A Role-based XACML Administration and Delegation Profile and Its Enforcement Architecture

Min Xu and Duminda Wijesekera
Department of Computer Science, George Mason University
Fairfax, VA, USA
mxu@gmu.edu, dwijesek@gmu.edu

ABSTRACT
The OASIS technical committee published the XACML v3.0 administration and delegation profile (XACML-Admin) working draft on 16 April 2009 [3] in order to provide policy administration and dynamic delegation services to the XACML runtime. We enhance this profile by adding role-based delegation by amalgamating the proposed profile with the XACML-ARBAC profile proposed in [19]. By doing so, we improve the scalability of the delegation mechanism. Second, we show how XACML-ARBAC enforcement mechanism proposed in [19] can be enhanced to enforce the proposed role-based administration and delegation XACML profile (XACML-ADRBC), therefore providing a method to enforce the XACML-Admin profile proposed in [3].

Categories and Subject Descriptors

General Terms
Security

Keywords
authorization, access control, administration, delegation, RBAC, ARBAC, XACML, Web service security

1. INTRODUCTION
The eXtensible Access Control Markup Language (XACML) [15] is emerging as the defacto standard to specify access control policies for Web services. The OASIS technical committee published the XACML v3.0 administration and delegation profile (XACML-Admin) working draft on 16 April 2009 [3] to support two Use Cases: (1) policy administration, and (2) dynamic delegation. The former controls the types of policies that individuals can create and modify, whereas the latter permits some users to create policies of limited duration to delegate selected capabilities to others. The delegation model used in [3] is a discretionary access control (DAC) model. Consequently, the profile only allows the owner of a permission to delegate it to a specific user, which is not scalable when permissions need to be delegated to a large number users with the same job function. In many cases in which the delegator is not available, or is unable to perform the delegation, it is more convenient to have a third party, such as the administrator, initiating the delegation on behalf of the user. This profile also lacks the support to allow delegators to delegate any subset of permissions assigned to him/her.

On a separate issue, this profile does not have an enforcement mechanism. Enforcing administrative or delegation operations will update relevant policies which results in read-write conflicts while the access controller attempts to evaluate a user’s access request. Also when an administrator or delegator attempts to revoke a permission granted to a user, the same user might still be exercising the permission to access a resource, which violates system safety. Consequently, we extend [3] to include the Use Cases of: (1) role-based delegation extending the delegation framework of [3], and (2) policy administration with or without delegation extending the Use Cases proposed in [19]. Furthermore, we show how these extended Use Cases can be realized by extending the design implemented in [19] that retains the system safety by revoking permissions invalidated by policy updates. To provide the extra Use Cases and enforce delegation, we divide the access requests into three categories as follows:

- Regular User Access Request: User requests access permission to a resource. This is the most common type of request made to an access control system.
- Administrative Request: Administrator requests an administrative request, such as changing privileges of a user etc.
- Delegation Request: Delegator (user or administrator) requests delegating one user’s permissions to another.

The first two category requests have been successfully specified and enforced in [19]. In this paper, we propose a role-based administration and delegation model, in which the delegators (user or administrator) are assigned to a delegable role granted with a set of delegation permissions. Our delegation permissions are different from the permissions to access resources. The delegation permissions are semi-administrative in nature such as creating/deleting a
delegated role (e.g., a role that can be assigned to a delegatee), granting/revoking permission to/from a delegated role, and assigning/removing a user to/from a delegated role. To provide multi-step delegation (e.g., to be able to delegate delegation permission), the delegable role can also create another delegable role. This role-based approach is scalable, because it facilitates permissions to be delegated to a large number of users that may want to be delegatees of the same permission set. It is flexible because it allows the delegators to delegate any subset of the permissions assigned to him/her and modify the delegated permissions in case of needed.

Specifically, we extend the XACML-ARBAC framework proposed in [19] as follows. First, we incorporate a delegation model into the ARBAC model, which adds a delegable role and a delegated role. The delegable role is granted a set of permissions to delegate any subset of permissions granted to the regular role which can be delegated in case of need. Second, we extend the XACML-ARBAC profile to cover the XACML-Admin profile, which we refer to as the XACML-ADRBAC profile adding appropriate syntax and constraints. Third, we extend the XACML-ARBAC enforcement architecture of [19] and specify the extra functionality required for the policy administration point (PAP) to enforce the extended XACML-ADRBAC profile. In order to achieve all these simultaneously, we direct different kind of access requests to different entities of the XACML runtime: the regular request to the policy enforcement point (PEP), the administrative request to the administrative PEP (A-PEP), and the delegation request to the PAP. We add a Lock Manager to enforce concurrency control necessary to maintain the transactional consistency between simultaneous operations among the policy decision point (PDP), A-PEP and PAP.

The rest of the paper is organized as follows. Section 2 briefly describes the preliminary information required to understand the rest, namely the OASIS administration and delegation profile, the ARBAC and session administration model, and some role-based delegation models. Section 3 introduces our role-based administration and delegation model. Section 4 presents our XACML-ADRBAC profile. Section 5 describes the enforcement architecture for the XACML-ADRBAC profile. Section 6 presents related work and Section 7 concludes the paper.

2. PRELIMINARIES

2.1 XACML v3.0 Administration and Delegation Profile

The delegation model proposed in [3] specify the permissions to create policies and methods to account for created polices against these permissions by tracing their lineage to a delegation chain. In order to do so, the proposed XACML-Admin profile adds a new key word <PolicyIssuer> element that identifies the source of the policy, where a missing <PolicyIssuer> element implies that the policy is trusted. A trusted policy is considered valid and its origin is assumed not to require verification by the PDP. Policies which have an issuer need to have their authority accounted for by using a delegation chain. If the authority of the policy issuer can be traced back to a trusted policy through a chain of delegations, the policy is used by the PDP, and discarded otherwise. The authority of the issuer depends on the context of the access request, therefore a policy can be valid or invalid depending on the request context. There are two access situations: the current attributes mode and the historic attributes mode. In the current attributes mode, when a delegated attribute is dynamic, the attribute value at the access request time must be used. In the historic attribute mode, when a delegate attribute is dynamic, the attribute value at the policy creation time must be used. Steps in the validation process are performed using a special XACML requests, called the administrative requests, which contain information about the policy issuers and the access mode.

2.2 ARBAC and Session Administration Model

We use the notation $RBAC = (U, O, A, R, P, \leq, U2R, R2P)$ for the model of an RBAC system, where the first four entities are the sets of users, objects, actions, and roles, respectively. $P$ is a subset of $O \times A$, representing the set of permissions. The partial ordering $\leq \subseteq R \times R$ is the role hierarchy. $U2R : U \mapsto 2^R$ and $R2P : R \mapsto 2^P$ are relations that are functional in their first coordinate, modeling user-to-role and role-to-permission assignments. That is, $U2R(u, M)$ and $R2P(r, N)$ are true iff user $u$ is allowed to play the set of roles $M$ and role $r$ can execute the set of permissions $N$ respectively. We use function $assignedPerm(\mu) = \bigcup_{r \in U2R(u), r \in \mathcal{R}2P(r)} R2P(r)$ to return the set of permissions that a given user obtains through his or her assigned roles. Administrative role based access control (ARBAC) models follows the spirit of administrating an RBAC model using another RBAC model. We use the ARBAC model proposed in [19] as follows.

**Definition 1 (ARBAC).** Let $(U, O, A, R, P, \leq, U2R, R2P)$ be an RBAC model. An administrative RBAC model is a tuple $ARBAC = (U, AO, AA, AR, AP, \leq_A, U2AR, U2AR2)$, where

- $AO = U \cup R \cup U2R \cup R2P \cup \leq$ is the set of administrative objects;
- $AA$ is the set of administrative actions given in Table 1;
- $AR$ is a set of administrative roles;
- $AP \subseteq (AO \times AA) \cup (AO \times AO \times AA)$ is the set of administrative permissions;
- $\leq_A \subseteq AR \times AR$ is the administrative role hierarchy;
- $U2AR : U \mapsto 2^{AR}$ is the user-to-administrative role assignment;
- $AR2AP : AR \mapsto 2^{AP}$ is the administrative role-to-administrative permission assignment.

As defined, administrative objects (AO) in ARBAC include the set of users ($U$), roles ($R$), user-to-role ($U2R$) and role-to-permission ($R2P$) mapping and the role inheritance relation ($\leq$) in RBAC, and administrative actions (in Table 1) create, update, and destroy these objects. For example, AssignUser and DeassignUser operation creates and removes entries in the user-to-role mapping $U2R$, respectively. An administrative permission is an application of an

---

1 The policy administration request in [3] is different from the administrative request in [19] which requests an administrative operation that results in changing the configuration of an RBAC system.
administrative action on one or two appropriate administrative objects. Execution of an administrative action changes the RBAC system configuration or condition to a new state. Our previous work [19] defined the following administrative model for session management.

**Definition 2 (Session Administrative Model).**
Let \((U, O, A, R, P, \leq, \leq, U2R, R2P)\) be the model of an RBAC system. A session administrative model is a tuple \(SAM = (ACTIVE-S, S \to ACTION, U2S, S2R, actRole, actPerms)\), where

- **ACTIVE-S** is the set of all active sessions at a given system state;
- **S \to ACTION** = \{(CreateSession(u, s), DeleteSession(u, s), ActivateRole(u, s, r), DeactivateRole(u, s, r), GrantPermission(u, r, P))\} is the set of session administrative actions, where \(u \in U\), \(r \in R\) and \(s \in ACTIVE-S\).
- \(U2S : U \mapsto 2^{ACTIVE-S}\) is a function mapping a user to a set of active sessions at a system state;
- **S2R : ACTIVE-S \mapsto 2^R** is a function mapping an active session to a set of activated roles at a system state;
- \(U2S \circ S2R(u) \subseteq U2R(u)\) is the constraint that at a system state, all activated roles of a user is a subset of or equal to the set of his or her assigned roles, where \(U2S \circ S2R(u) = \bigcup_{s \in U2S(u)} S2R(s)\);
- **activeRoles(u) = \bigcup_{s \in U2S(u)} S2R(s)\) is a function mapping a user to a set of active roles in all active sessions at a system state;
- **activePerms(u) = \bigcup_{s \in U2S(u), r \in S2R(s)} \bigcup_{r' \geq r} R2P(r')\) is a function mapping a user to a set of administrative permissions at a system state.

Each session administrative action changes the system to a new state, for example by creating/deleting a session for a user, or activating/deactivating a role within a session. When an administrative operation modifies a role, the administrative operation places write locks on the affected roles to prevent the PDP from “reading” the roles and other administrative operations from “writing” the roles. The affected roles are defined as lock scope as follows.

**Definition 3 (Lock Scope).** Let \((U, O, A, R, P, \leq, U2R, R2P)\) be the model of an RBAC system and \(r \in R\) be a role. We define the read scope and write scope of \(r\) respectively as \(r \to \text{Scope}(r) = \{r' \in R | r' \leq r\}\) and \(w\text{Scope}(r) = \{r' \in R | r' \geq r\}\).

### 2.3 Role-based Delegation

A number of models address various aspects of delegation, including [13, 10, 14, 5]. There is a series of role-based delegation models [7, 6, 20, 21]. RBDM0 [7] and RDM2000 [20] in particular are primarily based on roles, and address human-to-human delegation, whereby a user in a role (delegator role) delegates his/her role membership to another user in another role (delegatee role). RDM2000 extends RBDM0 by adding role hierarchies and multi-step delegation.

We base our work partially on PBDM [21], which extends RBAC [16] to include user-to-user delegations and role-to-role delegations. PBDM summarizes three scenarios that can use delegation: back up role, decentralization of authority, and collaboration. The first and third cases need temporary delegation, while the second case needs durable delegation. PBDM provides for single-step as well as multi-step delegation. In PBDM, there are three different types of roles: regular role (RR), delegable roles (DBR), and delegation roles (DTR). Permissions assigned to regular roles cannot be delegated to other roles or users. A delegable role can be delegated to other roles or users by creating delegation roles. Each delegable role has exact one base regular role. PBDM supports revocation by a user (1-3) or by an administrator (4-5) as follows.

1. Revoke the user-delegation role assignment.
2. Remove permissions from the delegation role.
3. Remove the delegation role.
4. Remove permissions from the delegable role.
5. Remove a user from a regular and its delegable role.

### 3. Administration and Delegation Model

In order to cover the XACML-Admin profile, we add the PBDM delegation model to the ARBAC model. We partition roles into regular roles (RR), delegable roles (DBR), and delegable roles (DR) (which are called delegation roles in PBDM [21]), and administrative roles (AR). A delegable role can be delegated to other roles or users by creating delegable roles or delegable roles (for multi-step delegation purpose). To support discretionary access control (DAC), that users that are assigned to a regular role are also assigned to the delegable role based on it. Administrators can be assigned to the delegable roles. The delegated roles are the role assigned to the delegator created by the delegable roles. This partition induces a partition of \(U2R\) and \(R2P\). \(U2R\) is separated into user-to-regular role assignment (U2RR), user-to-delegable role assignment (U2DBR), user-to-delegated role assignment (U2DR), and user-to-administrative role assignment (U2AR). Similarly \(R2P\) is separated into regular role-to-permission assignment (RR2P), delegable role-to-delegation permission assignment (DR2P), and delegable role-to-permission assignment (DDR2P). Administrative permissions are different from the regular permissions. Delegation permissions are also different from regular permission. A delegable role cannot have any senior regular role if it is placed into the role hierarchy. In general, definition of RR, DR, U2DR, RR2P, U2DR, DR2P, U2AR, ≤, and making changes to the delegable role hierarchy ≤ is the responsibility of security administrators. Definition of DR, U2DR, DR2P and delegable role hierarchy (≤) is the responsibility of users that can add a form of DAC to the ARBAC model. DRBR \(\to (DR \times DR)\) is the local delegated role hierarchy created by the DRBR.
configuration change affects the running system state, which
tions change the configuration of an RBAC system. Any
DBR \leq 2
R
DR
\leq \leq_2
U
 \leq_2
A
\leq \leq_2
B
\leq_2
D
\leq A
)
where
• \(R=RR \cup DBR \cup DR \cup AR\), where \(RR\) is a set of regular
roles, \(DBR\) is a set of delegable roles, \(DR\) is a set of delegated roles, and \(AR\) is a set of administrative roles,
with constraints: \(RR \cap DR = \emptyset\), \(RR \cap DBR = \emptyset\),
\(RR \cap AR = \emptyset\), \(DBR \cap DR = \emptyset\), and \(AR \cap DR = \emptyset\);
• \(AO = U \cup RR \cup DDBR \cup U_{2RR} \cup U_{2DBR} \cup U_{RRP} \cup
DBR_{2DP} \cup \leq U \leq 2DDBR\) is the set of administrative
objects;
• \(AA\) is the set of administrative actions given in Table 1;
• \(AP \subseteq (AO \times AA) \cup (AO \times AO \times AA)\) is the set of administrative
permissions;
• \(DP\) is the set of delegation permissions give in Table 2;
• \(P \subseteq (O \times A)\) is the set of regular permissions;
• \(U_{2RR}: U \mapsto 2RR\) is the user-to-regular role assignment;
• \(U_{2DBR}: U \mapsto 2_{DBR}\) is the user-to-delegable role assignment;
• \(U_{2DR}: U \mapsto 2_{DR}\) is the user-to-delegated role assignment;
• \(U_{2AR}: U \mapsto 2_{AR}\) is the user-to-administrative role assignment;
• \(RRP: RR \mapsto 2^p\) is the regular role-to-permission assignment;
• \(DBR_{2DP}: DBR \mapsto 2^DP\) is the delegable role-to-delegation
permission assignment;
• \(DR_{2P}: AR \mapsto 2^p\) is the delegated role-to-permission assignment;
• \(AR_{2P}: AR \mapsto 2^P\) is the administrative role-to-administrative
permission assignment;
• \(\leq \leq_2 RR \times RR\) is the regular role hierarchy;
• \(\leq_2 DBR \times DBR\) is the delegable role hierarchy;
• \(\leq_2 DR \times DR\) is the local delegated role hierarchy
created by the DBR;
• \(\leq_2 AR \times AR\) is the administrative role hierarchy;
• \(\text{senior}(r): R \mapsto 2^p\): a function mapping a role to all
its senior roles in role hierarchy;
• \(\text{base}(dr): DBR \mapsto RR\): a function mapping each
delegable role to a single regular role on which it is
based;
• \(\forall dbr \in DBR. \text{senior}(dbr) \cap RR = \emptyset\): no delegable role
has a senior regular role;
• \(\forall dr \in DR. \text{senior}(dr) \cap RR = \emptyset \land \text{senior}(dr) \cap DBR = \emptyset\): no delegated role has a senior
regular role or a delegable role;

Similar to administrative operations, delegation operations
change the configuration of an RBAC system. Any
configuration change affects the running system state, which
may demand session administrative actions (see Section 2.2)
to be invoked. The interaction between session administrative
actions and the delegation operations (i.e., the delegation
operation defined in Table 2) needs to be specified in order
to ensure the safety of the ADRBAC model. Because we
treat delegation operations as semi-administrative operations,
concurrency control requirements between the session
administrative model and delegation model for an RBAC
system extends the concurrency control requirements stated in
[19]: (1) revoke activated role or delete active session
immediately, and (2) delay administrative/delegation
operations. When a delegation operation modifies a role, it
might affect other roles. For example, removing a permission
from a delegated roles will affect the users granted to the
delegated role and all roles senior to the delegated role. We
can define the affected entities because of invoking a dele-
gation operation using lock scope defined in Section 2.2.
Algorithm 1 in Figure 2 shows this information for every
delegation operation listed in Table 2, although we do not
consider revoking delegations.

4. ROLE-BASED XACML ADMINISTRATION
AND DELEGATION PROFILE

We proposed an XACML-ARBAC profile where objects,
actions, and users are expressed as XACML <Resource>s,
<Action>s and <Subject>s respectively in our previous
work [19]. The XACML-ARBAC introduced a new data
type RoleType to the XACML syntax, and a roleType
attribute that can take value from \{userRole, adminRole\} to
distinguish administrative roles from user roles. This profile
also defines two generic XACML policies: a Permission
PolicySet, and a Role PolicySet. These are used to
express the entities of an RBAC model (i.e. permissions,

<table>
<thead>
<tr>
<th>Operations</th>
<th>Intuitive Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>DelegateRole(u,dr)</td>
<td>Delegate dr to u</td>
</tr>
<tr>
<td>DeassignUser(u,dr)</td>
<td>Deassign u from dr</td>
</tr>
<tr>
<td>GrantPermission(dr,p)</td>
<td>Grant p to dr</td>
</tr>
<tr>
<td>RevokePermission(dr,p)</td>
<td>Revoke p from dr</td>
</tr>
<tr>
<td>AddRole(dr)</td>
<td>Add dr</td>
</tr>
<tr>
<td>DeleteRole(dr)</td>
<td>Delete dr</td>
</tr>
<tr>
<td>AddEdge(dr,c,dr&quot;)</td>
<td>Make dr&quot; as a child of dr&quot;</td>
</tr>
<tr>
<td>DeleteEdge(dr&quot;,dr&quot;)</td>
<td>Remove dr&quot; as a child of dr&quot;</td>
</tr>
</tbody>
</table>

Table 2: Delegation Operations

Figure 1: Compute affected entities of a delegation operation.

Algorithm 1: Compute affected entities
Input: delegateOp
Output: Return affected to PAP
1: switch delegateOp do
2: case DeleteUser(u) { affected:=u; } 3: case DeassignRole(dr) { affected:=wScope(dr); } 4: case DeassignUser(u,dr) { affected:=(rScope(dr),u); } 5: case GrantPermission(dr,p) { affected:=wScope(dr); } 6: case RevokePermission(dr,p) { affected:=wScope(dr); } 7: case DeassignUser(u,dr) { affected:=wScope(dr); } 8: case DeassignRole(dr) { affected:=wScope(dr); } 9: case DeleteRole(dr) { affected:=wScope(dr); } 10: case DeassignUser(u,dr) { affected:=wScope(dr); } 11: case GrantPermission(dr,p) { affected:=wScope(dr); } 12: case RevokePermission(dr,p) { affected:=wScope(dr); } 13: case DeassignRole(dr) { affected:=wScope(dr); } 14: return affected;
A Permission \(<\text{PolicySet}\rangle\) is a \(<\text{PolicySet}\rangle\) used to define a set of permissions associated with a role. It may contain \(<\text{PolicySetIdReference}\rangle\) to other Permission \(<\text{PolicySet}\rangle\). Stated \(<\text{PolicySetIdReference}\rangle\)'s can be used to inherit permissions of a junior role.

A Role \(<\text{PolicySet}\rangle\) binds a set of attributes defining a role in a \(<\text{Target}\rangle\) to a \(<\text{PolicySetIdReference}\rangle\) outside of that \(<\text{Target}\rangle\). The latter points to the Permission \(<\text{PolicySet}\rangle\) of the role.

This XACML-ARBAC profile uses an XML file to maintain all user-to-role assignments in the policy repository which is different from the Role Assignment \(<\text{Policy}\rangle\) or \(<\text{PolicySet}\rangle\) in the XACML-RBAC [1] profile. The administrative policies uses the same machinery as the XACML-ARBAC profile, but with the additional constraints. The details of the additional constraints is presented in [19].

### 4.1 The XACML-ADRBAC Profile

For the XACML-ARDBAC profile, we add the values \(\text{delegatedRole}, \text{delegableRole}\) to the \(\text{roleType}\) attribute to distinguish delegated roles, delegable roles from user roles (regular roles) and administrative roles. We use all other primitive entities from the XACML-ARDBAC profile. We specify our administration and delegation profile (XACML-ADRBAC) using the same machinery as the XACML-ARBAC profile, but with the following added constraints for delegable roles and delegated roles.

**Constraining the Delegable Role \(<\text{PolicySet}\rangle\):** The Role \(<\text{PolicySet}\rangle\) of a delegable role must be a delegable Role \(<\text{PolicySet}\rangle\) with the following constraints:

1. All role names that appear in the \(<\text{Target}\rangle\) of the Role \(<\text{PolicySet}\rangle\) should be delegable roles, with the \(\text{roleType}\) set to \(\text{delegableRole}\).
2. The \(<\text{PolicySetIdReference}\rangle\) contained in the Role \(<\text{PolicySet}\rangle\) should point to a delegation Permission \(<\text{PolicySet}\rangle\), to be described shortly.
3. There should be exact one regular role matching this delegable role. We add a \(<\text{BaseRole}\rangle\) element to link the delegable role to the regular role.

**Constraining the Delegable Permission \(<\text{PolicySet}\rangle\):** All permissions listed in a \(<\text{PolicySet}\rangle\) of a delegable role must be delegation permissions as defined in Table 2. By enforcing the following constraints on the syntax used in a permission \(<\text{PolicySet}\rangle\), we ensure that it is a delegable Permission \(<\text{PolicySet}\rangle\).

1. The \(\langle\text{Action}, \text{Resource}\rangle\) pair listed in \(<\text{Rule}\rangle\) must form a delegation permission. That is, the \(\text{Action}\) attribute must be chosen from the operation names and the \(\text{Resource}\) attribute must be chosen from the operation parameters stated in Table 2.
2. This delegable role does not have any senior regular role.

**Constraining the Delegated Role \(<\text{PolicySet}\rangle\):** The Role \(<\text{PolicySet}\rangle\) of a delegated role must be a delegated Role \(<\text{PolicySet}\rangle\) with the following constraints:

1. All role names that appear in the \(<\text{Target}\rangle\) of the Role \(<\text{PolicySet}\rangle\) should be delegated roles. That is, their \(\text{roleType}\) should be set to \(\text{delegatedRole}\).
2. The \(<\text{PolicySetIdReference}\rangle\) contained in the Role \(<\text{PolicySet}\rangle\) should point to a delegated Permission \(<\text{PolicySet}\rangle\) to be described shortly.

**Constraining the Delegated Permission \(<\text{PolicySet}\rangle\):** All permissions listed in a \(<\text{PolicySet}\rangle\) of a delegated role must be a subset of permissions granted to the base role of the delegable role which created the delegated role. By enforcing the following constraints on the syntax used in a permission \(<\text{PolicySet}\rangle\), we ensure that it is a delegated Permission \(<\text{PolicySet}\rangle\).

1. The permissions are a subset of permissions of the base role of the delegable role which creates the delegated role. This can be checked by using the extended function \(\text{permission-in-permissionSet}(p,P)\) stated in Table 3.
2. This delegated role has no senior regular or delegable role.

The XML file recoding the user-to-role assignment is changed as follows. For any delegable role, we add a \(<\text{BaseRole}\rangle\) element. For example, Mary is assigned to the Manager role and a delegable ManagerDelegable role which has a base Manager role. Alice, the administrator, is also assigned to the ManagerDelegable role. Notice that Alice is not assigned to the Manager role, implying that the delegable may not have all the permissions granted to the base role. Jack is assigned to the ManagerDelegated role.

We enlarge the set of XACML condition functions (listed in Table 3) in order to check the constraints added in this section. Table 4 contrasts the elements of XACML-ADRBAC profile with the XACML-ADRBAC profile.

### 5. ENFORCEMENT ARCHITECTURE

In order to enforce the proposed XACML-ADRBAC profile, we extend the XACML-ARDBAC enforcement architecture as shown in Figure 5. For simplicity, we only draw the added interaction in Figure 2. The \(n.a\) shows the flow of the regular user access request, \(n.b\) shows the flow of the administrative request, and \(n.c\) shows the flow of the delegation request, where \(n\) is an integer that shows the stage of the control flow inside the enhanced XACML runtime.

The PDP evaluates the requests against the policies and renders one of \{permit, deny, indeterminate, not applicable\} as the outcome of the authorization decision.

The PEP receives a regular request, and enforces the authorization decision from the PDP. If needed, terminates the user sessions required due to enforcing an administrative operation or delegation operation.

The Administrative PEP (A-PEP) receives an administrative request stated in Table 1, returns a response to the administrator, and if needed, updates relevant policies as a consequence of enforcing the requested administrative operation. When a user is assigned to a role and revoked from a role, the A-PEP acts as a role enabler/disabler by invoking the appropriate administrative operation and updates the U2R mapping in an XML file. Details appear in [19].

The Policy administration point (PAP) creates policies at authoring time and administers (creates, updates, removes) policies during runtime. In this paper, we mainly discuss the runtime enforcement of these operations. When a delegator sends a policy delegation request to the PAP, the
PAP in turn send a request to the PDP, obtains the result and forwards the response to the delegator, and if needed, updates relevant policies as a consequence of enforcing the requested delegation operation.

The Lock Manager provides the concurrency control necessary to maintain the transactional consistency between simultaneous operations that are generated by (1) the PDP reading policies in order to evaluate access request, (2) the A-PEP modifying policies to enforce administrative operations, and (3) the PAP modifying policies to enforce delegation operations.

### 5.1 Concurrency Control

**Regular User Access Request:** When an access request arrives at the PEP (flow 1.a), the PEP forwards the request to the PDP (flow 2.a). The PDP requests a read lock on the policy that is found using the target matching algorithm (flow 3). Then the PDP evaluates the request (flow 4) and conveys the authorization decision to the PEP (flow 5.a). The details of evaluating authorization requests is described in Figure 4 of [19].

**Administrative Request:** When an administrative request arrives at the A-PEP (flow 1.b), the A-PEP forwards the request to the PDP (flow 2.b). The PDP requests a read lock on the policy that is found using the target matching algorithm (flow 3) from the lock manager. Then the PDP evaluates the request (flow 4) and conveys the authorization decision to the A-PEP (flow 5.b). The policy evaluation part (flows 3 and 4) is similar to that of evaluating a regular request. If the A-PEP gets a permit decision from the PDP, the A-PEP acquires a write lock on the policy (recall that administrative requests update XACML policies) that is to be updated (flow 6.b). After acquiring the write lock successfully, the A-PEP updates the policy to enforce the administrative operation (flow 7.b). The details of enforcing administrative operations is described in Figure 5 of [19].

**Delegation Request:** When a delegation request is submitted to the PAP (flow 1.c), the PAP forwards the request to the PDP for evaluation (flow 2.c). The PDP uses the same evaluation algorithm used to evaluate the access request (flows 3.4) (see Figure 4 in [19]) and returns the decision to the PAP (flow 5.c). If the returned value received at the PAP is not a permit, the PAP conveys the decision to the delegator (user or administrator). Otherwise (e.g., the return value is permit), the PAP uses the algorithm shown in Figure 3 to enforce that decision.

As the algorithm states, if the decision is not a permit, the PAP returns that decision to the delegator (lines 19). Otherwise, it acquires a write lock on the policy to be updated (line 3), calls the method getAffected(delegateOp) using the algorithm shown in Figure 1 to determine the parameters that are affected by the delegation operation (Line 5). Then, the PAP sends a request to all PEPs to terminate user sessions that can be affected by enforcing the delegation operation (lines 6-8), so that updating a policy while these users access permissions granted earlier do not render the access controller unsafe. Because the access controller cannot wait forever for those PEPs to confirm that the requested sessions have been terminated, the PAP sets up a timer (line 7). If all those PEPs returned successful answers (lines 12-14), the PAP updates the policy/user-role assignment XML file to enforce the delegation, releases the write lock on the policy (line 16), and finally informs the delegator that the delegation is enforced (the permit decision). Conversely, if any PEP fails to return a positive answer when the timer expires, the delegation is denied, the requestor is informed, and the write lock is released.

### 6. RELATED WORK

The XACML-ARBAC framework was proposed in our previous work [19], which proposed a session-aware administrative model for RBAC to manage interactions and potential conflicts between session management and administrative operations. The framework had an XACML-ARBAC profile to specify ARBAC polices and enforcement architec-

<table>
<thead>
<tr>
<th>Function</th>
<th>Intuitive Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>role-exist(r)</td>
<td>check the presence of the role r</td>
</tr>
<tr>
<td>inherited-by-assigned-role(r)</td>
<td>check if the given role r is inherited by a role already assigned to the subject</td>
</tr>
<tr>
<td>inherit-assigned-role(r)</td>
<td>check if the given role inherits a role already assigned to the subject</td>
</tr>
<tr>
<td>role-assigned-exist(s,r)</td>
<td>check if the subject s is already assigned to the role r</td>
</tr>
<tr>
<td>role-has-children(r)</td>
<td>check if the role r has any children</td>
</tr>
<tr>
<td>role-has-parent(r)</td>
<td>check if the given role has any parent</td>
</tr>
<tr>
<td>role-is-assigned(r)</td>
<td>check if the role r is assigned or not</td>
</tr>
<tr>
<td>role-is-inherited-by(r1,r2)</td>
<td>check if r1 is inherited by r2</td>
</tr>
<tr>
<td>role-is-parent-of(r1,r2)</td>
<td>check if r1 is parent of r2</td>
</tr>
<tr>
<td>permission-in-permissionSet(p,P)</td>
<td>check if p is an element of permission set P</td>
</tr>
</tbody>
</table>

Table 3: Extended functions applied in <Condition> in XACML-ARBAC profile

<table>
<thead>
<tr>
<th>XACML-ARBAC</th>
<th>XACML-ADRBAC</th>
</tr>
</thead>
<tbody>
<tr>
<td>user-role XML assignment</td>
<td>user-role XML assignment</td>
</tr>
<tr>
<td>constraints on administrative role &lt;PolicySet&gt;</td>
<td>constraints on administrative role &lt;PolicySet&gt;</td>
</tr>
<tr>
<td>constraints on administrative permission &lt;PolicySet&gt;</td>
<td>constraints on administrative permission &lt;PolicySet&gt;</td>
</tr>
<tr>
<td>constraints on delegable role &lt;PolicySet&gt;</td>
<td>constraints on delegable permission &lt;PolicySet&gt;</td>
</tr>
<tr>
<td>constraints on delegated role &lt;PolicySet&gt;</td>
<td>constraints on delegated permission &lt;PolicySet&gt;</td>
</tr>
</tbody>
</table>

Table 4: Contrasting elements in XACML-ARBAC with XACML-ADRBAC
Seitz et al. [17] presented a system permitting controlled policy administration and delegation using the XACML access control system. They used Delegent, an XML authorization server to provide delegation of administration. They added the delegation as an extra component while we added delegation and administration in a unified framework.

PERMIS [8] developed a role based access control infrastructure using X.509 [2] attribute certificates (ACs) to store the U2R relation. The PERMIS architecture included a Privilege Allocator GUI tool and a bulk loader tool that allowed administrators to construct and sign ACs and store them in a LDAP directory to be used by the PERMIS decision engine. Later PERMIS used an XML interface using Sun’s reference implementation [4] and added dynamic delegation of authority [9]. PERMIS assumed that privileges can be formulated as attributes and given to users. The dynamic delegation of authority was enacted via the issuing of credentials from one user to another. Our delegation model is role-based which is more scalable and flexible.

There have been many theoretical work on delegation. Delegation logic [11] was used to as a trust-management engine with credentials proving that a request complies with a policy. Delegation logic lacks the explicit subject abstraction which is desired for attributed-based delegation and role-based delegation. Li et al. [12] proposed a role-based trust-management framework (RT). In RT framework, role activation is delegated by issuing delegation credentials. Tamassia et al. [18] proposed a model for delegation of authority in decentralized trust management systems which combined the RT framework [12] with cascading delegation. Bandmann et al. [5] introduced a constrained delegation model which controlled the possible shapes of delegation chains. They used constraints to restrict the capability at each step of delegation.

7. CONCLUSION

We propose a role based administration and delegation model and an XACML-administration and delegation pro-

Algorithm 2: Enforcing Delegation Requests

Input: delegateRequest, PDPdecision
/*PDP returns policy decision to PAP*/

Data: PEPLList

Output: Return decision to the delegator

1 if PDPdecision==permit then
2   decision:=deny;
3 if AcquireLock(policy,write) then
4   if delegateOp is a (-) operation then
5     Affected:=getAffected(delegateOp);
6     forall PEP ∈ PEPLList do
7       set(timer, value);
8       sendRequest(PEP,(Affected,killSession));
9     if expires(timer) then
10       acceptFlag:=ok;
11     forall PEP ∈ PEPLList do
12       recv(PEP,(Affected, killsSession, NotOK));
13     acceptFlag:=reject;
14   if acceptFlag=ok then
15     modifyPolicy(policy, delegateRequest);
16     ReleaseLock(policy,write);
17     decision:=permit;
18 else
19     decision:=PDPdecision;
20 return (delegator, decision);

Figure 3: Enforcing Delegation Request.

Figure 2: Extended XACML architecture for XACML-ADRBAC enforcement.
file following the model which is more scalable, and flexible compared to the OASIS XACML-admin profile [3]. We present an enforcement architecture which elaborates the functionality of PAP and uses locking manager to address the concurrency issue at runtime. We are currently building a prototype by extending delegation to the XACML-ARBAC framework.

8. REFERENCES