Context Sensitivity in Role-based Access Control

Arun Kumar*, Neeran Karnik, Girish Chafle
IBM India Research Laboratory,
Block 1, Indian Institute of Technology,
Hauz Khas, New Delhi-110016, India.

Abstract

This paper describes an extended role-based access control (RBAC) model, which makes RBAC sensitive to the *context* of an attempted operation. Traditional RBAC does not specify whether the permissions associated with a role enable access to a *particular* object, or to some *subset* of objects belonging to a class. We extend the model by introducing the notions of role context and context filters. Context filters are Boolean expressions based on the context of the user attempting the operation, as well as the context of the object upon which the operation is attempted. By supplying context filters during the definition of a role, a security administrator can easily limit the applicability of users' role memberships to particular subsets of the target objects. We also describe our implementation of the model in a web-services platform, to illustrate how this technique is particularly valuable when the data is hierarchically structured.

1 Introduction

Role-based access control (RBAC) [SCFY96] is a popular methodology for specifying and applying authorization policies, to govern access to sensitive data. In RBAC, a security administrator assigns *users* to *roles* (see Figure 1). A role usually corresponds to an organizational function (e.g. Team Leader, Manager, Auditor), and is assigned the appropriate set of *permissions* by the security administrator. A permission can be thought of as the authority to perform an operation on one of the *objects* in the system. When a user attempts to perform some operation on a target object, the access control system intercepts this attempt and only allows it to proceed provided that the user is a member of some role that includes the necessary permission for that operation. The main benefits of RBAC are the ease of administration (permission assignment and revocation) of the security policy, and its scalability [SCFY96, SFK00].

However, the RBAC model does not specify whether the permission is applicable to a *particular* target object[1] or to *all* instances of a *class* of objects [Tho01]. This is usually left to the application to decide, and enforce. In most practical systems, the number of objects requiring access control is huge, making it impractical to define permissions for accessing each of them. The alternative therefore is to define permissions in terms of operations on classes of objects.

*All correspondence should be addressed to this author at kkarun@in.ibm.com

[1]We use the terms *instance* and *object* interchangeably hereafter.

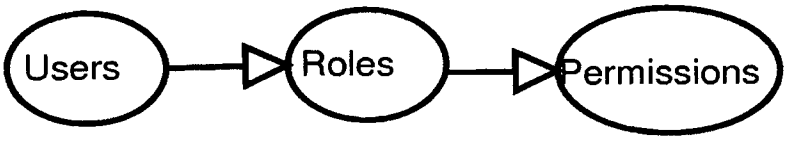

Figure 1: The basic Role Based Access Control model

For example, a permission may correspond to the `read` operation on the `StudentTranscript` class.

However, some ambiguity still remains, regarding which instances of the class are made accessible to a user by a permission. The user may be given the authority to assume a role while interacting with the system, but in general, his role membership provides the associated permissions only over a *subset* of the objects of a class. For example, a Student role has permission to read a transcript, but implicitly, only his/her own. The Principal of the school also has the same permission, but implicitly, over *all* students' transcripts, whereas a Teacher only has access to *some* transcripts – those of his/her own students.

An implementation of RBAC needs to enforce these implicit subset constraints in some application-specific manner. Various mechanisms have been used for this purpose, which we shall describe in Section 2. Our goal however, is to extend the RBAC model with a single construct that is powerful enough to encompass these disparate mechanisms. This would have the benefit of providing a standard mechanism that is not application-specific, and allowing a clean separation of access control code from the application logic. We start by categorizing earlier approaches to this problem in Section 2, and then describe a typical web-services environment that motivated this work, showing how plain vanilla RBAC is insufficient in that scenario. We then describe our model – *context-sensitive RBAC* – in Section 3, and briefly describe how we implemented it for our web-services system. A review of the related work follows, and finally we present some conclusions.

2 Background

2.1 Earlier approaches

The problem of limiting a user's access to a subset of objects has been tackled previously in various ways, which can be broadly categorized into the following:

- **Enumeration** involves a manual (and often static) specification of the subset of instances. In the student transcripts example, for each member of the Teacher role, the access control system may maintain a list of students in her class. The model described in [GMPT97] maintains a list of valid values for each of the security-relevant properties of the object classes in the system. These lists are effectively used to restrict the set of instances over which the user's permissions apply. Barkley et al. [BBU99] introduce the notion of *relationships* to determine whether a user's role-permissions hold good for a particular instance. However, this information is maintained by each instance as a list of authorized

users. Such lists are unwieldy, especially in large organizations, where there may be thousands of users and object instances. The implementation is also inefficient because these large lists have to be stored and searched frequently. This approach therefore, does not scale well.

- **Object grouping** involves creating groups of instances of the same class. A role-member acquires the associated permissions only for a specific set of such groups. This is more efficient and scalable than enumeration, since a list of groups is maintained rather than a list of individual instances. The criteria for including an instance in a group may be arbitrary, and the grouping is usually done manually, or outside the scope of the access control system. For example, on a Windows machine, a user places some files in a shared folder — corresponding to placing instances in a group — and gives certain roles access to the folder. This is easier than setting up an access control list for each shared file. Groups of objects may also be specified using wildcards, so that the membership of the groups is determined at runtime. The Generalized RBAC[MA01] model employs *object roles* that are basically groups of instances computed algorithmically on the basis of certain security-related properties of the objects. Similarly, *domains* are defined in [YLS96] as groups of objects over which a security policy applies, and roles are used for specifying these policies.

- **Constraints** are conditions that an object must satisfy in order that the user's attempt to perform an operation succeeds. These conditions involve security-relevant parameters of the attempted operation. This may include information gleaned from environment (such as the time of day, or whether it's a holiday), or state contained in the target object itself (e.g. a bank account's owner, its overdrawn status, etc.). These constraints are distinct from those defined in the base RBAC model itself [SCFY96], which constrain role definitions in order to avoid conflicting roles, promote separation of duties, etc. Systems such as [MA01, CLS+01] allow constraints, in the form of environment roles, that are purely dependent on external properties rather than the properties of the objects or subjects involved in the operation. The Role Object Model in [LS97] defines a role as a set of policies. Constraints involving properties of the objects are used to limit the applicability of those policies over object instances.

We now illustrate with the help of a real-life example, the need for a unifying mechanism that should be incorporated in the access control model itself, thereby relieving the application from the burden of implementing it in some non-standard way.

2.2 A Motivating Example

Consider a service delivery platform — an infrastructure for offering access to software in the form of services delivered across the Internet, on a pay-per-use basis. An organization must first register with the service provider (SP), thus becoming a *customer* of the SP. A customer can sign up for one or more of the services offered by the SP. Each such sign-up results in the creation of a *service instance* – a virtual copy of the service, dedicated to that customer. Each customer has its own *users* who may log in and use these service instances. The SP maintains a configuration database that includes such information as customer details, a list of the users,

their individual profiles, configuration parameters for various service instances, etc. The service delivery platform allows customers to administer their own details – e.g., creation of user profiles, configuration of service instances, etc. Therefore, an access control mechanism is needed to ensure that only authorized users can perform such administrative operations.

Each customer has (at least) one privileged user – a **Customer Administrator** – with overall authority over the customer's data. The customer administrator may designate some users as **User Administrators**. They have the authority to create, modify or delete user profiles for that customer. Similarly, **Service Administrators** may be designated, with the authority to sign up for new services, configure existing ones, etc. A service administrator may designate a user as the administrator of one or more service instances. Such an **Instance Administrator** can control and configure only those specific service instances.

In addition to these types of administrators, the SP may have one or more **Platform Administrators** with superuser-like privileges, including the ability to edit customer profiles, make new services available via the platform, etc. The SP may also have **HelpDesk Personnel**, who only have enough authority to enable them to assist customers in using the offered services. For example, they may be allowed to reset passwords for certain customers' users, but not to create new user profiles.

These categories of administrative users have well-defined permissions over specific classes of objects in the SP database. They can therefore be naturally modelled as roles in an RBAC system. Roles may be simulated by maintaining groups of users, using programming constructs such as arrays, sets, etc. In the service delivery platform however, it is not adequate to create one group of say, User Administrators (UA). This is because a UA must only be allowed to create or modify profiles for users belonging to the same customer as the UA himself. Therefore, one such group of UAs is created for each customer, as shown in Figure 2. The application must include the access control logic to ensure that the user attempting a `create-user` operation belongs to the appropriate UA group. Object grouping is used implicitly here – all user profiles of a customer constitute one object group. A particular member of the UA group then has access to the group of user profiles of that customer.

A more complex scenario arises when a user attempts to configure a service instance. Instance Administrators have this authority, but only for a specific subset of a customer's service instances. Therefore, the system needs to maintain one group of Instance Administrators per service instance (e.g. the `S1admins` group in Figure 2), and add the user to all such groups for which he is authorized. When the operation is attempted, the application must ensure that the user belongs to the same customer that owns the service instance, and further is an Instance Administrator of that service instance. This creates two problems:

- the proliferation of such groups of users to represent roles, which affects manageability (permission assignment, revocation and checking), and
- the embedding of access control logic in the application code.

This arises because of the limitation of RBAC identified in the previous section – the need

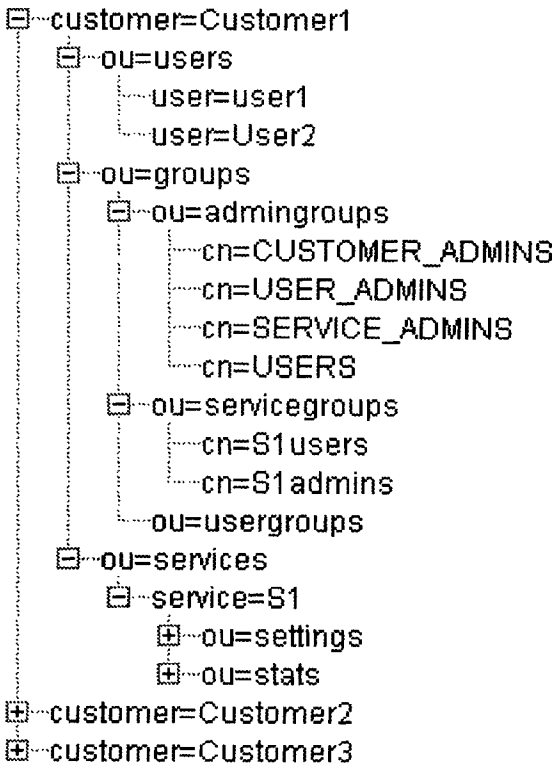

Figure 2: Customer data in the Service Delivery Platform

to identify a subset of instances to which a role membership applies. We now introduce an extended RBAC model that addresses these problems.

3 The Context-sensitive RBAC Model (CS-RBAC)

The previous section highlighted the need for support in the access control model to capture additional information about target objects. We now propose a unifying RBAC model, which not only captures such context information about target objects, but also does the same for users. This provides a comprehensive context to each operation, using which the access control system decides whether a role's permissions are valid for a given user-object pair.

In Context-sensitive RBAC (see Figure 3), we define a *permission* as the authority to perform a specific operation on a class of objects. In object-oriented systems, this corresponds to a method defined on a *class*. A *role* is then defined as a collection of such permissions. A *user* may be granted membership of a role, but the role membership is only valid within a certain *context*. This *role context* is a new construct that limits the applicability of the role's permissions to a subset of the instances. The advantage of this approach is that the role context can be specified by the system administrator at the time of role creation, and remains valid unless the role definition itself changes (a rare occurrence). Nevertheless, the decision whether

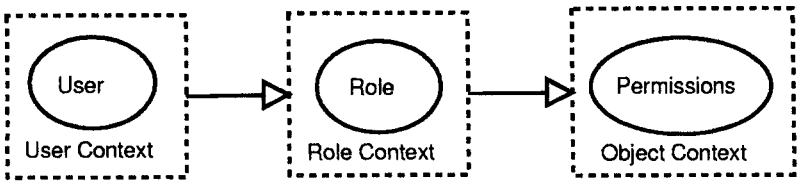

Figure 3: The Context-Sensitive RBAC Model

to allow/disallow an operation is made at runtime, based upon the current context at that time.

In order to define a role context, we first define two other forms of context. A *user context* captures all security-relevant information about a particular user. In our service delivery platform example, the user context includes the identity of the customer to which the user belongs. More generally, it may describe the organizational hierarchy under which the user exists. In similar fashion, the *object context* captures security-relevant information about the target object. For example, the context of a `ServiceInstance` object in our platform has attributes such as the identity of the customer that owns the service instance, the type of service, etc.

A role context can then be composed from user and object contexts by specifying a Boolean constraint expression, which we call a *context filter*. The operands in this expression are the attributes available in the user and object contexts, while the operators include the standard comparison[2] and logical[3] operators. In general, the attributes that constitute the user and object contexts are application-dependent. Therefore, we give the application developer the flexibility to define any security-relevant attributes in these contexts. For example, it may include an expiry date/time in order to enforce a limited-duration authorization of the user account. A subset of these attributes can then be used in composing the context filter for a role. The same filter is applicable to all users in that role. [The interested reader may refer to Appendix A for a formal treatment of the CS-RBAC model].

RBAC has traditionally been proposed as the appropriate model for large organizations, because role hierarchies map naturally onto the organizational hierarchy of the users in such organizations. The CS-RBAC model inherits this benefit from RBAC, and is particularly useful in scenarios where the target objects are similarly hierarchically organized. Objects may have hierarchical *containment* or *ownership* relationships amongst themselves. These can be easily captured within the object context in CS-RBAC and used in the context filter for controlling access. The filtering can be done by matching such attributes against specific values, or against attributes in the user context that draw values from the same domain.

For example, in the service delivery platform, the application semantics demand that a User Administrator only have access to user profiles belonging to the same customer as the UA himself. The object context of a user profile includes an `ownerId` attribute that identifies the customer in the hierarchy to which the object belongs. The filter expression can compare this `ownerId` with the corresponding attribute in the user domain – `custId` – in order to limit

[2]Comparison operators include =, ! =, <, >, <=, >=

[3]Logical operators are the Boolean AND, OR and NOT

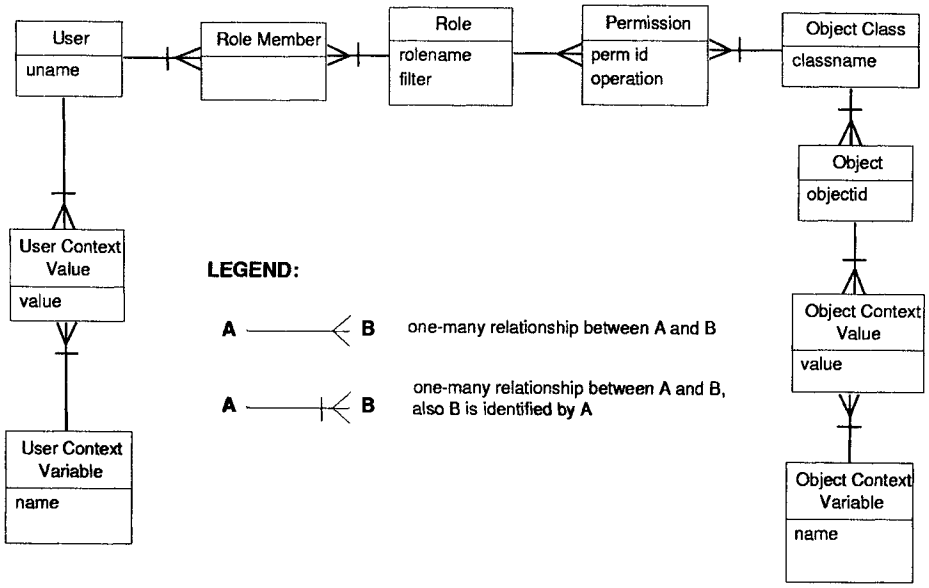

Figure 4: A data model for implementing CS-RBAC

a specific UA's access. However, a user in the HelpDesk role needs to perform the same task for users belonging to several customers. This could be implemented by providing an attribute in the user context that enumerates these customers. The context filter can then compare the `ownerId` of the object with each of the customers listed in the user context.

This context-sensitivity has the benefit of simplifying access control policy. The context filters can be easily replaced in a running system when the policy needs to be changed, and the change takes effect immediately. Further, the use of object groups or enumerations is limited in scope, to sub-trees in the object hierarchy – thus improving the scalability of the system as well.

4 Enforcement of CS-RBAC

We now describe how an access control subsystem can enforce the model described in Section 3. Figure 4 shows a schema for the access control data that needs to be maintained. The basic entities in the model are the User, Role and Permission. A Permission is defined as an operation on an ObjectClass, and a Role is then defined as a collection of Permission entities. The Role also contains the context filter expression. When a security administrator assigns a user to a role, a corresponding RoleMember entity is created. The user and object contexts are stored in the form of name-value pairs. The names of these context variables are determined

by the application and used in defining the filter. The values are supplied to the access control subsystem at runtime, when it is invoked to check an attempted operation. This access control check is modelled as a function call (see Figure 5). The application passes the name of the attempted operation, the caller and the target object to this function, and receives a Boolean value in return – indicating whether the attempted operation should be allowed, or not.

```
function allowOp (operation, caller, target_object) : boolean
begin
- P = perm_reqd (operation) // the permission required for
                            // the attempted operation
- RP = roles_with(P)        // set of roles that have
                            // permission P
- RU = roles(caller)        // set of caller's active roles
- R = intersection(RP,RU)
- allowFlag = false
- For each role r in R
    o UC = user_context(caller)
    o OC = object_context(target_object)
    o Evaluate CF(r, UC, OC)  // CF is the context filter
    o if result = true
      begin
         allowFlag = true
         exit the loop
      end
- return allowFlag
end
```

Figure 5: Algorithm for enforcing CS-RBAC

The access control system first identifies the user's role memberships that permit the attempted operation. For each such role, the corresponding role context is populated by plugging in values from the calling user's and target object's contexts, and its context filter is evaluated. If *any* of the filters evaluates to `true`, the operation is allowed to proceed.

The expression language used for defining context filters includes two keywords – `ObjectContext` and `UserContext`. A variable in the object context would be represented in the filter as `ObjectContext.`<*name*>. The expression language should include the usual comparison and logical operators, as well as any others that the implementation of CS-RBAC may choose to define. The `Evaluate CF()` function referred to in Figure 5 incorporates an expression evaluator for this language.

5 An Implementation of CS-RBAC

We implemented the CS-RBAC model in the web-services scenario discussed in Section 2.2. CS-RBAC is used to control access to the customer database – an LDAP[4] [WHK95] directory. The implementation setup is shown in Figure 6. The service delivery platform is accessed by users of various customers via a web interface, written using JSPs and servlets running on a Tomcat web server. These web operations are then translated into SOAP[5] [BEK+00] calls on the *administrative server*. The administrator server validates each SOAP call using the *access controller* object, before performing the appropriate operations on the customer database. The access controller maintains an *ACL database* that stores access control information using a schema similar to that shown in Figure 4.

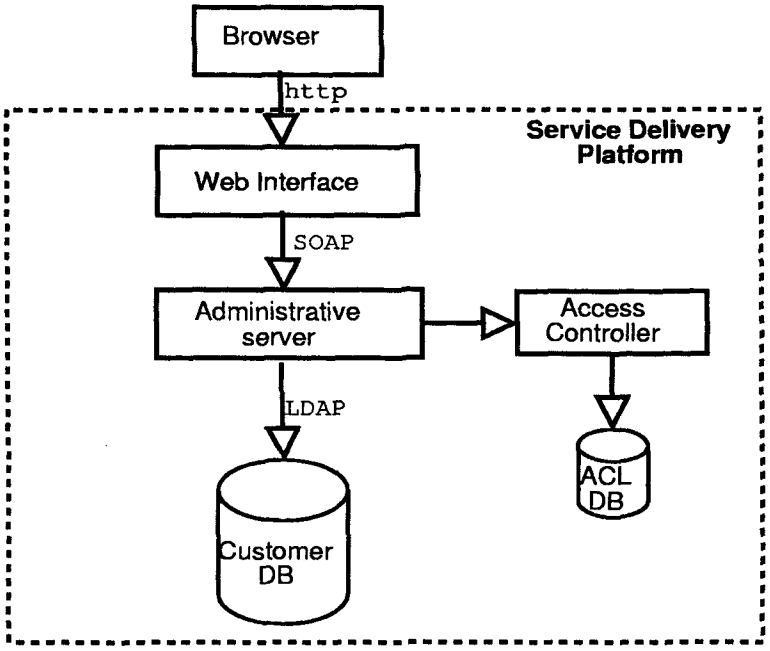

Figure 6: The implementation setup

We now describe how CS-RBAC was implemented, using an example from this web-services scenario. Consider the Service Administrator role defined in Section 2.2, which enables a user to sign up for new services and perform configuration-related operations on service instances. The permissions for this role are defined in terms of operations such as `create()`, `delete()` and `setUserLimit()` on the `ServiceInstance` class.

The user and object contexts supplied by the application are implemented as Hashtables, containing name-value pairs. The user context includes `custId` – the name of the customer to which the user belongs. The object context for the service instance object contains `ownerId`

[4]Lightweight Directory Access Protocol
[5]Simple Object Access Protocol

– the customer that owns the service instance. Thus, we can specify the context filter for the Service Administrator role using this simple expression:

```
ObjectContext.ownerId = UserContext.custId
```

When the user logs on through the web interface, the system determines her set of roles. Assume that the user invokes the `delete()` operation on a service instance object. The application then populates the user and object contexts with values corresponding to the variable names defined above, and invokes the access controller. The access controller executes the algorithm shown in Figure 5, and determines whether the operation should be allowed. If the algorithm returns `true`, the service instance is deleted. Otherwise, an error message indicating authorization failure is displayed.

6 Related Work

Context sensitivity has been explored for workflow and collaborative tasks in [CBE00] [Wan99], where *context* refers to the state of the workflow or task. In this section, we only consider related work that aims to restrict the applicability of roles over object instances.

The model proposed in [GMPT97] is intended to provide RBAC based permission assignment on object types at an enterprise level, and yet be able to activate and control permissions on individual users and object instances at a later time. However, the notion of a *team* – which is a set of users in various roles needed for a task – is the central and governing concept in the system. Their user context and object context are also defined for a team rather than a role. A team context, expressed in terms of ranges of values for certain security-relevant attributes, is used to restrict the object instances over which the permissions of a user apply. However, the valid ranges of values are maintained as lists, which essentially is enumeration and does not scale up.

Barkley et al. [BBU99] use the concept of relationships to identify whether a user U having a role R, that applies over a particular object type O, has an active relation with the instance of O under consideration. The requested operation is allowed only if the set of active relations between U and O contain the one required by the access control policy. However, their implementation also uses enumeration as the means to capture the relationship information.

The Policy based role object model presented in [LS97] defines a Role as a set of policies, where each policy represents an obligation or an authorization of a user over an object. Objects may belong to *domains*, which are groups formed for the purpose of management, configuration etc. These domains appear to be statically defined, as they are not programmatically computed at runtime. Constraints are used to restrict the applicability of the policies in a role, to a subset of objects in a domain. The users are not grouped and are not captured in the constraints. Therefore, the model fails to satisfy the requirements of the scenario described in Section 2.2. For example, the access control system cannot check that a user requesting a `create-user` operation is a UA of a specific customer. Each constraint applies to a single permission rather than a whole object. While this permits fine-grained control, it makes for a large and unwieldy

access control policy overall.

The Generalized RBAC model [MA01] introduces the notion of Object roles, Subject Roles and Environment Roles. Subject roles are traditional RBAC roles for users whereas object roles are basically groups of object instances computed algorithmically on the basis of certain security-related properties of the objects. Environment roles capture access-control relevant information from the environment, such as time of day, system load etc. The use of *transducers* [GJST91] is suggested to algorithmically compute the properties of an object. This enables automatic assignment of membership to object roles instead of having to do it manually. However, this technique only allows physical and temporal properties of an artifact (such as a file on disk) representing the real life entity to be captured, not the security-relevant properties contained by the entity itself. The automatic computation of properties does not address our requirements, which need the automatic computation of target objects based on their properties. The alternative is to define the object roles manually, resulting in loss of scalability. Furthermore, the model supports separate policies for users and objects. This leads to potential conflicts, and requires extra support that adds overhead.

The environmental context, as in GRBAC above, can be captured in our model through additional variables supplied by the application via object context or user context. The role context itself could be easily extended to include an environment context. Alternatively, keywords could be added to the filter language to allow environmental variables to become part of the filter expression. Since obtaining the values of these environmental variables would be platform-specific and is not the function of the access control subsystem, the onus of supplying their values lies on the application. We opted not to incorporate a separate environment context, in order to maintain the simplicity of the model.

7 Conclusions

We have proposed CS-RBAC — a unifying model that extends RBAC to make it sensitive to the contexts of both the user and the target object. This addresses the ambiguity in RBAC of identifying the subset of objects of a particular type, to which a permission should apply. In doing so, it encompasses implementation-dependent techniques such as enumeration and object grouping used in earlier efforts. In addition, it enables the filtering of access based upon the context of the user as well. This allows us to easily capture users' organizational hierarchy and objects' ownership hierarchy, as illustrated using the example of a service provider's database. The resultant access control policies are flexible, since context filters can be easily replaced in a running system. By extending the language used to define context filters, we can incorporate environmental information into the decision-making process as well.

An effort has been initiated recently to standardize various aspects of Role Based Access Control [SFK00]. The authors observe that the standardization of permissions is beyond the scope of a general purpose access control model. Providing support for context-sensitivity in the standard model, as proposed in this paper, would strengthen the expressiveness of RBAC policies.

In future, we plan to demonstrate the power of context sensitivity in RBAC for the administration of roles themselves. As noted in [SCFY96], RBAC can be used to manage RBAC itself. We believe that the scoping of authority of administrative roles can be easily captured using the proposed model.

Acknowledgements

We thank B. Nagender Reddy and Matulya Bansal of the Indian Institute of Technology, Guwahati, for their role in the initial stages of this work. We also extend our thanks to Ruby Arora of IBM India Research Lab., for her valuable help in implementing the model.

References

[BBU99] J. Barkley, K. Beznosov, and J. Uppal. Supporting Relationships in Access Control Using Role Based Access Control. In *Proceedings of the fourth ACM workshop on Role-based access control*, October 1999.

[BEK+00] Don Box, David Ehnebuske, Gopal Kakivaya, Andrew Layman, Noah Mendelsohn, Henrik Nielsen, Satish Thatte, and Dave Winer. Simple Object Access Protocol (SOAP) 1.1. http://www.w3.org/TR/SOAP, May 2000.

[CBE00] D. G. Cholewka, R. H. Botha, and J. H. P. Eloff. A Context Sensitive Access Control Model and Prototype Implementation. In *Proceedings of the IFIP TC11 15th International Conference on Information Security, Beijing, China (2000), pp. 341-350.*, 2000.

[CLS+01] M. J. Covington, W. Long, S. Srinivasan, A. K. Dey, M. Ahamad, and G. D. Abowd. Securing Context-Aware Application Using Environment Roles. In *Proceedings of SACMAT*, May 2001.

[GJST91] David K. Gifford, Pierre Jouvelot, Mark A. Sheldon, and James W.O. Toole. Semantic file systems. In *Proceedings of ACM SIGOPS Symposium on Operating Systems Principles*, October 1991.

[GMPT97] C. K. Georgiadis, I. Mavridis, G. Pangalos, and R. K. Thomas. Flexible Team-Based Access Control Using Contexts. In *Proceedings of ACM RBAC97*, 1997.

[LS97] Emil Lupu and Morris Sloman. Policy Based Role Object Model. In *Proceedings of the Enterprise Distributed Objects Conference*, October 1997.

[MA01] M. J. Moyer and Mustaque Ahamad. Generalized Role Based Access Control. In *Proceedings of the International Conference on Distributed Computing Systems*, October 2001.

[SCFY96] R. S. Sandhu, E. J. Coyne, H. L. Feinstein, and C. E. Youman. Role Based Access Control Models. In *IEEE Computer, Volume 29, Number 2, pages 38-47*, February 1996.

[SFK00] R. Sandhu, D. Ferraiolo, and R. Kuhn. The NIST Model for Role-Based Access Control: Towards A Unified Standard. In *Proceedings of the fifth ACM workshop on role-based access control on Role-based access control*, July 2000.

[Tho01] R. K. Thomas. Team-Based Access Control: A primitive for applying role-based access controls in collaborative environments. In *Proceedings of SACMAT*, May 2001.

[Wan99] Weigang Wang. Team-and-Role-Based Organizational Context and Access Control for Cooperative Hypermedia Environments. *Proceedings of ACM HyperText'99*, February 1999.

[WHK95] W.Yeong, T. Howes, and S. Kille. Lightweight Directory Access Protocol. http://www.ietf.org/rfc/rfc1777.txt, March 1995.

[YLS96] N. Yialelis, E. Lupu, and M. Sloman. Role-based Security for Distributed Object Systems. In *Proceedings of the IEEE fifth Workshop on Enabling Technology: Infrastructure for Collaborative Enterprises, WET ICE*, June 1996.

A A formal model for CS-RBAC

A formal definition of context-sensitive RBAC model is presented below :
R = set of roles defined in the system
U = set of users (subjects) of the system
C = set of classes of objects (targets) in the system
M = set of methods/operations on the object classes in C
O = set of all object instances
O_z = set of instances/objects of class Z

P $\subseteq$ M x C, i.e. { (m,c) | m $\in$ M, c $\in$ C, m $\in$ methods(c) }, is the set of permissions
PA $\subseteq$ P x R, the many-to-many permission-to-role assignment relation
UA $\subseteq$ U x R, the many-to-many user-to-role assignment relation
UC = the set of all security-relevant attributes of the user,
i.e. the user context
OC = the set of all security-relevant attributes of all target object classes
OCz = the set of all security-relevant attributes of target object class Z,
i.e. the object context

Contextual Constraint **CC**: $\bigcup_{L \in C} \{2^{UC} x 2^{OC_L}\} \rightarrow 2^{UxO}$, is a function, for a role, mapping a pair of user context and object context to a set of individual (user, object) pairs

Role Context then is defined as **RC** = <UC, OC, CF>, a three-tuple consisting of the user-context, the object-context and Context Filter, where

Context Filter **CF**: U x O $\rightarrow$ {0,1}, is a function, that returns *true* if, for a role, the given (user, object) pair belongs to the set of (user, object) pairs allowed by the contextual constraint CC of that role.

i.e., $\forall$ u $\in$ U, o $\in$ O,

$$\begin{aligned} \mathbf{CF(u,o)} &= 1, \text{ if (u,o)} \in \text{CC(uc,oc) where } \{ \text{ uc}|\text{uc} = \text{UC(u)}\} \wedge \{ \text{ oc}|\text{oc} = \text{OC(o)}\} \\ &= 0, \text{ otherwise} \end{aligned}$$

A *Context-Sensitive Permission* **S** is, therefore, defined over U x P x O as
{(u, p, o) |$\exists$ r $\in$ roles(u) such that (p, r) $\in$ PA $\wedge$ CF(u, o) = true $\wedge$ u $\in$ U $\wedge$ o $\in$ O $\wedge$ p = {(m,c) | m=method(c) and c=class(o)} }