Object Oriented Role-Based Access Control

Petr Stipek, Lukas Kralik, Roman Senkerik
Faculty of Applied Informatics
Tomas Bata University in Zlin
Zlin, Czech Republic
Email: stipek@fai.utb.cz, kralik@fai.utb.cz, senkerik@fai.utb.cz

Abstract — **This paper focuses on issues related to Security Design and Access Control in Object-Oriented Software projects by pointing out some common implementation problem sources, and their solutions. Further, the study presents an innovative way of extending the Role-Based Access Control (RBAC) Model for large and dynamically-growing projects. Specifically, the emphasis is placed on Scalability Allocation Rights to users, based on their roles. The proposed approach seeks to minimize the bindings of Application Logic from the Functional Logic Allocation and the Verification of Individual Rights.**

Keywords - software security; object-oriented programming; weakly-typed languages; ACL;RBAC; CRUD; ORM/ODM

I. INTRODUCTION

Ensuring security against unauthorized access is an integral part of nearly all systems. This is evident not only in security demands on simple claims relating to displaying selected parts of applications to user groups, but also in the very sophisticated - and interdependent relationship between the rights of users. Typically, the gradual expansion of systems allows modifications, which are consistent with the software evolution processes. Each phase of the evolution presents advantages - as well as difficulties that might potentially force developers to violate or abandon proven concepts regarding the fulfillment of the requirements of a final product. A common challenge that devlopers face is dealing with an inconsistent design that leads to a complex development and maintainance of the system. For example, while the maximum utilization of Integrated Development Environment (IDE) tools can perform code-refactoring, this is only effective in cases when careful documentation, via annotations, is adopted. This way, one avoids writing control symbols via primitive data types - which inherently may cause needless financial expenses.

There are many ways to design an Access Control List (ACL) [3]. Different combinations also exist for approaches - including RBAC [1][8]; Attribute Based Access Control (ABAC)[9]; or approaches based on the Create, Read, Update and Delete (CRUD) Operations [2][5][6]. Basically, it is either a user - or a group of users with allocated roles who can be assigned, or have permission to, or be withdrawn access to a part of a system.

In Section 1, the basic principles regarding what should be followed or held in the design of ACL are described. This is followed by a comparison between generally-used design patterns and mechanisms. Section 3, presents the main disadvantages of ACLs. In Section 4, an Object-Oriented Approach, suitable for appliactions using Object Relational Mapper/Object-Document Mapping, (ORM/ODM), is also presented. Finally - in Section 5, the Performance Impact of our proposal is discussed.

II. BASIC PRINCIPLES

This section presents some basic principles underlying the preparation of applications´ security structures. This approach helps to consider a few choices and takes into consideration our own requirements to select the best approach.

A basic presumption in effective design approaches is that all dependencies of the Application Logic from the users, user-accounts, and their roles, are removed. For instance, in an Invoice Price Calculation Model, the user or their role, is generally considered irrelevant. Rather, what is more important, is the knowledge of what operations can or cannot run. This means that whether or not the current user fulfills the conditions necessary for authorization, an authorization service that provides and manages the current user account, according to law and regulations has to be provided. At the moment, when a project reaches a state where it is necessary to set a security policy, quite a number of developers tend to advance the implementation of security policy in the code on the basis of customer specifications using an authentication service.

While there is nothing wrong with this process in principle, a common problem often surfaces in the later stages of development. This challenge, in particular, is related to creating information about user accounts - or their roles, in the code and in places where it would be needed to access user-roles instead of asking the authentication services; whether the specified permissions are assigned or not (See comparison in Figure 1).

Running both approaches will lead to the same results with negligible performance impact. Fixing roles in the model however, results in a scattered security policy throughout the system instead of being managed centrally [9][10]. In case of any change to the security policy, the entire code must be revised and all the potential occurrences must have to be checked. Such a system is more inclined to errors due to improper authentication – and, it is far more difficult to maintain the consistency of the overall security policy documentation of system roles.

Copyright (c) IARIA, 2016.

```php
class EntityModelOne
{
    /** @var IAuthorizator */
    private $authorizator;

    public function __construct(IAuthorizator $authorizator)
    {
        $this->authorizator = $authorizator;
    }

    /**
     * Creates new record only if user has a permission.
     * @param array $data
     * @throws AuthenticationException
     */
    public function createNew(array $data)
    {
        if (!$this->authorizator->isAllowed("createNewEntity"))
            throw new AuthenticationException();

        // Method content
    }
```

```php
class EntityModelTwo
{
    /** @var IAuthorizator */
    private $authorizator;

    public function __construct(IAuthorizator $authorizator)
    {
        $this->authorizator = $authorizator;
    }

    /**
     * Creates new record only if user has Admin role.
     * @param array $data
     * @throws AuthenticationException
     */
    public function createNewRecord(array $data)
    {
        if (!$this->authorizator->isInRole("AdminRole"))
            throw new AuthenticationException();

        // Method content
    }
```

Figure 1. Authentication during the Creating of a new Record in the Modeling, Verification of Rights (viz left); and Verification by Role (viz right)

A. Permission collision avoidance

A modern trend in applications development has to do with the design of modular applications with completely separate and independent components. This is particularly evident in Open Source projects, where hundreds of different developers create modules for a specific framework. This trend increases the potential risk of permission collision when composing the application. Since prefix titles are often used in prevention, there is always a real risk of missing these out in the assignment of prefixes. It is prudent therefore, to anticipate permission collisions during compiling or testing - when the application can fail, rather than in the production version - when full operation with client data is used.

This is consistent with the reasons advanced above for the introduction of the term “permissions resources”, which essentially divides privileges into smaller units – thus minimizing the risk of collisions. For weakly-typed languages, these resources are defined as a text-string. However, a much better way is to use objects like structures in strongly-typed languages, so that the textual expression resource name can be replaced for the entire class name; serving as a source of authority (See comparison in Figure 2). In case of building a program that would include two classes of the same name, an exception occurs when one compiles it - and the program will not even start.

```php
/**
 * Shows record if user has a permission.
 * @param int $id
 * @throws AuthenticationException
 */
public function showRecord($id)
{
    if (!$this->authorizator->isAllowed("entityResource"
        , "read")) {
        throw new AuthenticationException();
    }

    // Method content
}
```

```php
/**
 * Shows record if user has a permission.
 * @param MyEntity $entity
 * @throws AuthenticationException
 */
public function createEntity(MyEntity $entity)
{
    if (!$this->authorizator->isAllowed($entity
        , MyEntityResource::CREATE)) {
        throw new AuthenticationException();
    }

    // Method content
}
```

Figure 2. Avoiding Resource Permission Collisions; Text Form (viz left); Object Form (viz right)

B. Application of CRUD operations

With the entry of ORM [4] tools for mapping database data on the object-structure in applications, another layer nestled between the model and the database containing the repositories and services is formed. This is essentially designed to work with the entities. At the same time, there are attempts to unify the implementation of the authorization process with the interlayer - consistent with basic database operations, e.g creating, reading, editing and deleting records. For each entity, four permissions were created using for which the developers implemented the security function. This allowed the creation of generic class managing entities (Figure 3), thereby significantly reducing the spread of homogeneous source codes, and speeding up its development in a system with a large number of entities.

In most cases, these operations are quite enough. For example - in the Web-content management system context, in this way we manage the application development lifecycle. But which of the permissions does one require, for example, to publish a page by a Senior Editor?

Is it an operation to create or edit? In this case, we need help by creating additional permissions.

Copyright (c) IARIA, 2016. ISBN: 978-1-61208-493-0

```
abstract class GenericService
{
    /**
     * Returns class of generic entity
     * @return string
     */
    abstract protected function getEntityClass();

    /**
     * Creates new record from entity
     * @param $entity
     * @throws AuthenticationException
     */
    public function createEntity($entity)
    {
        // Check if entity is equal to generic entity class
        $this->assertEntitityType($entity);

        if (!$this->authorizator->isAllowed($this->getEntityClass(), "create"))
            throw new AuthenticationException();

        // Method content of creation
    }
    // Other methods
```

Figure 3. Demonstration of a Generic Service for Managing Weakly-typed Language Entities

C. *Misprint minimisation*

Man is a fallible creature, and it is very easy to make a misprint in writing code. If a programmer makes a mistake in the source code, the compiler reports an error. In most cases, the IDE in which the application is being developed, posts the error directly. However, if we connect the information controlling the program logic to text strings, then there is no better tool for performing such compilations. When this occurs, not even a robust IDE is able to estimate whether it is just text for later “bubbling” to the user; or to control characters. Object design is a popular approach in many systems - but not all developers can fully understand this approach and utilize all of the benefits that it brings. Occurrences of control character sequences are more advantageous to bind in constants tied to objects which have to be applied to them or semantically related. The added value is used for accuracy verification by the compiler so as to detect a misprint; while simultaneously, the IDE will offer its lists by enabling one to interactively cooperate with constants. Some developers however, reject this approach because it creates redundant writing – i.e. the extra burden to rethink how and where to place constants, or have no experience with good working practices (especially developers working with weakly-typed languages).

III. THE DISADVANTAGES AND LIMITATIONS OF ACL

The above-mentioned procedures are suitable for most applications in practice. However, owing to their functional principles, there are restrictive limits that are particularly felt in large and modular systems.

A. *Violation of the Single Responsibility Principle*

Too often, the open concept allows developers to design a system carelessly - instead of using best-practice principles, which would ensure the better sustainability of the system throughout its life-cycle. Most developers make errors to a varying degree when writing code and begin to merge the application´s object-structure, thereby limiting readability, scalability and testability. This increases the risk of error. An example can be data entities, to which constants are added and used for access control, instead of defining objects exclusively for this purpose and thus minimizing binding in the system.

B. *Gross Allocating Rights*

Access control does not necessarily influence the accessibility of specific records. If one needs to grant user access just to certain articles in the Content Management System, one can either set the rights for all - or for none. This can be done with definitions depending on the user (for example, a property right, the position of the head against the author, etc.). But if one wants to add access to an item that does not exist in the system´s logical connection to the user or their role, then this cannot be achieved. Further, the introduction of auxiliary information for approach management violates the Single Responsibility Principle (SRP).

C. *Keeping the documentation and permissions management*

It may seem that, in the documentation process, nothing is inherently damaged. A separate document is created that describes the rights of individual roles in the system, and appropriate comments are created in the source code. Unfortunately, experiences from practice demonstrate that these mechanisms do not always work. Often - under pressure, there are sudden changes, communication noises, and all these changes are either completely undocumented or not commented on in the system. Overall, the principle of keeping two documents is difficult to maintain. Rather, it is more suitable to structure an application so that both could be managed uniformly and centrally; though it will rely on information from source-codes. When designing a unified

standard that contains all the key information; the creation of automatically-generated documentation that is updated after every intervention in the system is not considered to be a big problem. This may simultaneously build more milestones for the development of safety measures.

The whole situation of the doubling of the documentation may still be complicated by the need for the creation of administration rights for management authority, where there is also a need to rewrite information about the function and impact of individual rights.

IV. THE OBJECT-ORIENTED DESIGN OF ACCESS CONTROL

In the previous section, it was shown how to use an object-oriented approach to improve the development of an ACL. So it is valuable considering how to compose a concept that would create some sort of framework to manage permissions. Additionally, framework features also include definitions of the scope of the proceedings - separated from the application logic, thus allowing scalability and self-documentation.

First, it is necessary to clarify several major changes and their impact on the structure of the safety logic.

A. Resource Permission Abstraction

The classical ACL model is restrictive due to the subtleties of how to assign permissions and owing to the fact that we have verified them against the object classes [3]. If we used a system where each entity uses a unique identifier (see Figure 4) within its own class, thereby defining their common interface to be able to obtain this key, and the transfer of specific instances of objects, one is able to obtain the name of the source that is its unique identifier. This then serves for the assignment of authorization services to obtain information on this source and to return a message saying if permission is set or not.

It must be noted that even standard entities either have a single identifier, or obtain one to perform a set of operations with relational-data or data-dependent objects. Solutions can be found in resolvers registrations for a particular object-type specified class or common interface. When creating a resource name, a resolver is necessary; and can be obtained from the specified object.

It is also necessary to convert the source-object into text or numbers so that the authentication service will be able to manage these objects in the database and to search for them. We could also store entire objects - but this approach is only suitable for document-oriented databases like MongoDB.

```
public function createEntity($entity)
{
    // Check if entity is equal to generic entity class
    $this->assertEntitityType($entity);

    if (!$this->authorizator->isAllowed($this->getEntityClass(), "create"))
        throw new AuthenticationException();

    // Method content of creation
}
```

Figure 4. Replacement of a Permission Resource by an Entity

B. Rights specification

The term 'resource permissions' represents a set of rules, settings and related information on how to handle data (see Figure 5) was introduced above. Principally, via this step, an attempt has been made to separate the security information objects outside the application logic and to form the basis for the documentation of the individual permissions. This source tells us - by entity or class of data shields, how to obtain information about a particular record (instance), under which the resource and its formal description fall hierarchically. It also allows one to create a collection of permissions that can be allocated over the object and verify whether they are associated with roles in the system. Another important benefit of this proposal is its ability to structurally rank these in hierarchies – not only as individual resources, but also permissions. Also, the entire system can be divided into logical units and an overall map of all privileges can be created. Ultimately, the outcome may generate documentation or create a tool that allows for the allocation and revocation of privileges because all of the information is managed in one place.

```
class MyEntityResource implements IResource {
    const READ = "read", CREATE = "create", UPDATE = "update", REMOVE = "remove";
    /** Name of parent resource for a hierarchical arrangement
     * @return string */
    function getParentResourceClass(){
        return MyModuleResource::class;
    }
    /** Returns target class type
     * @return string */
    function getTargetCLass(){
        return MyEntity::class;
    }
    /** Closure accessing entity primary key
     * @return Closure */
    function getPrimaryKeyGetter(){
        return function (IEntity $entity) {
            return $entity->getId();
        };
    }
    /** General name for documentation
     * @return string */
    function getName(){
        return "My entity";
    }
    /** General description for documentation
     * @return string */
    function getDescription(){
        return "CRUD operations over the My entity";
    }
    /** Setup all entity privileges definition
     * @param PrivilegeCollection $privileges */
    function setupPrivileges(PrivilegeCollection $privileges){
        $privileges->addGlobal(self::READ, "Show record", "Description");
        $privileges->addGlobal(self::CREATE, "Create op.", "Detailed description");
        $privileges->addGlobal(self::UPDATE, "Update op.", "Detailed description");
        $privileges->addGlobal(self::REMOVE, "Remove op.", "Detailed description");
    }
```

Figure 5. Sample Resource Permissions for One Entity

C. *Expansion Permission Problems*

Extending control permissions to a specific instance and data brings with it a big problem. This is termed the “default permissions” before first starting the system. The more options one has - the more “permissions” one needs to initialize. To simplify this process one has to rely on the advantage of the inheritance of both user roles and individual permissions. With a suitable algorithm, one can set the authorization service so that it is cumulatively associated with the higher-level roles and the permission subordinate roles [7].

This procedure is sufficient to define permissions on the lowest layers of the tree structure (Figure 6) and roles in the upper layers. This only defines additional permissions, which arise just for that role, and 'bubble up' to the other parent layer.

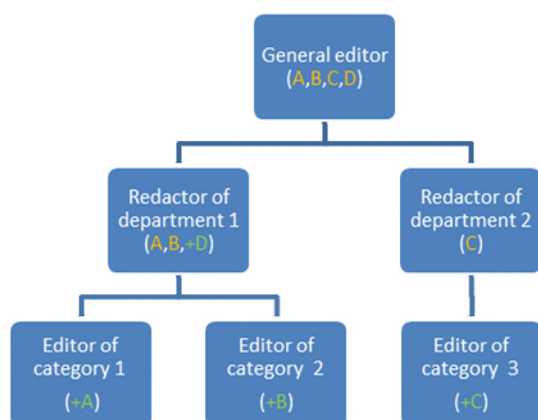

Figure 6. Cumulative Assignment of Permissions to Parent-roles (Green: Defined manually; Orange: inherited)

D. *The Multiple Assignment of Rights to Overall Resources of the Same Type*

By restricting the permissions to the distinction between instances, one loses the ability to mass configure the rights of target group resources. This limitation can be compensated for by the “inheritance” of individual permissions within one source - or privileges superior to the source. Logically, there is a possibility to divide these into Local Law (i.e. applied individually over instances); and Global Law (i.e. applicable to all instances) groups. Altering the setting of inheritance rights from local to global eliminates this restriction. For authentication services, it is necessary to know how to work with these additions. Inheritance can also be used to restrict the necessary definition permissions over their resources. For example, by assigning rights on the creation of article categories, one wants to enable someone to create individual articles within this category.

In contrast to roles, rights calculation must be performed in the opposite direction from the highest layer to the lowest. At the same time, it is not enough to work on only the grantgranting and refusing access - but a third, neutral state, must be introduced with the right to take over from the parent permissions.

E. *Creating Resources on the Run*

In order to fulfill the functionality of the above points, the implementation mechanism to manage these resources is still missing.

This requires a solution which offers a manual implementation approach to all services that manage entities and to data for which access needs to be managed. It would be necessary to implement at least a ‘create a resource’ entity after creating and deleting a source before deleting entities. One can include call changes and record any events; but it is not necessary to ensure that this concept will work.

Manual implementation can be dispensed with by using the abilities of some ORM/ODM implementations – i.e. so-called “listener” or “subscriber” services that invoke special application extensions on selected groups of objects that are triggered when changes in state entities occur. Their purpose is simple and built on objects´ additional events, without affecting the integrity of their content and functionality.

F. *Ownership of Resources and Events*

In some cases, we need to decide the access to a resource based on information regarding its ownership. It is very questionable whether the owner information should be part of the functional logic - or a component of the management approach, since this data is often only used as functional logic to filter the records or to access relational records. Their movement outside the influence entity would complicate querying databases. On the other hand, it would turn into a violation of the Single Responsibility Principle, so that the information about the owner should be stated at the source, not the entity. Both cases, however, can be resolved relatively quickly so that the resulting behavior will be similar. If we want to note the information about a property with entities, then we have to note the method by which the owner is obtained. In higher programming languages, we use the Lambda expressions or Closures for this purpose to advantage. Setting the property will be part of the functional logic.

For property management inside these resources purposes, we can automate it by just slipping the logged user object to the authorization service through which it obtains this identity and assigns it a new source. To simplify folding database queries, one needs to create an object that returns a partial database command connecting the required tables, which will simply be included into the desired filter command.

In the same way as a property, we can also keep information about the latest update, or delete the record.

V. OBJECT ACCESS PERFORMANCE IMPACTS

The crucial question however, is how this approach will have an impact on the application performance.

One can notice the significant impact when the call first acts on database queries. The percentage impact is very difficult to calculate and depends on the complexity and size of the entire model.

Negative impacts can partially cancel out the pre-calculations and there is a need for a suitable caching intermediate results application and for the results for each

Copyright (c) IARIA, 2016. ISBN: 978-1-61208-493-0

role in the system. After application caching, let us move on to the complexity of the search at list-level.

Another negative effect is due to the fact that there are doubled insert and delete commands to the database in the case of the creation and deletion of records. This concept is unsuitable for example, for monitoring systems - but rather, will assist in the development of Customer Relationship Management (CRM) systems.

The concrete results and ensuing comparison of the performance impact model applications - at least, are not yet known, because this model is currently in the testing phase.

VI. CONCLUSION

The main advantage of the above-mentioned approach is the centralization of security logic and related documentation in an ideal case as separated models from application logic. It allows one to have a greater detailed and more sensitive control of access to resources, (applications), without the side-effect of the expansion of privileges because of their structuring options due to heredity and to relations that are defined only in security logic. Additionally, the approach also helps in the production of more effective code by means of developer tools.

Future work will focus on three key areas, herein below:

Firstly, the work will focus on how to make preprocessed combinations of privileges, roles and all of the relations between them. This would be ideal for boosting the performance of authorization services and the minimization of latency.

Secondly, the focus will be on designing a security policy documentation generator, based on structure and definitions of all permissions - throughout all resources. This approach would generate feedback about the range and complexity of the (given) security policy.

Thirdly, we will focus on the creation of a Security Coverage measuring tool that will be able to analyze source codes and generate feedback about the degree of security (insecurity). It will also focus on the concrete role of access - or permission, requirements for accessing any part of a code. This would serve as a foundation of extant knowledge for developers.

ACKNOWLEDGMENT

This work was supported by:

Grant No.: IGA / Cebia Tech / 2015/036, Tomas Bata University in Zlin Internal Grant Agency.

REFERENCES

[1] D. Ferraiolo and R. Kuhn,“Role-Based Access Controls,” Baltimore, 15th National Computer Security Conference, 1992,pp. 554–563

[2] O. M. Pereira, M. Rui, R. L. Aguiar, and Y. M. Santos,“CRUD-DOM: A Model for Bridging the Gap between the Object-Oriented and the Relational Paradigms,” Fifth International Conference on Software Engineering Advances. IEEE, 2010,pp. 114-122, DOI: 10.1109/ICSEA.2010.25, ISBN 978-1-4244-7788-3

[3] R. S. Sandhu and P. Samarati, “Access control: principle and practice,” IEEE Communications Magazine (Volume 32, Issue:9),1994, pp. 40-48, DOI: 10.1109/35.312842, ISSN 0163-6804.

[4] H. Song and L. Gao,“Use ORM Middleware Realize Heterogeneous Database Connectivity,” Spring Congress on Engineering and Technology IEEE,” 2012, pp. 1-4, DOI: 10.1109/SCET.2012.6341925. ISBN 978-1-4577-1964-6.

[5] C. O. Truica, F. Radulescu, A. Boicea, and I. Bucur, “Performance Evaluation for CRUD Operations in Asynchronously Replicated Document Oriented Database,” 20th International Conference on Control Systems and Computer Science. IEEE, 2015, pp. 191-196, DOI: 10.1109/CSCS.2015.32, ISBN 978-1-4799-1780-8.

[6] O. M. Pereira, D. D. Regateiro, and R. L. Aguiar, “Distributed and Typed Role-based Access Control Mechanisms driven by CRUD Expression,” International Journal of Computer Science: Theory and Application, ORB Academic Publisher 2014, pp. 1-11, [Online] Available from: http://www.orb-academic.org/index.php/journal-of-computer-science

[7] A. A. Elliott and G. S. Knight, “Role Explosion: Acknowledging the Problem,” In Proceedings of the 2010 International Conference on Software Engineering Research & Practice, 2010

[8] C. Feltus, M. Petit, and M. Sloman, “Enhancement of Business IT Alignment by Including Responsibility Components in RBAC,“Proceedings of the CAiSE 2010 Workshop Business/IT Alignment and Interoperability (BUSITAL2010), 2010, pp. 61-75

[9] M. Munz, L Fuchs, M. Hummer, and G. Pernul, “Introducing Dynamic Identity and Access Managemen in Organisations,” 11th International Conference on Information Systems Security, 2015, pp. 139-158, DOI: 10.1007/978-3-319-26961-9_0

[10] R Sandhu, D Ferraiolo,and R Kuhn, “The NIST model for role-based access control: towards a unified standard,”, ACM workshop on Role-based access control, 2000

Copyright (c) IARIA, 2016. ISBN: 978-1-61208-493-0