ROAC: A Role-Oriented Access Control Model

Nezar Nassr and Eric Steegmans

Katholieke Universiteit Leuven,
Dept. of Computer Science and Engineering,
Leuven, Belgium
{nezar.nassr,eric.steegmans}@cs.Kuleuven.be

Abstract. Role-Based Access Control (RBAC) has become the de facto standard for realizing authorization requirements in a wide range of organizations. Existing RBAC models suffer from two main shortcomings; lack of expressiveness of roles/permissions and ambiguities of their hierarchies. Roles/permissions expressiveness is limited since roles do not have the ability to express behaviour and state, while hierarchical RBAC cannot reflect real organizational hierarchies. In this paper, we propose a novel access control model: The Role-Oriented Access Control Model (ROAC), which is based on the concepts of RBAC but inspired by the object-oriented paradigm. ROAC greatly enhances expressiveness of roles and permissions by introducing parameters and methods as members. The hierarchical ROAC model supports selective inheritance of permissions.

Keywords: Access Control, RBAC, Authorization, Role Hierarchies.

1 Introduction

The deployment of software applications on distributed networks and on the web has exposed them to many new security threats. One major risk is that an application can be accessed by unauthorized users in an easier way than in the past. Governments and commercial organizations are continuously seeking strong access control models that can help them prevent unauthorized access to their systems. Therefore, they maintain their reputation as safe institutions where confidential information is safeguarded. For example, WikiLeaks could have been prevented should better access controls have been in place [1]. Role based Access Control (RBAC) [2] has been used by organizations to protect resources in their software systems against unauthorized access. RBAC has become the dominant access control model that is widely accepted in enterprise, health, and governments systems.

RBAC is based on four principles: abstract privileges, separation of administrative functions, least privilege and separation of duties [3]. RBAC is expressed in terms of users, roles, permissions, objects and operations [4]. Permissions are assigned to roles and roles are assigned to users. Permissions are privileges to access objects or to execute operations. RBAC models often support role hierarchies. This feature is known as *hierarchical* RBAC. Role hierarchies define

I. Askoxylakis, H.C. Pöhls, and J. Posegga (Eds.): WISTP 2012, LNCS 7322, pp. 113–127, 2012.

partial orders on roles; this is analogous to inheritance in the object-oriented paradigm. The central advantage of RBAC is that it simplifies the management of access rights and offers a high level view on security in organizations by bridging the gap between functional requirements of organizations and the technical authorization aspects of their security policies [3], [5].

Despite robustness of RBAC, it has received a great academic attention from researchers. The literature shows many notable contributions that address limitations and suggest improvements to RBAC. However, in its current form, RBAC does not seem to have enough power to express a wide range of security requirements and capture fine access control granularity when put into practice [5]. Two main shortcomings of standard RBAC are its lack of expressiveness when defining roles [5], [6] and ambiguities that may arise in hierarchical role models [7]. Hierarchies in standard RBAC only support the *is-a* hierarchy which does not reflect real organizational hierarchies as we will see later. On the other hand, parametrized RBAC [5], [6],[8] has been proposed to address the lack of expressiveness by associating parameters to roles. Shortcomings related to role hierarchies were addressed by many initiatives. More discussions regarding this are contained in the next section.

Existing RBAC models consider roles as entities of a simple type that cannot have member attributes and operations, except parameters as suggested by parametrized RBAC. This provides a relatively simple and straightforward type for roles, but it lacks flexibility when defining roles. Roles in RBAC are blind in the sense that they are not aware of the application environment. They cannot access data in the system or perform any actions. Roles cannot hold variables, status, methods, etc. More so, the generalization concept in existing RBAC models does not reflect real organizational hierarchies. In most organizations, superiors do not need full access on permissions of their inferiors, and hence application of the *is-a* inheritance in these situations results in assignment of undesired privileges to superiors. This conflicts with the least privilege concept of RBAC. In many situations, senior users have supervision relations to junior users. Organizations are seeking flexibility when defining hierarchies in the access control model that can reflect the nuances above.

In this paper, we propose the Role Oriented Access Control model (ROAC) as a novel access control model. ROAC addresses limitations of existing RBAC models through benefiting from object-oriented concepts. ROAC makes analogies between roles and classes in object-oriented programming languages, then utilizes their concepts for constructing a new robust and extendible access control model. The main contributions of ROAC are:

1. To the best of our knowledge, ROAC is the most expressive access control model yet defined. ROAC greatly enhances the expressiveness of access control through associating variables and methods to permissions and roles. This architecture provides a means to defining one role and then defining multiple instances from the role with different levels of granularity. More so, application code is able to invoke methods to validate role parameters that are defined as part of roles permissions. This helps separating the access

control management from the application logic. This all results in stronger security and minimizes the risk of different interpretations of parameters among developers of the application.

2. ROAC greatly enhances RBAC hierarchies by adopting standard object-oriented inheritance concepts. At the same time, it extends hierarchical facilities with supervision relationships among roles. It also offers means for selective inheritance of permissions of junior roles by senior roles. In this way, ROAC better reflects organizational hierarchies. In other words, ROAC supports both the *is-a* and selective inheritance.

3. ROAC addresses scalability issues of existing RBAC models. In addition to the points mentioned above, ROAC provides a new kind of parameters, referred to as static parameters. Static parameters have common values for all instances of the role. Static parameters help updating all instances of the role at once. For example, if an organization often switches between two roles, it must be able to disable one type of role and enable the other type. A static parameter can then be introduced to specify whether or not all instances of the role are enabled. Moreover, by using validators, organizations can also provide assertions over the parameters and static parameters. Validators can also implement authorization policies that can be checked before authorizing operations. In ROAC, roles and permissions can hold state. Roles can connect to databases and can have data structures to hold data. This can be of great usage for auditing and tracking authorizations.

The remainder of this paper is organized as follows: In the next section we review related work, then in the third section we overview the ROAC model. In section four, we provide the data model of ROAC. In the fifth section we explain the generalization model of ROAC. Section six provides a discussion about how ROAC can implement next generation RBAC concepts and the trade-off between complexity and fine granularity. Finally, section five concludes our work and highlights future tracks.

2 Background and Motivation

RBAC has received a lot of attention from academic researchers and from commercial organizations. This has lead to many improvements to the standard RBAC model [4]. RBAC research can be broadly classified into two main categories: improvements to features existing in standard RBAC and extensions to standard RBAC. Improvements to standard RBAC have been mainly focusing on improving role hierarchies of the standard RBAC model and on improving expressiveness of roles by parametrization. Extensions to standard RBAC have been focusing on adding new features to RBAC such as supporting cross domain roles, role delegation models, etc. In this paper, we focus on improvements to RBAC.

In standard RBAC, role hierarchies support multiple inheritance; meaning that a role can inherit permissions from multiple roles. The general roles hierarchies concept in standard RBAC has two main properties; firstly, the possibility to derive roles from multiple roles, and secondly, the role hierarchies concept provides a uniform treatment of user/role and role/role relations. Users can be included in the role hierarchy, using the same relation to denote the user assignment to roles. More so, standard RBAC supports the limited role hierarchy concept, in which hierarchies are limited to the single immediate descendent [4]. The roles hierarchies concept in standard RBAC suggests that when a senior role inherits from a junior role, all permissions of the junior role are transferred to the senior role.

The most familiar form of collaborative working is hierarchical in nature. In organizational hierarchies, the superior may not take part in the details of a task, but rather acts as the instigator of the task [9]. In other words, the most typical form of hierarchy in organizations is the supervision hierarchy [11]. More so, in some situations it is required to keep a role private and inhibit others from extending it. Sandhu [3], [10] has introduced the concept of the private role, which is a role that cannot be further extended. In situations where users have private documents that they need to protect from their superiors, a new private role has to be introduced for each user. This results in an increased number of roles in the system. This counter-balances the advantage gained by using hierarchies which is reducing number of roles in the system [11]. Xuexiong et al [12] have proposed an approach to tackle excessive inheritance that occurs when users get more permissions than they should have by permission inheritance. They resolve the issue by segregating role permissions into private permissions and public permissions. Then only public permissions are transferred through inheritance to superiors. If a role r has a set of permissions P, then P is divided into two sets P_{prv} for private roles, and P_{pub} for public roles. When a senior role r_s inherits from r, only P_{pub} are transferred to r_s. The drawbacks of this approach are that the private permissions of a role won't be inherited by any other role. In organizations, it might be the case that private permissions are different between two superiors of a junior role. In this situation, it won't be possible to define the inheritance for the two roles.

Lack of expressiveness in role definition has received attention from researchers as well. In many organizations, different users may require different granularity levels of the same role. For example, two tellers in a bank might have the same role that enables them to perform transactions. But the maximum amount of the transactions both of them can perform might be different depending on their seniority. Standard RBAC can be adapted to capture such fine grained authorizations by dramatically increasing the number of distinct roles. Parametrized roles [5], [6], [8] have been proposed to address the lack of expressiveness of roles. One of the good attempts to address lack of expressiveness of RBAC by using parametrized roles was defined by Jaeger et al. [8]. The formal definition of parametrized RBAC was introduced by Abdallah et al. [5]. Parametrized RBAC provides finer granularity by creating instances of RBAC components according

to the contexts of their use [5]. This is achieved by associating parameters with roles. Parameters are used to define the granularity level of the role. In the example of the bank tellers presented previously, the teller role can be parametrized by an amount limit parameter. Then each teller can be assigned a maximum amount limit when assigned to the role.

Fischer et al. [6] have proposed the object-sensitive RBAC (ORBAC), which is a generalized RBAC model for object-oriented languages. ORBAC addresses the lack of expressiveness of RBAC by using parametrized roles. In ORBAC, privileged operations are parametrized by a set of index values, which are used to distinguish the granularity level of the roles between users. A privileged operation can only be invoked if both the required role is assigned to the user who invokes the operation and the role's index values matches the operation's index values.

Parametrized RBAC was the first initiative to address the lack of expressiveness in role definitions, but parametrized RBAC is still not sufficient to express many authorization requirements. In the example discussed above, it is not possible to check the amount against currencies and to find the amount value against the home currency of the bank. Expressiveness of RBAC can be further improved should we introduce possibilities to make validations on parameters. In addition, we provide a new type of parameters that can have values common to all instances of roles. In our proposed access control model (ROAC), we address these limitations and further improve the concept of roles and permissions.

3 The Role-Oriented Access Control Model Overview

In the previous section, we have shown that parametrized RBAC was proposed to address standard RBAC's lack of expressiveness when defining roles. The proposed approach adds some flexibility when defining roles. In parametrized RBAC, computations involving parameters of roles must be performed at the application side. This is similar to plain old record types of *structs* in procedural programming languages. Object oriented programming languages have introduced the notion of encapsulation that is wrapping data and methods within classes in combination with implementation hiding [13]. The idea here is to transplant those ideas to the definition of roles. With parametrized RBAC it is possible, for example, to specify an amount limit and a currency as parameters to the teller role. But it cannot provide further possibilities to compute the amount against the home currency. As an example, if we pass to the teller role 1000 as an amount and *EUR* as a currency, the amount is not equivalent to 1000 with *USD* currency. Moreover, it provides no way of adding static parameters to roles, i.e. when the static parameter is changed it takes effect on all instances of the role. If an organization requires to disable a role from the access control system, but the organization cannot delete it, since it is associated with records in their audit trail. Deleting the role causes inconsistencies within the system. A better way to cope with this issue is to flag the role as deleted.

In ROAC, we address limitations of existing RBAC models by adjusting and transplanting concepts of object oriented programming languages to the context of roles and permissions. Roles and permissions in ROAC are analogous to objects in object oriented programming languages. Like objects, roles and permissions can hold data (variables) and operations (methods). Similarly, objects can inherit from other objects typically expressing an *is-a* relation, roles can be organized into hierarchies with different relationships between superior nodes and their subnodes.

The core ROAC model consists of three main elements, users, roles and permissions. Users are principals requiring access to a software system. Roles project job functions within organizations. Roles can be further fine grained to represent sub-functions e.g. a job function can be a *teller* and a sub-function can be *AccountHolder*. Permissions are privileges to execute operations or access objects in the system. Users are assigned to role instances and permissions instances are assigned to roles. Since permissions usually correspond to operations and/or objects in a software system, parameters and validators should be included in permissions and propagated back to roles when permissions are assigned to roles. This means that roles combine all parameters of their assigned permissions. The values of parameters are set during users to roles assignment. The structure of the ROAC model is shown in Fig. 1.

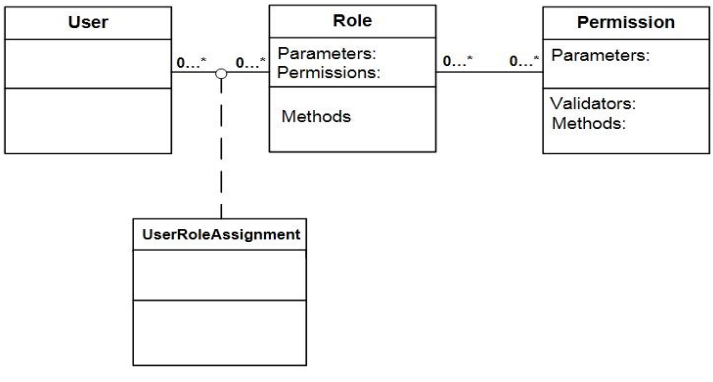

Fig. 1. UML diagram of the ROAC model

ROAC hierarchical model supports two hierarchies; the *is-a* hierarchy and the supervision hierarchy. In the *is-a* hierarchy, senior roles inherit all permissions and definitions of junior roles. The *is-a* hierarchy in ROAC does not necessarily reflect roles hierarchies defined in the standard RBAC model, it could be also used for deriving new roles and re-using definitions of existing roles. The other kind of hierarchy supported by ROAC is the supervision hierarchy. The supervision hierarchy reflects organizational hierarchies. Hierarchical ROAC model is explained in more detail in the fifth section.

4 ROAC Reference Data Model

The central notion of ROAC is that instances of roles and permissions are considered as objects, and hence, they are able to encapsulate data and perform operations. In this section, we summarize the main features of ROAC in a reference data model.

In the ROAC model, we extend the principle of roles and permissions to become analogous to object oriented classes. Both roles and permissions are equipped with variables and methods. Parameters are firstly defined in permissions and then propagated back to roles. Parameters are attributes (also called fields or data members) as in object-oriented languages. Once permissions and roles are defined, instances of both roles and permissions can be created. Roles are assigned in a many-to-many relation with permissions instances. Roles can also have extra parameters defined that are not in permissions instances assigned to the roles instances. These parameters are of type static. Static parameters can be defined in permissions and in roles. Once a value is set for a static parameter, it takes effect for all instances of the role or permission. Static parameters are similar to static variables in object oriented languages. Static variables in object oriented languages store values for the variables in a common memory location, all objects of the same class are affected if one object changes the value of a static variable [14]. Static parameter values can be initialized when static parameters are defined and their values can be changed by static *setter* methods. Roles and permissions can also have private parameters which are variables defined to be used in methods or validators internally.

Definition 1: Role and Permission Parameters. Role and permission parameters are attributes of roles and permissions. Parameters are declared by specifying the parameter name, data type and modifiers.

Methods are either used as validators or administrative functions such as setters and getters. Validators are methods defined in permissions for computing the authorization decision. The simplest form of a validator, is an empty validator. An empty validator grants authorization on an operation in a software system to any user that possesses a role that is assigned the permission that contains the validator definition. Extended form of validators takes inputs from the environment, and may connect to external systems to compute the authorization decision. Validators always return a *Boolean* value, true if it grants authorization and false otherwise. The convenient operations that validators most often perform are to check parameters values extracted from user/role assignment against parameters passed to operations in software systems protected with the ROAC model. Validators can also implement authorization policies. The choice of parameters, static parameters and validators often depends on the organization and the type of operations and objects they need to protect. Static parameters can hold temporal information about the roles. These temporal properties can be validated by validators. It could be useful in an organization when they, for example, add a new role in their access control system and they decide to start

using the role on a specific date. The organization can define the role with a static parameter *StartDate* and assign the role to users. Then they can validate the *StartDate* before granting access on an operation. There are many scenarios where static parameters can help organizations maintain dynamic properties of their access control system.

Methods in ROAC are of great importance. Methods can be defined in roles and in permissions. The purpose of methods in ROAC is to provide administrative functions over roles and permissions and to handle operations on role and permission parameters.

Definition 2: Permissions Validators. A validator is a permission member operation that provides an authorization decision. Validators definitions consist of a signature and a body. The signature specifies the validator name and input parameters. Validators always return Boolean values. True if authorization is granted and false if denied. The body of the validator is the implementation of validator that consists of a sequence of programming statements implementing the authorization conditions.

Definition 3: Permissions and Roles methods. Methods in permissions and roles are member operations. Methods definitions consist of a signature and a body. The signature specifies method name, input parameters and a return value. The body of the method represents the method's business logic implementation by a sequence of programming statements.

Definition 4: ROAC Permissions. A Permission is a data type characterized by operations and attributes. Operations and attribute definitions are the same for all instances of a given permission. Permission non-static attributes values are specific to instances derived from a given permission. A permission determines an access authorization on one or more objects or one or more operations in a software system. Permissions in ROAC consist of: parameters, validators and methods. Parameters are attributes, while validators and methods are operations.

Definition 5: ROAC Roles. A Role is a data type characterized by operations and attributes. Attributes in roles correspond to role parameters propagated from permissions assigned to the role, and to static attributes of the role. Role operations correspond to role methods that provide administrative operations.

Definition 6: ROAC Data Types. Data types in ROAC correspond to the type of parameters in permissions and roles. Data types supported by ROAC depend on the programming language at stake, which usually are primitive data types and reference data types (objects).

Definition 7: User-Role Assignment. Let U be a set of user instances from user, R be a set of role instances created from different roles. Let M be a set of parameters of roles and V be a set of possible values for parameters. The user-role assignment is a many-to-many relation, given by the following mapping:

UA = ([u,r] (m_1=v_1, ., m_n=v_n)) , u ∈ U , r ∈ R, $m_1..m_n$ ∈ M,$v_1..v_2$ ∈ V

Definition 8: Role-Permission association. Let R be a set of different roles, let P a set of permissions instances, let M be a set of permissions parameters and let R be a role instance created from R. The role-permission association is given by the following mapping:

RA = (r,p($m_1..m_n$)) r ∈ R ,$m_1..m_n$∈M , r = r(p_{pre},p) r∈R, p_{pre} is the existing role permissions

We have until now defined the different elements of the role-oriented access control model. We now discuss how interactions between the different elements are established. Afterwards we use an example to explain these interactions.

In ROAC, users are assigned to roles and permissions are assigned to roles. Actually, one of the major advantages of RBAC is simplification of permissions management. Users can be easily reassigned from one role to another. Roles can be granted new permissions as new applications and systems are incorporated, and permissions can be revoked from roles as needed [3]. This is a great advantage that can be provided if user-role associations and role-permission assignments can be achieved dynamically. We have designed relations between ROAC elements to be implemented dynamically. In user-role association, users are associated to roles instances by role administrative methods. The role definition is not changed during this process. Parameter values of roles are set also during user-role assignment. This enables organizations to define different parameter values for different users, and hence provide different granularity levels of roles. The roles-permissions associations are also achieved similarly. If a new permission is to be added to the access control system, it does not need redefinition of roles that need to be assigned the new permission. Roles have an enumeration that contains all permissions assigned to roles. The enumeration can be dynamically updated by administrative roles methods for associating new permissions instances to roles. Roles also have enumerations that contain the parameters of permissions. Parameters of a role are the set of parameters of all permissions associated to the role. The parameters enumeration is updated each time a new permission instance is associated to the role. As well, since multiple permissions may share similar parameters, such as an *amount* value of a bank teller role permissions; all similar permissions parameters are considered as one parameter. The only condition is that those parameters must share the same name and data type. It might happen that a new permission is added to a role in a live environment where the role is already assigned to users, so an administrative function is also provided to set and update particular parameter values for particular users. More so, depending on authorization requirements, more administrative methods can be added to roles. When permissions are assigned to roles, static parameters are not propagated back to roles. Since static parameters are corresponding to the permission and their values are common to all instances of permissions.

Fig. 2. shows an example of a role definition and a permission definition. The role reflects a junior *teller* role in a bank. The permission is a privilege for withdrawing money from a bank account. The *Withdraw* permission has

two parameters; *AmountLimit* represents the maximum amount of a transaction the teller can perform. *ListOfCurrencies* represents the allowed currencies for the teller. The static parameters of the role are: *StartDate*: determines when the role is activated. *ExpiryDate* determines when the role expires and is retired. The *Disabled* flag determines if the role is enabled or disabled, the *withdraw* permission has also a *disabled* flag. The *Withdraw* permission has one validator to validate the *Amount* specified in the transaction against the *AmountLimit* of the role and to check if the currency of the transaction is in *ListOfCurrencies* of the role. The *ValidateHomeAmount()* validator converts the currency of the transaction to the home currency of the bank, and then it compares the transaction amount with the *AmountLimit*. This computation is required as the home amount value depends on the currency of the transaction. For example; if the user has an *AmountLimit=10000*, and the home amount is *EUR*, and the amount of the transaction is 20000 with the *YEN* currency. Then the transaction should be authorized. The static methods defined in the role are used for setting and getting values of static parameters and for modifying and querying permissions. The *ValidateHomeAmount* validator may check if the permission is enabled or not before deciding to authorize.

5 Generalization in Role Oriented Access Control Model

In the object-oriented paradigm, inheritance is a mechanism that implements *is-a* relationships between classes. Inheritance allows hierarchically related classes to reuse and absorb features by inheriting class members (variables and methods). Most existing RBAC models support role hierarchies based on a similar inheritance mechanism found in object-oriented languages.

The advantages most commonly associated with inheritance in the object oriented paradigm are: malleability and reusability, malleability facilitates program construction, maintenance, and extension through factoring the definitions common to a set of classes into a single class called the superclass and then any change required in the common behavior can be done only once in the superclass. Reusability facilitates the reuse of code and data by defining abstractions in terms of existing abstractions. This greatly reduces development efforts by reusing existing software components [15].

Hierarchical ROAC supports multiple inheritance by allowing a role to have more than one parent. Despite advantages of multiple inheritance, it introduces a new complexity: two or more parents may define identifiers with the same name [15]. Roles have multiple members such as parameters, static parameters and validators. The definition of roles in ROAC might introduce some challenges as encountered when defining inheritance in object-oriented languages. One challenge is name conflicts. When two junior roles are to be inherited by a third senior role, where the two junior roles have two parameters (or static parameters) that have identical names. This problem has been exposed in object-oriented languages and there have been some approaches put together to address this problem. In object oriented languages, strategies for resolving conflicts in multiple inheritance are divided into two main categories, depending on whether resolving the

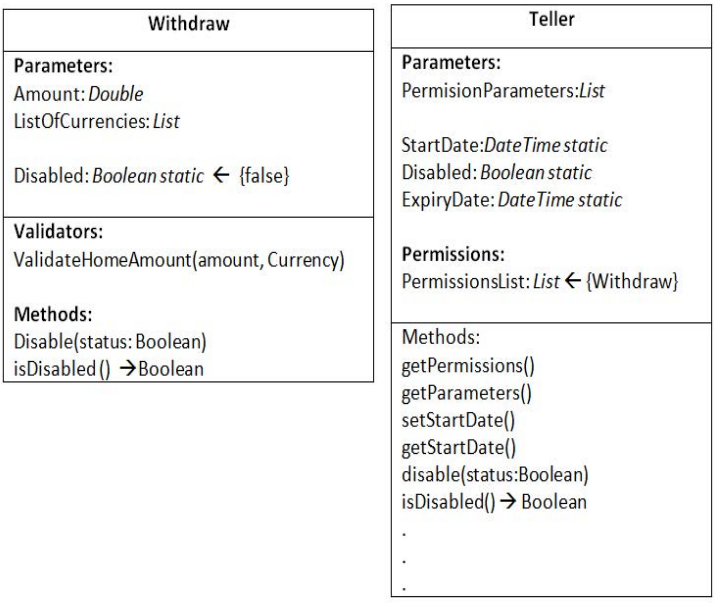

Fig. 2. An example role and permission

conflict requires interaction with the user or not [16]. In the category where no interactions are required from the users, object-oriented languages automatically resolve the conflict. They rank the objects parent and take the property with highest rank. They use linearization to construct a total ordering of all classes. Linearization solves runtime conflicts without human interventions, but it has two drawbacks: it masks ambiguities between otherwise unordered ancestors, and it fails with inheritance graphs that it deems inconsistent [15]. The other technique used to solve name conflicts requires interventions from users, such as explicit designation as in C++, exclusion as in *CommonObjects* and renaming as in Eiffel [16]. Renaming gives the developer the power to decide on properties names and to choose appropriate names. It also avoids complexity and inefficiency of linearization. In our approach to role inheritance, we adopt the renaming approach. If a role is inheriting from two roles that have the same parameter or static parameter names, then we rename the parameter if the two parameters are different and we retain parameter names if they are identical and hence combined into one parameter, in this case the two parameters must have identical data types. Fig. 3 shows an example of how conflicts are solved in ROAC by renaming. In part one of the figure, role R_3 is inheriting roles R_1 and R_2. R_1 has two parameters P_1 and P_2. R_2 has two parameters P_2 and P_3. If P_2 of R_1 is identical to P_2 of R_2, then R_3 inherits only three parameters P_1, P_2 and P_3. In part two of the figure, P_2 of R_1 is different from P_2 of R_2. Then R_3 inherits four parameters, and P_2 of R_1 and P_2 of R_2 must be renamed as shown in Fig. 3.

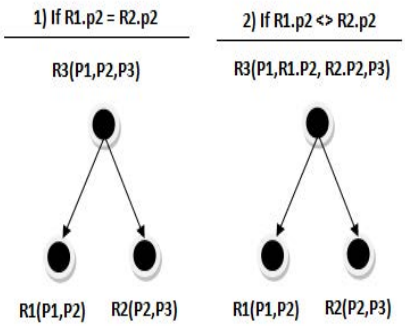

Fig. 3. Name conflict resolution in ROAC

Generalization in ROAC has two sides: roles and permissions definition inheritance and permissions inheritance. In the roles and permissions definition inheritance, the objective is re-usability by factoring the definitions common to a set of roles or permissions into a single role or permission. Permissions instances associated to roles are not considered in roles definitions inheritance. In permissions inheritance, senior roles can inherit subsets of permissions from the junior roles. In many organizations, the actual hierarchies are supervision hierarchies rather than *is-a* hierarchies. As an example, in a bank, the branch manager could inherit the *teller* role, but he might not need to inherit the permission of initiating payments of the *teller* role. While it might be required those other senior users inherit the teller role, and require the permission of initiating payments, but they do not need the permission of initiating transfers. So, the inheritance model must enable selective inheritance of roles, to enable selecting permissions from junior roles.

In the object-oriented paradigm, encapsulation is a technique used for hiding data within classes and preventing outsiders from manipulating class members directly. Some object-oriented languages such as Java define access control rules that restrict the members of a class from being used outside the class. This is achieved by access control modifiers. The encapsulation model in object-oriented languages is not satisfactory for access control. As in access control, it is required to be more selective regarding permissions when performing inheritance hierarchies. As a consequence, we have designed ROAC with two sides inheritance. Firstly, the inheritance for roles and permissions, in which only the definition of the role or permission is transferred to subnodes. This is useful for re-using already defined roles and permissions. Another advantage is that a basic role and a basic permission can be defined and equipped by all administrative methods needed to manipulate administrative functions over roles and permissions. Then all other roles and permissions in the system can inherit from the basic role and the basic permission. Secondly, the permission inheritance has to be defined which applicable only for roles. In permission inheritance, permissions of super roles are transferred to sub-roles. Permission inheritance supports selective inheritance, where a set of role permissions can be excluded from being

transferred by inheritance. Permissions can be excluded by providing the permissions exclude list when de"fining permissions inheritance. For example, let X be a role defined with a set of permissions (p_1,p_2,p_3,..) and let Y be a descendent of X, the exclusion list is (p_1, p_2).

Our target is to provide a mechanism for specifying which permissions can be inherited from a junior role by a senior role. In ROAC, permissions list is defined as a enumeration in the role. We can now specify what permissions can be inherited by what senior roles. This enables us to implement supervision as well as the *is-a* hierarchies.

6 Discussion

RBAC supports three well-known security principles: least privilege, separation of duties, and data abstraction [3]. RBAC suggests that users are assigned to roles, roles are assigned to permissions and recommends that roles are assigned only the minimum set of permissions required for tasks needed by members of the roles.

The advances in software systems and the high dependability of organizations on software systems have demanded more requirements on access control. Sandhu and Bhamidipati [17] have offered five founding principles for next-generation access control including next-generation RBAC, summarized as ASCAA for Abstraction, Separation, Containment, Automation and Accountability. The first two are included in RBAC96 [3]. Containment includes three principles; least privilege, separation of duty from RBAC96 and incorporates usage limits. Usage limits are constraints on how users can use roles. ROAC directly supports the user limits concept. Conditions on role usage can be easily implemented in roles by specifying them in permissions validators, and using static parameters to set values for global parameters. As an example, if it is required to restrict the number of times a role can be exercised in a time frame, we can define two static parameters; one for the time frame and the other for number of exercises, then in the validators we can assert this condition. Similarly we can define a time frame where the role can be exercised and the role becomes inactive outside the time frame. Similarly, the automation principle can be implemented in ROAC. Constraints can be defined in administrative methods for user-role assignment. For example expiry of assignment can be defined by using parameters to hold the expiry dates and then implementing the condition in the administrative method that is used to assign users to the role. Different conditions can be implemented for each role. Accountability can be implemented in a combination of three ways. Firstly, sensitive operations require enhanced audit trail, secondly, by notification that requires sensitive operations to trigger a message to an appropriate user, and finally, by escalating the authentication required for sensitive operations [17]. The first and second ways can be incorporated in ROAC. Developers are can add any required definitions for roles in validators. Audit information can be supplied to validators in applications and then validators can store them in data bases or send them to audit trail systems.

ROAC is an expressive access control model that helps large organizations to provide fine granularity of roles while reducing the number of roles. However, this is applicable when multiple roles can be reduced to single role by using parameters. There is a trade-off between simplifying the management of access rights and providing fine granularity [5]. So, organizations should pay attention to the design of roles in a way that provides more granularity but reducing the number of roles. The hierarchical form of ROAC can be used to reflect organizational hierarchies which also simplify the management and the view of roles.

We have validated the ROAC model by simulating an implementation using the Java programming language. We have tested the implementation on a security service that was implemented by the authors of the paper.

7 Conclusion and Future Work

The contribution of this paper is proposing a new access control model, the role-oriented access control model (ROAC). In ROAC roles and permissions are defined as object-oriented classes, where they can have member attributes and operations. ROAC has many advantages compared to existing access control models. One of the main advantages is expressiveness and the possibility to reflect precise organizational hierarchies by ROAC. Another advantage is that organizations can implement any specific requirements for granting authorizations on operations by using validators. The permissions and roles implementation can contain access to external systems like databases and audit log systems to either extract or provide information.

We have discussed some related work on existing RBAC models. We have explained how existing models attempted to tackle shortcomings of access control models that are encountered when they are put into practice. We have focused on two points which are expressiveness of RBAC models and on hierarchical RBAC models.

ROAC concepts were validated by an implementation using the Java programming language. In the implementation we have created an API that can be used for creating roles and permissions, as well as defining relations between the different elements of ROAC such as user/role assignment, role/permissions assignments, and the administrative functions of ROAC. Moreover, the implementation has simulated the hierarchical ROAC model. Our future direction from this point is to provide a full feature access control system based on ROAC. Our ideas are to encapsulate separation of duty, role delegation, as well as other features. The target is to make an API that can used by organizations and researchers, which they can use for constructing their customized access control systems.

References

1. eWeek, `http://www.eweek.com/c/a/Security/Rethinking-Access-Controls-How-WikiLeaks-Could-Have-Been-Prevented/1/`
2. Ferraiolo, D., Kuhn, D.: Role-based access control. In: Proceedings of the 15th National Computer Security Conference (1992)

3. Sandhu, R., Coyne, E., Feinstein, H., Youman, C.: Role-based access control models. IEEE Computer, 38–47 (1996)
4. ANSI INCITS 359, Standard for Role Based Access Control (2004)
5. Abdallah, A., Khayat, E.: A Formal Model for Parameterized Role-Based Access Control. In: Dimitrakos, T., Martinelli, F. (eds.) FAST 2004. IFIP, vol. 173, pp. 233–246. Springer, Boston (2005)
6. Fischer, J., Marino, D., Majumdar, R., Millstein, T.: Fine-Grained Access Control with Object-Sensitive Roles. In: Drossopoulou, S. (ed.) ECOOP 2009. LNCS, vol. 5653, pp. 173–194. Springer, Heidelberg (2009)
7. Kalam, A., Benferhat, S., Miege, A., Baida, R., Cuppens, F., Saurel, C., Balbiani, P., Deswarte, Y., Trouessin, G.: Organization based access control. In: Proceedings of the 4th IEEE International Workshop on Policies for Distributed Systems and Networks (POLICY 2003). IEEE Computer Society, Washington, DC (2003)
8. Jaeger, T., Michailidis, T., Rada, R.: Access Control in a Virtual University. In: Proc. of the 8th International IEEE Workshops on Enabling Technologies: Infrastructure for Collaborative Enterprises, California, USA, pp. 135–140 (1999)
9. Barka, E.: Framework for Role-Based Delegation Models. PhD Thesis, George Mason University (2002)
10. Sandhu, R.: Role activation hierarchies. In: Proceedings of the Third ACM Workshop on Role-Based Access Control (RBAC 1998), pp. 33–40. ACM, New York (1998)
11. Moffett, J., Lupu, E.: The uses of role hierarchies in access control. In: Proceedings of the Fourth ACM Workshop on Role-Based Access Control (RBAC 1999), pp. 153–160. ACM, New York (1999)
12. Xuexiong, Y., Qinxian, W., Changzheng, X.: A Multiple Hierarchies RBAC Model. In: International Conference on Communications and Mobile Computing (2010)
13. Eckel, B.: Thinking in Java, 2nd edn., p. 261. Prentice-Hall (2000)
14. Liang, D.: Introduction to Java Programming, Comprehensive Version, 5th edn. Prentice Hall (2006)
15. Chambers, C., Ungar, D., Chang, B., Holzle, U.: Parents are shared parts of objects: inheritance and encapsulation in SELF. Lisp Symb. Comput., pp. 207–222 (1991)
16. Ducournau, R., Habib, M., Huchard, M., Mugnier, M.L.: Monotonic conflict resolution mechanisms for inheritance. In: Conference Proceedings on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA 1992). ACM, New York (1992)
17. Sandhu, R., Bhamidipati, V.: The ASCAA Principles for Next-Generation Role-Based Access Control. In: Proc. 3rd International Conference on Availability, Reliability and Security (ARES), Barcelona, Spain (2008)