Elements of A Language for Role-Based Access Control

M. HITCHENS & V. VARADHARAJAN
Distributed Systems and Network Security Research Group,
University of Western Sydney Nepean

Key words: Role Based Access Control, Object-Oriented, Language

Abstract: A language for specifying role-based access control (RBAC) policies is presented. The language is designed to support the range of access control policies of commercial object systems. The basic structures of RBAC, such as role, users and permission, are present in the language as basic constructs. The language is flexible and is able to capture meta-level operations. The language also provides a mechanism for tracking actions and basing access control decisions on past events.

1. INTRODUCTION

In a computing system, when a request for a certain service is received by one principal (an agent) from another, the receiving principal needs to address two questions. First, is the requesting principal the one it claims to be? Second, does the requesting principal have the appropriate privileges for the requested service? These two questions relate to the issues of authentication and access control (authorisation). Recently, there has been extensive interest in Role Based Access Control (RBAC) [1,2] as an alternative to the more traditional discretionary access control (DAC) and mandatory access control (MAC) approaches. In RBAC models the attributes used in the access control are the roles associated with the principals and the privileges associated with the roles. In a previous paper [10] we have discussed some of the issues which need to be considered in the design of a language for specifying RBAC policies. These issues include the basic constructs of the language, the question of ownership and the recording of the history actions so that access control decision may be based on past events. In this paper we present some of the details of the language Tower, which has been specifically designed for the expression of RBAC policies in object systems and which addresses these issues.

The original version of this chapter was revised: The copyright line was incorrect. This has been corrected. The Erratum to this chapter is available at DOI: 10.1007/978-0-387-35515-3_53

S. Qing et al. (eds.), *Information Security for Global Information Infrastructures*

The next section describes the basic structure of the language. Section 3 outlines the basic constructs of Tower, including roles, users, and permissions.. Finally section 4 gives concluding remarks. A full version of this paper, including further motivation and examples in the language specifying commonly used access policies is in [11].

2. ACCESS CONTROL LANGUAGE: TOWER

The most important structures in Tower are the definitions of users, roles, and permissions. The details of these structures are discussed in subsequent sections but first we wish to make the following general comments. Each structure is declared and is given a name. The name is used to identify the structure throughout the access control system. The problems of a single level name space are avoided by employing block structured scoping. Each of these names must be unique within a particular scope. A new structure instance may be created and assigned to a structure variable. The closure of a structure includes any variables declared in the same scope. The structures are immediately available upon creation for evaluating access requests. They may also have their values modified in code that is subsequently executed. In this paper we do not specify the management interface of the access control system. We envisage that both users and administrators can enter policies (in the form of Tower expressions) into the system. Whether this is in a form similar to the Adage VPB [8] or by some other means is not relevant to the design of the language itself.

For an access control system to function it will require some capacity for storing information about the objects it manages and the access policies to be enforced. The Tower language allows the specification of information internal to the access control system in the form of variables. There are two distinct categories of variables in Tower, which differ in the type of information stored, their scope and use. These categories are simple variables (henceforth referred to as variables) and structures. The types of (simple) variables supported are standard ones such as integer, real, boolean, string, userid (user identity) and sets.

From the point of view of such variables, Tower is a block-structured language. A block consists of the definition of either a role or permission or statements between matching *begin* and *end* statements. Within a block, variables are declared before any roles, permissions or interior blocks. A variable is in scope within the block in which it is declared, within any structures declared within that block and within any interior blocks (except for further declarations using the same variable names) and any constructs defined within them. Variables declared within permissions or roles are only in scope within those constructs. Variable declarations have the following syntax:

var_name [= *value*], var_name [= *value*], ... **:** var_type

except for set variables, which are declared as

setvar_name [= *value*], setvar_name [= *value*], **: set of** element_type

A set variable of any set type may be empty.

The optional section after each variable name allows the value of variables to be initialised when declared. The value of variables can be altered in subsequent code,

especially in the action sections of privileges (see below). The values of variables may be tested within condition expressions and constraints. Any attempt to access a value of a variable before it is initialised results in an error.

Each variable name may be followed by a * or a & (or both). These control the actual number of instances of the named variable and their effect within the current scope. If neither symbol follows the variable name in the declaration then only a single variable is created. If the variable name in the declaration is followed by either or both of these symbols then more than one variable, each with the same name, is (potentially) created within this scope. If a variable's name is followed by a *, then a separate such variable is created for each object covered by the permission(s) within the scope of the declaration. If a variable's name is followed by a &, then a separate such variable is created for each user whose access requests involve this scope. If both symbols occur, then a separate variable is created for each user/object pair. As it cannot be always known in advance which users and objects will be involved, these variables are created dynamically as required. As accesses to variables only occur when a request to a specific object by a specific user occurs, it is straightforward for the system to determine which variable is to be used in any particular case.

Structure variables cover the following constructs within Tower: privileges, permissions, roles, users, ownership and blocks. Their values must be initialised before use (the exception to these provisions is blocks). The details of how values for these structures are created are covered in the following sections. Apart from the obvious differences between structures and variables in terms of syntax and value, the chief difference between them is the scope of structures. Unlike variables, which are only in scope within the block or structure within which they are declared, structures can be in scope within the entire access control system. The decision on scoping must be made when the structure variable is declared. Global scope is the default; if a structure's scope is not to be global then its name must be followed by a '@' character in the declaration. The unique user identification of the user who created the structure can be pre-pended to its declared name to ensure uniqueness. The exception to the above are blocks defined by *begin* and *end* keywords. Any such block is considered to be global if it is not defined within another block. Blocks do not need declaration but can be given a name.

The name of a block can be used to add additional structures or variables to the scope it represents. That is, Tower is not a statically scoped language but to some extent is dynamically scoped. This is related to database schema evolution.

Many constructs within Tower are based upon sets. The language provides a number of operations upon sets for all of these constructs; union, difference, intersection, test for inclusion, cardinality, equality, subset test The operators are type-sensitive, i.e. the types of all the sets involved must match and the types of the elements must match the declared element type of the sets.

3. BASIC RBAC STRUCTURES

In a previous paper [10] we refined the basic RBAC model and proposed one which explicitly includes the objects to which access is being controlled. We also introduced another structure, for which we use the term *privilege*. Ownership of an

object may be vested in a user, a role or any combination thereof. Permissions apply to one or more objects and privileges specify the methods to which the privilege grants access and the conditions under which they may be accessed. In this section, we describe the representation of the basic RBAC elements in Tower.

3.1 Privileges

In an object-oriented system, it is reasonable to base the lowest level construct of the access control system at the method level. In Tower, a privilege is a triple, consisting of the set of names of the methods to which it gives access, the condition under which access is granted and any action to be taken within the access control system if access is granted. A new privilege is created as follows:

```
privilege_name := privilege
    [condition_expression]
    [action_statement,action_statement,...]
    {method_clause,method_clause,...}
end_privilege
```

The condition expression and the set of action statements are optional. The condition expression is a Boolean expression (of arbitrary complexity) which must evaluate to true if any of the methods is to be invoked under the authority of this privilege. A condition expression can test both the values of parameters passed and access control system variables in scope. The action statement (or statements) is (are) executed if the invocation of any of the methods is allowed under the authority of the privilege (the default) or *whenever* the condition expression is tested, by preceeding each action statement to be executed with the keyword **always**. When an action statement is executed, the state of the access control system is altered. A *method_clause* is either a method name or a set of method names.

Note that there is no specification within a privilege as to the objects to which it applies. This is handled at the permission level. While users will probably have access to multiple methods of each object, they will not necessarily be able to access those methods under the same condition. We therefore associate conditions and methods in privileges and group privileges together with a specification of which objects they apply to within the permissions. For those methods of an object to which the same conditions apply, they may be grouped together in the method set of a privilege.

3.2 Permissions

Permissions encapsulate the access to objects of a single class. A permission consists of a specification of the objects to which it gives access and how these objects can be accessed. The latter is specified as a set of privileges. A permission will give access to some subset of the objects of the class to which the permission applies. Normally the subset will be a proper subset and not all the objects of the class. This restriction reflects the observation that normally a user will not have access to all the objects of a class (unless they are the only user who can access objects of that class). It would be an unusual situation where, for example, a single

user would have access to all spreadsheets or all text documents in a multi-user system. However, it is usually impossible to specify in advance the names (or other identifiers) of all the objects of a given class to which a user will have access. A permission can specify that it allows access to objects of a class owned by a given set of users. This allows access control to be specified for objects which have not yet come into existence. The syntax for a creating a new permission is as follows:

```
permission_name := permission
    class_name
    [owner]
    [users user_set]
    [roles role_set]
    [objects object_set]
    [variable_declarations]
    privileges {privilege_clause,privilege_clause,...}
end_permission
```

The *class_name* gives the name of the class of the object to which this permission grants access. After that, we have clauses specifying the objects covered by the permission. A permission may contain one or more of these clauses. These clauses are the first test on whether access will be granted by the permission.

The objects to which the permission will grant access may be specified in terms of their ownership. If the keyword **owner** is employed then the permission can grant access to objects of the named class owned (singly or jointly) by the user attempting to gain access. The permission may grant access to objects of the named class owned by any of the listed entities. This may be a set of explicitly named users or users which currently have the named role as an active role. The permission may be defined to give access to a set of existing objects by explicitly naming them. The permission can then be used to access those objects and no others. Finally the object set may be a named object set, which can be dynamically updated without directly accessing the permission.

If an access to an object is attempted which is not to one of the specified objects then this permission will not grant access.

Of course, even if the object that is being accessed is one covered by the permission, access may still be denied according to the privileges included within the permission. Tests for ownership may also occur in the condition sections of privileges, but such tests are additional (not an alternative) to the permission level tests.

After the specification of the objects to which the permission applies any variables that are in scope within the permission are declared. Finally, there is a set of privileges which define the exact access allowed by the permission. A *privilege_clause* is either a privilege, a privilege set or a *privilege_expression*. A *privilege_expression* is an expression specifying changes to a privilege (such as adding or subtracting methods, conditions or actions).

The following gives an example of the initialisation of a permission and the effects of ownership. A user, *a*, wishes to access the objects of class *text_object* owned by user *b*. *a* enters the following code:

```
b_text := permission
    text_object
```

```
    users b
    {privilege,privilege,...}
end_permission
```

The code is syntactically correct and the permission will be created *if* both the owner of the class definition for *text_objects* and user *b* give their permission. The method by which they would do this relates to the management interface and is outside the scope of this paper. The management interface and operations are addressed in a separate paper that is currently in preparation.

3.3 Roles

The syntax for creating a new role value is as follows:

```
role_name := role
    [variable_declarations]
    [authorised constraint_expression
    [constraint_action]]
    [active constraint_expression
    [constraint_action]]
    [session constraint_expression
    [constraint_action]]
    [roles {role_clause,role_clause,...}]
    [permissions {permission_clause,permission_clause,...}]
end_role
```

Role constraints may be used to affect the roles of a user at three different levels

- the roles that a user may be authorised to have as active,
- the roles that a user has active across concurrent sessions
- the roles that a user has authorised within a particular session

These are in increasing level of refinement – if a role specifies that no user can have both it and another role as authorised roles, then obviously the user can not have both those roles as active roles (either in the same session or in another one). Constraints-may be used to impose restrictions upon whether a user may have this role added to his/her set of roles, or whether a user may add another role while possessing this one. The constraint tests in a role are checked when a user to role mapping is made (the role is to become an authorised role) and when a session to role mapping is made (the role becomes active). The constraints are also checked whenever any relevant user mappings are altered. For example, this avoids the necessity of specifying exclusion in both roles. A constraint test is a Boolean function which must evaluate to true if the role is to be added. A short hand is provided for the common case of exclusion, which is that possession of the current role is mutually exclusive with the roles in the role set.

```
exclude role_set
```

This set can be explicitly listed in the constraint expression or represented by a set variable, allowing easier dynamic update.

The constraint action allows for updating of any variables relevant to the constraint. Variables have been discussed above. The role and permission sections define the access allowed by the newly created role. The definitions of *role_clause* and *permission_clause* are analogous to that of *privilege_clause* in section 4.2. Role inheritance is modelled by allowing roles to be formed, in part, from other roles. These roles may already exist, and are referred to by name, or are defined within the new role.

3.4 Users and Sessions

The syntax for creating a new user structure is as follows:

user_name := **user**
    name
    uuid
    [variable_declarations]
    [{*role*,*role*,...}]
    [{*session*,*session*,...}]
end_user

Note that the roles are those which the user *may* take on (known as the authorised roles of that user). When a new user is created this set may often be empty. In addition to explicitly naming roles, one or more role sets could also be given. The variable declaration section allows attributes to be assigned to the user. The sessions of the user will only be updated by the system, reflecting the current sessions of the user.

For each log-in session of a user, it is also necessary to record the actual roles that are current (known as the active roles). It is the active roles that are used to check whether any attempted method invocation should be allowed.

The syntax for a session is

session_name := **session**
    user_name
    uuid
    [{*role*,*role*,...}]
end_session

Note that in some sense this a conceptual syntax, as such structures would be implicitly created whenever a new user session is commenced. However, they have an actual existence and are used in checking role constraints as well as actual method invocation. Here the roles are the active roles for the particular session.

3.5 Ownership of Objects and Structures

The concept of ownership can simplify the expression of access control policies. Many systems limit ownership to a single user. This does not match many real world situations, where ownership is often equally shared between many people. For example, all members of the committee may jointly own a document produced

by a committee. Vesting ownership in more than a single entity leads to the question of how many of these entities must co-operate for successful performance of actions restricted to an owner. Many discussions of RBAC ignore the question of ownership completely. In Tower we employ a relatively simple answer to the question: for each object, the number (or fraction) of the joint owners who must agree before an action can be performed is stored along with the ownership information.

Each object (and class specification) stored in the system has a corresponding access control structure. These structures record the owner(s) of the corresponding object and other related information (such as attributes). While the creation of the ownership structures is automatic on the creation of the corresponding object, they have a conceptual Tower syntax. This allows updating of the ownership information within the scope of the language.

name := **object**
    owners
    {*uuid, uuid, ...*} | {*role, role, ...*} | {*uuid, uuid, ...*} {*role, role, ...*}
    quorum positive integer | real between 0 and 1 | **all**
    creation {*uuid, uuid, ...*} | {*role, role, ...*} | {*uuid, uuid, ...*} {*role, role,...*}
    [variable_declarations]
end_object

The name of the structure is the system dependent unique object identifier. The first clause specifies the owner of the object, as one or more specified users and/or the members of named roles. The second option allows for a dynamic concept of ownership, as it grants joint ownership to all users who currently have at least one of the named roles as an active role[1].

The second clause specifies how many of the owners must agree if any operation requiring owner approval is to be carried out. For an object there are only three such operations

- ξ changing any of the information stored in the ownership structure (including the specification of the owner),
- ξ allowing the object (or class specification) to be referenced from within a permission, and
- ξ revoking the allowance for the object (or class specification) to be referenced from within a permission.

The second operation prevents users from including objects within a permission when they do not own that object. The third operation allows for revocation of access.

The third *creation* clause specifies the owner of any object created as a direct result (i.e. without subsequent accesses to other objects) of access to this object. For example, while the owner of a text editor may be the system manager, any files created using the text editor can be specified as belonging to the user who accessed it.

[1] While we could have simply allowed the role to be an authorised role, insisting that it must be an active role helps to protect untrusted code running used limited permissions.

The same principles of ownership can be applied to structures of the access control system (roles, permissions, privileges, users). The syntax is the same as that given above, except that keyword *object* is replaced with *structure*. The name of the ownership structure is that of the structure to which it applies, followed by the special character "^". This allows us to control access to the access control system itself in a conceptually efficient manner. Each structure in Tower has an associated ownership structure. The ownership information in such an ownership structure also applies to itself, avoiding infinite recursion. Thus it is possible to specify who owns each structure and can therefore modify it. This also allows us to restrict the use of access control structures; they can only be altered or used (included in the values of other structures) either by their owner, or with their owner's permission. In the case of removing one structure from another (such as removing a role from a user's list of authorised roles), the permission of the owner of either structure is sufficient.

3.6 System Evolution : Alterations to Structure Values

The previous sections have described how the various structures of the language are given their initial values. As the system evolves, any of these structures may need to have their values updated. Set operations may be applied to each of these structures, for example

```
P1 := P1 + {Pr1,Pr2}
```

Permission *P1* now has privileges *Pr1* and *Pr2* added to its set of privileges. The type of *Pr1* and *Pr2* (i.e., privilege) means that the update must be to the privilege set of the permission. Therefore we can simply use the permission name without further qualification. This applies to all the components of structures that can be unambiguously identified. Where a structure consists of two or more sets of the same element type, such as the record of the owners of an object and the owners of any new objects, further qualification, and updates occur as follows:

```
object1.owners := object1.owners + {michael}
object1.creation := object1.creation + {vijay}
```

The first statement adds the user *michael* to the set of users who own object *object1*. The second statement adds the user *vijay* to the set of users who will own any objects created using *object1*.

From the above, the set operations applied to a privilege alter the contents of its set of method names (as the only set contained in a privilege is the method set). Similarly, the roles and permissions which make up a role can be altered, as in the following examples:

```
R1 := {P1,P2}
```

The permissions in *R1* are now *P1* and *P2*.

```
R1 := R1 + {R2,R3}
```

R1 has *R2* and *R3* added to its roles

```
R1 := R1 - {R3}
```

R3 is no longer one of *R1's* roles

The system can determine if the roles or permissions of a role are being updated by resolving the names on the right hand side of the assignment statements.

The other information held in a structure may also be updated within assignment statements. Additions (for example) may be made to the condition within a privilege, as:

```
Pr1 := Pr1 + condition_expression
```

The new condition expression for the privilege is formed by joining the previous expression and that in the assignment statement with the *and* conjunction.

4. CONCLUDING REMARKS

Implementation of Tower is in its early stages. We rejected any implementation on top of other access control mechanisms, such as access control lists, as being too inefficient and probably incapable of supporting the full expressive power of the language. Instead we have chosen to directly implement it. The chosen vehicle is the CORBA interceptor mechanism [3,9]. This allows the access control to be independent of the rest of the system while still being able to allow or deny access. The implementations in each ORB can communicate, allowing distributed access control. However, several of the issues related to implementation of RBAC management in a distributed environment still need to be solved. We will report on the implementation when it is completed.

5. REFERENCES

[1] R.Sandhu, E.J.Coyne, H.L.Feinstein, "Role based Access Control Models", IEEE Computer, Vol. 29, no. 2, Feb.1996, pp. 38-47.

[2] D.Ferraiolo, R.Kuhn, "Role based Access Controls", 15th NIST-NCSC National Computer Security Conference, Oct.1992, USA.

[3] Object Management Group (OMG) : Security Services in Common Object Request Broker Architecture, 1996.

[4] S.Jajodia, P.Smarati, V.Subrahmanian, "A Logical Language for Expressing Authorizations", IEEE Proceedings on Security and Information Privacy, 1997.

[5] R. Sandhu, E. Coyne, H. Feinstein & C. Youman, "Role-Based Access Control: A Multi-Dimensional View", 10th Annual Computer Security Applications Conference, 1994, IEEE CS Press, pp. 54-61.

[6] B. Hilchenbach, "Observations on the Real-World Implementation of Role-Based Access Control", National Information Systems Security Conference, 1997, pp. 341-52.

[7] V.Varadharajan, C.Crall, J.Pato, "Authorization for Enterprise wide Distributed Systems: Design and Application", IEEE Computer Security Applications Conference, ACSAC'98, 1998.

[8] M. Zurko, R. Simon & T. Sanfilippo, "A user-Centered, Modular Authorization Service Built on an RBAC Foundation", IEEE Symposium on Security and Privacy, 1999.

[9] Object Management Group (OMG), "CORBAservices: Common Object Services Specification", OMG Document 97-07-04.

[10] M. Hitchens & V. Varadharajan, "Issues in the Design of a Language for Role Based Access Control", ICICS'99, pp. 22-38.

[11] M. Hitchens & V. Varadharajan, "Tower: a Language for Role Based Access Control", *submitted for publication.*