A Framework for Implementing Role-based Access Control Using CORBA Security Service*

Konstantin Beznosov and Yi Deng

Center for Advanced Distributed Systems Engineering
School of Computer Science
Florida International University

Abstract

The paper shows how role-based access control (RBAC) models could be implemented using CORBA Security service. A configuration of CORBA protection system is defined. We provide definitions of RBAC0 and RBAC1 implementations in the framework of CORBA Security and describe what is required from an implementation of CORBA Security service in order to support RBAC0-RBAC3 models.

1 Introduction

Role-based access control (RBAC) [SCFY96] is a family of reference models in which permissions are associated with roles and users are assigned to appropriate roles. A role can represent competency, authority, responsibility or specific duty assignments. Some variations of RBAC include the capability to establish relations between roles, between permissions and roles, and between users and roles. There are four established RBAC reference models: unrelated roles (RBAC0), role-hierarchies (RBAC1), user and role assignment constraints (RBAC2), and both hierarchies and constraints (RBAC3). RBAC supports three security principals: least privilege, separation of duties and data abstraction.

A major purpose of RBAC is to facilitate access control administration and review. RBAC is a promising approach to address the needs of the commercial enterprises better than lattice-based MAC [BL75] and owner-based DAC [Lam71]. Recent series of papers describe ways to model or implement RBAC using the technologies employed by the commercial users: Oracle [Not95], NetWare [ES95], Java [Giu98], DG/UX [Mey97], object-oriented systems [Bar95], object-oriented databases [Won97], MS Windows NT [BC98], enterprise security management systems [Awi97]. Evidence of RBAC recognition in the US government is the fact that the proposed rules on security from the Department of Health and Human Services [Dep98] include RBAC as one of the required alternatives for access control.

At the same time, the commercial market is experiencing the spread of systems based on Common Object Request Broker Architecture (CORBA) technology. CORBA is a versatile object-based distributed computing technology, which is becoming a worldwide industry standard for constructing distributed software systems. CORBA standardization process is based on the consensus of over 800 software companies. The computing model that CORBA adheres to is outlined in Object Management Architecture (OMA) [SS95]. The CORBA environment, including CORBA Security Service, provides a general-purpose infrastructure for developing and deploying distributed object systems in a broad range of specialized vertical domains. CORBA Security service (CS) defines the interfaces to a collection of objects for enforcing a range of security policies using diverse security mechanisms. It provides abstraction from an underlying
security technology so that CORBA-based applications could be independent from the particular security infrastructure provided by a user enterprise computing environment. Due to its general nature, CS is not tailored to any particular access control model. Instead, it defines a general mechanism which is supposed to be adequate for the majority of cases and could be configured to support various access control models. For example, it is shown in [Kar96] how to implement lattice-based MAC using the CORBA authorization model. In the next few years we expect to witness significant financial investments in the enterprise-wide deployment of CS in commercial and government organizations, including those who will construct their security policies utilizing RBAC concepts. It is important to foresee if CS will fully support RBAC models. However, we are not aware of any work in the research community that has explored the potential of CS for support of RBAC reference models.

In this paper we present an approach for implementing RBAC models using the access control mechanism provided by CS. We define a configuration of CS protection system. Then we define RBAC_0 and RBAC_1 implementations in terms of CS framework and describe how RBAC_0-RBAC_3 could be implemented in CS. We illustrate the discussion with several examples. Our approach allows an implementation compliant with CS specification to support RBAC_0. Additional functionality, which is beyond CS specification scope, should be implemented in order to support RBAC_1 and/or RBAC_2.

The paper is organized as follows: Section 2 describes the access control model of CS and defines a configuration of the CORBA protection system; Section 3 defines RBAC models using CS concepts and shows a possible implementation of RBAC_0-RBAC_3 using CS with illustration on an example role hierarchy; Section 4 concludes the paper.

2 CORBA Access Control Model

In this section, we first informally describe the CORBA Access Control model. Then, we formally define a configuration of the CORBA Protection System state.

2.1 Informal Description

The CORBA environment, including CORBA Security Service, provides a general-purpose infrastructure for developing and deploying distributed object-based systems in a broad range of specialized vertical domains. All entities in the CORBA computing model are identified with interfaces defined in the OMG Interface Definition Language (IDL). A CORBA interface is a collection of three things: operations, attributes, and exceptions. An implementation of a CORBA interface is called a CORBA object. Hence, we use “CORBA object” or just “object” to mean “implementation of a CORBA interface”, where it does not cause confusion. Object functionality is exposed to other CORBA-based applications only through the corresponding interfaces. Objects have object references by which they can be referenced. An object reference is a handle through which one requests operations on the object.

The CS model comprises the following functionalities visible to application developers and security administrators: identification and authentication, authorization and access control, auditing, integrity and confidentiality protection, authentication of clients and target objects, optional non-repudiation, administration of security policies and related information.

One of the objectives of CS is to be totally unobtrusive to application developers. Security-unaware objects should be able to run securely on a secure ORB without any active involvement on the site of application objects. In the meantime, it must be possible for security-aware objects to exercise stricter security policies than the ones enforced by CS. In the CS model, all object invocations are mediated by the appropriate security functions in order to enforce various security policies such as access control. A simplified schema of control points in CS model is represented in Figure 1. Those functions are part of CS and are tightly integrated with the ORB because all messages between CORBA objects and clients are passed through the ORB.

CS uses the notion of principal. “A principal is a human user or system entity that is registered in and authentic to the system” [Obj98]. In translation to the traditional security terminology, a principal is a subject. CS manages access control policies based on the security attributes of principals and attributes of objects as well as operations implemented by those objects. Objects that have common security requirements are grouped in security policy domains. Access control policies control what principals can invoke what operations on what objects in the domain the policies are defined on. Policies can be enforced either by the ORB or by the application. In the latter case, such an application is called a security-aware application. Domains allow application of access control policies to security-unaware objects without requiring changes to their implementations or interfaces.
As it can be seen in Figure 1, the client-side and target-side invocation access policy governs whether the client can invoke the requested operation on the target object on behalf of the current principal. This policy is enforced by the ORB in cooperation with the security service it uses for all (security-aware and unaware) applications. A client may invoke an operation on the target object as specified in the request only if this is allowed by the object invocation access policy.

A user uses a UserSponsor\(^1\) to authenticate to the CS environment (Figure 2). A UserSponsor authenticates on behalf of a user with and obtains authenticated credentials from a PrincipalAuthenticator. Instances of UserSponsor implement user interface specific to the authentication method supported by the concrete implementation of CS. For example, for password-based authentication, it prompts the user for user name and password. For authentication based on smart-cards, it interacts with a smart-card reader and (probably) prompts the user to insert the card in the reader. CS standard does not mandate any particular authentication method. What it does specify is the interface of a PrincipalAuthenticator. A PrincipalAuthenticator conducts the actual authentication and creates Credentials object for a new principal. Based on the authentication data it received from a UserSponsor and on the underlying security technology (Kerberos, SESAME, or any other capable technology) as well as on any rules it adheres to, PrincipalAuthenticator instantiates Credentials with various information. The information in Credentials constitute the identity of the new principal which initiates requests on CORBA objects on behalf of the user. Principal authenticated security attributes are part of the information stored in the Credentials object.

The concept of a user is absent from CS AC model. Instead a principal represents the user completely. The notion of a session is indistinguishable from the notion of a principal. Thus multiple principals can act on behalf of a single user. They all potentially have different sets of credentials and therefore exist in CS as completely independent entities. Among other data, principal credentials contain security attributes. Hereafter, we understand attribute to mean security attribute. From the CS AC model point of view, a principal is nothing but an unordered collection of authenticated attributes. All attributes are typed. Attribute types are partitioned into two families: privilege attributes and identity attributes. The family of privilege attributes enumerates attribute

\(^1\)A UserSponsor is an implementation artifact which handles the user authentication process.
types that identify principal privileges: access identifier, primary and secondary groups the principal is a member of, clearance, capabilities, etc. Identity attributes, if present, provide additional information about the principal: audit id, accounting id, and non-repudiation id, reflecting the fact that a principal might have various identities used for different purposes. Principal credentials may contain zero or more attributes of the same family or type.\(^2\) An example of security attributes assigned to authenticated principals is provided in Table 1. One of the standard CORBA attribute types is the role attribute. Due to the extensibility of the schema for defining security attributes, an implementation of CS can support attribute types that are not defined by the CORBA Security standard. Although the normative part of CS does not mandate the way attributes are managed, assignment of such attributes to users is meant to be performed by user administrators.

<table>
<thead>
<tr>
<th>Principal</th>
<th>Attributes</th>
</tr>
</thead>
<tbody>
<tr>
<td>p(_1)</td>
<td>a(_1)</td>
</tr>
<tr>
<td>p(_2)</td>
<td>a(_2), a(_6)</td>
</tr>
<tr>
<td>p(_3)</td>
<td>a(_2), a(_3)</td>
</tr>
<tr>
<td>p(_4)</td>
<td>a(_4), a(_5)</td>
</tr>
</tbody>
</table>

Table 1: Security Attributes Possessed by Authenticated Principals

All a principal does in the CORBA computational model is invoke operations on corresponding objects. In order to make a request one needs to know two things: object reference, which uniquely identifies an object, and operation name. CORBA interfaces can inherit from other CORBA interfaces via interface inheritance. An operation name is unique for an interface.\(^3\) Thus, any operation is uniquely identified by its name and by the name of the interface it is defined in.

In this paper, we use notation \(i_k m_n\), to refer to \(n\)-th operation on \(k\)-th interface.

There is a global\(^4\) set of required rights for each operation defined by its interface’s required rights mapping. This set, together with a combinator (all or any right), defines what rights a principal has to have in order to invoke the operation. Table 2 provides an example of required rights for operations on three interfaces \(i_1, i_2,\) and \(i_3\). It is assumed that required rights are defined and their semantics are precisely documented by application developers who know the best what each operation does. Depending on the access policy (DomainAccessPolicy) enforced in a particular AC policy domain,\(^5\) a principal is granted different rights (GrantedRights) according to what SecurityAttributes it has.\(^6\) Each DomainAccessPolicy defines what rights are granted for each security attribute. An example of a mapping between principal privilege attributes and granted rights is provided in Table 3. Security administrators are responsible for defining what rights are granted to what security attributes in what delegation state on domain per domain basis. Whenever a principal attempts an operation invocation, principal’s effective rights are computed via operation AccessPolicy:get_effective_rights.\(^7\) CS specification purposefully does not define how the operation combines rights granted through different privilege attribute entries in Table 3. The specifiers let CS implementors define the operation’s internal behavior ([1098, p. 122]). A simplest implementation of get_effective_rights could be when the set of rights granted to a principal is a union of rights granted to every security attribute the principal has. For our examples, we will assume exactly this implementation of the operation. If we use our example of security attributes assigned to principals \(p_1, p_2, p_3,\) and \(p_4\) (Table 1), and the examples of required (Table 2) and granted (Table 3) rights, then Table 4 shows what rights the principals are granted in each domain.

\(^2\)This rule applies to all attribute types including access id, although it is hard to foresee a useful implementation of CS where a principal would have multiple or no access identities.

\(^3\)Interface inheritance in CORBA does not allow to inherit from interfaces with operations of the same type. This rule resolves the problem of operation name overlapping.

\(^4\)I.e. not dependent on a policy domain in which the object is located.

\(^5\)In the CORBA security model, a security policy domain is just a collection of objects.

\(^6\)For the sake of brevity, we omit delegation state qualifier for granted rights. This does not change the correctness of the discussion, as we show below.

\(^7\)Regular caching techniques can be used by an implementation to avoid repetitive computations.
<table>
<thead>
<tr>
<th>Operations</th>
<th>Required Rights</th>
<th>Combinator</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>( i_1m_1 )</td>
<td>( r_1 )</td>
<td>all</td>
<td>Only a principal who is granted right ( r_1 ) can invoke the operation.</td>
</tr>
<tr>
<td>( i_1m_2 )</td>
<td>( r_1, r_2 )</td>
<td>any</td>
<td>Any principal who is granted either ( r_1 ) or ( r_2 ) right can invoke the operation.</td>
</tr>
<tr>
<td>( i_2m_1 )</td>
<td>( r_2, r_3 )</td>
<td>all</td>
<td>Only a principal who is granted both ( r_2 ) and ( r_3 ) rights can invoke the operation.</td>
</tr>
<tr>
<td>( i_2m_2 )</td>
<td>( r_2, r_3, r_4 )</td>
<td>all</td>
<td>Only a principal who is granted all ( r_2, r_3, r_4 ) rights can invoke the operation.</td>
</tr>
<tr>
<td>( i_3m_1 )</td>
<td>( r_1, r_2, r_3, r_4 )</td>
<td>any</td>
<td>Any principal who is granted either of ( r_1, r_2, r_3, r_4 ) rights can invoke the operation.</td>
</tr>
</tbody>
</table>

Table 2: Required Rights Matrix

<table>
<thead>
<tr>
<th>Principal</th>
<th>Granted Rights</th>
<th>Domains</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>( d_1 )</td>
<td>( d_2 )</td>
</tr>
<tr>
<td>( p_1 )</td>
<td>( r_1 )</td>
<td>( r_2 )</td>
</tr>
<tr>
<td>( p_2 )</td>
<td>( r_6 )</td>
<td>( r_1 )</td>
</tr>
<tr>
<td>( p_3 )</td>
<td>( r_2, r_3 )</td>
<td>( r_1 )</td>
</tr>
<tr>
<td>( p_4 )</td>
<td>( r_1, r_2, r_3 )</td>
<td>( r_1, r_2, r_3, r_4 )</td>
</tr>
</tbody>
</table>

Table 4: Granted Rights Per Principal

2.2 CORBA Protection State Configuration

We summarize the above description of the CS AC model by defining the protection state configuration of a CORBA system:

**Definition 2.1** A configuration of a CORBA system protection state is the thirteen-tuple \((A, IM, O, R, D, C, RRM, DS, IDM, GRM, effective\_rights, combine, interface\_operation)\) interpreted as follows:

- \( A \) is the set of privilege attributes.
- \( IM \) is the set of operations uniquely identified by interfaces that they are defined on.
- \( O \) is a set of distinguishable interface instances.
- \( R \) is the set of rights.
- \( D \) is the set of access policy domains.
- \( C = \{ all, any \} \) is a set of rights combiners.

- \( RRM \) is the required rights matrix, with a row for every interface operation from \( IM \) and two columns. For the first column (Required Rights), we have \([IM, Rights] \subseteq R\). For the second column (Combinator), we have \([IM, Combinator] \subseteq C\).

- \( DS = \{ i, d \} \) is a set of delegation states.

- \( IDM \) is the matrix of domain membership for interface instances with a row for every domain from \( D \) and a column for every interface instance from \( O \). We denote the contents of \((D, O)\) cell of \( IDM \) by \([D, O]\). We have \([D, O] \subseteq \{T,F\}^8, [d, o] = T \implies o \in d\).

- \( GRM \) is the granted rights matrix, with a row for every attribute from \( A \) and a column for every access policy domain from \( D \). We denote the contents of the \((A, D)\) cell of \( GRM \) by \([A, D]\). We have \([A, D] \subseteq R\).

- \( effective\_rights: D \times 2^A \rightarrow 2^R \), a function mapping a set \( a_1, a_2, ... \) of privilege attributes (where \( \forall i < i < i < i \in A \) in a domain \( d_j \in D \) to a set of rights \( r_1, r_2, ... r_p \) (where \( \forall i, 1 < i < p: r_i \in R \)) that are in effect for the given set of attributes.

- \( combine: 2^D \times 2^R \rightarrow 2^R \), a function mapping sets of rights returned from \( effective\_rights \) for every domain in \( D \) the interface instance is a member of, to a set of effective rights.

- \( interface\_operation: M \times O \rightarrow IM \), a function mapping an operation name \( m \) and an interface instance \( o \in O \) into an interface operation uniquely identified on the interface, which \( o \) implements.

\[ ^8 \right \]
Function effective_rights looks up GRM to obtain granted rights for each attribute in all domains to which \( o \) belongs. It combines those rights according to its implementation and returns effective rights for each domain. Results returned from effective_rights serve as input parameters for the function combine. The latter combines them according to its implementation. Rights returned by combine are checked against RRM. If the match succeeds, then access is granted. Otherwise, access is denied.

Table 5 shows what operations can be invoked by the principals from our example. For each domain, an access matrix from [Lam71], such as in Table 6, could be constructed.

<table>
<thead>
<tr>
<th>Principals</th>
<th>Operations</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Table 5: Operations that a Principal Can Invoke

<table>
<thead>
<tr>
<th>Subjects</th>
<th>Objects</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>( i_1 )</td>
</tr>
<tr>
<td>( p_1 )</td>
<td>( i_1 m_2 )</td>
</tr>
<tr>
<td>( p_2 )</td>
<td>( i_1 m_1, i_1 m_2 )</td>
</tr>
<tr>
<td>( p_3 )</td>
<td>( i_1 m_1, i_1 m_2 )</td>
</tr>
<tr>
<td>( p_4 )</td>
<td>( i_1 m_1, i_1 m_2 )</td>
</tr>
</tbody>
</table>

Table 6: Access Matrix for Domain \( d_2 \)

Three general observations are worth noting for an access matrix constructed for any CS system. First, subjects cannot be objects, i.e. the CORBA access control model does not have the concept of operations on principals. It only has the concept of operations on interfaces, which are objects according to the terminology of the access matrix [Lam71]. Second, since \( i_k m_p \equiv i_l m_q \iff i_k \equiv l \land p \equiv q \) (i.e. just \( p \equiv q \) is not enough for \( i_k m_p \equiv i_l m_q \)), as in Table 6, the semantics of operations in a general case might be different. Thus, for each subject \( s \) and object \( o \), the content of cell \([s,o]\) is specific to the object, i.e. no operations permitted on one object could be permitted on another object because operations are semantically different for every interface unless interfaces are related via inheritance. Third, all implementations of the same interface in a given access policy domain are represented by the same object in the access matrix; therefore, implementations of the same interface are indistinguishable from the access control point of view. This is one of the reasons policy domains are important in the CORBA access control model.

3 Support of RBAC by the CORBA Access Control Model

Among the four RBAC reference models defined by Sandhu et al [SCFY96], RBAC0 is the base model. It requires only that a system has notions of users, roles, permissions and sessions. There are no constraints on the assignment of permissions to roles and users to roles. RBAC1 has hierarchies of roles in addition to everything RBAC0 has. RBAC2 has constraints on the assignment of users to roles and permissions to roles in addition to everything RBAC0 has. RBAC3 combines RBAC1 and RBAC2. In this section, we define RBAC0 and RBAC1 using the language of Definition 2.1 of the CORBA protection state configuration. This will help us show the correctness of our approach to configuring a CORBA system for supporting various RBAC models.

3.1 RBAC0: Base Model

For the base model RBAC0, the four sets of identities are represented in CS as follows:9 Users in RBAC map to users in CS; Roles are represented by set \( A \) of privilege attributes of type role; Permissions are equivalent to the set of rights \( R \) in CS; Sessions are equivalent to principals, which are nothing but sets of security attributes, from CS AC point of view.

RBAC0 definition (reprint is available in Appendix) in the language of CS is formally defined as follows:

Definition 3.1

- \( U, A, R, P \) (users, attributes of type role, rights, and principals, respectively)
- \( PA \subseteq R \times A \), a many-to-many assignment of granted rights to security attributes of type role relation.
- \( UA \subseteq U \times A \), a many-to-many user to security attributes of type role assignment relation.

9We do not mention CS AC domains because, as it will be shown in the example on Page 9, RBAC models can be supported in CORBA using a single domain.
• user : P → U, a function mapping each principal \( p_i \) to the single user user\((p_i)\), constant for the principal lifetime, and

• roles : P → \( 2^A \), a function mapping each principal \( p_i \) to a set of privilege attributes of type role \( \text{roles}(p_i) \subseteq \{ a \mid (\text{user}(p_i), a) \in A \} \) and principal \( p_i \) has the granted rights \( \bigcup_{a \in \text{roles}(p_i)} \{ r \mid (r, a) \in \text{PA} \} \)

\[ \square \]

It is easy to see that the definition describes a system compliant with the RBAC0 definition provided in [SCFY96]. Given the definition, we will show how a CORBA protection system specified by a configuration language from Definition 2.1 could be used to implement a security system compliant to this definition of RBAC0. \( \text{PA} \) relation is specified by granted rights matrix \( \text{GRM} \). \( \text{UA} \) relation is managed by user administrators in CS that define what values of attributes of type \( \text{role} \) are assigned to users. However such management functionality is beyond the scope of CS specification, which means that functionality defined by \( \text{UA} \) relation is implementation-specific. An implementation of \text{PrincipalAuthenticator} \footnote{As it was described in Section 2, a \text{PrincipalAuthenticator} conducts the actual authentication and creates \text{Credentials} object for a new principal.} initializes new principal credentials with security attributes according to \( \text{UA} \) An example is provided in Table 1, where attributes \( a_1 \) through \( a_6 \) have the type \( \text{role} \). The value of the principal’s privilege attribute of the type \( \text{AccessId} \) is equivalent to the return value from the function \text{user}. An implementation of \text{PrincipalAuthenticator} should initialize principal credentials according to the function \( \text{roles} \). Since a user in RBAC0 can activate any subset of roles the user is assigned to, implementation of \( \text{UA} \) ensures implementation of RBAC0. Thus, we have shown that all relations, functions and sets specified in Definition 3.1 can be directly supported by CS-compliant implementations. In order for a CS implementation to support RBAC0 it should:

1. comply with CS standard, and

2. provide a means to administrate user-to-role assignment relation \( \text{UA} \), and

3. provide a means for users to select through \text{UserSponsor} a set of roles with which they would like to activate the new principal, and

4. implement \text{PrincipalAuthenticator} which creates principal credentials containing privilege attributes of type \( \text{role} \) according to relation \( \text{UA} \), and

5. implement \text{PrincipalAuthenticator} which creates principal credentials containing one and only one privilege attribute of type \( \text{AccessId} \).

A straightforward implementation of RBAC0 in CS would be the one that uses privilege attributes of only type \( \text{role} \) for constructing granted rights tables, such as Table 3.

### 3.2 RBAC1: Role Hierarchies

RBAC1 is RBAC0 with role hierarchies. RBAC1 (the definition reprint is available in Appendix) in the language of CS is formally defined as follows:

**Definition 3.2**

- \( U, A, R, P, \text{PA}, \text{UA} \) and \text{user} are unchanged from RBAC0.

- \( \text{RH} \subseteq A \times A \) is a partial order on \( R \) called the role hierarchy, written as \( \geq \). It is the same as in [SCFY96].

- \( \text{roles} : P \rightarrow 2^A \) is modified from RBAC0 to require \( \text{roles}(p_i) \subseteq \{ a \mid (\exists a' \geq a) \left( (\text{users}(p_i), a') \in \text{UA} \right) \} \) and principal \( p_i \) has the granted rights \( \bigcup_{a \in \text{roles}(p_i)} \{ r \mid (\exists a'' \leq a) \left( (r, a'') \in \text{PA} \right) \} \)

\[ \square \]

The function \( \text{roles} \) is to be implemented and enforced by a \text{Principal Authenticator} (Figure 2 on Page 3). A user provides to a \text{UserSponsor} a set of roles which they want the principal to be activated with. The \text{PrincipalAuthenticator}, during the authentication phase with the \text{UserSponsor}, creates new credentials of the principal. The credentials have requested by user roles provided that they satisfy the definition of function \( \text{roles} \) for RBAC1.

A valid implementation of RBAC1 could be one that allows a user to specify any role junior to those the user is a member of. In this case, an implementation of \text{PrincipalAuthenticator} activates all roles which are junior to the specified role.

In order for a CS implementation to support RBAC1 it should:

1. implement RBAC0, and

2. provide a means to administrate the role hierarchy relation \( \text{RH} \), and
3. implement **PrincipalAuthenticator** which creates principal credentials containing privilege attributes of the type role according to relations $UA$ and $RH$, as well as the function roles.

### 3.3 RBAC$_2$: Constraints

Constraints in RBAC are predicates that apply to $UA$ and $PA$ relations and the user and roles functions ([SCFY96]). Constraints on $UA$ relation are to be enforced by an implementation of user administrator tools. Constraints on the functions user and roles are the responsibility of **PrincipalAuthenticator** implementation. Constraints on $PA$ relation are to be enforced by an implementation of security administrator tools.

In order for a CS implementation to support RBAC$_2$ it should:

1. implement RBAC$_0$, and
2. implement support of constraints on $UA$ relation user administrator tools, and
3. implement **PrincipalAuthenticator** with support of constraints on functions user and roles, and
4. enable enforcement of constraints on $PA$ relation by security administration tools.

### 3.4 RBAC$_3$: RBAC$_1$ + RBAC$_2$

RBAC$_3$ is a combination of RBAC$_1$ and RBAC$_2$ along with possibly additional constraints on the role hierarchy. It can be implemented in CS as well. Obviously, in order for a CS implementation to support RBAC$_3$ it should:

1. implement RBAC$_1$,
2. implement RBAC$_2$, and
3. implement possible additional constrains on the role hierarchy.

Requirements for support of RBAC$_1$ and RBAC$_2$ by CORBA Security service implementation have been already discussed. Implementation of additional static constrains on the RBAC$_1$ role hierarchy is to be done by user administrator tools. For the support dynamic constrains, additional functionality in the implementation of **PrincipalAuthenticator** is required, in addition to the administrator tools.

### 3.5 Example

To illustrate the points made in this section, we describe a protection state of a CORBA system defined by Definition 2.1 that implements an example role hierarchy. We use an example hierarchy from [SP98] shown in Figure 3. We will show how a CORBA-based distributed system could be configured to support RBAC$_1$ with an example hierarchy shown on Figure 3 and to protect access to implementations of CORBA interfaces shown in Figures 4 and 5. The following access control policies describe what

![Figure 4: Example CORBA Interfaces](image)

![Figure 5: Employee Interface](image)

actions are allowed. All other actions are denied.

1. Anyone can look up an employee’s name and experience.
2. Everyone in the engineering department can get a description of and report problems regarding any project.
3. Engineers, assigned to projects, can make changes and review changes related to their projects.
4. Quality engineers can inspect the quality of projects they are assigned to.
5. Production engineers can create new releases.
6. Project leaders can close problems.
7. The director can manage employees (assign them to projects, un-assign them from projects, add new records to their experience, and fire) and close engineering projects.

We define that effective rights returns a union of granted rights per attribute. We define that combine returns a union of rights granted in each domain.

**Single Access Policy Domain Solution**

In order to implement the role hierarchy in CS without using access policy domains, we introduce two new interfaces `EngineeringProject1` and `EngineeringProject2`, as shown in Figure 6. The following configuration of a system protection state could be used:

- $A = \{e, ed, e1, e2, pe1, pe2, qe1, qe2, pl1, pl2, dir\}$. All these attributes have type *role*.

- $IM = \{\text{Employee::get\_name, Employee::assign\_to\_project, Employee::unassign\_from\_project, Employee::add\_experience, Employee::get\_experience, Employee::fire, EngineeringProject1::inspect\_quality, EngineeringProject1::make\_changes, EngineeringProject1::report\_problem, EngineeringProject1::review\_changes, EngineeringProject1::close, EngineeringProject1::close\_problem, EngineeringProject1::create\_new\_release, EngineeringProject1::get\_description, EngineeringProject2::inspect\_quality, EngineeringProject2::make\_changes, EngineeringProject2::report\_problem, \}$
Figure 6: EngineeringProject Interface Hierarchy

EngineeringProject2::review_changes, EngineeringProject2::close, EngineeringProject2::report_problem, EngineeringProject2::create_new_release, EngineeringProject2::get_description};

We do not use any implementations of interface EngineeringProject. Only derived interfaces are used.

- \( O = \{ e, ed, e1, e2, pe1, pe2, qe1, qe2, pl1, pl2, dir, prj1, prj2 \}. prj1 is an instance of EngineeringProject1, and prj2 is an instance of EngineeringProject2. All other elements of \( O \) are instances of interface Employee.

- \( R = \{ gn, atp, ufp, ae, ge, f, mc1, rc1, iq1, rl1, cl1, cnr1, gd1, c1, mc2, rc2, iq2, rl2, cl2, cnr2, gd2, c2 \}\(^{11}\)

- \( D = \{ d1 \} \)

- \( C = \{ all \} - we use only one combinator.\)

- \( RRM \) is shown in Table 7. We omitted column with rights combinators because required rights for all operations have the same combinator - “all”.\(^{12}\)

- \( DS = \{ i, d \} \)

- In the IDM, all interface instances are in members of the only access policy domain.

- \( GRM \) is shown in Table 8.

<table>
<thead>
<tr>
<th>Operations</th>
<th>Rights</th>
</tr>
</thead>
<tbody>
<tr>
<td>Employee::get_name</td>
<td>gn</td>
</tr>
<tr>
<td>Employee::assign_to_project</td>
<td>atp</td>
</tr>
<tr>
<td>Employee::unassign_from_project</td>
<td>ufp</td>
</tr>
<tr>
<td>Employee::add_experience</td>
<td>ae</td>
</tr>
<tr>
<td>Employee::get_experience</td>
<td>ge</td>
</tr>
<tr>
<td>Employee::fire</td>
<td>f</td>
</tr>
<tr>
<td>EngineeringProject1::get_description</td>
<td>gd1</td>
</tr>
<tr>
<td>EngineeringProject1::inspect_quality</td>
<td>iq1</td>
</tr>
<tr>
<td>EngineeringProject1::make_changes</td>
<td>mc1</td>
</tr>
<tr>
<td>EngineeringProject1::review_changes</td>
<td>rc1</td>
</tr>
<tr>
<td>EngineeringProject1::report_problem</td>
<td>rp1</td>
</tr>
<tr>
<td>EngineeringProject1::close_problem</td>
<td>cp1</td>
</tr>
<tr>
<td>EngineeringProject1::create_new_release</td>
<td>cmr1</td>
</tr>
<tr>
<td>EngineeringProject1::close</td>
<td>c1</td>
</tr>
<tr>
<td>EngineeringProject2::get_description</td>
<td>gd2</td>
</tr>
<tr>
<td>EngineeringProject2::inspect_quality</td>
<td>iq2</td>
</tr>
<tr>
<td>EngineeringProject2::make_changes</td>
<td>mc2</td>
</tr>
<tr>
<td>EngineeringProject2::review_changes</td>
<td>rc2</td>
</tr>
<tr>
<td>EngineeringProject2::report_problem</td>
<td>rp2</td>
</tr>
<tr>
<td>EngineeringProject2::close_problem</td>
<td>cp2</td>
</tr>
<tr>
<td>EngineeringProject2::create_new_release</td>
<td>cmr2</td>
</tr>
<tr>
<td>EngineeringProject2::close</td>
<td>c2</td>
</tr>
</tbody>
</table>

The CORBA protection system configuration described above allows enforcement of the sample policies listed on Page 8. For example, a lead of project 1 with role pl became able to invoke operations get_name and get_experience on all implementations of interface Employee as well as all but close operations on all implementations of interface EngineeringProject1.

\(^{11}\)We used first letters of each operation to create a corresponding right.

\(^{12}\)We could have used “any” as well. When an operation’s required rights set consists of only one right, the effect of either combinator is the same.
<table>
<thead>
<tr>
<th>Privilege Attribute</th>
<th>Rights</th>
</tr>
</thead>
<tbody>
<tr>
<td>e</td>
<td>gn, ge</td>
</tr>
<tr>
<td>ed</td>
<td>gd1, gd2, rp1, rp2</td>
</tr>
<tr>
<td>e1</td>
<td>mcl, rc1</td>
</tr>
<tr>
<td>pel</td>
<td>cn1</td>
</tr>
<tr>
<td>qe1</td>
<td>k1</td>
</tr>
<tr>
<td>pl1</td>
<td>cpl</td>
</tr>
<tr>
<td>e2</td>
<td>mc2, rc2</td>
</tr>
<tr>
<td>pe2</td>
<td>cn1</td>
</tr>
<tr>
<td>qe2</td>
<td>k1</td>
</tr>
<tr>
<td>pl2</td>
<td>cpl</td>
</tr>
<tr>
<td>dir</td>
<td>atp, up, af, f, c1, c2</td>
</tr>
</tbody>
</table>

Table 8: Granted Rights Matrix for Single Domain Solution.

From observing the configuration of the CORBA protection system in this solution, significant administrative overhead could be noticed. The overhead is due to the gratuitous use of a separate interface (EngineeringProject(1,2)) per project. This is because we purposefully limited our solution to a single access policy domain. It could be easily shown how the unnecessary redundancy of protection system configuration data is eliminated by using access policy domains and a hierarchy of such domains. We omit the description of a solution with multiple domains due to space limitation.

4 Conclusions

In this paper, we provided a definition of protection system configuration for CORBA Security service (CS). We defined RBAC0 and RBAC1 models in the language of CS and described how RBAC0-RBAC3 could be implemented in CS. We discussed what functionality needs to be implemented, besides compliance with CS standard, in order to support RBAC models by CS. We illustrated the discussion with a single access policy domain example of CS protection system configuration, which supports a simple role-hierarchy and access policies.

Implementations compliant with the CS specification can support RBAC0-RBAC3. However, additional functionality non-specified by CS is required. Implementations of PrincipalAuthenticator interface and User-Sponsor need to be aware of roles and their hierarchies (RBAC1). To support constraints (RBAC2), a PrincipalAuthenticator has to enforce corresponding constraints. Tools to administer user-to-role and role-to-rights relations are also required.

The work presented in this paper sets up a framework for implementing as well as for assessing implementations of RBAC models using CS. It provides directions for CS developers to realizing RBAC in their systems. It gives criteria to users for selecting such CS implementations that support models from the RBAC0-RBAC3 family.

Acknowledgements

We are grateful for very helpful comments from the anonymous reviewers. We also thank the OMG security special interest group (SecSIG) for feedback received during the presentation, in December 1997, on supporting RBAC in CORBA Security. Special thanks to Bob Blalder from DASCOM Inc. for insightful comments on the first draft of Section 2.

References

[ACM95] ACM. Proceedings of the First ACM Workshop on Role-Based Access Control, Gaithersburg, Maryland, USA, November 1995.


Appendix

Definitions of RBAC models from [SCFY96]:

**Definition 4.1** The $RBAC_0$ model has the following components:

- $U, R, P, and S$ (users, roles, permissions and sessions respectively),
- $PA \subseteq P \times R$, a many-to-many permission to role assignment relation,
- $UA \subseteq U \times R$, a many-to-many user to role assignment relation,
- $user : S \rightarrow U$, a function mapping each session $s_i$ to the single user $user(s_i)$ (constant for the session’s lifetime), and
- $roles : S \rightarrow 2^R$, a function mapping each session $s_i$ to a set of roles $roles(s_i) \subseteq \{ r | (user(s_i), r) \in UA \}$ (which can change with time) and session $s_i$ has the permissions $\bigcup_{r \in roles(s_i)}\{ p | (p, r) \in PA \}$

**Definition 4.2** The $RBAC_1$ model has the following components:

- $U, R, P, S, PA, UA$, and $user$ are unchanged from $RBAC_0$,
- $RH \subseteq R \times R$ is a partial order on $R$ called the role hierarchy or role dominance relation, also written as $\geq$, and
- $roles : S \rightarrow 2^R$ is modified from $RBAC_0$ to require $roles(s_i) \subseteq \{ r | (\exists r' \geq r) [ (users(s_i), r') \in UA ] \}$ (which can change with time) and session $s_i$ has the permissions $\bigcup_{r \in roles(s_i)}\{ p | (\exists r'' \leq r) [ (p, r'') \in PA ] \}$