Attack Surface Reduction for Web Services based on Authorization Patterns

Roland Steinegger, Johannes Schäfer, Max Vogler, and Sebastian Abeck
Research Group Cooperation & Management (C&M)
Karlsruhe Institute of Technology (KIT)
Karlsruhe, Germany
{ abeck, steinegger }@kit.edu, { johannes.schaefer, max.vogler }@student.kit.edu

***Abstract*—During the design of a security architecture for a web application, the usage of security patterns can assist with fulfilling quality attributes, such as increasing reusability or safety. The attack surface is a common indicator for the safety of a web application, thus, reducing it is a problem during design. Today's methods for attack surface reduction are not connected to security patterns and have an unknown impact on quality attributes, e.g., come with an undesirable trade-off in functionality. This paper introduces a systematic and deterministic method to reduce the attack surface of web services by deriving service interface methods from authorization patterns. We applied the method to the Participation Service that is part of the KIT Smart Campus system. The resulting RESTful web services of the application are presented and validated.**

Keywords-security pattern, attack surface, authorization, web service, rest

I. INTRODUCTION

Every web application has assets needing protection from threats, e.g., web services. Thus, securing web applications is a major issue. Security must be considered during the whole software development life cycle to build secure software [1]. In such a security-based software development life cycle, security patterns are used during the design phase to solve common security problems and build a security architecture [2].

Security patterns in the security architecture can have an impact on non-security quality attributes of the whole software system, such as loose coupling or discoverability [2]. When using security patterns, it is helpful to know this influence on the quality of the application [3]. Additional, security should be applied as early as possible to increase overall security [3]. Developers are generally not security experts and a systematical approach can help them reaching quality requirements [4]. Regarding a concrete quality attribute, the attack surface, several metrics have been introduced to measure the attack surface of whole software systems [5], object oriented designs [3][6] and web applications [7].

In addition to metrics, there are methods to reduce the attack surface, e.g., by using the Top 10 most critical applications security flaws of the Open Web Application Security Project (OWASP) [8], by removing or disabling less important or unnecessary functionality [9][10] or by reducing the permissions of the application [11]. These methods do not offer the possibility to systematically reduce the attack surface and they do not describe their influence on other quality attributes. Additionally, there is no connection to security patterns that are commonly used in a security-based development process.

Thus, we propose a method based on security patterns for authorization to reduce the attack surface of web services. The method has direct impact on the service interface. It mainly focuses on web services having a manageable amount of authorization rules that do not change periodically. It reduces the attack surface, by reducing the privileges for methods on the interface to the minimum needed, according to authorization. Furthermore, the client can choose under which privilege a service interface method should be called. Both increase the security by following the principle of least privilege and secure interaction design [12]. Our approach additionally leads to service interfaces, which are compliant with the Representational State Transfer (REST) paradigm [13].

The method is applied on the Participation Service of the KIT Smart Campus system. The service uses an Attribute-Based Access Control (ABAC) for authorization due to complex security requirements. The resulting web services of the Participation Service are introduced. The web services are analyzed using the attack surface metric of [7].

The article is structured as follows: Firstly, the needed background and related work are introduced in Section II. The approach is presented in Section III for two commonly used authorization patterns. The next Section IV shows the evaluation of the approach by applying it on the Participation Service. After the evaluation Section V discusses limitations of the approach. The paper gives conclusions and an outlook on future work in the last Section VI.

II. BACKGROUND AND RELATED WORK

In this section, the needed background for our approach is presented. This includes the software system used for evaluating the approach, the Participation Service, security patterns used for our approach, and related work on the attack surface, as well as on REST and its constraints.

A. Participation Service of the KIT Smart Campus

The KIT Smart Campus (KIT-SC) system is a web application developed at the Karlsruhe Institute of Technology (KIT). A detailed description of the KIT-SC and its features is given in [14]. The KIT-SC pursues the goal to support students and employees at learning, teaching and other activities related to the KIT campus.

The Participation Service represents a part of the KIT-SC. It provides a forum with voting and discussion features.

Following the principles of systemic consensus, this enables groups of users to make decisions on campus-related issues by using the modern, responsive web application.

B. Security patterns for authorization

With our approach, service interfaces are derived from authorization patterns. The steps are shown for two common security patterns: Role-Based Access Control (RBAC) and ABAC.

RBAC takes advantage of the fact, that organizations are often structured in roles, e.g., students, employees and administration [2]. These roles have certain rights and duties. The rights of these roles can be used to model the access rights in the system. Thus, subjects get all rights through their roles. In this way, the process of assigning access rights is simplified by the usage of global roles instead of individual rights [2].

The structure of RBAC shows Figure 1. Subjects have certain roles and these roles are directly connected with resources. The concrete right is associated to the connection between role and resource. As soon as roles are not applicable or a more flexible access control is required, RBAC has strong limitations [15].

ABAC is a more flexible approach because of the usage of attributes as information source for access control [15]. In addition to static roles, which can still be realized with ABAC, access control can be defined for dynamic attribute combinations of subjects, resources and environments [15]. This structure shows Figure 2. Subjects are directly connected to resources. The right is associated to this connection and uses the attributes.

Yuan et al.'s formal definition [15] is: S, R and E are subjects, resources and environments with pre-defined attribute sets SAn, RAn and EAn. A policy rule that decides on whether a subject s can access a resource r in an environment e is a Boolean function of s, r and e's attributes:

canAccess(s, r, e) ← f(ATTR(s), ATTR(r), ATTR(e))

where ATTR() is a function that assigns every currently valid attribute to a subject, resource or environment.

Authorization using ABAC is, thus, more fine-grained than RBAC. But as negative aspect, it is more complex to implement.

C. Attack Surface

With our approach, we connect security patterns with software product quality according to ISO/IEC 25010 [16]. These are on the one hand the quality attribute attack surface and on the other hand quality attributes connected to the REST paradigm. In this section, we introduce the attack surface.

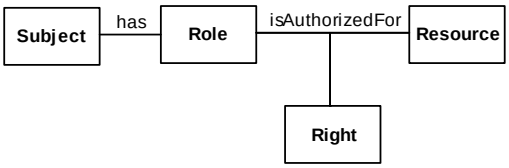

Figure 2. Scheme of the Roled-Based Access Control based on [17]

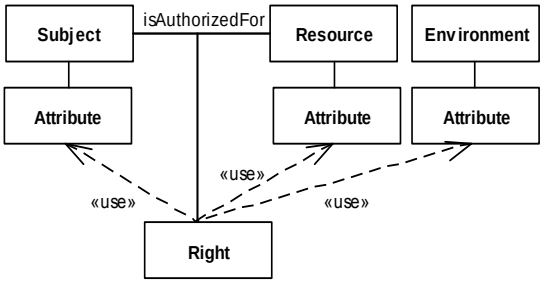

Figure 1. Scheme of the Attribute-Based Access Control based on [17]

Developers wish to anticipate the vulnerability of their software system prior to deployment. The popular concept of loose coupling and the distribution of systems or web applications lead to an increasing number of interfaces [18]. These are natural security boundaries that augment the attack surface, an indicator for measuring a system's vulnerability towards external attacks [7][9].

The attack surface does not give information on code quality or high-value architectural design. And neither does a large attack surface imply that a system has much vulnerability, nor does a small attack surface mean little vulnerability. But a large attack surface indicates that an attacker presumably needs less effort for exploiting vulnerabilities [5]. The reduction of the attack surface, therefore, reduces the overall security risk – a product of the probability, the consequences of occurrence of a hazardous event and the asset value: Risk = Threat × Vulnerability × Asset Value [19]. Think of two web applications with similar functionality and value – the one with a higher attack surface is more likely to be chosen to attack amongst these opportunities.

We use the attack surface metric for web applications [7] to evaluate our approach. The metric is based on parameters grouped into parameter families. These parameter families are *Degree of Distribution*, *Dynamic Creation*, *Security Features*, *Input Vectors*, *Active Content*, *Cookies* and *Access Control*. Parameters are, e.g., *Role* and *Privileges* for the parameter family *Access Control*. For each of the parameters a value is assigned, depending on the application. The higher the value, the greater is the attack surface and the higher is the risk for attacks, e.g., accessing the application as unauthenticated user has value 0, whereas accessing as authenticated or root user have value 5 and 10. The metric is calculated by calculating the Euclidian norm for each value of a parameter family. The value of the parameter family is the Euclidian norm calculated for each value of parameter in the family. The maximum attack surface is 60.79.

In the next sections, we discuss methods for reducing the attack surface regarding our goals and service interface design. The author of [9] suggests several methods for reducing the attack surface of an operating system. His 80/20 rule (according to the Pareto principle) to reduce the amount of running code contradicts our goal to not reduce functionality. Further, he offers no systematical way to find

code to remove. The methods for applying least privileges and reducing access for untrusted users mainly focus on the system running the application. According to this method, we suggest that for service interfaces least privileges also means reducing the amount of accessible operations. Authorization defines who shall access operations and is, therefore, our starting point for securing access by reducing the attack surface.

Reference [10] introduces an approach for removing or disabling unused code in operating systems. This corresponds to finding the 20 percent in the 80/20 rule of [9] and therefore, it aims to reduce functionality. Their general approach consists of two phases, the analysis and enforcement phase. In the analysis phase, unused code is found. The enforcement phase aims to avoid execution of unused code. They identify unused code by running the application and executing all available methods. Thus, this approach needs a running application and is firstly applicable in the implementation phase. We think that seldom-used or unused code could be avoided by considering security earlier.

Methods for reducing the attack surface of a web application based on the Top 10 vulnerabilities published by the OWASP are introduced by [8]. The authors use security measures mitigating these vulnerabilities. The Top 10 entries are related to security vulnerabilities in web applications and therefore, they do not have to be connected to the attack surface. Thus, not all of the applied measures, such as input validation and secret tokens, affect the attack surface directly. A systematical way to reduce the attack surface needs to ensure this reduction.

The discussed approaches aim to reduce the attack surface of in several ways. They do not offer a systematical way with concrete transformations to reduce the attack surface. Often the functionality of the application is reduced to ensure a smaller attack surface. Using security patterns is not part of any of these approaches. We tackle these limitations with our approach.

D. Web Services based on REST

According to the W3C, the term web service refers to a software system designed to support interoperable machine-to-machine interaction over a network [21][20]. It is frequently regarded more as a system's function of providing web access to its inner purpose rather than the whole system itself. Furthermore, a web application consists of web services, e.g., the web browser uses web services.

The W3C distinguishes two types of web services: Those using REST-compliant interfaces and those providing arbitrary access [20]. While the latter have been primarily used in the past – presumably because of the ease of implementation – RESTful interfaces become increasingly popular, mainly for their lightweight and universal deployment [21].

REST is an architectural style for the communication of web services proposed by Fielding [13]. It relies on existing standards, such as the Hypertext Transfer Protocol (HTTP), and defines six constraints for RESTful interfaces rather than concrete implementation specifications: The Client-Server principle, the concept statelessness, the usage of a cache, the uniformity of the interface, the layered system and the optional Code-On-Demand feature [13].

The uniform interface is the centerpiece of the REST architectural style: The interface describes every aspect trough resources. Every resource is identified by a unique address, which is in most cases a URI. Those resources are retrieved or manipulated via representations. A set of valid operations on these representations is available. Requests and responses are self-descriptive and semantic and hypermedia is used to describe them [13]. Hereby, a high degree of universality is achieved. However, it comes with a compromise in efficiency since the standardized information transfer leads to an overhead [21].

Since our approach alters the operations allowed on the resources, the compliance of the new interface to the uniformity concept is focus of validation.

III. Deriving Service Interface Methods from Authorization Patterns

In this section, we introduce our method to reduce the attack surface. We developed the approach based on the following assumptions and formulated goals 1 to 6. First, current methods for attack surface reduction have inacceptable deficits, such as decreasing functionality (goal 1 and 4). Second, non-security experts can apply the method and ensure security [4] (goal 2). Third, the method must be applicable at an early stage [3] (goal 3) on the KIT Smart Campus (goal 5, 6).

1. Security patterns shall be connected to software product quality not related to security.
2. A systematic way shall ensure certain quality attributes, including the attack surface.
3. The method shall be applicable in an early software development phase.
4. The method shall not reduce application functionality.
5. The method shall be applicable on web applications.
6. It shall apply for web services similar to the RESTful web services of the Participation Service.

Before introducing the method, we align the term attack surface according to ISO/IEC 25000 and 25010. The attack surface is an inherent characteristic of software, because it can be measured with several metrics introduced. Thus, speaking in the language of ISO/IEC 25000 [22], it is a software quality attribute. We suggest to assign it to the quality characteristic freedom from risk and its sub characteristic economic risk mitigation according to ISO/IEC 25010 [16]. Therefore, it belongs to the quality in use model.

Concerning the method, the starting point is the authorization of the application and corresponding security patterns. These patterns describe who can access resources in which way. Thus, authorization can be used to reduce the attack surface to exactly the functionality that shall be offered. Regarding the metric for web applications introduced in [7], our approach reduces the parameter family of access control. Other parameter families are not influenced by the approach and, thus, a reduction is ensured.

Our approach consists of the following three steps:

1. Set up an access control matrix.
2. Derive services from the access control matrix.
3. Create REST-compliant web services based on the derived services.

The access matrix of the first step contains resources and operations as columns and policy rules as rows. For every operation allowed by a policy rule, the corresponding table cell is filled with a dot. See Table 1, Table 2 and Table 3 as examples. In the second step, a web service is introduced for each resource. Its service interface has an operation for every table cell having at least one marked row. Figure 3 is an example for this. In the last step, the resulting web services are mapped to a REST-compliant web service. Each step is introduced in the next sections. First, the main idea of deriving technology independent web services and its service interfaces is explained in depth. Second, the mapping from the abstract web service to a REST-compliant web service.

A. *Deriving Abstract Service Interfaces from Role-Based Access Control*

A role-based scheme for the access control with n different resources and m roles can be depicted as a two-dimensional matrix (see example on Table 1). With the REST paradigm's resource-oriented interface style kept in mind, we assume that four operations are possible per resource: Creating, retrieving, updating and deleting (CRUD). A bullet indicates that the specified role is allowed to use the specified operation on the specified resource.

While in an ordinary RESTful implementation the interface would have provided access for all roles on all operations and all resources, our approach aims to reduce the overall number of accessible operations to a minimum. In the context of Table 1, this would lead to a reduction of the attack surface by the number of unfilled table cells.

TABLE I. EXEMPLARY MATRIX FOR RBAC WITH ROLES

	Resource #1				**Resource #2**			
	C	***R***	***U***	***D***	***C***	***R***	***U***	***D***
Role #1		●				●		
Role #2		●	●		●	●		●
Role #3	●	●	●	●	●	●		●

TABLE II. EXEMPLARY MATRIX FOR ABAC WITH EXPRESSIONS

	Resource #1				**Resource #2**			
canAccess(s, r, e)	***C***	***R***	***U***	***D***	***C***	***R***	***U***	***D***
attribute1(r)		●				●		
!attribute2(r)		●	●		●	●		●
attribute2(r) ∧ attribute3(r)	●	●	●		●	●	●	●

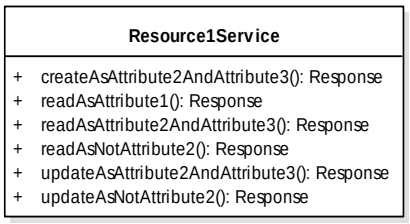

Figure 3. Entity Service for Resource #1 of Table 2

This is achieved by the creation of additional methods: Usually, one method is implemented for each operation on a resource. But by using our approach, methods are not only generated per operation but per operation and role (GetAsRole1, GetAsRole2, GetAsRole3, PostAsRole1, etc.). The difference is that each method can only validly be used by exactly one role and not by all roles possible. So far, the attack surface stays the same. The reduction is then reached by not implementing those methods that do not have a bullet in the access control matrix of, e.g., Table 1.

B. *Deriving Abstract Service Interfaces from Attribute-Based Access Control*

Applying the approach to ABAC extends the principles of the application to RBAC.

In the first step, all applicable operations for each resource of R are listed as columns in the access control matrix. Every policy rule of the canAccess() functions is listed as row. Every cell for which a canAccess() function is true is marked. A possible result shows Table 2.

Deriving the interface from Table 2 works similarly to the role-based approach: A service interface is created for each resource. In every service, operations are created for all allowed operation. Example operations from Table 2 are readIfIsAttribute1, updateIfIsNotAttribute2 and deleteIfIsAttribute2AndAttribute3 (see Figure 3). To prevent long and complicated method names, it is best practice to derive canAccess() rules from single attributes only whenever possible.

C. *Application on Authorization Patterns*

Sections III.A and III.B show how service interface methods can be derived for ABAC and RBAC. This section shows that the method is applicable for any kind of authorization.

In the sections on RBAC and ABAC, there are two limitations. First, the service interface methods are derived from access control matrixes for RBAC and ABAC. Second, because of the scenario and REST compliance, we used entity services [23] using only basic CRUD-operations. Both limitations are not necessary and can be generalized.

Concerning the first limitation, the abstract security pattern Authorization defines who may access protected resources in which way [2]. The access control matrix contains the description of the entity (who) on the first column of a row, the resource to access (what) on top of the

column and how the resource shall be accessible below the resource. Therefore, an access control matrix, as used it before, can be created for every kind of authorization.

Deriving the abstract service interfaces from these access control matrixes can be achieved as previously shown. Create a service interface for each service with operations combined to the permission. The name of the operations can be of any kind, thus not only CRUD-operations are applicable.

D. Maintaining REST Compliance

In order to comply with the previously presented REST constraints, we propose to not realize the derived service interface methods with extended HTTP-operations. Quite the contrary: REST relies on a defined and pre-known set of operations – namely GET, POST, PUT, DELETE, etc. when using HTTP. Introducing new operations restricts the API usage to insiders, thus, adversely affects the interface's uniformity and universality. It is also hardly possible in practice when using HTTP, since custom methods are not supported by browsers or most clients [21].

It is furthermore not advisable to realize the derived methods by using custom HTTP headers. To send a "X-Role: Administrator" header with every request seems practical on the first sight. But whitelist-based firewalls and proxy servers will skip those custom headers [24] limiting the API usage to clients that don't rely on a firewall. This kind of limitation is not acceptable.

However, a third way exists: We propose adding the service operation name to the request URI. Illustrating HTTP requests using the examples from above could then look like this:

```
POST /resource1/?authorization=createAsRole3
DELETE /resource2/?authorization=deleteAsRole3
…
GET  /resource1/?authorization=readIfIsAttribute1
PUT  /resource1/?authorization=updateIfNotAttribute2
…
```

This is legal in the HTTP standard and does not violate the interface uniformity constraint of REST compliance. The server extracts the information from the parameter – a task possible with every framework and scripting language. Diligence is required in the implementation: The parameter must not have a fallback for an invalid or missing value. If that is the case, an error has to be thrown. Otherwise, the attack surface is not reduced for the simple reason that it does not differ from the traditional implementation.

TABLE III. ACCESS CONTROL MATRIX OF USER AND GROUP RESOURCES OF THE PARTICIPATION SERVICE

	User				**Group**			
canAccess(s, r, e)	***C***	***R***	***U***	***D***	***C***	***R***	***U***	***D***
Guest(s)	●							
Authenticated(s)		●			●	●		
User(s) = r			●					
User(s) = Owner(r)							●	
Admin (s)		●	●			●	●	

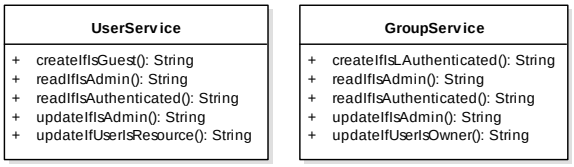

Figure 4. User and Group Service derived from access control matrix shown in table 3

An appropriate error communication for that case and for the case of using a not allowed permission on the specific resource, is responding with HTTP's status 405 Method Not Allowed. At first sight it seems uncommon to respond with a method-related error code to a missing or falsely specified parameter. However, as the parameter is merely an extension of the method according to the approach of this paper, it is suitable here. The list of "allowed methods" (more precisely: method and value for the authorization parameter) can be supplied in the body of the HTTP response. As a result it is possible to follow the Hypertext-As-The-Engine-Of-Application-State (HATEOAS) paradigm.

IV. EVALUATION

In this section, we apply the method on the Participation Service of the KIT-SC system, show the resulting web services and give an evaluation. The Participation Service is developed by seven students during a practical course at the KIT. The group was divided into two teams, one focusing on the HTML 5 frontend and the other focusing on the Java backend.

At the beginning the requirements for the service were collected. All required subjects S, resources R, environments E and their attributes SAn, RAn and EAn were identified and the access control matrix was built. Possible subjects are anonymous users and authenticated users. This publication demonstrates the method on the User and Group resources only, leaving out all other resources of the Participation Service for the sake of shortness.

According to the requirements, both, users and groups, can be created, edited and displayed. Deletion is solved by setting a status flag to deactivated, thus, by updating the resource. The access control matrix in Table 3 shows the authorization rules based on ABAC. Users can be created by guests. An authenticated user can read user account data, create groups and read them. The owner of an user or group account can update its information. User with the admin flag are allowed to read and update users and groups.

Figure 4 shows the derived abstract service interfaces from the access control matrix of Table 3. For each resource a service is modeled with the operations according to the access control matrix. This implies, that the services do not have operations for deleting the resources, because no authorization rule exists for this operation. Typically the delete operation would still be implemented, but inaccessible due to the enforced authorization. According to [9], this mapping is a reduction of the attack surface.

The abstract service interfaces are then mapped to the REST services with URLs as follows:

For the User Service:
POST /user/?authorization=createIfIsGuest
GET /user/?authorization=readIfIsAdmin
GET /user/?authorization=readIfIsAuthenticated
PUT /user/?authorization=updateIfIsAdmin
PUT /user/?authorization=updateIfUserIsResource

For the Group Service:
POST /group/?authorization=createIfIsAuthenticated
GET /group/?authorization=readIfIsAdmin
GET /group/?authorization=readIfIsAuthenticated
PUT /group/?authorization=readIfIsAdmin
PUT /group/?authorization=readIfUserIsOwner

The Spring Security project was chosen to enforce the authentication and authorization of the KIT-SC. Authorization is implemented by adding the annotation *PreAuthorize* to each entry point of the corresponding URL. These annotations contain the access policies as Spring EL expressions, which are evaluated by Spring Security to enforce access control. Spring EL offers the possibility to state expressions on the attributes of resource and subject. Thus, the patterns delivered in the request, formerly introduced by our method, can be used to formulate the Spring EL statement.

Using the approach of this paper in combination with Spring Security proved to be a good choice for many reasons:

The attack surface metric of [7] has been improved. The access control parameter *rights* of the parameter family *access control* has been reduced from 10 to 0 or 5, depending on the privileges of the operation.

Moreover, enforcing the authorization is easier, because testing functionality and access decision can be combined. For example look at the third row of Table 3. The user shall only be able to update its account. This constraint can be implemented and tested quite easily. Further, for enforcement of this policy, just the ownership has to be validated. This is quite easy, because the user data is delivered in the request. Without this limitation, the information must be collected separately. Thus, with a generic update operation, for each user touched by an operation call, every policy has to be enforced and corresponding data has to be fetched.

Additionally, frontend developers benefited from associating the authorization to HTML forms, buttons and links. By choosing which operation to call, they get sensitized to security. Following the principles of secure interaction design [12], they added confirmation messages, warnings, colors and icons to the user interface according to the security level of the different operations used.

V. Limitations

Regarding goal 6, the method is based on at least three assumptions. First, the authorization may be exposed to the users of the web service and, thus, also to attackers. This may be a threat for the web service or even a problem regarding federation. We assume, that the system is secure, even if the attackers have this information, according to Kerckhoffs's principle for crypto-systems [27]. Thus, this information may be exposed without making the web service insecure. Despite this, exposing the information can be impossible. In this case, the web service operation name has to be obfuscated or the introduced method cannot be applied.

Second assumption is, that the count of authorization rules for a single web service does not exceed. The policies defined by ABAC can be fine-grained using complex expressions. All these fine-grained policies lead through our approach to at least one service interface operation. In large systems this may be a great overhead. Many operations with potentially long names could be introduced. For example operations with similar functionality need an agnostic internal method to avoid redundancy and more methods and tests have to be implemented by the developers.

Third assumption raised by goal 6 is, that the authorization rules do not change periodically or often. A change in the authorization rules may lead to changes in the web service operations and can cause changes in systems using the web service, when using the method. This depends on the change and on the mapping of the abstract interface to the language depend web service interface. In our REST mapping, the URL does not change, but a new parameter may be introduced. In this case, changing authorization rules do not lead to changes in systems using the web service. Even so, the web service has to be enhanced including overhead.

Additionally, the approach introduced is systematical, but we have not used a language to describe access control policies. This is because we could not find a suitable language. Possible candidates are the Unified Modeling Language with SecureUML [4] and UMLsec [25] or the Ponder Policy Specification Language (PPSL) [26]. But UMLsec and SecureUML need to be enhanced, to support every kind of authorization. PPSL is not based on the UML and has no visual representation, but we think both are important prerequisites so that the approach is used.

Another limitation concerning REST is the restricted functionality of HTTP's OPTIONS method. An OPTIONS call to a resource is responded with a list of allowed methods on that resources and using one of them should not result in a 405 Method Not Allowed error code. However, after applying this paper's approach, the method name is not sufficient to formulate valid requests – information about valid authorization parameter values are required (see Section III). The response is expressed in a list of comma-separated HTTP methods and there seems to be no possibility to additionally provide parameter values.

VI. Conclusion and Future Work

We introduced a new way of designing interface methods by using security patterns. For this method, we showed that the attack surface on the interface is minimized according to the least privilege needed. Additionally, we showed how to combine the method with the REST paradigm and therefore, create REST-compliant web services.

The application of the method was shown within the Participation Service of the KIT-SC. In this application, at least the disadvantage of creating many interface methods by applying our approach arose. However, the attack surface has been reduced. By giving a mapping from the technology independent web service to a RESTful web service, the approach facilitates a REST-compliant Participation Service.

The approach gives software architects the possibility to improve the safety of web services using authorization patterns. They can follow instructions to improve quality attributes of the application in a systematic way without having a security background or knowledge.

Software developers using the derived service interface are aware of the privileges when using interface methods. This increases the security according to secure interaction design. Furthermore, the implementation of the service interface can be easier tested, because the authorization offers constraints for the operation to be implemented.

The disadvantage of creating many service interface methods may be the focus of future work. For instance, this phenomenon could be avoided by combining similar rights for the same object to one service interface method. Another starting point for future work is to research the advantages of the static in contrast to the dynamic access decisions. This can lead to an improved performance, improved security through easier testing and easier externalization of access decisions.

Our main goal is to combine the usage of security patterns with quality attributes. This can lead to more precise predictions on the quality of software. Therefore, non-functional requirements of stakeholders can be considered during the design of an application. By offering systematical methods, the quality can be ensured among the phases of the software development.

REFERENCES

[1] G. McGraw, "Software Security," IEEE Security & Privacy, pp. 80-83, Mar.-Apr. 2004, doi:10.1109/MSECP.2004.1281254.

[2] M. Schumacher, E. Fernandez-Buglioni, D. Hybertson, F. Buschmann, and P. Sommerlad, "Security Patterns: Integrating Security and Systems Engineering," John Wiley & Sons, Dec. 2005, ISBN: 978-0-470-85884-4.

[3] B. Alshammari, C. Fidge, and D. Corney, "Security Metrics for Object-Oriented Designs," IEEE 21. Australian Software Engineering Conference (ASWEC), Apr. 2010, pp. 55-64, doi:10.1109/ASWEC.2010.34.

[4] D. Basin, J. Doser, and T. Lodderstedt, "Model Driven Security: from UML Models to Access Control Infrastructures," ACM Transactions on Software Engineering and Methodology (TOSEM), vol. 15, Jan. 2006, pp. 39-91, doi:10.1145/1125808.1125810.

[5] P. Manadhata, K. Tan, R. Maxion, and J. Wing, "An Approach to Measuring A System's Attack Surface," Carnegie Mellon University, Aug. 2007 [online]. Available from: http://www.dtic.mil/cgi-bin/GetTRDoc?Location=U2&doc=GetTRDoc.pdf&AD=ADA476805 [retrieved: 23.09.2014]

[6] B. Alshammari, C. Fidge, and D. Corney, "Security Metrics for Object-Oriented Class Designs," IEEE 9th International Conference on Quality Software, Aug. 2009, pp. 11-20, doi:10.1109/QSIC.2009.11.

[7] T. Heumann, J. Keller, and S. Türpe, "Quantifying the Attack Surface of a Web Application," In Proceedings of Sicherheit 2010, vol. 170, 2010, pp. 305-316, ISBN: 978-3-88579-264-2.

[8] G. Sumit, R. K. Nabanita, Mukesh, S. Saurabh, and M. Pallavi, "Reducing Attack Surface of a Web Application by Open Web Application Security Project Compliance," Defence Science Journal, vol. 62(5), Sep. 2012, pp. 324-330, doi: 10.14429/dsj.62.1291.

[9] M. Howard, "Attack Surface – Mitigate Security Risks by Minimizing the Code You Expose to Untrusted Users," MSDN Magazine, November 2004. [Online]. Available from: http://msdn.microsoft.com/en-us/magazine/cc163882.aspx [retrieved: 23.09.2014]

[10] A. Kurmus, A. Sorniotti, and R. Kapitza, "Attack Surface Reduction For Commodity OS Kernels: trimmed garden plants may attract less bugs," in Proceedings of the Fourth European Workshop on System Security (EUROSEC '11), Apr. 2011, pp. 1-6, doi:10.1145/1972551.1972557.

[11] A. Bartel, J. Klein, and M. Monperrus: "Automatically Securing Permission-Based Software by Reducing the Attack Surface: An Application to Android," in Proceedings of the 27th IEEE/ACM International Conference on Automated Software Engineering (ASE 2012), Sep. 2012, pp. 274-277, doi: 10.1145/2351676.2351722.

[12] K. Yee, "Guidelines and Strategies for Secure Interaction Design," Security and Usability: Designing Secure Systems That People Can Use, pp. 247.273, Aug. 2005, ISBN: 978-0-596-00827-7.

[13] R. Fielding, "Architectural Styles and the Design of Network-based Software Architectures," Dissertation, University of California, Irvine, 2000, ISBN: 0-599-87118-0.

[14] M. Gebhart, P. Giessler, and P. Burkhardt, "Quality-Oriented Requirements Engineering for Agile Development of RESTful Participation Service," in press.

[15] E. Yuan and J. Tong, "Attribute Based Access Control (ABAC) for Web Services," in Proceedings of the International Conference on Web Services (ICWS), Jul. 2005, pp. 561–569, doi:10.1109/ICWS.2005.25.

[16] ISO/IEC, "ISO/IEC 25010:2011(E) Systems and software engineering – Systems and software Quality Requirements and Evaluation (SQuaRE) – System and software quality models," 2011.

[17] R. Steinegger, "Authentication and authorization patterns in existing security frameworks [Authentifizierungs- und Autorisierungsmuster in bestehenden Sicherheits-Frameworks]," diploma thesis, Karlsruhe Institute of Technology, Karlsruhe, Germany, 2012. German.

[18] C. Pautasso and E. Wilde, "Why is the Web Loosely Coupled? A Multi-Faceted Metric for Service Design," in Proceedings of the 18th international conference on World wide web (WWW '09), Apr. 2009, pp. 911-920, doi:10.1145/1526709.1526832.

[19] A. Caballero, "Computer and Information Security Handbook," Morgan Kaufmann Publications, 2009, ISBN: 978-0123743541.

[20] W3C, "Web Services Glossary," Feb. 2004. [Online]. Available from: http://www.w3.org/TR/2004/NOTE-ws-gloss-20040211/#webservice [retrieved: 23.09.2014]

[21] L. Richardson and S. Ruby, "RESTful Web Services", O'Reilly Media, May 2007, ISBN: 978-0596529260.

[22] ISO/IEC, "ISO/IEC 25000:2005(E) Software Engineering – Software Product Quality Requirements and Evaluation (SQuarE) – Guide to SQuaRE," 2005.

[23] S. Cohen, "Ontology and Taxonomy of Services in a Service-Oriented Architecture," Microsoft Architect Journal, Apr. 2007.

[24] A. van Kesteren, “HTTP methods, Web browsers and XMLHttpRequest,” Oct. 2007. [Online]. Available from: http://annevankesteren.nl/2007/10/http-method-support [retrieved: 23.09.2014]

[25] J. Jürjens, “UMLsec: Extending UML for Secure Systems Development,” Lecture Notes in Computer Science, vol. 2460, pp. 412-425, Sep, 2002, doi:10.1007/3-540-45800-X_32.

[26] N. Damianou, N. Dulay, E. Lupu, and M. Sloman, “The Ponder Policy Specification Language,” in Proceedings of the International Workshop on Policies for Distributed Systems and Networks (POLICY '01), Jan. 2001, pp. 19-37, ISBN: 3-540-41610-2.

[27] Auguste Kerckhoffs, “La cryptographie militaire,” Journal des sciences militaires, vol. IX, Jan. 1883, pp. 5–38.