When using more complex ABAC expressions, SpEL in combination with *PermissionEvaluators* can be used. For that, the expression "*hasPermission*" can be used [5], for each of which the processing *AccessDecisionVoter* calls appropriate *PermissionEvaluators*. Implementing a *PermissionEvaluator* closes the gap between the needs of ABAC and the Spring Security access control implementation. The implementation of the *Permission-Evaluator* interface can access any *Attribute* of the *Subject*, *Resource* and *Environment*.

Spring Security offers the use of *Access Control Lists (ACL)*, which are commonly used to implement IBAC [2]. In [5], the set up of a database, holding the *ACL* and the configuration of Spring Security to use a database, is shown. For each *Resource* an *Access Control Entry* can be added to the database, giving specific *Permissions* to a *Subject*. Built-in permissions are read, write, create, delete and administer. These *Permissions* can be enhanced or replaced [5]. Besides *ACL* and its *Entries*, the protected URLs have to be configured or methods have to be annotated. This is done using the "*hasPermission*" SpEL expression [5].

2) Architectural Patterns

The previous section showed the definition of authorization policies with Spring Security and merely parts of their enforcement. The framework uses a concrete *PEP* for URLs and for method access control respectively. The *PEP* has to handle all requests on a *Protected Object*. A filter (*FilterSecurityInterceptor*) is used to intercept requests on URLs and to control access on the URL. The filter implements the *AbstractSecurityInterceptor*. Thus requests on URLs are handled as described in Section II.A.

Requests on methods are intercepted using the Spring Aspect-Oriented Programming (AOP) [23] 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 [5].

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 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.

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

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] R. J. Anderson, Security Engineering, 2nd ed. Indianapolis, Ind.: Wiley, 2008, p. 1040.

[2] M. Schumacher, E. B. Fernandez, D. Hybertson, F. Buschmann, and P. Sommerlad, Security Patterns. Chichester, England: John Wiley & Sons Ltd, 2005, p. 565.

[3] 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.

[4] “Spring Security.” SpringSource Community, p. Apache License, Apr. 2008.

[5] P. Mularien, Spring Security 3. Birmingham, Packt Publishing, 2010.

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

[7] 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.

[8] 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.

[9] C. Steel, R. Nagappan, and R. Lai, Core Security Patterns, 1st ed. Upper Saddle River, N. J.: Prentice Hall International, 2005, p. 1088.

[10] N. A. Delessy and E. B. Fernandez, “Patterns for the eXtensible Access Control Markup Language,” Proc. 12th Pattern Languages of Programs Conference (PLoP2005), Monticello, Illinois, USA, Sept. 2005, pp. 7–10.

[11] E. B. Fernandez, “A Methodology for Secure Software Design,” Proc. International Conference on Software Engineering Research and Practice (SERP’04), pp. 21–24, 2004.

[12] N. A. Delessy and E. B. Fernandez, “A Pattern-Driven Security Process for SOA Applications,” Proc. ACM Symposium on Applied computing, pp. 2226–2227, 2008.

[13] M. Bunke and K. Sohr, “An Architecture-Centric Approach to Detecting Security Patterns in Software,” Proc. Third International Conference on Engineering Secure Software and Systems (ESSoS'11), Madrid, 2011, pp. 156–166.

[14] 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.

[15] T. Lodderstedt, D. Basin, and J. Doser, “SecureUML: A UML-Based Modeling Language for Model-Driven Security,” LNCS, vol. 2460, pp. 426–441, 2002.

[16] M. Alam, R. Breu, and M. Hafner, “Modeling Permissions in a (U/X) ML World,” Proc. First International Conference on Availability, Reliability and Security (ARES), Vienna, April 2006,pp. 685–692.

[17] C. Emig, S. Kreuzer, S. Abeck, J. Biermann, and H. Klarl, “Model-Driven Development of Access Control Policies for Web Services,” Proc. 9th International Conference Software Engineering and Applications (IASTED), vol. 632, pp. 069–165, 2008.

[18] “Java Servlet 2.3 Specifications,” Palo Alto, 53, Sep. 2001, last accessed March 2012.

[19] T. Berners-Lee, R. Fielding, U. C. Irvine, and H. Frystyk, “Hypertext Transfer Protocol (HTTP 1.0),” 1945, May 1996.

[20] E. Yuan, J. Tong, B. Inc, and V. McLean, “Attributed Based Access Control (ABAC) for Web Services,” Proc. IEEE International Conference on Web Services (ICWS), Orlanda, Florida, July 2005.

[21] T. E. Fægri and S. O. Hallenstein, “A Software Product Line Reference Architecture for Security,” in Software Product Lines, no. 8, T. Käkölä and J. C. Dueñas, Eds. Berlin, Heidelberg: Springer, 2006, pp. 276–326.

[22] R. T. Fielding, Architectural Styles and the Design of Network-based Software Architectures, Irvine: University of California, Irvine, 2010.

[23] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. Lopes, J.-M. Loingtier, and J. Irwin, “Aspect-Oriented Programming,” in Lecture Notes in Computer Science, vol. 1241, no. 10, M. Akşit and S. Matsuoka, Eds. Berlin/Heidelberg: Springer-Verlag, 1997, pp. 220–242.

[24] Oracle, Java Platform, Enterprise Edition (Java EE) 6, <http://www.oracle.com/technetwork/java/javaee/overview/index.html>, last access March 2012.