

Implementing ACL-based Policies in XACML

Günter Karjoth, Andreas Schade and Els Van Herreweghen
 IBM Research
 Zurich Research Laboratory

Abstract

XACML is commonly used as a policy exchange mechanism, decision engines are available, and verification tools are under development. However, no support for legacy access control systems exists yet. To explore the feasibility to support legacy systems, we designed and implemented a mapping of the IBM® Tivoli® Access Manager policy language into XACML. Although the Tivoli Access Manager policy language, being ACL-based, is simpler in general, it turned out to be a non-trivial task to encode the interplay of the Tivoli Access Manager policy elements and decision logic within XACML. To achieve this task, we had to come up with a novel use of XACML features.

1 Introduction

The *extensible access control markup language* (XACML), an OASIS standard formally ratified in 2005, defines both a policy language and an access control decision request/response language, both given in XML [6]. It also includes a high-level architecture defining the roles of and the data flow between the entities involved to make an authorization decision. Authorizations are expressed in form of positive or negative rules, possibly subject to a condition. Rules are combined into policies and policies may be combined into policy sets. Several combining algorithms exist to aggregate the individual results yielded by rules, policies, or policy sets. These composition operators express logical operations on the results, supporting federated administration of policies on shared resources.

Recently XACML has received considerable attention from academia as well as from industry. At Burton Group's 2007 Catalyst Conference North America¹, eight XACML vendors showed fundamental interoperability in two usage scenarios: policy exchange and authorization decision processing.

¹<http://www.oasis-open.org/news/xacml-interop-2007-press-release.pdf>

Whereas interest in and adoption of XACML continues to increase across the industry, not all application areas of XACML are yet clearly identified. XACML as a policy exchange mechanism is already commonly used. Decision engines are currently developed [10] but XACML policy editing tools are missing. A number of tools, Margrave [4] for example, have been developed for the verification of XACML policies. In [14], XACML policies are derived from business process models. However, to gain wide market penetration, XACML has also to support existing legacy access control systems; i.e., to show its platform independence. To manage access control policies of heterogeneous computing environment centrally, there is the need to import (translate) these policies into XACML. Whereas several translations of XACML into other formats exist, mainly for verification purposes, to the best of our knowledge there is no work in the other direction with the exception of translating Java® PolicyFile policies into XACML [1].

IBM Tivoli Access Manager (AM) is a system for (1) centrally managing policy to govern access to resources over geographically dispersed intranets and extranets and (2) providing authorization services to applications [8]. Applications that are part of the Tivoli Access Manager family include WebSEAL (for Internet resources), Access Manager for Business Integration (for MQSeries queues and applications), and Access Manager for Operating Systems (for system resources). Third-party applications can use Tivoli Access Manager's authorization service by calling its standard-based Authorization API [13].

AM performs authorization checks (i.e., access control decisions) on protected URL-addressable objects, including "dynamic URLs" generated by applications, based on the user's credentials. Policies are defined over an object space. ACL inheritance allows to manage these policies efficiently. Access control lists (ACLs) attached to resources assign permissions to users and groups and control their reachability. Predefined or customer-defined conditions, called protected object policies (POPs) respectively authorization rules (ARules), may further restrict access or give additional instructions to the resource manager.

To study the feasibility to support such legacy systems,

we designed and implemented a mapping of the AM policy language into XACML. Although AM's language, being ACL-based, is simpler in general, it turned out to be a non-trivial task to encode the interplay of AM's data structure and decision logic within XACML. To achieve this task, we had to come up with a novel use of XACML features. An AM policy and an XACML policy are equivalent if for a given query both decision engines return the same result. As opposed to AM's ability to support multiple actions per query, queries have to be restricted to only contain a single action as prescribed by the XACML model for decision evaluation. The described XACML mapping covers the full range of Tivoli Access Manager policies provided that XACML condition expressions exist as counterparts for all authorization rules.

In the remainder of this paper, we briefly describe the Tivoli Access Manager authorization model in Section 2 and that of XACML in Section 3. In Section 4, we outline the encoding of the AM policy elements ACL, POP and ARule. The complete translation algorithm is given in Section 5, where we show how we have captured the special effects of permission inheritance and accessibility. Section 6 describes the implementation of the AM to XACML translator and discusses software requirements to support general XACML policies. We conclude in Section 7.

2 Tivoli Access Manager Policy Semantics

In this section, we describe the part of AM, which is relevant for the translation into XACML. A detailed and formal description of Tivoli Access Manager's underlying access control model is given in [8].

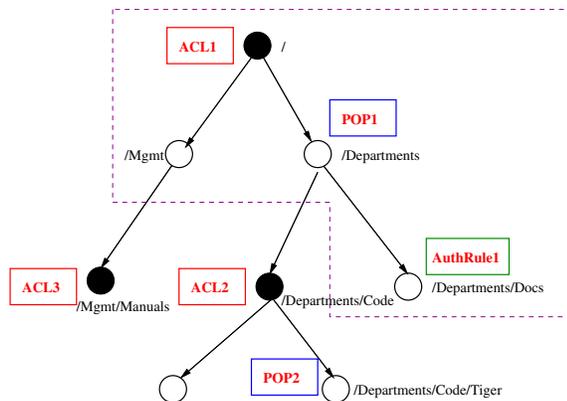


Figure 1. Object space populated with ACLs, POPs, and ARule.

To control access, AM deploys access control lists (ACL), protected object policies (POP), and authorization rules (ARule). An ACL grants authorizations to subjects,

which are single users or groups. Groups are sets of individual users and cannot be nested but a user may belong to several groups. Besides users and groups, there are two additional ACL entries: *any-other* matches any authenticated user and *unauthenticated* matches any unauthenticated user. A POP is a set of predefined conditions evaluated on attribute-value pairs. POPs impose restrictions on the request or denote information to be passed back to the calling resource manager. For example, a POP may limit access to a specific time period or may instruct the resource manager to enforce a certain audit level.

Objects are denoted by strings whose syntax and structure are similar to absolute filenames in a hierarchical file system. The slash character ('/') is used to delimit, from left to right, hierarchical components of the object's name. Thus, the strings `/Mgmt` and `/Mgmt/Manuals` are examples of object names given in Figure 1. AM's object space is open; i.e., any object without a specific policy is subject to the policy attached to its nearest ancestor in the object hierarchy.

Tivoli Access Manager gathers resources that require protection along with the associated policy into a *domain*. Resources within a domain are represented by objects that span a hierarchical portrayal of its members called *protected object space*. In Fig. 1, the object space is given as a tree. To control access, ACLs, POPs, and authorization rules are attached to objects. If an object does not have an ACL, POP, or ARule explicitly attached then it inherits that policy from an object higher up in the hierarchy. A policy attached to an object defines a *region*; that object is called container object. A region is a subtree whose root is the the container object together with all descendants that do not have a policy attached to themselves. For example, the region defined by *ACL1* consists of the objects `/`, `/Mgmt`, `/Departments` and `/Departments/Docs` (see the dashed polygon in Fig. 1).

An object within a region is accessible if either it is the container object of that region or the user (explicitly or implicitly) has the Traverse right on the container object. An object is accessible if all regions on the path from the root to this object are accessible. Thus, in checking for a primary authorization corresponding to an access query for an object, Traverse authorizations on parent objects must be checked as auxiliary (or secondary) authorizations.

In summary, to determine whether an access request is permitted, Tivoli Access Manager checks whether (1) the ACL of the container node grants the required permission(s), (2) all regions on the path to the root are accessible (the Traverse permission is granted), and (3) evaluates the POP, and (4) the ARule. If any of the ACL, POP, or ARule evaluation fails, the request is denied access. For example, the object `/Departments/Code/Tiger` is controlled by two ACLs and one POP. Access is granted if the region

spawned by ACL *ACL1* is accessible, ACL *ACL2* grants the necessary permission, and the evaluation of POP *POP2* does not fail.

3 XACML Policies

Each XACML policy contains exactly one Policy or PolicySet root XML element. A PolicySet is a container that can hold other Policies or PolicySets, as well as references to policies found in remote locations. A Policy represents a single access control policy, expressed through a set of Rules with Permit or Deny effect. A Policy or PolicySet may hence contain multiple policies or rules, each of which may evaluate to different access control decisions. Therefore, XACML has a collection of Combining Algorithms to reconcile the decisions made by these rules. Each algorithm represents a different way of combining multiple decisions into a single result. There are Policy Combining Algorithms (used by PolicySet) and Rule Combining Algorithms (used by Policy). An example of these is the Deny Overrides algorithm, which says that no matter what, if any evaluation returns Deny, or no evaluation permits, then the final result is also Deny. Thus, this combiner can be regarded as a logical conjunction of permits. There are seven Combining Algorithms to build up increasingly complex policies.

A Target is basically a set of simplified conditions for Subject, Resource and Action. These conditions use boolean functions to compare values found in a request with those included in the Target. If all the conditions of a Target are satisfied, then its associated PolicySet, Policy, or Rule applies to the request.

Once a Policy has been found and verified to apply to a given request, its Rules are evaluated. Rules have an Effect – a value of Permit or Deny that is associated with successful evaluation of the Rule. Rules may also have a condition. If this condition evaluates to true then the Rule’s Effect is returned. Evaluation of a Condition can also result in an error (Indeterminate) or discovery that the Condition doesn’t apply to the request (NotApplicable). Conditions can be quite complex, built from an arbitrary nesting of functions and attributes.

Attributes are named values of known types that may include an issuer identifier or an issue date and time. Specifically, attributes are characteristics of the Subject, Resource, Action, or Environment in which the access request is made. A user’s name, its security clearance, the file the user want to access, and the time of day are all attribute values. When a request is sent from a Policy Enforcement Point (PEP) to a Policy Decision Point (PDP), that request is formed almost exclusively of attributes, and their actual values will be compared to attribute values in a policy to make the access decisions.

In summary, authorizations are expressed in XACML by

access rules which specify the subject, resource and action elements of an authorization. These elements may define the applicable subjects, resources, and actions specifically, or may be wildcard elements, which match all specific elements in the corresponding category. For example, a rule expressing authorization of a user Alice to read a data object “Manuals” can be defined by (using some simplifications in the XML syntax):

```
<Rule Effect="Permit">
  <Target>
    <Subject "Alice">
      <Resource "Manuals">
        <Action "read">
          </Action>
        </Resource>
      </Subject>
    </Target>
  </Rule>
```

Groups of Rules can be combined via logical algorithms into Policies, and groups of policies can be similarly combined into Policy sets. Thus, the overall access control policy data structure in XACML typically comprises multiple Policy sets, each specifying the logical algorithms to be applied by the decision logic to Policies, and, within those Policies, Rules, which apply to particular combinations of subject/resource/action elements. In operation, the decision logic compares the subject/resource/action triple in an access query to the targets in the data structure to identify the applicable policy sets, policies and rules, and then evaluates these accordingly. This evaluation yields a permit or deny decision in response to the access request.

4 Translating Tivoli Access Manager Policy Elements

In Tivoli Access Manager (AM), accessibility of a resource depends on the permissions assigned by the ACL controlling the region of the resource, the reachability of that region, a protected object policy, and an authorization rule. In particular, the AM access decision function grants the request of a user to perform a given action on a given resource if all of the following conditions are fulfilled:

1. the user has the required permissions on the resource as determined by the user’s permissions according to the ACL applying to the resource itself;
2. the user can access the resource’s region, as determined by Traverse permissions in the ACLs applying to the nodes between the resource and the root;
3. all the conditions as expressed in the POP applying to the resource are fulfilled;
4. all the authorization rules applying to the resource evaluate to true.

In the remainder of this section, we explain in detail the structure of the above AM policy elements, how they are interpreted by the AM access decision function, and how they can be expressed in XACML.

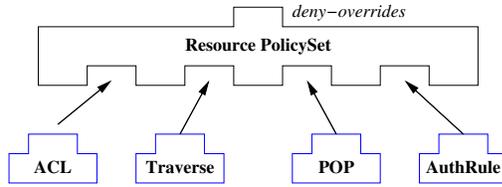


Figure 2. Structure of a Resource policy.

Having an XACML policy for each of the above items, the overall policy for a particular resource can be modeled by an XACML PolicySet that combines the four subpolicies in a `deny-overrides` mode (see Figure 2). This policy set contains a target that matches the particular resource being modeled.

4.1 Access Control Lists

AM grants authorizations to subjects, which are either single *users* or flat *groups* of users. The ACL applying to a resource determines the permissions of a user (authenticated or unauthenticated, with possible group memberships) to perform certain actions on that resource. An ACL has four entry types: *user entries*, *group entries*, and the special entries *any-other* and *unauthenticated*. The *any-other* entry contains default permissions of all authenticated users while the *unauthenticated* entry contains the permissions of unauthenticated users. The ACL shown in Figure 3 gives, for example, user Alice the read (r) and write (w) permission, whereas user Bob has just the read permission. The absence of a permission is indicated by the symbol '-'. Note that the Traverse (T) permission is only assigned to entries *any-other* and *unauthenticated*; its particular importance to control accessibility of nodes in the object space is described in Section 4.2.

```

user Alice      -rw
user Bob        -r-
group Admin     --w
group Physician -r-
any-other      Tr-
unauthenticated T--

```

Figure 3. Content of ACL *ACL3*.

For a given client, determined by its user identifier and a possibly empty set of group identifiers, the set of permis-

sions granted by a specific ACL is determined by performing a sequence of *attempted matches* against ACL entry types. First, it checks whether the user identifier matches one of the ACL’s user entries. If so, it returns the associated set of permissions. Otherwise, if any of the user’s group credentials match any of the ACL’s group entries, the algorithm returns the union of all permissions the user holds via matching group entries. If there is neither a user nor a group match but the user is authenticated, the permissions of the *any-other* entry are returned (or an empty permission set if this entry is absent). If the user is unauthenticated, the algorithm returns the permissions of the *unauthenticated* entry after computing a bitwise “and” operation against the *any-other* entry. The latter operation ensures that unauthenticated users do not have more permissions than those matched by the *any-other* entry. The above evaluation scheme on ACLs is similar to schemes found in Posix or DCE [11].

Note that Bob shall not be able to write (w) even when his credentials contain the group id Admin. Also, an authenticated user (not Alice or Bob) belonging to the group Physician but not to Admin shall not be able to write even if an authenticated user (not Alice or Bob) not belonging to any of these groups can. The latter examples show that there is the possibility to express negative permissions for particular users due to the “pre-emptiveness” of the evaluation algorithm.

ACL Policy. We translate an ACL into an XACML policy with a *first-applicable* algorithm for combining rules. That is, the effect (Deny or Permit) of the first rule target match determines the result of the policy. Further, we generate two rules for each ACL Entry. For the respective subject, the first rule grants the assigned permissions whereas the second rule denies any (other) permission. As each rule shall be evaluated in the order in which it is listed in the policy, the second rule denies access if the subject does not have the required permission in the preceding rule. Whereas the rules for user entries are next to each other, positive group rules precede negative group rules to implement the union semantics of groups. Finally, a catch rule at the end of the rule set assures that access is denied for every requestor not addressed in the ACL, implementing the closed world semantics of Tivoli Access Manager.

To incorporate the special entries *any-other* and *unauthenticated*, we impose the convention that there are two special subjects. If a requestor is unauthenticated it shall be denoted by the user *Unauthenticated*. By definition, every user is member of the group *Any-other* except user *Unauthenticated* who is not member of any group. The structure of a XACML Policy for ACL is given in Figure 4). Criss-cross lines indicate negative rules.

The below XACML policy definition describes the au-

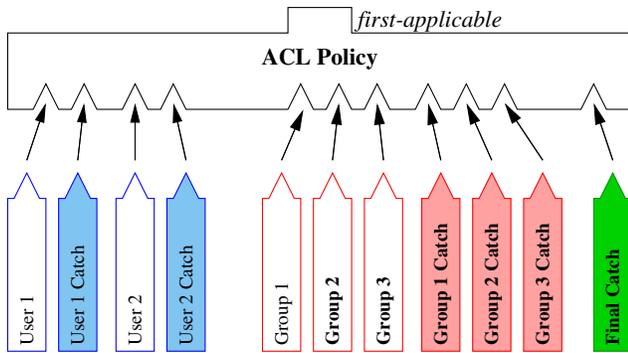


Figure 4. Structure of an ACL.

thorizations given by the ACL of Fig. 3, using a short-hand notation for the XACML Target elements.

```
<Policy
  RuleCombiningAlgId="first-applicable">
  <Rule Effect="Permit"> Alice, (r,w) </Rule>
  <Rule Effect="Deny"> Alice </Rule>
  <Rule Effect="Permit"> Bob,w </Rule>
  <Rule Effect="Deny"> Bob </Rule>
  <Rule Effect="Permit"> Admin,w </Rule>
  <Rule Effect="Permit"> Physician,r </Rule>
  <Rule Effect="Deny"> Admin </Rule>
  <Rule Effect="Deny"> Physician </Rule>
  <Rule Effect="Permit"> Any-other, (T,r) </Rule>
  <Rule Effect="Permit"> Unauthenticated,T </Rule>
  <Rule Effect="Deny"/>
</Policy>
```

The Deny rules for Alice and Bob ensure that, after a successful user id match, no other matches are attempted. All the user rules of course precede all the group rules. Each of the user Deny rules follows its corresponding permit rule. The Deny rules for Admin and Physician ensure that granting a request based on the Any-other rule can occur only if none of the requester's credentials matched a user id or group name. As a request can match multiple group rules and the requester should be granted access if any of his group credentials give the right permission, all the group Permit rules precede all the group Deny rules. The Any-other rule follows any group rule such that permissions based on the any-other ACL entry will only be granted if no specific user or group match occurred. The Unauthenticated rule can be at any position except the last.

Note that if there is an ACL entry without permissions, for example 'user Charles ---', only the negative rule would be generated.

4.2 Traverse Permission

In Tivoli Access Manager, it is not sufficient that the requester holds the necessary permissions on the object *but* the object must also be accessible for the requester. It is the

Traverse ('T') permission that controls who can traverse a particular node to access nodes lower in the tree of the object space. Checking object accessibility thus includes the search for all ACLs attached to ancestors on the path to the root,² and, on each level, to check whether the Traverse permission is granted. Let us call the ACL attached to the node to be the primary ACL and the ACL(s) on the path to be the secondary ACLs.

To illustrate this particular evaluation strategy, let us consider the accessibility of node /Mgmt/Manuals in Figure 1. ACL3 is defined in Figure 3 and the content of ACL1 is given below:

```
user Alice          -rw
group Admin         T-w
group Physician     -r-
any-other           Tr-
```

If we assume that Alice is member of group Physician, Bob is member of group Admin, and Charles is not member of group Physician, then we get following effective permissions:

```
Alice  ---
Bob    -r-
Charles Tr-
```

According to ACLs ACL1 and ACL3, user Alice has sufficient permissions to write on node /Mgmt/Manuals (ACL3) but does not have the Traverse permission on node / (ACL1). Even if Alice would be member of group Admin she cannot access any node below the root because of the preemption after evaluation of her user entry in ACL1. The same holds for unauthenticated users as there is no unauthenticated entry in ACL1.

Traverse Permission PolicySet. Evaluating a request of Alice to read a resource requires the checking of two access rights – whether the primary ACL grants the read action and the secondary ACLs grant the Traverse action. However, XACML does only allow to check for one of the two actions [6, XACML Context Request]. The challenge is therefore to enforce the checking of Traverse permissions in parallel to the evaluation of an access request for another action.

The crucial observation is the fact that it is not necessary to explicitly match against the Traverse permission. It is sufficient to have a check that fails if the requester *does not* hold the Traverse permission in any secondary ACL. Our solution is to shift up the evaluation of secondary ACLs

²The ACL attached to the requested object is not considered.

from rules to policies. Each secondary ACL lists the combination of users and groups that hold the Traverse action. First, we can safely remove all actions in the secondary ACL except Traverse. Next, we translate each ACL into an XACML secondary policy as described in Section 4.1. In the policy, we remove all action elements such that the policy permits any action only if it would permit a Traverse action. Thus, it is sufficient to check whether there is a matching user or group entry, which is implemented by rules with a Subject element only. The set of all secondary ACL policies is aggregated into a policy set with *deny-overrides*.

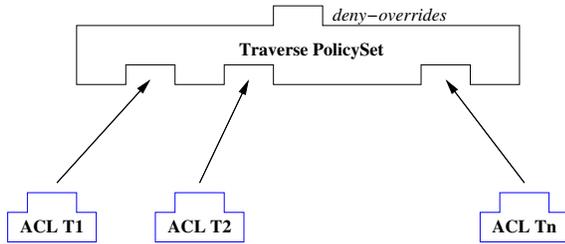


Figure 5. Structure of a Traverse policy.

To access node `/Mgmt/Manuals`, a requester is required to have Traverse permission on node `/`, which is controlled by ACL *ACLI*. Following the above approach, an equivalent check is encoded in the below XACML Policy.

```
<Policy RuleCombiningAlgId="first-applicable">
  <Rule Effect="Deny"> Alice </Rule>
  <Rule Effect="Permit"> Admin </Rule>
  <Rule Effect="Deny"> Physician </Rule>
  <Rule Effect="Permit"> Any-other </Rule>
  <Rule Effect="Deny"/>
</Policy>
```

Note that the structure of the above XACML policy for ACL *ACLI* follows the approach given in Section 4.1 except that if a User or Group Entry holds the Traverse permission then only a Permit rule is created with the Subject as its only element else only a Deny rule is created.

Finally, accessibility of a node is encoded by a PolicySet, where each policy contains the information on subjects with Traverse permissions at a particular node on the path to the root. The number of policies in the policy set is the number of ACLs on the path from the node to the root without the node itself. These policies are combined with the *deny-overrides* algorithm: if one of the nodes cannot be traversed (one of the policies evaluates to a Deny), the requested access is denied.

Figure 5 shows the structure of the Traverse PolicySet, “anding” the results of the policies for the secondary ACLs. In our example, the accessibility to the region of node `/Mgmt/Manuals` is only controlled by ACL *ACLI* attached to the root; therefore, the PolicySet contains only one Policy holding the Traverse information of *ACLI*.

4.3 Protected Object Policies

A *protected object policy* (POP) is a set of predefined attributes whose values either additionally restrict access

- to a specific time period or weekday,
 - to an IP address range or to minimal authentication levels for certain IP addresses,
- or impose additional provisional actions (see also [9]) on
- warning,
 - audit levels,
 - quality of protection: none, integrity, privacy.

A provisional action is passed back to the resource manager³ along with the answer. Conditions imposed by a POP apply to all principals.

Although Tivoli Access Manager provides these POP attributes, it only enforces the warning mode, audit level, and time-of-day access. WebSEAL, a resource manager for Web-based information and included with Tivoli Access Manager, uses POPs to enforce quality of protection⁴ (*qop*) and to restrict certain IP addresses (or IP address ranges) to access any resource in the secure domain (*ipauth*).

POP attribute	Value
audit-level	Deny
ipauth	9.0.0.0, 255.0.0.0, 1
ipauth	anyothernw Forbidden
qop	Integrity
tod-access	mon, tue, wed: 0800-1800
warning	No

Table 1. Attributes of a POP

To illustrate the policies that can be expressed by POPs, we consider an example given in Table 1. The *audit-level* attribute Deny instructs the authorization service to log all unsuccessful requests. The *ipauth* attribute specifies that any request coming from an IP address `9.*.*.*` has to have an authentication level of at least 1. The *qop* attribute demands the use of integrity checking mechanism such as MACs to ensure that the data of the request has not changed. The *tod-access* setting specifies that access can only be granted on Mondays, Tuesdays and Wednesdays between 8 a.m. and 6 p.m. Finally, the value of the *warning* attribute switches off the warning mode that would allow bypassing of the access decisions for testing purposes.

POP PolicySet. To encode POPs in XACML, we observe that each POP attribute is a parameter of (provides input for)

³In the XACML model, it corresponds to a Policy Enforcement Point (PEP)

⁴The required level of data protection, determined by a combination of authentication, integrity, and privacy conditions, when performing an operation on an object.

a predefined condition or provision. For example, the evaluation of the tod-access part of the POP shown in Table 1 includes the evaluation of the following condition:

```
date() IsIn tod-access
```

where function *date()* returns the current date and time. For the above example, this condition would return true if the current time is Mon May 26 14:45:42 CEST 2008.

Attributes *warning*, *tod-access*, and *ipauth* correspond to XACML conditions, and attributes *audit-level* and *qop* correspond to XACML obligations. For reasons of simplicity, we implement each of the POP attributes as a policy and combine them in a PolicySet (representing the collection of POP constraints) using a deny-overrides policy combining algorithm (see Figure 6). This structure also works for policies with obligations because the XACML PDP combines obligations of policies with the same effect [6, Sect. 7.14].

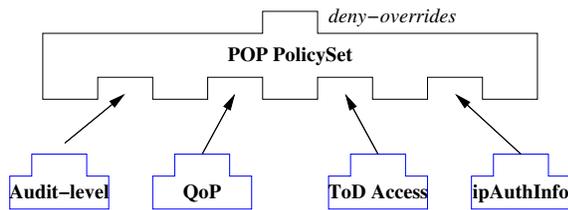


Figure 6. Structure of a POP.

Each policy consists of a Permit rule, which tests the condition, and a Deny rule triggered only if the condition failed. We now show the details of each of the policies. The *TIME.IN.RANGE* policy consists of one Permit rule testing the time condition, and a Deny rule which is triggered only if the condition does not hold. Checking the time interval is easily expressed by XACML built-in functions where current time is evaluated by selecting the *environment:current-time* environment attribute, which must either be present in the decision request or its value must be supplied by the PDP.

```
<Policy PolicyId="POP_TIME_IN_RANGE"
RuleCombiningAlgId="first-applicable">
<Rule Effect="Permit" RuleId="Details_POPrule1">
<Condition>
<Apply FunctionId="time-in-range">
<Apply FunctionId="time-one-and-only">
<EnvironmentAttributeDesignator
AttributeId="current-time"
DataType="time"
MustBePresent="true" />
</Apply>
</Apply>
<AttributeValue DataType="time">
08:00:00
</AttributeValue>
<AttributeValue DataType="time">
18:00:00
</AttributeValue>
</Rule>
</Policy>
```

```
</Apply>
</Condition>
</Rule>
<Rule Effect="Deny"/>
</Policy>
```

In the *DAY.OF.WEEK* policy, there is an environment variable called *day-of-week*, whose value we assume to be present in the XACML Request Context or otherwise supplied by the PDP. Note that checking whether the value of the existing *environment:current-date* environment attribute indicates a particular weekday would either require a complicated XACML condition expression or the definition of a non-standard function.

```
<Policy PolicyId="DAY-OF-WEEK"
RuleCombiningAlgId="first-applicable">
<Rule Effect="Permit" RuleId="Details_POPrule2">
<Condition>
<Apply FunctionId="any-of">
<Function FunctionId="string-equal" />
<Apply FunctionId="string-one-and-only">
<EnvironmentAttributeDesignator
AttributeId="day-of-week"
DataType="string"
MustBePresent="true" />
</Apply>
<Apply FunctionId="string-bag">
<AttributeValue DataType="string">
Mon
</AttributeValue>
<AttributeValue DataType="string">
Tue
</AttributeValue>
<AttributeValue DataType="string">
Wed
</AttributeValue>
</Apply>
</Apply>
</Condition>
</Rule>
<Rule Effect="Deny" />
</Policy>
```

For space limitation, we do not show the XACML condition to encode the POP attribute *ipauth*. IP addresses are represented as bit strings and regular expressions are used to matching addresses. This makes reading cumbersome but allows for finer-grained condition creation on the network address. Catch rules are used as well to handle the case where the network address matches but the authlevel did not satisfy the condition.

POP attributes that represent provisions are encoded as XACML obligations, which are operations that must be performed by the PEP in conjunction with an authorization decision. Obligations may have associated arguments, whose values are interpreted by the PEP. An example policy is given below.

```
<Policy PolicyId="QoP"
RuleCombiningAlgId="first-applicable">
<Rule Effect="Permit" \>
<Obligations>
```

```

<Obligation
  ObligationId="QoP"
  FulfillOn="Permit">
  <AttributeAssignment
    AttributeId="QoP"
    DataType="string" >
    Integrity
  </AttributeAssignment>
</Obligation>
</Obligations>
</Policy>

```

The obligation in the above policy instructs the PEP to use a particular QoP mechanism, namely integrity checks. Audit level and Warning attributes are implemented similarly.

4.4 Authorization Rules

Like POPs, an authorization rule when attached to a protected object imposes conditions that must be met before access is permitted. These conditions are based on data supplied to the authorization engine within the user credential, from the resource manager application, from the encompassing business environment, or from trusted third parties [12, Chapter 10].

In Tivoli Access Manager, the eXtensible Stylesheet Language (XSL) is used to specify rules and XML is the language used for the data that forms input to the rules. For example, the below XSL Transformation (XSLT) template checks whether the sum of the requested amount and of the current credit card balance is lower than the credit card limit and that the requester's member status is '100k'.

```

<xsl:if
  test="(AmountReqd + JohnSmith/CreditCard/Balance)
  &lt; JohnSmith/CreditCard/Limit
  and JohnSmith/MilagePlus/MemberStatus = '100k'">
  !TRUE!
</xsl:if>

```

Authorization PolicySet. The main difference between an authorization rule and a POP attribute is the fact that the latter represents a predefined condition. Thus, authorization rules have the the same XACML policy structure. As XSLT programs are more general than the XACML condition language, the translation must be restricted to the subset of XSLT expressions that can be implemented in XACML. Of course, new functions and data types can always be added to XACML if needed.

The above XSLT contains four variables, `/AmountReqd`, `JohnSmith/CreditCard/Balance`, `JohnSmith/CreditCard/Limit`, and `/JohnSmith/MilagePlus/MemberStatus` whose values must be retrieved for evaluation. In the XACML condition below we assume that they are of types string and integer and are contained in the XACML request context.

```

<Policy RuleCombiningAlgId="first-applicable">
  <Rule Effect="Permit" >
    <Condition>
      <Apply FunctionId="and">
        <Apply FunctionId="integer-less-than">
          <Apply FunctionId="integer-plus">
            <EnvironmentAttributeDesignator
              AttributeId="/AmountReqd" />
            <EnvironmentAttributeDesignator
              AttributeId="JohnSmith/CreditCard/Balance"/>
          </Apply>
          <EnvironmentAttributeDesignator
            AttributeId="JohnSmith/CreditCard/Limit" />
        </Apply>
      <Apply FunctionId="string-equal">
        <EnvironmentAttributeDesignator
          AttributeId="/JohnSmith/MilagePlus/MemberStatus"/>
        <AttributeValue DataType="string">
          100k
        </AttributeValue>
      </Apply>
    </Apply>
  </Condition>
</Rule>
<Rule Effect="Deny"/>
</Policy>

```

If an AM rule evaluation fails, the decision engine returns “access denied” with a reason code, a string defined by the administrator at rule creation. In XACML, the absence of matching attributes in the request context leads to a XACML response context where the Decision element contains the “Indeterminate” value, accompanied with a status code of `missing-attribute`. Because it is recommended that the resource manager reacts on the failure reason we regard both behaviors to be equivalent.

5 Sparse Object Space

Tivoli Access Manager's object space constitutes a set of hierarchical objects. However, not all resources in the “real world” correspond to an object in the object space. ACLs, POPs, and ARules are assigned only to those nodes in the hierarchy where the policy changes. Any object without explicitly attached ACL (POP, ARule) inherits the policy of its nearest object with an explicitly set ACL (POP, ARule). In other words, objects without explicit ACL, POP, or ARule assigned inherit their ACL (POP, ARule) policy from the closest preceding container object in the hierarchy.

Tivoli Access Manager's ACL inheritance can be modeled in XACML by requiring that the PolicySets are ordered with respect to their targets; i.e., PolicySets with longer resource names in the target come first. The Rule combining algorithm `first-applicable` assures that (only) the PolicySet whose resource name constitutes the longest matching prefix w.r.t. the requested resource will be evaluated. Figure 7 illustrates the structure of the generated XACML policy. For each node in that AM PolicySet, the node Resource PolicySet consists of ACL Policy, Traverse

PolicySet, POP PolicySet, and ARules PolicySet as defined in Section 4. If any of these policies fails then access to the corresponding resource should be denied.

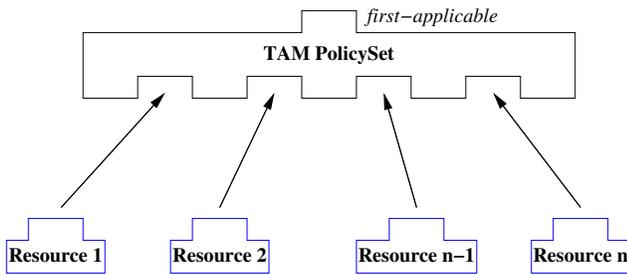


Figure 7. Structure of an AM policy.

ACL assignment is implemented by the target definition of the overall policy. The below XACML target definition matches any object (resource) that is below object /Mgmt/Manuals/.

```
<Target>
  <Resources>
    <Resource>
      <ResourceMatch MatchId="anyURI-regexp-match">
        <AttributeValue DataType="string">
          ^/Mgmt/Manuals/.+$
        </AttributeValue>
        <ResourceAttributeDesignator DataType="anyURI"
          AttributeId="resource-id" />
      </ResourceMatch>
    </Resource>
  </Resources>
</Target>
```

To model Tivoli Access Manager's inheritance of ACLs, POPs, and ARules, we create a hierarchical representation of the object space, pruning all the leaf nodes which do not have an ACL, POP or ARule explicitly attached. Next, we attach to each node of the tree a 'wildcard' child node representing all possible children in addition to its explicit children already defined. We then inherit ACLs throughout this tree; i.e. each of the nodes without an ACL explicitly attached inherits the ACL of its nearest ancestor that has an ACL explicitly attached. We repeat this step also for POPs and ARules. Finally, the nodes in this policy-enhanced tree are then linearized in a depth-first manner such that all children precede their parents, and that a wildcard node always comes after all its siblings. Below we show this linearization for the tree of Figure 1.

```
/Departments/Code/Tiger      ACL2, POP2
^/Departments/Code/Tiger/.+$  ACL2, POP2
/Departments/Code           ACL2, POP1
^/Departments/Code/.+$      ACL2, POP1
/Departments/Docs          ACL1, POP1, ARule1
^/Departments/Docs/.+$     ACL2, POP1, ARule1
/Departments                ACL1, POP1
```

```
^/Departments/.+$          ACL1, POP1
/Mgmt/Manuals              ACL3
^/Mgmt/Manuals/.+$        ACL3
/                          ACL1
^/.+$                      ACL1
```

To determine the access rights on a specific resource, it is now sufficient to locate the first node whose name matches the resource name and to evaluate the corresponding Resource PolicySet. An XACML representation for the full object space can thus be constructed as a 'first-applicable' PolicySet of individual PolicySets each representing one of the resources and ordered by the length of the fully qualified resource name. For example, the resource /Departments/CodeA, a sibling of (the explicit) nodes /Departments/Code and /Departments/Docs would be matched by target ^/Departments/.+\$.

The above mapping follows the Hierarchical Resource Profile of XACML v2.0 [2], which treats nodes in a hierarchical resource as individual resources and makes no assumption about the accessibility of descendent nodes. The profile takes also into consideration that a node's position within the hierarchy may be part of the node's identity, denoted by the separator character '/', and that node identities do not terminate with the '/' character. To state policies for a particular resource, it also recommends a regular match function on URIs. However, using the resource as the only element of policy (set) targets and their specific ordering for a First-applicable evaluation is not given.

6 Implementation

We have implemented the AM to XACML policy translator as a Java program. It uses the AM Java Administration API to extract the AM policy elements from the Tivoli Access Manager policy database. Exploiting the Eclipse Meta Format EMF, we have created a Java XACML library, generated from the XACML schema definition, to programmatically represent the appropriate XACML terms.

The size of the generated XACML policy file reaches double digit megabytes for even moderate AM policies. With XACML v2.0, policy (set) references for ACL, POP, and ARule policies can be used to avoid duplication. When implemented, it drastically reduced the file size by 75%. In retrospect, policy references correspond to the AM concept of templates – definitions for ACLs, POPs, and ARules. Thus, an administrator has to define first a template and then assign the template to resources. A change to the template effects all nodes where the template is attached.

Compared to the time needed for retrieval of the policy information (in the seconds range) from the Tivoli Access

Manager installation, the generation of the XACML policy takes only a small fraction of time.

7 Conclusions

We described a mapping of the IBM Tivoli Access Manager policy language (AM) into XACML. Our effort was motivated by the idea to make XACML support available for a powerful and widely used legacy access control system. To encode the interplay of AM's policy elements and decision logic within XACML, we came up with the use of wildcards to link the check for multiple access rights. To this end, the generated XACML policy clearly reflects the AM policy elements and the associated evaluation logic.

There are some AM functions for which we did not find an equivalent representation in XACML. For example, it is possible to ask for the possession of several permissions in a single AM access request – the read and write permission for example. CORBAsec has a similar feature called 'RequiredRights' [7]. However, XACML allows only to specify a single action in a request context. Of course, there is always the possibility to query for each element and to combine the results but for the price of a big performance penalty.

Besides AM, we implemented a translator from the IBM WebSphere® Portal authorization language PAC to XACML [3]. This makes us confident that many other legacy access control systems could also be supported in XACML by following our approach.

Being able to translate policies of legacy systems to XACML provides many opportunities. For example, sophisticated analysis tools developed for XACML become available. An example is our Separation of Duty analysis tool [5] that works on AM policies generated by the described translation. This work is part of our long-term goal to provide an infrastructure that enables the centralized management of access control policies from heterogeneous computer platforms. This goal requires that modifications in the XACML policies can be transferred back into the legacy system, requiring a mapping in the opposite direction. The challenge is that the XACML normative specifications, however, include not only negative permissions and conditions but also powerful rule and policy combinators. Thus, XACML policies can have a multitude of different forms that make the translation of an arbitrary XACML policy very hard if not impossible even if the target policy language is as expressive.

Acknowledgments

We would like to thank Craig Forster and Michiharu Kudo for their insights into the usage of XACML. Chris Giblin implemented the XACML library.

IBM, the IBM logo, and [ibm.com](http://www.ibm.com) are trademarks or registered trademarks of International Business Machines Corporation in the United States, other countries, or both. If these and other IBM trademarked terms are marked on their first occurrence in this information with a trademark symbol (® or ™), these symbols indicate U.S. registered or common law trademarks owned by IBM at the time this information was published. Such trademarks may also be registered or common law trademarks in other countries. A current list of IBM trademarks is available on the Web at "Copyright and trademark information" at www.ibm.com/legal/copytrade.shtml.

Java is a trademark of Sun Microsystems, Inc. in the United States, other countries, or both. Other company, product, or service names may be trademarks or service marks of others.

References

- [1] A. Anderson. XACML J2SE[TM] Platform Policy Profile. Version: 1.28 Updated: 03/07/21 (yy/mm/dd) research.sun.com/projects/xacml/J2SEPolicyProvider.html
- [2] A. Anderson (Ed.). Hierarchical Resource Profile of XACML v2.0. OASIS Standard, Feb. 2005. docs.oasis-open.org/access_control-hier-profile-2.0-spec-os.pdf,
- [3] S. Burri. PAC to XACML – translating IBM WebSphere Portal Server's access control model to standard model XACML. Semesterarbeit, ETH Zurich, 2007. www.infsec.ethz.ch/people/burrisa/PACtoXACML.pdf.
- [4] K. Fisler, S. Krishnamurthi, L. A. Meyerovich, and M.C. Tschantz. Verification and change impact-analysis of access-control policies. In *27th International Conference on Software Engineering ICSE '05*, pages 196–205. ACM Press, 2005.
- [5] C. Giblin, S. Hada, G. Karjoth, A. Schade, Y. Sodha and E. Van Herreweghen. Separation of Duties and Entitlement Analyzer for Tivoli Access Manager. IBM alphaWorks, 2008. <http://www.alphaworks.ibm.com/tech/sod4tam>
- [6] S. Godik and T. Moses (Eds.). eXtensible Access Control Markup Language (XACML). Version 2.0, OASIS Standard, Feb. 2005.
- [7] G. Karjoth. Authorization in CORBA security. *Journal of Computer Security*, 8(2/3):89–108, 2000.
- [8] G. Karjoth. Access control with IBM Tivoli Access Manager. *ACM Transactions on Information and System Security*, 6(2):232–257, 2003.
- [9] M. Kudo and S. Hada. XML document security based on provisional authorization. In *ACM Conference on Computer and Communications Security*, pages 87–96. ACM Press, 2000.
- [10] A. X. Liu, F. Chen, J. Hwang, and T. Xie. Xengine: A fast and scalable XACML policy evaluation engine. In *Measurement and Modeling of Computer Systems (SIGMETRICS 2008)*. ACM Press, 2008.
- [11] J. Pato. *DCE Access Control Lists (ACL's)*. OSF DCE Specifications, 1990.
- [12] *IBM Tivoli Access Manager – Administrator's Guide*, 2008. Version 6.1. publib.boulder.ibm.com/infocenter/tivihelp/v2r1/topic/com.ibm.itame.doc/am61_admin.pdf.
- [13] The Open Group. Authorization (AZN) API. Open Group Technical Standard C908, Jan. 2000.
- [14] C. Wolter, A. Schaad, and C. Meinel. Deriving XACML policies from business process models. In M. Weske, M.-S. Hacid, and C. Godart, editors, *Web Information Systems Engineering (WISE 2007)*, Lecture Notes in Computer Science #4832, pages 142–153. Springer, 2007.