Policy Storage for Role-Based Access Control Systems

András Belokosztolszki, David M. Eyers, Wei Wang, Ken Moody
University of Cambridge Computer Laboratory
JJ Thomson Avenue, Cambridge, United Kingdom
{firstname.lastname}@cl.cam.ac.uk

Abstract

Role-based access control has been a focal area for many security researchers over the last decade. There have been a large number of models, and many rich specification languages. However there has been little attention paid to the way in which access control policy is stored persistently. This paper investigates policy storage from the perspective of access control to the policy itself, and of its distributed administration.

1. Introduction

By introducing roles as an indirection between the mapping of users to privileges, role-based access control (RBAC) can simplify security administration tasks. Because users are mapped to roles and roles are then mapped to privileges, the addition or deletion of users is a separate process from adding and deleting the privileges connected to those roles. This simplifies management compared to approaches such as access control lists, access matrices, or DAC [7]. However, most RBAC models have numerous extensions beyond the basic role abstraction, sometimes leading to complex policy specification languages. For example, these languages might support separation of duty constraints, fast revocation and/or role hierarchies [1, 7, 8, 9, 13]. Although policy design has drawn significant research attention, often little thought has been given to how these policies might actually be *stored*. Many prototype implementations use a flat file structure, for example a XML file. Some store their policies via a directory service; however, the emphasis is still on the use of such policy, and not how it is created in the first place, nor subsequently maintained during system operation.

Clearly utilising a text file provides little fine-grained control over its access control. Generally, binary access will be employed, allowing anyone with access to the file complete policy update privileges. This is clearly inadequate, especially when one considers the requirements for any large-scale organisation. Our research is partly motivated by Electronic Health Record (EHR) access control within the UK National Health Service. In the NHS, access control on patients' records must be enforced within a large-scale distributed environment where local domains (e.g. hospitals) have varying degrees of autonomy. To be scalable, we must support the distribution of policy throughout these domains. We also have further requirements from two extremes of policy granularity; on one hand there may be NHS-wide policy standards required, and on the other, UK data protection laws may lead to the requirement that patients can elect to restrict access to their own records, in this way becoming administrators of certain policy components. Thus for such highly distributed administration, we need to apply access control over the access control policy representation itself – segmenting policy storage to give control to principals with varied *trustworthiness*, *competence*, and *privilege*.

Restricting the modification of access control policies can be done at two main levels. First, a principal's requests to modify policy are intercepted, and policy is used to determine whether such an operation is permitted or denied. Assuming sufficient privilege to make a policy modification, this new policy version can then be checked as to whether it satisfies certain constraints (see [4]). We consider both these approaches in this paper.

The frequency of policy change in large-scale systems may vary significantly. Some systems might modify policy as infrequently as once per year whereas others may make changes many times a day. In either case multiple versions of a policy may be active at any given time, and this poses problems.

Our goal is a scalable policy store which is *distributed* and *active*, by which we mean that notification of change to one policy component is distributed as soon as possible to the manager of each policy component that may depend on it. The policy store must support fine-grained *access control* over its modification. The policy management system also must provide *version control* and handle any version conflict situations.

This paper presents our experience designing and implementing a policy store for our particular model of role-based access control, which is introduced in Section 2. Note that most of our work is relevant to general RBAC models; we indicate which parts of our policy store are particularly specific to our RBAC architecture. In the second part of this section we describe work done in this area by other researchers and compare their research with ours. Section 3 examines the policy structure we want to store. This includes both the information that encodes the meaning of policy itself, and the meta-information needed to manage policy versioning, facilitate policy access control, and to ease policy administration.

In Section 4 we present implementation details relating to activity support, and discuss how our policy store manages version control and maintains policy consistency. Finally, Section 5 concludes the paper and provides an overview of our future research goals.

2. Related Work

This section provides a quick overview of the technologies used within our research. We begin by introducing the OASIS RBAC model.

2.1. OASIS RBAC

As mentioned in the introduction, Role-based access control introduces a role indirection between users and privileges. Many RBAC models (see [7, 11, 13]) extend this basic approach with constraints that require a user to hold a number of particular prerequisites, probably other roles or references to external predicates, in order to make further role activations. Extensions might also include support for delegation; enabling roles or principals to give temporary rights to other principals.

Without loss of generality we will look at one particular implementation of role-based access control. The Open Architecture for Secure Interworking Services (OASIS) [1], includes an RBAC model which supports most of the extensions commonly used at present.

In OASIS users enter a role (within a given OASIS *session*) by satisfying some *role activation rule*. These rules include a sequence of parameterised prerequisites, each prerequisite being a role, an environmental predicate or an appointment. Environmental predicates form a link to services external to the OASIS world, for example database systems or time services. Appointments are credentials like roles, but their lifetime is not session-bounded. Thus they can cater for concepts like delegation or privilege assignment as well as real world qualifications (e.g. a doctor's certification). Apart from having a lifetime that is longer that of a session appointment certificates (instances of appointments) can impose additional, policy independent restrictions. An important feature of OASIS is its fast revocation mechanism. Prerequisites can be marked as membership conditions – any membership condition becoming false (e.g. a role is revoked) will cause immediate revocation of the roles dependent on this prerequisite. An activation rule might look like:

$$\begin{aligned} &local\_user(h\_id?),\\ &employed\_medic(h\_id?)*,\\ &on\_duty(h\_id) \quad \vdash \quad doctor\_on\_duty(h\_id) \end{aligned}$$

In this rule there are three prerequisites: a role (*local_user*) with one parameter *(h_id?)*, an appointment *(employed_medic)* that also has a parameter *(h_id?)* and an environmental predicate *on_duty*, also with one parameter *(h_id)*. If a principal holds these prerequisites then according to this rule it may enter the target role *doctor_on_duty(h_id)*, where the *h_id* parameter is set by the rule. Note, that the appointment is also a membership condition, therefore the target role is revoked if the validity of the appointment certificate for *employed_medic* is revoked.

Privileges are assigned to roles by means of *authorisation rules*. Any such rule assigns a single privilege to a single role, but may also contain environmental predicate prerequisites. Note that these prerequisites' parameters can bind privilege parameters. An example authorisation rule is:

$$\begin{aligned} &doctor\_on\_duty(h\_id),\\ &\quad patient\_of(h\_id,x) \quad \vdash \quad accessEHR('general\ inf.',x). \end{aligned}$$

In this example a privilege *accessEHR* is assigned to the *doctor_on_duty(h_id)* role if the environmental condition *patient_of(h_id, x)* evaluates to true. In this rule *h_id* and *x* are variables and *'general inf.'* is a constant.

Full details of the OASIS model can be found in [1].

2.2. Related work

An important motivation for our research is the need for policy administration and access control to it. Most models use simple text files for policy specification or just completely ignore this important issue.

Sandhu introduces the concept of administrative domains in [12] about ARBAC. This work provides the basic means required to control access to a policy store. The granularity of access is too coarse for our needs, however, we have extended their domain concept into our notion of *contexts*. Contexts are discussed in section 4.4.

Many RBAC models use more sophisticated means to store a policy than simple text files. Ponder [6] reads its

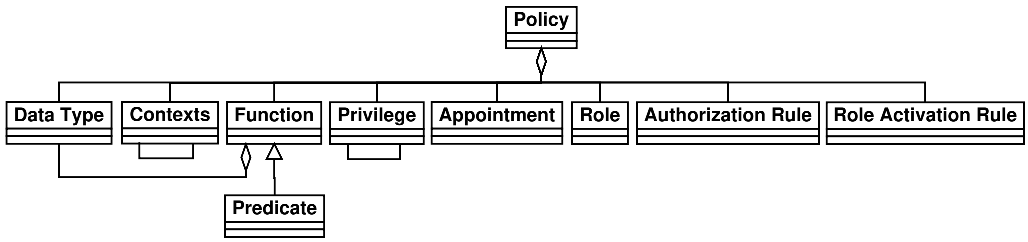

Figure 1. The major components of a policy.

policy from a directory accessed via LDAP. Whilst this allows fine-grained access control to a policy, the hierarchical structure of the directory is rigid. Our approach permits several, orthogonal policy classification schemes to operate in parallel, achieving functions such as information flow restrictions as well as assisting distributed administration.

Some models encapsulate policy components into certificates. An example for this is PERMIS [5] that uses X.509 certificates, which it stores in a directory service. This approach supports a less expressive access control to policy components. Also, due to the nature of certificates the removal of selected policy components is more difficult.

3. Policy Components

RBAC policy storage will clearly depend on the components which make up a policy representation. In this section we describe the OASIS RBAC components and talk about relationship between them. The main components are shown in figure 1. In many cases this is without too much loss of generality in terms of relevance to other RBAC approaches. The individual components are as follows:

Data types describe the types used in the policy for function return values, and prerequisite parameter types – the types for role, appointment, and environmental predicate parameters. From a storage perspective these data types are handled similarly to data types in many standard relational database management systems, e.g. PostgreSQL [10].

Note the specification of a data type must include some functions that enable management of its values. Such functions include tests for equality, serialisation and type conversion.

Contexts provide a naming and classification mechanism on which policy relating to policy modification can be based. Contexts are labels that can be attached to, and thus group together, sets of policy components including rules, roles, even role parameters and contexts themselves. Any policy component can have a number of contexts associated with it. At policy specification time, specification errors or policy modification privilege violations can be detected by using contexts to check information flow constraints between policy components. Contexts can also be used as a classification scheme for restricting access to the policy elements themselves. Such policy element classification is essential for distributed policy administration.

Although largely used for policy administration, clearly this context information needs to be included in our policy store, along with the permissible relationships between contexts. For more information on contexts see [3].

Functions and predicates are mainly used as prerequisite conditions in a policy. Predicates are an extension of functions as they can refer to entities outside the access control policy engine (for example they can access external databases or web services). Predicates and functions both have a return data type, although in the case of predicates the return type is fixed to the built-in `boolean` type. All the parameters of functions and predicates are also typed.

In some contexts, the optimisation of function usage can be greatly assisted by further meta-information, such as whether functions are invertible (and what the inverse function is if it exists), whether backtracking through this function can generate multiple results, and so forth.

Privileges specify the name and typed parameters of the privileges that can be the target of authorisation rules.

Appointments are a concept specific to OASIS RBAC, but from storage point of view they behave like roles that are the part of all RBAC models.

Roles are used either as rule prerequisites or as the targets of activation rules. As in case of privileges and appointments, information about the parameter types and context must be stored for each role.

Authorisation rules, as described in our section about OASIS, broker the role to privilege mapping. The structure of these rules is visualised in figure 2. In this figure we refer to `Terms` which are either constants or variables.

The *container objects* are present to represent instances of prerequisites or privileges. In addition to the reference of the prerequisite or privilege type, these instances contain rule specific information about the parameter binding of prerequisites to either variables within the scope of each rule, or constants. For example a role specification $local\_user(id\ integer)$ gives information only about the data types and names of its parameters, but a role container binds the parameters of this role to terms that hold a value

($local\_user(X)$, where X is a variable of a rule this role instance is part of).

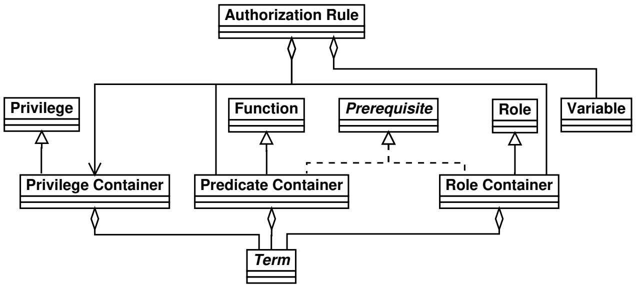

Figure 2. Authorisation rule structure

When storing an authorisation rule in a file either as text or an XML document the sequential nature of files imposes an ordering on the rules. This quality is lost when these rules are stored in a data base like system.

Many RBAC implementations including OASIS do not require ordering, but some RBAC use this information to prioritise among rules. To support both these models rules can be partially ordered, and this information must be stored with them.

Activation rules are similar to authorisation rules, and are depicted in figure 3. Here the containers store additional

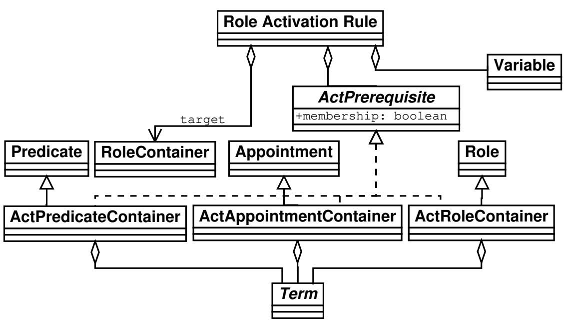

Figure 3. Role activation rule structure

information such as whether a prerequisite is a membership condition.

4. Policy Store

In this section we look at policy storage from an architectural perspective. We present a design which uses three levels of abstraction (see figure 4).

The lowest level, described in Section 4.1, provides advanced notification and triggering support in the underlying storage database. In our implementation we extended the PostgreSQL [10] RDBMS, integrating our `t5` predicate store. The API of this layer is targeted to the needs of our middle layer.

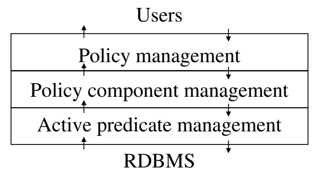

Figure 4. Our three layer policy store architecture.

As described in Section 4.2, our middle layer uses the lowest layer active predicate store to assist it in mapping policy components onto database tables and monitoring their modification. The API presented to the highest layer provides addition, deletion and modification operations over policy components whilst hiding all the low-level notifications caused by such operations.

The highest layer is responsible for grouping policy modifications into a larger transaction-like units, providing version support and ensuring consistency. We discuss this layer in Section 4.3.

OASIS makes extensive use of a publish/subscribe event middle-ware for managing revocation and for handling problems associated with unavailability of OASIS servers. By using self-administration, our particular policy store implementation inherits many of these desirable OASIS features.

4.1. The `t5` Active Predicate Store

We originally implemented the `t5` active predicate store to provide efficient support for environmental predicates that accessed databases in OASIS. `t5` predicates, just like relational database tables, are specified by a name and a type schema.

Treating relational database tables as predicates, `t5` adds advanced, automatically generated triggering based on a small amount of meta-information provided for each table.

Note that to support the RBAC policy component data types, we use `t5`'s extensible data type system to specify and check predicate parameters. A simple expression language describes the particular actions to perform based on changes to the predicates. These expressions are automatically translated into SQL triggers and into meta information that is used by `t5` triggers.

We store all the `t5` conditions in *template* tables (see figure 5) which relate to each predicate of interest. Storing conditions in this way has many advantages over explicitly setting up triggers for each. Firstly the semantics of trigger conditions are restricted, meaning we know about invertibility and other conditions of the available template evaluation functions. We can thus increase efficiency by filtering which templates need evaluation.

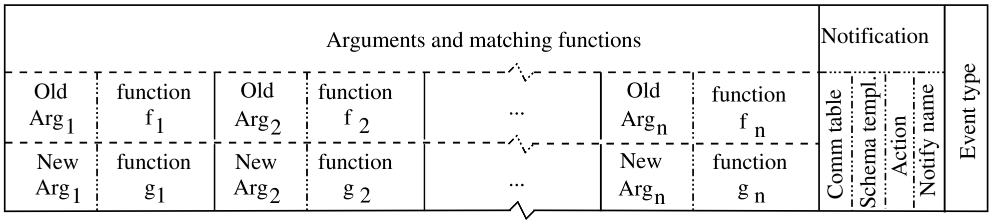

Figure 5. The structure of a template.

Whenever a predicate changes (e.g. tuple insertion, deletion or modification), it is matched against the relevant template(s). These templates include old and new value expressions which are matched via template functions to old and new values of the predicate. These functions (shown as f_i and g_i in Figure 5) are user-specified and can even be added at runtime. If a checked template matches a change then the action part of the template is executed. This can publish an event to a registered principal. `t5` supports both push and pull style messaging, and can optionally guarantee delivery. A *post action* is executed after such a template match. This post action can be used, for example, to specify that a template should be automatically deleted after a particular number of successful matches.

A major advantage of `t5` is that its features are accessible from any SQL console. Users merely need to know about a few extra table structures. For example, to set up a new `t5` predicate it is sufficient to add a tuple to the `t5_predicates` table. This will automatically set up all the relevant tables that store meta-information, templates, triggers, and so forth, if they do not already exist. From a user perspective, the predicate tables behave exactly the same as other tables in the DBMS. `t5` also comes with a set of helper functions which perform tasks such as adding wrappers to the data types, or adding access control restrictions.

Note that `t5` can also be accessed explicitly from OASIS via a set of environmental predicates, although to support the policy storage in this paper this extra interface is not necessary.

4.2. Policy Component Storage

We build on the predicate store presented above to map policy components into predicates that are stored persistently in the database, and react to changes immediately.

Due to space limitations we are only able to examine a few of the fourteen tables we use to store our policy in a relational database system, although this is sufficient to illustrate our approach.

Activation rules, as for any of the other policy components, may be associated with some number of contexts. This association is rendered in the standard manner for n-to-n relations, as in:

```
act_rules ( ar_id ID PRIMARY KEY, ...)
contexts ( c_id ID PRIMARY KEY, name text, ...)
actrule_contexts(ar_id ID REFERENCES act_rules,
                 c_id  ID  REFERENCES contexts)
```

These tables would normally be accessed indirectly via an API that concerns policy components described in Section 3. For example, the method `add_context(...)` modifies the `Contexts` table. These methods are responsible for maintaining consistency of the underlying tables.

Before invocation of these methods leads to any modifications, the requesting principal must be verified to have appropriate authorisation.

4.3. Policy Management Layer

Although the methods provided by the API of the middle layer in themselves perform simple steps, the semantics of the policy elements may lead to many method calls being required to make a single change. For example, to add a new rule one must specify the variables for this rule as well as the prerequisites, and bind the parameters of the prerequisites (the number of which may vary) to variables or constant values. It is not appropriate to provide a single middle layer method to perform all these operations, however not doing so might lead to an operation like adding a rule being non-atomic. Thus for policy evolution we must provide a means of grouping such a method call sequence into a transaction, and to identify each such consistent policy version. The necessary functions are provided by the policy storage middle layer.

4.4. Consistency

Policies are distributed over a number or OASIS servers, so, naturally, it is necessary to disseminate policy updates. To avoid inconsistencies due to use of different policy versions the lower layers of policy storage provide us with notification of policy modification. For disseminating these notifications we use the notification mechanisms that are already present in the OASIS middleware. Issues concerning notification failures and attack handling are described in [2].

The style of policy modification transaction at a given server is a check in/check out system rather than utilising other automatic conflict resolution methods common in databases. Once a policy is checked out at a particular policy store server, modifications can be performed on it. There are many ways to constrain such modifications:

(1.) Fine-grained access control based on the modification primitives. We do not have space to detail all the fine-grained modification methods of the middle layer here, however each is associated with a parameterised RBAC privilege. Whenever such methods are invoked, the OASIS engine will have checked that the principal requesting the change holds the relevant privilege.

(2.) As each policy component can be tagged as a member of some number of contexts, modification privileges can make use of context information as discussed in [3]. When policy components are grouped into units based on contexts, the number of privileges required for specifying access to the policy itself can be reduced significantly.

(3.) Once a policy modification has been completed, that is the new policy is ready to be checked in into the policy store, the policy may be checked against a set of high-level constraints, so called meta-policies [4]. Meta-policies can express constraints such as static separation of duties, and can insist on the existence of certain types, roles, and rules, among other tests. These consistency checks occur at the top level in our policy store architecture.

After a policy has been successfully checked into the policy store it is assigned a new version tag, and appropriate notification events are sent out regarding this policy change. To achieve fast policy dissemination we use a publish/subscribe middleware that is already integrated into OASIS. Information about the differences between policy versions may also be issued. This information is used by tools that monitor policy evolution. For example, it may be acceptable that a role issued based on a previous policy version be retained and accepted for access requests, but otherwise all roles issued by the obsolete policy version will need to be revoked.

5. Conclusion

This paper described our work on active RBAC policy storage. We described the three-level architecture used in our implementation. At each level we discussed the kinds of policy consistency being checked, and what policy meta-information was required. Overall we have designed and implemented a system to provide comprehensive support for RBAC access control, policy management and policy evolution.

6. Acknowledgements

András Belokosztolszki is supported by King's College Cambridge, the John Stanley Graduate Fund, and the United Kingdom Overseas Research Students (ORS) Awards Scheme. David Eyers is funded by the Cambridge Australia Trust and the ORS Awards Scheme. Wei Wang's research is supported by EPSRC.

References

[1] J. Bacon, K. Moody, and W. Yao. A model of OASIS role-based access control and its support for active security. *ACM Transactions on Information and System Security (TISSEC)*, 5(4):492–540, Nov. 2002.

[2] A. Belokosztolszki and D. Eyers. Shielding RBAC infrastructures from cyberterrorism. In *Research Directions in Data and Applications Security*, pages 3–14. Kluwer Academic Publishers, 2003.

[3] A. Belokosztolszki, D. M. Eyers, and K. Moody. Policy contexts: Controlling information flow in parameterised RBAC. In *Policy 2003: IEEE 4th International Workshop on Policies for Distributed Systems and Networks*, 2003.

[4] A. Belokosztolszki and K. Moody. Meta-policies for distributed role-based access control systems. In *Policy 2002: IEEE 3rd International Workshop on Policies for Distributed Systems and Networks*, pages 106–115, 2002.

[5] D. W. Chadwick and A. Otenko. The permis x.509 role based privilege management infrastructure. In *Seventh ACM Symposium on Access Control Models and Technologies*, pages 135–140. ACM Press, 2002.

[6] N. Damianou, N. Dulay, E. Lupu, and M. Sloman. The ponder policy specification language. In *Policies for Distributed Systems and Networks,International Workshop, POLICY 2001, Bristol, UK*, pages 18–38, 2001.

[7] D. F. Ferraiolo, J. F. Barkley, and D. R. Kuhn. A role-based access control model and reference implementation within a corporate internet. In *ACM Transactions on Information and System Security*, volume 2, pages 34–64, 1999.

[8] L. Giuri. Role-based access control: a natural approach. In *Proceedings of the first ACM workshop on Role-based access control*, pages II–33–37, 1995.

[9] C. Goh and A. Baldwin. Towards a more complete model of role. In *Proceedings of the third ACM workshop on Role-based access control*, pages 55–62, 1998.

[10] T. P. G. D. Group. *PostgreSQL 7.3 Programmer's Guide*. `http://www.postgresql.org`.

[11] M. Nyanchama and S. Osborn. Access rights administration in role-based security systems. In *IFIP Workshop on Database Security*, pages 37–56, 1994.

[12] R. Sandhu, V. Bhamidipati, E. Coyne, S. Ganta, and C. Youman. The ARBAC97 model for role-based administration of roles: preliminary description and outline. In *Proceedings of the second ACM workshop on Role-based access control*, pages 41–50, 1997.

[13] R. Sandhu, E. Coyne, H. L. Feinstein, and C. E. Youman. Role-based access control models. *IEEE Computer*, 29(2):38–47, 1996.