RBAC Administration in Distributed Systems

M.A.C. Dekker[*]
DIES, Twente University
The Netherlands

J. Crampton
ISG, Royal Holloway
University of London
United Kingdom

S. Etalle
DIES, Twente University and
SEC, Technical University of
Eindhoven
The Netherlands

ABSTRACT

Large and distributed access control systems are increasingly common, for example in health care. In such settings, access control policies may become very complex, thus complicating correct and efficient adminstration of the access control system. Despite being one of the most widely used access control standards, RBAC does not include an administration model for distributed systems. In this paper we fill this gap. We present a model for the administration of RBAC in a distributed system and propose an administration procedure supporting the principle that different systems protect different sets of objects. We demonstrate that our procedure fulfills the formal requirements deriving from safety and availability, and we show how it can be translated to a practical implementation. Finally, we show how our model can be extended with multiple decentralized administrative systems.

Categories and Subject Descriptors

D.4.6 [**Security and Protection**]: Access controls; K.6.5 [**Management of Computing and Information Systems**]: Security and Protection

General Terms

Security, Theory

Keywords

Access control, Distributed System, RBAC, Administration

1. INTRODUCTION

Large and distributed information systems employing access control to protect data are increasingly common. For example, most large hospitals run a variety of systems that process medical data, which (by law) must be protected from unauthorized access. Role-based access control (RBAC) [1, 15] is one of the most prominent access control standards, simplifying the specification of access control policies, by grouping users in a number of *roles*, which are ordered in a *role-hierarchy*. However, practice has pointed out that – in organizations – RBAC policies can become very large, involving hundreds of roles [7], and this makes administration of an RBAC-based system a difficult task. Particularly in large and distributed systems.

Consider for example a hospital with a distributed system composed of various subsystems that store and process confidential medical data. Both *safety*, and *availability* are key in this setting. Let us suppose that the hospital's security officer, to fulfill data-protection requirements, has deployed a set of different RBAC policies at different subsystems in the hospital. Over time some of these policies need to change. For example, a nurse may need to be assigned to a new role because of a changed hospital shift, or a database role may need to get access to additional tables, because some database application changed. Now who can make the policy changes? Which subsystems need to update their access control policies following policy changes? How can the update of the various subsystems take place efficiently? How can multiple administrative systems be used concurrently?

Although administration of RBAC has received much attention recently, and numerous researchers have proposed different ways of choosing administrative RBAC policies [2, 4, 5, 6, 7, 11, 14, 16, 17, 18, 19], there is no literature on the more practical issue of administration of a distributed RBAC system. The RBAC standard does not address this either. In this paper we present a model for the administration of a distributed RBAC system and we show how it can be translated to a practical implementation.

- We present a distributed system model with a central administrative system. A key component of our model is a *mapping* based on the fact that different subsystems protect different subsets of data, and that therefore only some policy changes are relevant to certain subsystems. We use this in Section 3 to define precise *safety and availability requirements* for the administration of an RBAC policy across the subsystems.
- We present an administration procedure, which is efficient in the sense that subsystems are only updated about *relevant* policy changes, and correct in the sense that it preserves the formal safety and availability requirements (Section 4).
- We translate the administration procedure to practi-

[*] Research performed at the Security group of TNO ICT. Currently consultant for KPMG CT, The Hague.

SACMAT'08, June 11–13, 2008, Estes Park, Colorado, USA.
Copyright 2008 ACM 978-1-60558-129-3/08/06 ...$5.00.

cal *pseudo-code* to demonstrate how it can be implemented (Section 5).

- We show how our model can be extended with *multiple* administrative subsystems (Section 6) and we sketch the additional steps that are required here. This addresses advanced settings with for example a human resources system for assigning users to roles, and a database management system (DBMS) for assigning database privileges to database roles.

2. PRELIMINARIES

While there are many different RBAC models, in this paper we restrict our attention to General Hierarchical RBAC [1], as it is the most common RBAC model of the standard. The standard assumes the existence of a set names for users, U (ranged over by u, u',...), for roles R (ranged over by $r, r', \dots$), a set of actions A and a set of objects O. Privileges in RBAC are permissions to perform actions on objects (being data or other resources). They form a set $P \subseteq A \times O$ (ranged over by p, p'), and we refer to these as *user privileges* (as opposed to the administrative privileges to be introduced below). A standard RBAC policy is a triple (UA, RH, PA), where $UA \subseteq U \times R$, $RH \subseteq R \times R$, and $PA \subseteq R \times P$. The set of policies is denoted Φ. For the sake of brevity we treat $\phi \in \Phi$ as a single directed graph (a single set of edges $UA \cup RH \cup PA$).

Contrary to the RBAC standard, we do not require that the RH relation is transitive, or that the graph of the RH relation is acyclic. We believe that transitivity complicates administration unnecessarily, in agreement with Li et al. [10]. Cycles in RBAC policies are sometimes considered to be redundant. On the other hand, there are no strong reasons for explicitly excluding such policies.

A standard RBAC reference monitor is a system that decides whether or not a user is allowed to perform a certain action. Basically, user $u \in U$ can perform an action $a \in A$ on an object $o \in O$ if and only if $u \rightarrow_\phi (a, o)$, where $\rightarrow_\phi$ denotes the transitive closure of ϕ. For a detailed description of the reference monitor (e.g. sessions, role-activation) we refer to the RBAC standard [1]. In the following we focus on *administrative reference monitors.*

The RBAC standard includes *administrative functions and controls* but it does not mention how to specify administrative policies about who can use these functions (which is addressed by existing literature [5, 7, 17]), nor how to practically implement administrative policy in a distributed system (which is not addressed in existing literature). We define the following administrative commands for changes to UA, RH or PA, ignoring (for the sake of clarity) changes to the sets U, R, and P, as we can assume these *name spaces* to be sufficiently large and fixed.

Definition 1 (Administrative Commands). *Let U, R, P, be sets of users, roles, and privileges. Then the set of* administrative commands *is defined to be*

$$\{\blacksquare_u(u', r'), \blacklozenge_u(u', r'), \blacksquare_u(r, r'), \blacklozenge_u(r, r'), \blacksquare_u(r, p), \blacklozenge_u(r, p)\}.$$

The commands $\blacksquare$ and $\blacklozenge$ change ϕ by adding and removing edges, respectively. For example, given a policy ϕ, the command $\blacksquare_u(r, r')$ corresponds to a user u that changes ϕ to $\phi \cup (r, r')$. The corresponding *administrative privileges* form a set P°. We give an example of our notation at the end of this section.

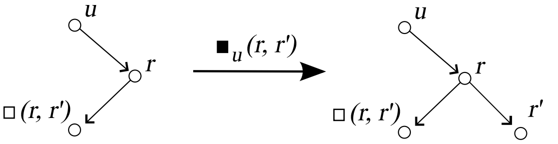

Figure 1: An administrative policy and an administrative action.

Definition 2 (Administrative Privileges). *The* administrative privileges *form a set*

$$P^\circ = \{\Box(u, r), \Diamond(u, r), \Box(r, r'), \Diamond(r, r'), \Box(r, p), \Diamond(r, p)\}.$$

The privileges $\Box(.,.)$ and $\Diamond(.,.)$ are read as *mayAssign* and *mayRevoke*, respectively. We assign administrative privileges to roles in the role-hierarchy, just like the ordinary privileges, which yields the following policy set.

Definition 3 (Administrative Policies). *An administrative policy ϕ is a tuple*

$$(UA,\ RH,\ PA \cup PA^\circ),$$

where $PA^\circ \subseteq R \times P^\circ$ are assignments to administrative privileges.

The set of administrative policies is denoted by Φ°, which is a superset of the standard RBAC policies. There are several lines of research concerned with defining 'suitable' subsets of Φ° [5, 7, 17], but for the sake of generality, we do not make choices in this regard.

Example 1 (Administrative policy and action). *As a simple example of administrative policies, and administrative actions, consider a policy containing the edges (u, r) and $(r, \Box(r, r'))$, as depicted in Figure 1. This policy allows user u to add an edge from role r to role r', i.e. it allows the command $\blacksquare_u(r, r')$.*

3. DISTRIBUTED SYSTEM MODEL

In this section we present the basic distributed system model of this paper. The model consists of a central administrative subsystem and a number of non-administrative subsystems. In section 6 we show how it can be generalized to a system with multiple administrative subsystems.

Consider a heterogeneous distributed system composed of databases, file systems etc, like one may find in an organization such as a hospital. In such a system it is inconvenient to use a central reference monitor to decide all user access requests, as each request would involve contacting the central reference monitor creating a bottleneck and a single point of failure. On the other hand, when each subsystem has its own reference monitor, and a separate policy, then one needs to manage those policies consistently. For example, if a user is assigned to the role of employee, all subsystems in the organization should allow the user to use the privileges of the employee role and vice versa when a user is revoked from a role. Inconsistencies in the definition of roles across the distributed system cause confusion and may affect safety and availability of data across the system.

So one could argue that a procedure is needed that maintains exact copies of a single system-wide policy at the different subsystems, and that updates all the subsystems after each policy change. At the same time, it is unnecessary to

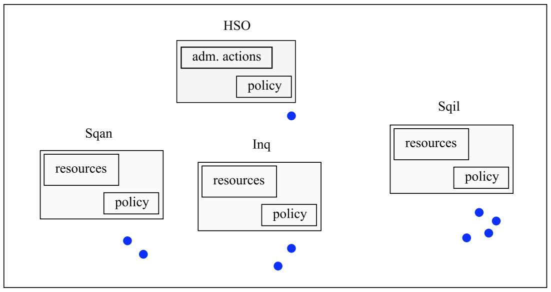

Figure 2: A hospital's distributed system.

send updates to, say, a printer about changed database table privileges, particularly given the fact that in practice RBAC policies can be large and policy changes frequent [7]. Additionally, for example in health care, policy definitions may even be *sensitive*. In this section we define a distributed system model, and basic safety and availability requirements, allowing us to derive a more efficient administration procedure (in Section 4).

The privilege mapping *pm* is a key component of our model, allowing us to capitalize on the fact that different subsystems offer access to different (largely disjoint) subsets of the resources, and avoid excessive updates about irrelevant policy changes. Formally it is defined as follows.

Definition 4 (Privilege Mapping). *The* privilege mapping*, denoted pm, is a mapping* $pm : S \to \mathcal{P}(P)$*. We say that subsystem s* protects *object o, if* $(a, o) \in pm(s)$.

The privileges in $pm(s)$ are referred to as the *relevant user privileges* for subsystem s. We do not require that $pm(s)$ and $pm(s')$ are disjoint for different subsystems s and s' (see also remark 1). The privilege mapping can be used as a tool for the security officer to evaluate and implement policy changes. Let us give a practical example.

Example 2 (A hospital's distributed system). *A hospital has a network consisting of a database named Sqil, a medical system Sqan, a printer Inq, and an administrative system HSO for administrative tasks, such as policy changes. The system is depicted in Figure 2, where the blue dots denote ordinary users of the system.*

The hospital's security officer enforces a number of RBAC policies across the different subsystems that protect resources such as an electronic health record table of Sqil denoted ehrtable*, and a scan job of Sqan called* job*. The hospital's security officer has defined the following privilege mapping:*

$$pm(Sqil) = \{(ehrtable, view), (ehrtable, insert)\}$$
$$pm(Sqan) = \{(job, halt), (job, start)\}$$
$$pm(Inq) = \{(black, print), (color, print)\}$$

Remark 1. *One could argue that implementing the privilege mapping introduces overhead, but we believe that in many practical situations the object mapping* follows *largely from the names of the objects. Like in the next example, ehrtable1, ehrtable2, and so on, would all map to the database system Sqil, while job1, job2, and so on, would all map to the medical system Sqan.*

The privilege mapping could map some of the objects to multiple subsystems. For instance when the same resource is present at multiple subsystems. For example, an emergency procedure that is present on all subsystems, or the same authorization table that is present one a cluster of databases.

There may exist practical settings where it is difficult to keep track of which objects or resources reside on which subsystems, for instance because users can move them freely from one subsystem to another. In such settings it may be more convenient, but in principle less precise, to map the resources to all the subsystems they may be moved to.

Our model of a distributed system comprises a set of systems (each with a reference monitor), denoted S and ranged over by $s_0, s_1, \ldots$, and a single administrative system, denoted s_a, from which (administrative) users can make policy changes. The policy of the administrative system is denoted ϕ, while the policy of subsystem s is denoted $\psi(s)$, where $\psi : S \to \Phi$ defines the distribution of policies across the subsystems.

Definition 5 (Distributed System). *A* distributed system *is a tuple*

$$(S,\ pm,\ \phi,\ \psi\ ),$$

where S *is a set of systems,* $pm : S \to \mathcal{P}(P)$ *is the distributed system's privilege mapping,* $\phi \in \Phi^{\circ}$ *is an administrative policy, and* $\psi : S \to \Phi$ *is a function that maps each subsystem to a non-administrative policy.*

Using the privilege mapping we can define two formal requirements for the distribution of policies across the subsystems.

The first is the basic requirement that each subsystem's policy is included in the policy of the administrative system. This is in the interest of safety and administration, so that the security officer, or some other user of *HSO*, in Example 2, can correctly asess the impact of policy changes. The second captures that the relevant parts of the administrative system's policy should be present at the subsystems. This is in the interest of availability of resources, so that, as in Example 2, if the security officer has given to users the privilege to access a resources, then they are also granted access by the subsystem.

Definition 6 (Soundness and Completeness). *Given a distributed system* $(S,\ pm,\ \phi,\ \psi)$*, we say that* ψ *is* sound *with respect to the central policy* ϕ*, if*

$$\bigcup_{s \in S} \psi(s) \subseteq \phi.$$

On the other, we say that the distribution ψ *is* complete *with respect to the central policy* ϕ *if and only if for any subsystem* $s \in S$*, and any privilege* $p \in pm(s)$

$$u \to_{\phi} p \text{ implies } u \to_{\psi(s)} p.$$

Soundness is important from the viewpoint of safety: it ensures that subsystems grant access only when it is allowed by the administrative policy ϕ. It may seem that a weaker requirement suffices: *For any* $s \in S$*, and any privilege* $p \in pm(s)$*,* $u \to_{\psi(s)} p$ *implies* $u \to_{\phi} p$. However, such a weak requirement would complicate the implementation of policy changes, as will become clear in the next section.

Completeness, on the other hand, is important from the viewpoint of availability: it ensures that the subsystem protecting object o grants access to the object o, whenever it is

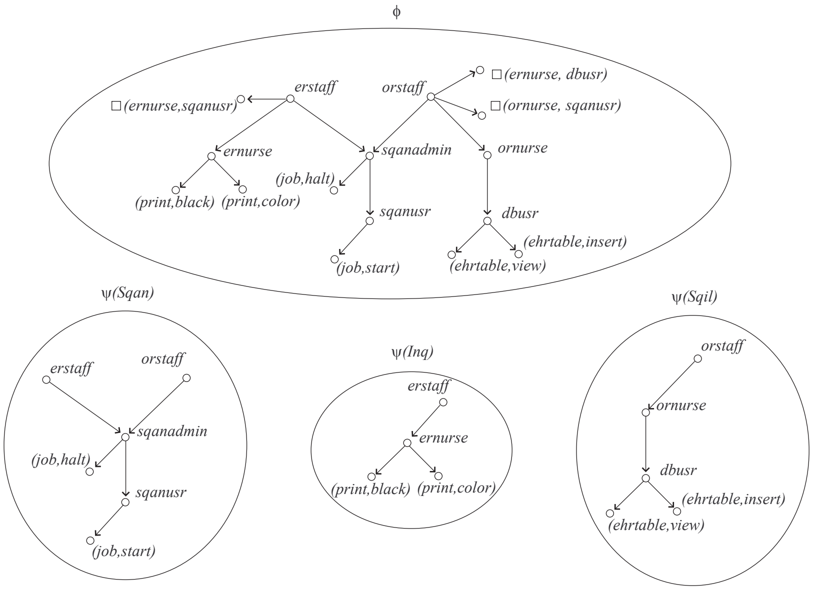

Figure 3: Sound and complete policy distribution.

allowed by the administrative policy. Before defining an administration procedure that implements policy changes, preserving soundness and completeness (in the next section) let us introduce the running example of our paper, and demonstrate the practical usefulness of the definitions above.

Example 3 (RBAC policies in the hospital). *Let us build on the Example 2. The hospital's security officer has defined a number of roles for staff and nurses of the operation room (OR) ($orstaff$, $ornurse$), and staff and nurses of the emergency room (ER) ($erstaff$, $ernurse$). For the sake of brevity we do not elaborate on which users are assigned to these roles.*

The hospital's security officer has prepared the distributed system as shown in Figure 3. The policy ϕ is the (administrative) policy of system HSO, and the policies $\psi(Sqil)$, $\psi(Sqan)$, and $\psi(Inq)$ are the (non-administrative) policies of the database Sqil, the medical system Sqan, and the printer Inq.

The distribution ψ is sound with respect to to the central policy ϕ, because each subsystem policy is a subset of the administrative policy ϕ. It is also clear that the distribution ψ is complete. So although the subsystems in the hospital do not enforce the hospital's policy in its entirety, this does not affect availability nor safety of the resources. Indeed, most parts of the hospital policy are in practice *irrelevant for the printer Inq, as Inq does not protect database tables of Sqil, nor the resources of Sqan. Also the administrative privileges $\Box(.,.)$, and $\Diamond(.,.)$ are irrelevant for the subsystems, because they can not be used for administrative actions anyway. These subsystems implement standard RBAC policies from Φ.*

Soundness and completeness are, so to speak, the minimal requirements that must be fulfilled. The *largest* distribution ψ, that is both complete and sound, is the distribution where $\psi(s) = \phi$ for all $s \in S$, where all subsystems have the same policy. We can also define the *smallest* policy distribution that satisfies soundness and completeness.

Definition 7 (Upper and Lower Closure). *The* upper closure *of a vertex v in ϕ, denoted, $(\uparrow_\phi v)$, is $\{(v', v'') \in \phi \mid v'' \rightarrow_\phi v\}$, and the* lower closure *of a vertex v in ϕ, denoted, $(\downarrow_\phi v)$, is $\{(v', v'') \in \phi \mid v \rightarrow_\phi v'\}$.*

The smallest distribution ψ that is sound and complete is such that for every subsystem $s \in S$, the following holds,

$$\psi(s) = \bigcup_{p \in pm(s)} (\uparrow_\phi p).$$

We call this the *lean* distribution. The *lean* policy distribution has the advantage that components of the distributed system have the parts of the policy that are strictly necessary to decide about allowing or denying user actions.

4. ADMINISTRATION PROCEDURE

Having specified the formal requirements in Definition 6 for the distribution of RBAC policies across a distributed system, we take a more practical approach in this section, by defining an administration procedure for the administrative reference monitor s_a that *preserves* these requirements.

We model the system s_a by defining a command queue, containing administrative commands ($\blacksquare$, or $\blacklozenge$) for policy changes, and *message commands*. The message commands are needed to model the propagation of policy changes across the subsystems. They form a set $\{\oplus_s(\delta), \ominus_s(\delta)\}$, where $s \in S$ denotes the recipient subsystem and $\delta \in \Phi$ is a policy, and $\oplus$ denotes addition and $\ominus$ denotes removal. Let us first sketch the procedure by giving an example: a user u of the administrative system s_a places the administrative command $\blacksquare_u(r, r')$ in the queue. The administrative subsystem processes it by (1) checking that ϕ allows u to make this policy change, (2) changing its policy ϕ to $\phi \cup (r, r')$, (3) replacing the administrative command with message commands $\oplus_s\big((r, r') \cup (\uparrow_\phi r)\big)$ for each subsystem $s \in S$ that has relevant privileges in the lower closure of r' in ϕ, and (4) processing the message commands. In the sequel we will show that sending $\big((r, r') \cup (\uparrow_\phi r)\big)$ suffices to preserve completeness. Moreover we will show that the procedure preserves soundness. Let CQ denote the set of all command queues. We define the administration procedure by a formal transition function.

Definition 8 (Distributed Administration). *Given a distributed system* $(S,\ pm,\ \phi,\ \psi)$, *let* $cq \in CQ$ *be a command queue, and* N *be the number of systems in* S. *The transition function* $\Rightarrow\colon\ CQ \times \Phi^{\circ} \times (\Phi)^N \ \rightarrow\ CQ \times \Phi^{\circ} \times (\Phi)^N$, *is defined as follows.*

$\langle cq,\ \phi,\ \psi\rangle \ \Rightarrow\ \langle cq',\ \phi',\ \psi'\rangle$ *holds when:*

if $cq = [\blacksquare_u(v, v') : cq'']$ *and* $u \rightarrow_{\phi} \Box(v, v')$, *then*
$cq' = [\oplus_{s_1}\big((v, v') \cup (\uparrow_{\phi} v)\big) : \cdots : \oplus_{s_k}\big((v, v') \cup (\uparrow_{\phi} v)\big) : cq'']$,
where $\{s_1, \ldots, s_k\}$ *are all the subsystems with relevant privileges in the lower closure of* v', *that is* $\{s_1, \ldots, s_k\} = \{s \in S \mid \textit{if } \exists p \in pm(s).v' \rightarrow_{\phi} p)\}$.
$\phi' = \phi \cup (v, v')$, *and* $\psi' = \psi$.

if $cq = [\blacklozenge_u(v, v') : cq'']$ *and* $u \rightarrow_{\phi} \Diamond(v, v')$, *then*
$cq' = [\ominus_{s_1}\big((v, v')\big) : \cdots : \ominus_{s_k}\big((v, v')\big) : cq'']$,
where $\{s_1, \ldots, s_k\}$ *are all the subsystems.*
$\phi' = \phi \setminus (v, v')$, *and* $\psi' = \psi$.

if $cq = [\oplus_s(\delta) : cq'']$, *then* $cq' = cq''$,
$\phi' = \phi$, $\psi'(s) = \psi(s) \cup \delta$ *and* $\psi'(s') = \psi(s')$ *for* $s' \neq s$.

if $cq = [\ominus_s(\delta) : cq'']$, *then* $cq' = cq''$,
$\phi' = \phi$ *and* $\psi'(s) = \psi(s) \setminus \delta$ *and* $\psi'(s') = \psi(s')$ *for* $s' \neq s$.

Notice that the messages are the smallest when the command is a user assignment, since the upper closure of a user is always empty. This is also the most frequently used administrative command [7]. Message commands following an assignment (■) only involve those subsystems that are 'affected' by it. Revocations (⧫) on the other hand are broadcast to all subsystems to ensure soundness of ψ.

One could make this procedure even more efficient by keeping a *history* of sent policy definitions per subsystem, to avoid sending revocations of definitions to subsystems that never received them (or to avoid sending the same policy definitions twice). We do not go into details about this, for the sake of brevity. Although it is common in literature on distributed systems to use an *expiration mechanism* to reduce the number of revocations [13], we refrain from going into details about time or expiration here, because we believe they are out of the scope of the RBAC standard. We would like to mention however that, because soundness and completeness are preserved when edges expire, it seems straightforward to add expiration to our model. The procedure preserves soundness and completeness, but it does not preserve leanness, for example. To preserve leanness subsystems could remove irrelevant parts of the subsystem's policy, independently of the administrative reference monitor. System s can check for each edge (v, v') in $\psi(s)$ whether or not $v' \rightarrow_{\psi(s)} p$ for a relevant privilege $p \in pm(s)$.

Let us return to our running example to demonstrate a practical instance of the administration procedure.

Example 4 (Administration in the Hospital). *Suppose Bob, a member of* orstaff, *wants to grant all members of* ornurse *the right to use the medical system Sqan, say for a new type of operation. To do so, Bob puts an administrative command in the queue of the administrative system HSO. He is allowed to do so, by the administrative policy* ϕ *in Figure 3.*

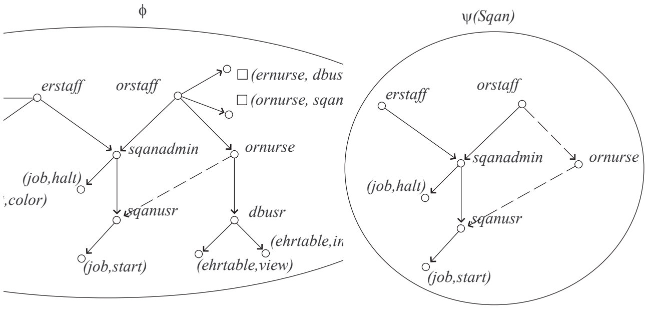

Figure 4: Update for subsystem ***Sqan*****.**

The administrative system HSO now takes the following steps: The command in the queue is

$$\blacksquare_{Bob}(\mathit{ornurse}, \mathit{sqanusr}).$$

After executing this command, the new policy ϕ' *contains the new edge* $(\mathit{ornurse}, \mathit{sqanusr})$ *and the command on the queue is replaced by the message command*

$$\oplus_{Sqan}\big((\mathit{ornurse}, \mathit{sqanusr}) \cup (\uparrow_{\phi} \mathit{ornurse})\big).$$

The message command is executed, updating also the policy of Sqan. The new policy for Sqan includes the upper closure of ornurse, i.e. the new edge $(\mathit{ornurse}, \mathit{sqanusr})$, *as well as the 'members' of* ornurse. *So Bob's administrative command changes the policies* ϕ *and* $\psi(Sqan)$, *but not the policies of Inq or Sqil. The policy changes corresponding to Bob's action are depicted in Figure 4 by dashed edges.*

It is important noticing that the administration procedure preserves soundness and completeness. It does so without sending irrelevant parts of ϕ to subsystems. We denote a sequential execution of administrative commands (a *run*) by $\Rightarrow^*$ and an empty queue by ε.

Theorem 1. *Let* $(S,\ pm,\ \phi,\ \psi\ )$ *be a distributed system. For any command queue* $cq \in CQ$ *that contains only administrative commands (of the form* $\blacksquare_{\cdot}(.,.)$, *or* $\blacklozenge_{\cdot}(.,.)$*), the run to an empty queue*

$$\langle cq,\ \phi,\ \psi\ \rangle \Rightarrow^* \langle \varepsilon,\ \phi',\ \psi'\ \rangle,$$

yields a policy ϕ' *and a distribution* ψ' *for which the following statements hold:*

1. *If* ψ *is sound with respect to* ϕ, *then also* ψ' *is sound with respect to* ϕ'.
2. *If* ψ *is complete with respect to* ϕ, *then also* ψ' *is complete with respect to* ϕ'.

Proof. We have to show that an arbitrary queue of administrative commands preserves soundness and completeness. We prove the result by induction on the number of commands in the queue. Let us sketch the proof briefly. We assume that the distribution ψ is initially sound and complete with respect to ϕ.

The base case (the empty queue) is trivial. The induction hypothesis is that soundness and completeness are preserved by queues with n commands, and we show that this holds for queues with $n + 1$ commands. Consider a queue containing $n + 1$ commands. We enumerate the different possibilities for the first command in the queue.

- If the first command is of the form $\blacksquare_u(v, v')$, and it is replaced by the message commands $\oplus_{s_1}\big((v, v') \cup (\uparrow_\phi v)\big):\ldots:\oplus_{s_k}\big((v, v') \cup (\uparrow_\phi\ \ v)\big)$ on the queue, where $\{s_1, \ldots, s_k\} = \{s \in S \mid \text{if } \exists p \in pm(s).v' \rightarrow_\phi p)\}$. The administrative policy is changed to $\phi'' = \phi \cup (v, v')$ (cf. the first item in Definition 8), and after processing the message commands, the distribution changes to ψ''.

  Soundness follows since the difference between $\psi(s)$ and $\psi'(s)$ is at most $(v, v') \cup (\uparrow_\phi v)$, which is a subset of ϕ'. The remaining queue is shorter and preserves soundness by induction hypothesis.

  Completeness follows trivially for subsystems outside $\{s_1, \ldots, s_k\}$, as the policy change does not affect the upper closure of their privileges. The other subsystems are complete before the change, so they already have the upper closure up till v_2. The update message adds to this also the rest of the upper closure (v_1, v_2), the new edge, and $(\uparrow_\phi v_1)$.

  The rest of the queue preserves completeness by induction hypothesis.

- If the first command is of the form $\blacklozenge_u(v, v')$, and it is replaced by the message commands that remove the edge (v, v') from all systems in S.

  Soundness is straightforward, since the distribution was sound before processing this command, and the edge (v, v') is removed from all the policies of the subsystems.

  For *completeness* observe that ψ is initially complete with respect to ϕ, and that by removing an edge the upper closure only shrinks.

  By the induction hypothesis, both soundness and completeness are also preserved by the commands on the remaining (shorter) queue.

This completes the proof. $\square$

5. IMPLEMENTATION

The formal procedure of Definition 8 can be translated into an actual implementation. Here we report procedures, by using pseudo code, both for the administrative reference monitor and for the non-administrative reference monitors of the subsystems.

Let us introduce the syntax of the code. Vertices `v1, v2,...` (users, roles, and privileges) are assumed to be unique strings, and edges are pairs of such strings `(v1, v2)`. Policies are represented as *lists* of edges. Below the expression `a in b` checks whether `a` is in the list `b` or not. The functions `add`, `remove`, and `join` denote adding, and removing elements from lists, and joining two lists, respectively, and `[]` denotes the empty list. We use sub-procedures for finding the upper and lower closure (see Definition 7) of a vertex in a policy, for later use. The function `lower(a, b)` returns a list of elements from the policy `a` which are in the lower closure of `b`, i.e. $(\downarrow_a b)$. Both `lower` and `upper`, its converse, are implemented by a basic depth-first search.

```
procedure dfs(policy, v1, visited)
  visited := add(visited, v1).
  list l1 := [].
  for (v1,v2) in policy and v2 not in visited
    list l2 := dfs(policy, v2, visited).
    l1 := join(l1, l2).
  return with l1.

procedure lower(policy, v)
  return with dfs(policy, v, []).
```

The depth-first search `dfs` picks an edge from `v1` to another vertex `v2`, and continuous the search. To avoid cycles we mark which vertices where visited already. The same algorithm `dfs` can be used for the upper closure `upper` by inverting the direction of the edges in `policy`. Administrative commands are denoted as `(user, action, (v1, v2))`, where the second parameter is either ■ or ♦, and `(v1, v2)` is the edge being assigned or revoked. Queues are lists of administrative commands, and `shift` returns and removes the first element of the queue. The main procedure for the administrative system is as follows.

```
procedure admin (policy, queue)
  if queue= []
    return with policy.
  endif
  shift(queue) := (user, action, (v1, v2)).
  list lowu := lower(policy, user).
  if action = ■ and □(v1,v2) in second(lowu)
    list uppv1 := upper(policy, v1).
    list lowv2 := lower(policy, v2).
    list dest := [].
    for priv in second(lowv2)
      for s in systems
        if pm(s, priv) and s not in dest
          dest := add(dest, s).
        endif
    for s in dest
      send(s, ⊕, add(uppv1, (v1, v2))).
    return with admin(add(policy, (v1,v2)), queue).
  endif
  if action = ♦ and ◇(v1,v2) in second(lowu)
    for s in systems
      send(s, ⊖, [(v1,v2)])
    return with admin(remove(policy, (v1,v2)),queue).
  endif
    return with admin(policy, queue).
```

Let us explain the procedure in detail. In case the action is ■ it is checked whether or not the user is allowed to perform that command. This is only true when the corresponding privilege □(v1,v2) is in the lower closure of `user`. The function `second`, used here, takes a list of pairs, and returns a list of the second element of every pair. The next step takes care of sending the proper update messages. The list of subsystems is denoted `systems`, and the privilege mapping is a function `pm` that takes a privilege and a system name as input and returns true if the privilege is a relevant privilege for the system. The lower closure of `v2` is used to select which list of subsystems `dest` will receive a message (denoted by `send`). The upper closure of `v1`, on the other hand, constitutes the contents of the update message (cf. Definition 8). The steps for ♦ can be explained in the same way. The procedure recurs through the queue, untill it returns the new administrative policy, or if no command was allowed the same administrative policy.

The procedure for the non-administrative system is more simple. There are two types of commands: A message command by an administrative system, denoted by `receive`, and a basic user command by a user who wants to perform an action on an object, denoted by `do`.

```
procedure subsystem(policy, queue)
  if queue= []
    return with policy.
  endif
  shift(queue) := cmd.
  if cmd = receive(⊕, delta)
    return with subsystem(add(policy, delta),queue).
  endif
  if cmd = receive(⊖, delta)
    return with subsystem(remove(policy, delta), queue).
  endif
  if cmd = (user, action, object)
    list lowu := lower(policy,user).
    if (action,object) in second(lowu)
      do(action, object).
    endif
  endif
  return with subsystem(policy,queue).
```

Note that in this procedure the lower closure of `user` in `policy` is calculated at every user access request. This may be time-consuming (each search involves $O(E)$ steps, where E is the number of edges in $\psi(s)$). One could instead calculate the full transitive closure for the policy once, and update it only when update messages arrive.

6. DECENTRALIZED ADMINISTRATION

In the previous sections we have modeled systems with a single administrative reference monitor. In this section we show how our model can be extended to deal with systems with multiple administrative reference monitors.

In practice, administrative actions (e.g. assigning a user to a role) are relatively rare; for instance, they are much less frequent than ordinary user actions (e.g. accessing a database table). This suggests that for many practical distributed systems a single administrative reference monitor should be sufficient. Still, there may be scenarios where multiple administrative monitors are needed. For example when two different organizations share a common infrastructure, and at the same time prefer to use their own separate administrative systems. In this setting, one could use identical administrative subsystems augmented with standard mutual exclusion techniques to coordinate policy changes. More challenging are the settings in which the administrative reference monitors are not identical (i.e. with different administrative policies), for instance because they are not equally trustworthy.

In this section we briefly describe the additional steps needed to extend our distributed model to these settings, and we define an additional requirement for the distribution of policies across the administrative systems. The extension, although not entirely straightforward, makes use of the same formal structure of the previous section.

Let us assume that – in addition to the set of ordinary subsystems S – there exist a set of administrative reference monitors S° (ranged over by $s_a, s_b, \dots$), and an administrative privilege mapping

$$pm^\circ : S^\circ \to \mathcal{P}(P^\circ)$$

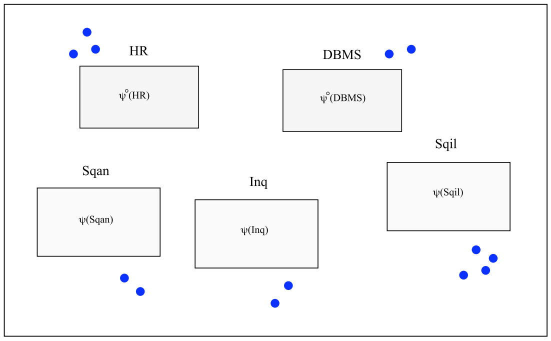

Figure 5: Decentralized administration in a hospital.

As before, we say that $p \in P^\circ$ is a *relevant administrative privilege* for system s, if and only if $p \in pm^\circ(s)$. The mapping pm° corresponds to the intuitive idea that certain administrative reference monitors can only be used for certain administrative actions. A distributed system is defined as a tuple

$$(S^\circ,\ S,\ pm^\circ,\ pm,\ \psi^\circ,\ \psi),$$

where $\psi^\circ : S^\circ \to \Phi^\circ$ is the distribution function of the administrative policies across the administrative reference monitors. Let us see an example (refer to Figure 5): the system *HR* is used at the human resources department for changing user-role assignments, while the system *DBMS* is used at the hospital's data center for changing database privileges. Here, there are multiple distinct administrative subsystems, and multiple distinct non-administrative subsystems, and there is no central administrative system.

Let us define ϕ by

$$\phi = \bigcup_{s \in S^\circ} \psi^\circ(s) \cup \bigcup_{s \in S} \psi(s).$$

The policy ϕ is here no longer the policy of a central administrative system (as in the previous sections), but only an abstract notion of the full system-wide policy. Like before, the subsystems each hold parts of ϕ.

Let us now define requirements for ψ and ψ° concerning safety and availability of objects, similar to the ones presented in Section 3.

The safety requirement (soundness) remains the same, but the availability requirement (completeness) becomes more complex. The policy of an administrative reference monitor should be (1) complete for its relevant administrative privileges, and – in addition – (2) it should contain the parts of ϕ needed to produce the message commands described in Section 5. We call the first *standard completeness* and the second *administrative completeness.* Let us show an example of standard completeness and administrative completeness.

EXAMPLE 5. *Consider an administrative system $s_b \in S^\circ$, with $\Box(v, v') \in pm^\circ(s_b)$, and that $(r, \Box(v, v')) \in \phi$.*

Standard completeness with respect to to the privilege $\Box(v, v')$ requires that ψ contains $(\uparrow_\phi r)$.

Administrative completeness additionally requires that ψ

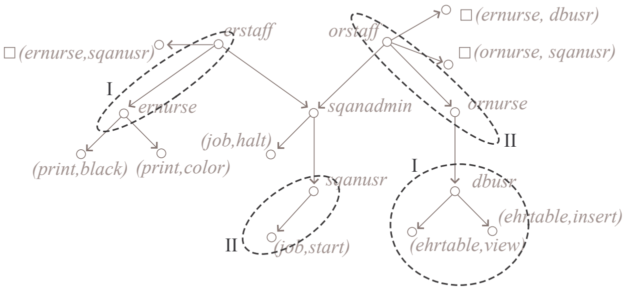

Figure 6: Policy support for different administrative privileges.

contains $\left((\uparrow_\phi\ v) \cup (\downarrow_\phi\ v')\right)$*, i.e. the parts of* ϕ *needed to perform the message commands described in Section 4.*

We call $\left((\uparrow_\phi v) \cup (\downarrow_\phi v')\right)$, in the example, the *policy support* of the administrative privilege $\Box(v, v')$. Basically, the policy support of an administrative privilege ensures that the administrative subsystem can perform administrative operations, *and* propagate the relevant additional parts of ϕ to subsystems. Let us show an example of policy support.

Example 6. *Figure 6 shows the policy support of two administrative privileges.*

The edges in the areas marked with I form the policy support of the privilege $\Box(ernurse, dbusr)$*, and those marked with II the policy support of the privilege* $\Box(ornurse, sqanusr)$*.*

It is now clear how a 'correct' and 'efficient' administration procedure can be defined in this model. It depends on the administrative privilege mapping pm° as follows: an administrative subsystems s_a must send an update to an administrative subsystem s_b every time the policy support for relevant administrative privileges of s_b changes.

7. RELATED WORK

The administration of RBAC policies is an issue that attracts considerable attention from the research community. In particular, there is a large body of literature on how to choose administrative policies (informally, about which roles should get what authority to change the RBAC policy) [2, 4, 5, 6, 7, 11, 14, 16, 18, 19]. The considerations that motivate the choices in these proposals are diverse. Crampton and Loizou, for example motivate their choice by considering responsibility in a organization hierarchy [5], whereas Li and Mao consider for example flexibility, and psychological acceptability [11]. None of these proposals address how to distribute RBAC policies across a distributed system, in a *correct* and *efficient* way (which is the scope of this paper). We now consider some of these proposals in more detail.

Wang and Osborn [17] introduce administrative domains for *role graphs*, a class of RBAC policies with a single lowest and single highest role, called *minrole* and *maxrole* respectively. Each administrative domain is defined by one role, and contains all the roles below it, except *minrole*. Administrative domains may not overlap, unless one domain includes the other completely. Wang and Osborn justify this restriction by arguing that it should not be allowed for different domain administrators to make changes to the same roles. On the other hand they also stress that this is a disadvantage of their model, arguing that in practice one would like to have overlapping domains, for example when one resource is shared by different departments (see Figure 3). Implementation of RBAC in distributed systems is not at the basis of this choice. For example, the administrative policy depicted in Figure 3 is not admitted by the administrative domains model of Wang and Osborn. In their model, administrative privileges about the role *sqanusr* can only be assigned to a *domain administrator*, which also has administrative privileges about *ehrstaff* and *orstaff*. Although we agree that there may be practical settings where administrative domains may be useful, we do not adopt such restrictions here.

Closely related is the work by Crampton and Loizou [5], who define the concept of *administrative scope*. Basically a role r is in the scope of a role r' if there is no role above r that is not comparable to r'. They show how administrative scope can be used to constrain delegations to evolve in a natural progression in the role hierarchy. Administrative scopes can be used as a basis for a policy distribution, but this does not yield the sound and complete administration procedures defined in our model.

Similarly, in the ERBAC model, proposed by Kern et al. *scopes* are used to define over which RBAC objects and administrator has authority [9]. The ERBAC model focuses on administrating RBAC policies in a commercial enterprise security software. Although ERBAC has been verified against a business case involving multiple remote company sites, the main goal of the administrative component of ERBAC is to allow for delegation of administrative authority, and does not deal with the issue of distributing (parts) of RBAC policies in a proper way.

Li and Mao design three main requirements (flexibility and scalability, psychological acceptability, economy of mechanism) and analyze them in different existing administrative models, and they design UARBAC, a new family of models. As mentioned earlier, none of the above-mentioned models address the issue of distributing (administrative) policies across a distributed system.

Somewhat related to our work is the paper by Bhamidipati and Sandhu which discusses how RBAC can be used in a number of different architectures with multiple servers in a network [3]. They focus on the capabilities of the servers, specifically on whether or not RBAC is *supported*, and treat the role-hierarchy as a central service. In our model on the other hand we distinguish which policy changes are relevant, and we do not update subsystems about irrelevant policy changes. dRBAC is a decentralized trust management and access control mechanism for systems that span multiple administrative domains [8]. It is targeted at settings where independent organizations form dynamic coalitions. Similar settings are also addressed by the TM models to be discussed below. In dRBAC, local policies in one administrative domain can be used in another domain; on the other hand, dRBAC does not address the issue of distributing policies (efficiently) across systems within an administrative domain.

Role-based Trust Management (TM) [12] and distributed certificate systems, such as SDSI [13], are remotely related lines of research. In these systems, a number of agents exchange *security statements* and may create hierarchies similar to those used in RBAC. Issuing TM credentials corresponds to administrative commands in RBAC. In TM however it is generally assumed that users are free to utter security statements, while the focus is on whether to *trust* such statements (which involves some trust calculation by the receiver of such statements). In RBAC this assumption is inappropriate, because policy changes are explicitly guarded by administrative privileges. The central issue of this paper, that is, to ensure that users can perform the actions they are allowed to, without broadcasting the entire security policy, has also been researched in TM. In some TM models the user is expected to collect the credentials needed for access by itself. In others *credential chain discovery* algorithms are used, which are automatic procedures to retrieve missing credentials. In this paper we describe a model that prevents situations where policy definitions must be retrieved ad-hoc by a subsystem, by pushing them to the interested subsystems upon the issuing of policy changes.

8. CONCLUSION

Despite a large body of literature on the administration of RBAC policies [5, 7, 14, 17], there is no proposal for RBAC administration in distributed systems. In this paper we fill this gap: we present a model for the implementation of a common RBAC standard in a distributed system. We focus on the formal requirements for such implementation, and we propose an administration procedure for the deployment of policy changes across the distributed system, which is efficient and preserves the formal requirements. A key part of our model is a privilege mapping, which captures the intuitive idea that different systems protect different objects. To demonstrate how the procedure can be implemented in practice, we translate our procedure to practical pseudo-code, and finally we also indicate how to extend our model to cover settings with multiple administrative systems, which are rather common in practice.

Acknowledgements

We would like to thank the anonymous reviewers for their helpful comments. We thank Jan Cederquist for discussions about early drafts of this paper. Marnix Dekker was funded by TNO and SenterNovem through the IOP Gencom project *PAW*. Sandro Etalle was partly funded by the projects EU-Serenity, and EU-NOE-ARTIST2.

9. REFERENCES

[1] RBAC Standard, ANSI INCITS 359-2004, 2004.

[2] E. Barka and R. S. Sandhu. Framework for role-based delegation models. In *Proceedings of the 16th Annual Computer Security Applications Conference (ACSAC)*, pages 168–176. IEEE Computer Society Press, 2000.

[3] V. Bhamidipati and R. Sandhu. Push architectures for user role assignment. In *Proceedings of the 23rd National Information Systems Security Conference (NISSC)*, pages 89–100, 2000.

[4] J. Crampton and H. Khambhammettu. Delegation in role-based access control. In *Proceedings of the 11th European Symposium on Research in Computer Security (ESORICS)*, LNCS, pages 174–191. Springer, Berlin, 2006.

[5] J. Crampton and G. Loizou. Administrative scope: A foundation for role-based administrative models. *ACM Transactions on Information System Security (TISSEC)*, 6(2):201–231, 2003.

[6] M. A. C. Dekker, J. Cederquist, J. Crampton, and S. Etalle. Extended privilege inheritance in RBAC. In *Proceedings of the 2007 ACM Symposium on Information, Computer and Communications Security (ASIACCS)*, pages 383–385. ACM Press, 2007.

[7] D. F. Ferraiolo, D. R. Kuhn, and R. Chandramouli. *Role-based Access Control*. Computer Security Series. Artech House, 2003.

[8] E. Freudenthal, T. Pesin, L. Port, E. Keenan, and V. Karamcheti. dRBAC: Distributed role-based access control for dynamic coalition environments. In *Proceedings of the 22nd International Conference on Distributed Computing Systems (ICDCS)*, pages 411–420, IEEE Computer Society Press, 2002.

[9] A. Kern, A. Schaad, and J. Moffett. An administration concept for the enterprise role-based access control model. In *Proceedings of the 8th ACM Symposium on Access Control Models and Technologies (SACMAT)*, pages 3–11, 2003.

[10] N. Li, J. Byun, and E. Bertino. A critique of the ANSI standard on role based access control. *IEEE Security and Privacy*, pages 1540-7993.

[11] N. Li and Z. Mao. Administration in role-based access control. In *Proceedings of the 2007 ACM Symposium on Information, Computer and Communications Security (ASIACCS)*, pages 127–138. ACM Press, 2007.

[12] N. Li, W. H. Winsborough, and J. C. Mitchell. Distributed credential chain discovery in trust management: extended abstract. In *Proceedings of the 8th ACM Conference on Computer and Communications Security (CCS)*, pages 156–165. ACM Press, 2001.

[13] R. L. Rivest and B. Lampson. SDSI – A simple distributed security infrastructure. Presented at CRYPTO'96 Rump session, 1996.

[14] R. S. Sandhu, V. Bhamidipati, and Q. Munawer. The ARBAC97 model for role-based administration of roles. *ACM Transactions on Information and System Security (TISSEC)*, 2(1):105–135, 1999.

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

[16] J. Wainer and A. Kumar. A fine-grained, controllable, user-to-user delegation method in RBAC. In *Proceedings of the 10th ACM Symposium on Access Control Models and Technologies (SACMAT)*, pages 59–66. ACM Press, 2005.

[17] H. Wang and S. L. Osborn. An administrative model for role graphs. In *Proceedings of the IFIP TC-11 WG 11.3 Annual Working Conference on Data and Application Security (DBSec)*, pages 302–315. Kluwer, 2003.

[18] L. Zhang, G. Ahn, and B. Chu. A rule-based framework for role-based delegation and revocation. *ACM Transactions on Information and System Security (TISSEC)*, 6(3):404–441, 2003.

[19] X. Zhang, S. Oh, and R. S. Sandhu. PBDM: a flexible delegation model in RBAC. In *Proceedings of the 8th ACM Symposium on Access Control Models and Technologies (SACMAT)*, pages 149–157. ACM Press, 2003.