XACML Policies for Exclusive Resource Usage

Vijayant Dhankhar, Saket Kaushik, and Duminda Wijesekera

Department of Information & Software Engineering
George Mason University
Fairfax, VA 22030, U.S.A
{vdhankha | skaushik | dwijesek}@gmu.edu

Abstract

The *extensible access control markup language (XACML)* is the standard access control policy specification language of the World Wide Web. XACML does not provide exclusive accesses to globally resources, and we do so by enhancing the policy execution framework with locking.

1 Introduction

The *extensible access control markup language (XACML)* [17] is the standard language to specify accesses to resources available on the world wide web. However, XACML normative specifications lack necessary syntax to specify *exclusive access* to resources, and no exforcement framework provides so. Given that new web services can be constructed by composing and orchestrating existing ones using languages such as BPEL [15]), concurrent request for resources on the WWW can occur. For example, updating an XML schema requires exclusive write access. We enahnce XACML syntax and enforcement mechanism using locks.

Perils of not using a synchronization mechanism (such as the *dirty read* [20] in distributed systems) in exclusive accesses are well known. Consequently, we advocate to make a distinction in granting exclusive access and non-exclusive accesses by access controllers. Thus we add appropriate syntax to XACML and an enfofrcement mechanism using locks. Consequently, if and when granted, the access control policy is aware that such permissions are exclusive. This enrichment to XACML has no relationship to application level concurrency control, but not surprisingly, due to the enforced semantic distinction between exclusive and non exclusive acccesse, aids in enforcing *separation of duty* principles [11, 8, 9, 19].

To enforce enhanced XACML policies, we add a lock manager to the policy enforcement module and require that all globally accessible resource register with unique lock manager. In order to ensure starvation avoidence, we assume that resource requesters give up such resource after their usage - although this is beging made policy driven in our ongoing work.

The rest of the paper is organized as follows. Section 2 has related work. Section 3 presents sample Use Cases for exclusive access. Syntactic extensions to XACML appear in Section 4 and Section 5 details the architectural enhancements and some implementation details. Section 8 concludes the paper.

2 Related Work

Motivated by a desire to to introduce trust-based, context-aware access control framework for Web Service invocations, including support for RBAC sessions, Bhatti et al., *et al.* [4, 6, 5, 7] define X-RBAC and X-GTRBAC models for access control frameworks for Web Services, respectively based on RBAC [11] and GTRBAC [13] models of access control. However, they do not provide mechanisms to enforce dynamic separation of duty (DSoD) policies, which exists in current XACML RBAC profile [16]. Cardea by Lepro et al [14] offers a dynamic access control system for the Web, where the *dynamism* means that the request is not bound to local identities at runtime, but instead uses a remote requester's context instead. However, Cardea does not explicitly address concurrent access to exclusively used resources nor dynamic separation of duty policies.

3 Use Cases, Misuse Cases and Requirements

Although some existing work on Web Services orchestration argues the need to lock shared resources [3, 12], to the best of our knowledge they only reserving syntax for locks [3]. Following Use Cases show the need.

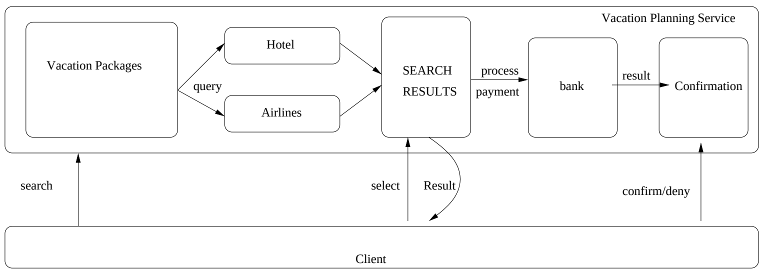

Figure 1: *Vacation Planning Service*

3.1 Use Case 1: exclusive access

Consider an example, *vacation planning service (VPlanner)* that reserves hotel rooms and air tickets for its clients, whose workflow is given in figure 1 [22]. As seen, this interactive service is used to first *searches* for available rooms and air tickets for specified dates and destinations and presents various alternatives to its clients, from which the latter *chooses* alternative for reservations. The service then *initiates* a monetary transfer request to the credit granting agency. On success, the room(s) and air tickets are reserved and aborted otherwise. An efficient implementation should invoke airline and hotel room searches concurrently, while, work-flow dependencies require that monetary transfer request should wait till other parts of the procedure are complete.

Now, suppose two clients are searching for reservations from the VPlanner and are shown the same tickets and hotel rooms. This is potentially dangerous because both can choose the same room or ticket, where simultaneous requests can deadlock two

BPEL server processes. One way of avoiding this situation is to not offer a second client the choices while a precedent client s in the process of reserving a package - thus requiring the VPlanner to lock rooms and tickets during ongoing reservations, referred to as *tentative locking of resources* in the Web Services literature [3].

3.2 Use Case 2: enforcing dynamic constraints

Example 1 (DSoD [8])**.** *Consider a DSoD constraint* an employee cannot invoke role 1 in a session if another role, role 2, is already invoked in some other session. Assuming a data structure maintained by the system *'sessions' with following XML schema:*

```
<user id="ID">
  <sessions>
    <session id="123123">
       <role name="role1"/>
       <role name="role3"/>
    </session> ...
  </sessions>
</user>
```

An abbreviated DSoD XACML policy as follows:

```
1<Rule RuleId="DSoD:role1-role2:requirements" Effect="Deny">
2 <!-- SoD Rule for Example 1 (begin) -->
3 <Target>
4   <Subjects>
5     <AnySubject/>
6   </Subjects>
7   <Resources>
8     <AnyResource/>
9   </Resources>
10  <Actions>
11    <Action>
12      <ActionMatch MatchId="urn:oasis:names:tc:xacml:1.0:function:string-equal">
13        <AttributeValue DataType="http://www.w3.org/2001/XMLSchema#string">activate-role</AttributeValue>
14        <ActionAttributeDesignator DataType="http://www.w3.org/2001/XMLSchema#string" AttributeId="urn:oasis:names
              :tc:xacml:1.0:action:action-id"/>
15      </ActionMatch>
16    </Action>
17  </Actions>
18 </Target>
19 <!-- SOD check -->
20 <Condition FunctionId="urn:oasis:names:tc:xacml:1.0:function:seperationOfDutyCheck">
21  <!-- sessions -->
22  <Apply FunctionId="urn:oasis:names:tc:xacml:1.0:function:getSubjectSessions">
23    <!-- subject-id -->
24    <AttributeSelector RequestContextPath="//Request/Subject/Attribute[1]/AttributeValue/text()" DataType="http:/
          /www.w3.org/2001/XMLSchema#string"/>
25  </Apply>
26  <!-- role-id -->
27  <AttributeSelector RequestContextPath="//Request/Resource/Attribute[2]/AttributeValue/text()" DataType="http://
          www.w3.org/2001/XMLSchema#string"/>
28  <!-- comma delimited set of conflicting roles -->
29  <AttributeValue DataType="http://www.w3.org/2001/XMLSchema#string">role1,role2</AttributeValue>
30 </Condition>
31</Rule>
```

Policy 1: *DSoD policy*

Example 1 above is a DSoD policy expressed in terms of the XACML RBAC profile [16], where as stated in lines 20-30, role 1 and role 2 cannot be co-activated. However, this policy is not currently enforceable because XACML enforcement does not consider concurrent requests. To be fair, the XACML RBAC profile *out sources* the process of enabling roles to the *Role Enablement Authority* module.

3.3 Use Case 3: enforcing history based constraints

This use case enables evaluation of history based constraints in XACML specifying that current access to a resource is contingent upon the history of earlier accesses. A classic example is that of a Chinese Wall policy [8], expressed in XACML in example 2.

Example 2 (Chinese Wall [8])**.** *Consider the following history-based constraint – an employee cannot service a request from company B if (s)he has already serviced a request by company A. We assume the existence of a system resource –* `service-history`*, as the following:*

```
<employee accountId="emp-ID">
  <service-history>
    <service-id="svc-id" >
      <client-id>client-id</client>
      <completed>date</completed>
    </service-id>
  </service-history>
</employee>
```

An abbreviated XACML policy that specifies history based constraint is as follows:

```
 1 <Policy xmlns= ...
 2    PolicyId="ChineseWall:Policy"
 3    RuleCombiningAlgId="&rule-combine;deny-overrides">
 4  <Rule RuleId="CW:COI-A-vs-B:" Effect="Deny">
 5    <Target>
 6      <Subjects>
 7        <Subject>
 8          <SubjectMatch
 9           MatchId="&function;checkHistory">>
10           <AttributeValue
11              DataType="&xml;anyURI">
12              Company-A
13           </AttributeValue>
14           <SubjectAttributeDesignator
15              AttributeId="&history;service-history"
16              DataType="&xml;anyURI"/>
17          </SubjectMatch>
18        </Subject>
19      </Subjects>
20      <Resources>
21        <AnyResource/>
22      </Resources>
23      <Actions>
24        <Action>
25          <ActionMatch
26           MatchId="&function;anyURI-equal">
27           <AttributeValue
28              DataType="&xml;anyURI">
29              &actions;serviceRequest
30           </AttributeValue>
31           <ActionAttributeDesignator
32              AttributeId="&action;action-id"
33              DataType="&xml;anyURI"/>
34          </ActionMatch>
35        </Action>
36      </Actions>
37    </Target>
38  </Rule>
39 </Policy>
```

Policy 2: *Chinese Wall policy*

Example 2 above shows a conflict of interest XACML policy stating that an employee cannot service a request from company B if (s)he has already serviced a request from company A. The policy enforcement mechanism should check the *history of service* (lines 6-19) before authorizing any request involving company B. If the *service history update operation* is interleaved with *service checkHistory* operation, an incorrect authorization may execute. Clearly, this requires that read/write access to history of employee service records be synchronized for correct evaluation of the policy, and is not enforced currently.

Our design enables the following Use cases:

Secure registration of resources: A resource may register itself to a unique lock manager.

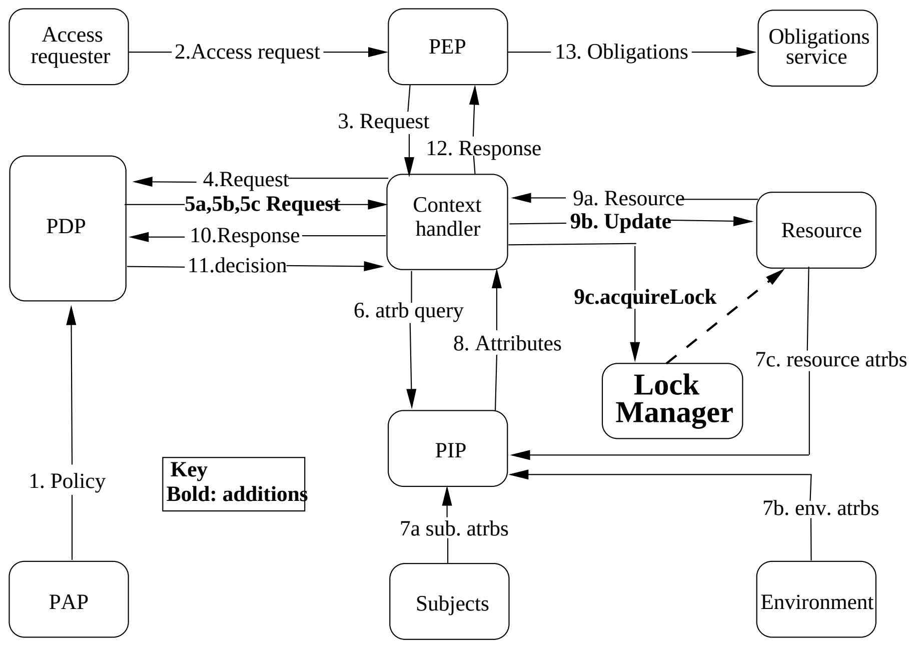

Figure 2: *Extended XACML data flow diagram*

Secure deregistration of resources: Only a resource is able to securely deregister itself from the lock manager.

Exclusive access /relinquish resources: Exclusive use of a resource must be granted to a uniques requester at a time.

3.4 Preventing Misuse Cases

Our design prevents following Misuse Cases:

Registering a resource with multiple lock managers: An exclusively usable resource being registered with multiple lock managers, referred to as *atomic registration*.

Spoofing a resource: Others (de)registering an exclusively accessible resource.

Preventing simultaneous exclusive access: Multiple requesters simultaneous accessing an exclusively usable resource.

4 Enhancing the XACML syntax

Because, our solution for exclusive usage is locking, we enhance XACML syntax for locks. Each of the following elements are specified within `<Rule/>`, `<Policy/>` and `<PolicySet/>` elements of XACML.

– `<PreAction />` specifies a set of locks to be acquired before rule evaluation.

`<AcquireLocks />` specifies a set of locks to be acquired and is a sub element of `<PreAction/>` element, where the `<AcquireLock>` sub element specifies an individual lock.

- `<PostAction />` identifies a set of actions to be performed after (a) rule evaluation leads to a permitted request, (b) rule evaluation leads to a denied request. The set of actions may include releasing locks or updating system resources. For different evaluation results multiple `<PostAction />` elements may be defined.

  `Effect` attribute indicate the *effect* of a post action, as discussed above.

  `<Updates />` specifies updates to be performed in a `<PostAction/>`. `<Update>` sub element specifies an individual change.

  `<ReleaseLocks />` specifies a set of locks to release and is a sub element of `<PostAction/>` element. `<ReleaseLock>` sub element specifies an individual lock.

Introduced elements specify lock acquisition prerequisite for evaluating a rule and post evaluation steps to be taken. The following example policy extends policy 1 with proposed syntactic enhancements.

```
<Rule RuleId="DSoD:role1-role2:requirements" Effect="Deny">
 <PreAction>
  <AquireLocks>
   <AquireLock>
    <!-- sessions -->
    <Apply FunctionId="urn:oasis:names:tc:xacml:1.0:function:getSubjectSessions">
     <!-- subject-id -->
     <AttributeSelector RequestContextPath="//Request/Subject/Attribute[1]/AttributeValue/text()" DataType="
            http://www.w3.org/2001/XMLSchema#string"/>
    </Apply>
   </AquireLock>
   .
   .
   .
  </AquireLocks>
 </PreAction>
    .
    . <!-- DSoD Rule in Example 1 -->
    .
 <PostAction Effect="Permit">
  <Updates>
   <Update>
    <Apply FunctionId="urn:oasis:names:tc:xacml:1.0:function:addRoleToSession">
      <!-- role-id -->
     <AttributeSelector RequestContextPath="//Request/Resource/Attribute[2]/AttributeValue/text()" DataType="
            http://www.w3.org/2001/XMLSchema#string"/>
     <!-- session-id -->
     <AttributeSelector RequestContextPath="//Request/Resource/Attribute[3]/AttributeValue/text()" DataType="
            http://www.w3.org/2001/XMLSchema#string"/>
     <!-- sessions -->
     <Apply FunctionId="urn:oasis:names:tc:xacml:1.0:function:getSubjectSessions">
      <!-- subject-id -->
      <AttributeSelector RequestContextPath="//Request/Subject/Attribute[1]/AttributeValue/text()" DataType="
             http://www.w3.org/2001/XMLSchema#string"/>
     </Apply>
    </Apply>
   </Update>
   .
   .
   .
  </Updates>
  <ReleaseLocks>
   <ReleaseLock>
    <!-- sessions -->
    <Apply FunctionId="urn:oasis:names:tc:xacml:1.0:function:getSubjectSessions">
     <!-- subject-id -->
     <AttributeSelector RequestContextPath="//Request/Subject/Attribute[1]/AttributeValue/text()" DataType="
            http://www.w3.org/2001/XMLSchema#string"/>
    </Apply>
   </ReleaseLock>
   .
   .
   .
  </ReleaseLocks>
 </PostAction>
</Rule>
```

Policy 3: *Enhancements to DSoD policy*

The `PreAction` element in lines (3-12) of policy 3 above states that before evaluating a rule in the DSoD policy, the user session must be locked (lines 5-10). Similarly, `PostAction` element in lines (16-42) requires that after the policy has been evaluated, the locks acquired earlier be released for future concurrency-free changes to user sessions. We assume here that resources are not created during XACML evaluation (they already exist and are registered with a lock manager), however their usage status, *i.e.*, open for read/write, *etc.*, may be modified during a policy evaluation. For example, an XACML evaluation can modify a log file, *etc.*

4.1 Implemented semantics of syntactic extensions

Postaction elements are intended to be evaluated in the following manner.

– Post action can only update resources for locks obtained at the corresponding level or those obtained at the level of the container.

– If two rules within a policy require same lock then they must be acquired and released at Policy level. Such locks are visible within all embedded rules. Similarly, if a lock is acquired at the `Policy Set` level then it is visible to all embedded policies.

– If a resource must be updated in multiple rules, then corresponding lock must be acquired at their container level, *i.e.*, `Policy`.

– Rule evaluation within a `Policy` element must be evaluated by a single thread of execution.

– If locks are required at only the rule level, they must be released at the rule level `<PostAction/>`, otherwise, they must be released at the `<Policy/>` level `<PostAction/>`

4.2 XACML functions

In order to extend the policies with our syntactic extensions, we use the following functions:

function:getSubjectSessions($subject-id as string) Accepts a subject-id as an input and returns a bag of session objects used by this subject. If the subject's sessions have been acquired by PDP through exclusive access, *i.e.*, locked, then the sessions can be cached till the lock to the sessions is released.

function:addRoleToSession($role-id as string, $session-id as string, $sessions as bag) Accepts role-id, session-id and a bag of sessions as input and it adds the passed role to the particular session. Sessions data structure contains all the sessions and session-id is used to locate the relevant session in the current implementation. More efficient implementations are possible.

5 The extended architecture

Figure 2 shows the existing XACML execution model with data flow for policy control [17]. The data flow begins with the Policy Administration Point (PAP) that authors the policies evaluated by the XACML framework, shown in *Flow 1*. Next, access requests (*Flow 2*), initiated by resource requesters, are intercepted by the Policy Enforcement Point (PEP). PEP forwards them, *Flow 3*, to the Context Handler (CH) with optional requester attributes and environmental conditions required for processing. Context handler has following three functions:

– *Flow 4*: Translate access requests into a format understood by the Policy Decision Point (PDP).

– *Flow 10*: Generate the *context* for policy evaluation (response to *Flow 5*), by gathering resource and requester attributes along with the current state of the system from the policy information point (PIP) (*Flow 6,7,8 and 9*), and pass them to the PDP.

– *Flow 12*: Receive policy decisions from the PDP and translate them back to the PEP.

PDP evaluates an XACML policy applicable to the access request and accompanying context. If the evaluation fails, the access is denied, and granted otherwise. This decision is made available to the context handler (*Flow 11*) and relayed to the PEP for enforcement.

Figure 2 shows the *extended* XACML data flow diagram that introduces a *lock manager (LM)* to augment XACML access control decisions. Lock Manager is an entity that grants and revokes locks for accessing resources registered with itself, requiring extra data flows in the extended framework as follows:

Flow 5b: Update System Request (USR) Update system request may be initiated by the PDP to update system resources for setting up an access. For instance, enabling a role may require that user session – a system resource – be updated with the role added.

Flow 5c: Create Lock Request (CRL) This request is initiated by the PDP on behalf of the requesting process – whenever the requesting process asks for exclusive access to an available resource. This is finally refined to acquireLock operation (9c.), where, the lock is *owned* by the requesting process.

Flow 10: Response to PDP queries (overloaded) We reuse the response sent by the context handler to the PDP queries for sending USR and CRL responses in addition to resource query response.

Flow 9b: Resource update, 9b. The PEP upon enforcing the access control decision, updates an internal resource for a log of accesses. This data flow ensures that XACML policies can now support access control decisions based on history of resource usage.

Flow 9c: AcquireLock, 9c. Instructs the LM to invoke a lock on behalf of a requester. Based on the availability of a resource, this operation may succeed or fail.

As in normative XACML specification, here we assume that all attributes have been authenticated (using attribute certificate authenticity) prior to policy evaluation. We discuss the additional complexity due to these addition later in section 5.2. First, we describe the Lock Manager design.

5.1 Lock Manager (LM)

The Lock Manager (see figure 2) is a *privileged process* that, at any given time, has only a single instance running. Lock Manager maintains and creates locks for resources it manages. The functionality of the Lock Manager that provides and maintains locks is summarized below:

Lock Acquisition Simple lock acquisition method is used to obtain simple lock on a resource. This can be used for history-based policies.

Lock Verification Simple lock verification method can be used to verify the validity of the simple lock, that is, the lock is recognized by the lock manager as being owned by the rightful owner. s

Release Lock Release lock method is used to relinquish active locks.

5.1.1 Lock acquisition

Lock acquisition is an atomic operation that cannot be interleaved. That is, testing if a lock is available *and* acquiring it, should both be done in a single atomic operation, akin to the unix `test&set` operation [21]. The LM implements a Service, called *acquireLock*, to achieve this. We associate two strings – `requesterId` and `resourceId` – with the `acquireLock` operation. These strings can be qualified names or URIs of network entities. Lock acquisition requests must be initiated and mediated by XACML policies to ensure that entities with privileges to acquire lock alone will get their requests serviced. The actual call is made by the lock manager within an synchronized critical section (as shown in section 6).

5.1.2 Releasing a lock

Lock revocation is an atomic process implemented as the service *releaseLock*. As before, `releaseLock` is a secure operation (akin to kernel primitive in unix), that is implemented by the LM privileged process. Again, XACML policies are required to ascertain whether a release lock request is valid, *i.e.*, only those release requests that are initiated on the behest of corresponding lock owners are serviced.

5.1.3 Verifying a lock

`verifyLock` operation verifies the validity of a lock presented to it when it is 'invoked' for usage. The execution model states that for every resource update must be preceded by a call to `verifyLock`.

`verifyLock` is invoked by the resource manager for verifying the validity of a lock. This is essential to ensure that resource update request is a valid and the update has been generated by the entity that 'holds' a corresponding lock. The execution

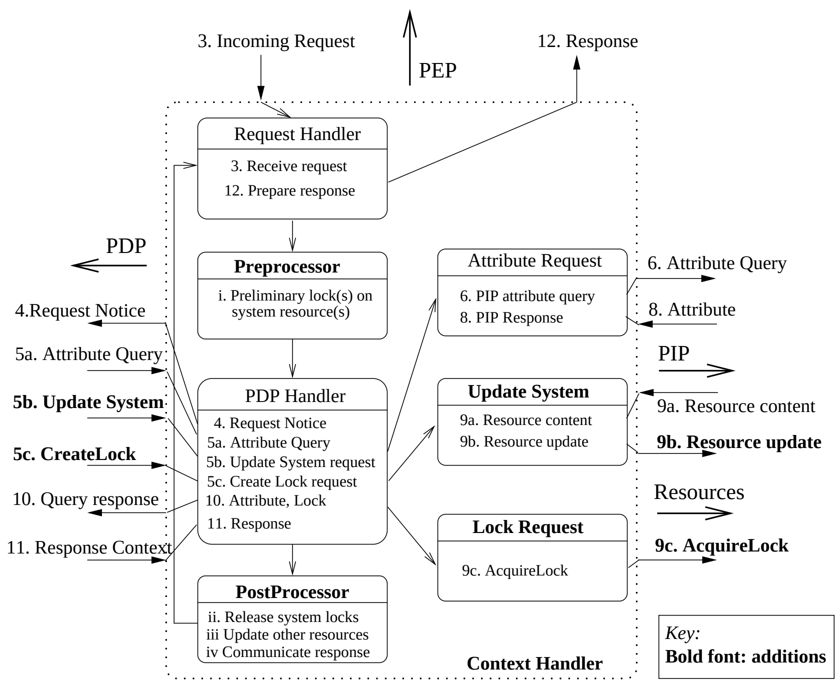

Figure 3: *Context Handler*

model is that a requester gets the lock and at the time of resource usage, it presents the locking permission to the manager of that resource, that in turn verifies the validity of the presenter claim with the lock manager and or PDP.

5.1.4 Registering with a single LM

Registration of a resource with a single lock manager is a basic design requirement. We achieve this by requiring an attribute certificate. That is, each resource is assigned (by a local certificate authority) an X.509 attribute certificate [10] that ties the resource to a *single* Lock Manager. (We assume readers are familiar with X.509 attribute certificates. Additional information on these may be found in RFC 3281). During the registration and deregistration phase, this attribute certificate serves the dual purpose of:

1. Authenticating the resource to the Lock Manager

2. Identifying resource's handling lock manager.

5.2 Extensions to the context handler

In this section we present two extensions required to a context handler to support the additional use cases, namely, resource pool bootstrapping, termination and maintenance, *i.e.*, performing atomic registration, atomic deregistration, *etc.*, and extending the business logic for additional functionality.

5.2.1 Resource pool maintenance

Here we extend the design of a context handler to support lock manager bootstrapping and maintenance of the resource pool. The algorithms in this section are written in pseudo code, with '–>' symbol indicating a call to a sub-module within the context handler.

Secure registration of resources

One of the stated misuses is multiple registration of a resource with one or more lock managers. It is possible that multiple concurrent registration requests are invoked by a resource. To prevent the misuse, we need to ensure that only one among many such requests succeeds. To do so, we introduce the secure registration of resource procedure. Secure registration begins with a registration request by a resource. The input message to this request identifies (a) the resource, and (b) the lock manager, by an attribute certificate. This request is serviced according to the sequence of steps given in figure 3, shown in algorithm 1 below.

```
 1 register(R,C,LM)
 2 Inputs: R(resource), C(certificate), LM(lock manager)
 3 Output: Lock information, Exception
 4
 5 Translate request to XML document
 6 Pass translated XML to PDP Handler
 7 -->Invoke PDP to verify R,C and LM
 8  if (decision == accept)
 9    Lock Request for lock to global.lock
10    -->Acquire lock to global.lock
11    Update System Request for variable
12    -->Create variable lock.R
13      if (lock.R NOT IN global.lock)//
14        Assign lock.R.subjectID = ""
15        Insert lock.R in global.lock
16        Assign message = lock.R
17        Hand request to PostProcessor
18        -->Lock Request for release lock
19           -->Release lock
20           Return request+message to Request Handler
21      else
22        Assign exception = Already Registered
23        Hand Request to PostProcessor
24        -->Lock Request for release lock
25           -->Release lock
26           Return request+exception to Request Handler
27  else
28    Assign exception = Invalid Request
29    Hand the request to PostProcessor
30    -->Return request+exception to Request Handler
```

Algorithm 1: *Secure Registration*

The `register` method accepts three inputs – R, the registering resource; C, its attribute certificate, and LM, the Lock Manager. This request is handed to the 'Request handler' module of CH by the PEP (line 5). The request handler translates the request into XML and hands it to PDP handler for further processing (line 6). The PDP handler invokes the PDP and processes the response (lines 7,8). If the decision is accept, a lock is acquired (lines 9,10) and an update request is fired (line 11). Based on the success of this call, lock is release and the relevant message is returned to the request handler (lines 18-26). If PDP denies the register request, an appropriate response is constructed as well (line 27-30).

Secure deregistration of resources

Similar to secure registration, deregistration of a resource requires locking support. This is because of consistency requirements – a lock resource should not be deregistered while in use. The process flow is as follows:

```
1 deregister(R,C,LM)
2 Inputs: R(resource), C(certificate), LM(lock manager)
3 Output: boolean, Exception
4
5 Translate request to XML document
6 Pass translated XML to PDP Handler
7 -->Invoke PDP to verify R,C and LM
8    if (decision == accept)
9      Lock Request for lock to global.lock
10     -->Acquire lock to global.lock
11     if (acquirelock()== false)
12       Assign exception = In use // no waiting!!
13       Hand request to PostProcessor
14       -->Send request+exception to Request Handler
15     else
16       if(lock.R IN global.lock && lock.R.subjectId="")
17         Update System Request
18         -->Assign global.lock = global.lock -lock.R
19            Assign message = true
20         Hand request to PostProcessor
21         -->Lock Request for release lock
22            -->Release Lock // will succeed
23            Return request+message to Request Handler
24       else
25         Assign exception = Does not exist/In use
26         Hand Request to PostProcessor
27       -->Lock Request for release lock
28          -->Release lock
29          Return request+exception to Request Handler
30   else
31     Assign exception = Invalid Request
32     Hand the request to PostProcessor
33     -->Return request+exception to Request Handler
```

Algorithm 2: *Secure Deregistration*

Similar to the `register` method `deregister` accepts the same three inputs and securely removes the resource from LM. The difference here is that the update request removes the lock from global lock data structure in a critical section (line 18).

5.2.2 Extending business logic

Gaining exclusive access to resources

Once a resource is registered, exclusive access to it can be guaranteed by a process very similar to the above processes, as follows:

```
exclusiveAccess(R,S)
Input: R (resource), S (Subject)
Output: boolean, Exception

Translate request to XML document
Pass translated XML to Pre processor
-->Lock Request for lock to global.lock
   -->Acquire lock to global.lock
   if (acquireLock() == false)
     Assign exception = In use // no wait
     Hand Request to PostProcessor
     -->Send Request+exception to Request Handler
   else
     Hand request to PDP Handler
     -->Invoke PDP for authorizing S to R
     -->Attribute Query for lock.R
        -->Read lock.R
          if(lock.R IN global.lock)
            send lock.R to PDP
          else
            exception = resource unknown
            Hand request to PostProcessor
            -->Return Request+exception // to RH
       if (decision == accept)
         Update System Request
         -->Assign lock.R.subjectId = S
            Assign message = true //
         Hand request to PostProcessor
         -->Lock Request for release lock
           -->Release Lock
           Return request+message to Request Handler
       else
         Assign exception = Invalid Request
         Hand the request to PostProcessor
         -->Return request+exception to Request Handler
```

Algorithm 3: *Exclusive access*

The `exclusiveAccess` method invokes a similar CH workflow for granting access to externally usable resources. The main difference here includes a call to the pre processor module that acquires locks before beginning any PDP evaluation (lines 7-12). In addition, the PDP may query for lock attributes (lines 16-23). Finally, if the request is granted, the lock manager is asked to update `lock.R` to indicate the new owner (line 26).

Use resource: Exercising exclusive access, once a requester has gained such an access to a resource, is a simple call to the resource (through the PEP). It is the resource's responsibility to ensure that the requester owns a valid lock to it. This is done by a *verifyLock* call to the lock manager, of which the details are omitted due to lack of space.

Guaranteeing dynamic/history-based constraints

Steps for enforcing dynamic constraints are as follows:

```
accessResource(R,S)
Inputs: R (resource), S (Subject)
Output: boolean, Exception

Translate request to XML document
Pass translated XML to Pre processor
-->Locate internal resources for the request
   Lock Request for locks to all internal resources
   -->while(more resources)
        { Acquire lock } // locking phase
   if (all locks acquired == false)
     Assign exception = Cannot process // no wait
     Hand Request to PostProcessor
     -->Send Request+exception to Request Handler
   else
     Invoke PDP for authorizing S to R
     Attribute Queries (optionally)
     -->Read attribute
        if(attribute present)
          send attribute to PDP
        else
          exception = resource unknown
          Hand request to PostProcessor
          -->Send Request+exception to Request Handler
     if (decision == accept)
       Update System Request
       -->while(more resources need updation)
            { update ith resource }
          Assign message = true //
       Hand request to PostProcessor
       -->Lock Request for releasing all locks
         -->Release Lock
         Return request+message to Request Handler
     else
       Assign exception = Invalid Request
       Hand the request to PostProcessor
       -->Return request+exception to Request Handler
```

Algorithm 4: *Dynamic constraints*

Release exclusive-use resource A resource requester can *release* a resource for which it holds an exclusive use right. This again is a simple modification of algorithms presented above, of which the details are omitted due to lack of space.

6 Implementation

In this section we briefly present the salient features of our LM implementation. We begin our discussion with LM-data structures to implement locks, followed by a sample Java snippet and WSDL interaction to acquire locks.

```
 1 <xs:element name="Lock"
 2  type="xacml-context:LockType"/>
 3 <xs:complexType name="LockType">
 4   <xs:sequence>
 5     <xs:element name="resourceId" type="xs:string"
 6       minOccurs="1" maxOccurs="1"/>
 7     <xs:element name="ownerId" type="xs:string"
 8       minOccurs="0" maxOccurs="1"/>
 9   </xs:sequence>
10 </xs:complexType>
```

Listing 1: *The Lock Data Structure*

Listing 1 shows the 'LockType' XML Schema (lines 3-10) that represents a lock for a *single* shared resource. The data structure identifies the lock through a `resourceID` and an `ownerId`. An available lock has `ownerId` as an empty string, while a locked resource has a non-empty `ownerId`. Several such locks (one for each shared resource) are stored in a global LM-data structure, called 'GlobalLock', *i.e.*, a *set* of Locks, shown next.

```
<xs:element name="GlobalLock"
 type="xacml-context:GlobalLockType"/>
<xs:complexType name="GlobalLockType">
 <xs:sequence>
   <xs:element ref="xacml-context:GlobalLockEntryType"
     maxOccurs="unbounded"/>
 </xs:sequence>
</xs:complexType>
```

Listing 2: *The Global Lock Data Structure*

```
<xs:complexType name="GlobalLockEntryType">
 <xs:sequence>
   <xs:element name="resource" type="xacml-context:Resource"
     minOccurs="1" maxOccurs="1"/>
   <xs:element name="lock" type="xacml-context:Lock"
     minOccurs="0" maxOccurs="1"/>
 </xs:sequence>
</xs:complexType>
```

Listing 3: *The Global Lock Entry Type*

The 'Global Lock Entry Type' data structure is essentially a means to store key-value pairs, and more specifically, here is used to store a resource and its corresponding lock.

```
<wsdl:message name="aquireLockRequest">
  <wsdl:part name="correlationSet" element=
     "xacml-context:CorrelationSet" />
  <wsdl:part name="lock" element=
     "xacml-context:Lock" />
</wsdl:message>

<wsdl:message name="lockResult">
  <wsdl:part name="correlationSet" element=
     "xacml-context:CorrelationSet" />
  <wsdl:part name="result" element=
     "xacml-context:LockResult" />
</wsdl:message>
```

Listing 4: *Lock acquisition/response message*

Listing 3 shows WSDL message definitions for 'acquireLock' request and the 'result' response.

```
1 public class LockManager {
2
3 // list of all the resources and locks
4 public static Map GlobalLock = new HashMap();
5
6 /**
7 * Aquire a lock
8 */
9 public static LockResult aquireLock (AquireLockRequest request) {
10
11 Lock lock = null;
12 boolean aquireSuccess = false;
13
14 // aquire global lock
15 synchronized (GlobalLock) {
16
17   // if lock already acquired or not registered then fail
18   if (GlobalLock.containsKey(request.getResourceId())) {
19     lock = GlobalLock.get(request.getResourceId());
20     if (lock==null) {
21       // if no locks exist on the resource
22       // then create a new resource
23       lock = new Lock();
24       lock.setOwnerId(request.getActorId());
25       lock.setResourceId(request.getResourceId());
26       GlobalLock.put(request.getResourceId(), lock);
27       aquireSuccess = true;
28     }
29   }
30
31 } // release the global lock
32
33 LockResult result = new LockResult();
34 if (!aquireSuccess) {
35   // couldnt aquire lock
36   result.setStatus("fail");
37 } else {
38   // lock aquired
39   result.setStatus("pass");
40   result.setLock(lock);
41 }
42 return result;
43 }
```

Listing 5: *AcquireLock method*

Listing 5 shows a `Java` implementation of the `AcquireLock` method for acquiring a lock to an existing resource. Java allows *synchronization* through exclusive access to objects that we leverage upon in this implementation (line 17). In Line 16 we acquire exclusive access to the GlobalLock data structure till Line 33. In line 19 we check if the resource is registered with the Lock Manager. Line 21: We check if the lock has already been granted, if not then we create a new Lock with the owner and resource specified in request and add it to GlobalLock (Line 24-27). Finally after updating the GlobalLock we remove our exclusive access to it (Line 33) Line 35: we create a result data structure. Line 36-43: we construct the result for the request accordingly and return it in line 44.

7 Safety and Liveliness

Because we allow *concurrent requests* and locks to serialize access to 'critical sections' of the security monitor, we must ensure that we protect against stated (rather well-known) misuses, requiring us to ensure liveliness and safety properties. In this section, we informally argue that our framework ensures them.

Lemma 1. *Given a resource pool $\mathcal{R}$ and a set of lock managers $\mathcal{L}$, a resource $R_i \in \mathcal{R}$ can only be registered with a single lock manager $L_j \in \mathcal{L}$*

Proof Sketch: Ensured by checking the attribute certificate mapping for R_i before registering it with any lock manager (algorithm 1, line 7). We assume a single certificate authority to issue attribute certificates to all local resources.

Lemma 2. *Given a resource R_i and a lock manager L_K to which R_i can register itself,* `register` *method ensures that R_i can be registered only once with L_K.*

Proof Sketch: `register` implementation (see algorithm 1) acquires a lock to `global.lock` before updating it. This ensures that only a single registration request succeeds.

Lemma 3. *Given a resource R_i and a lock manager L_K to which R_i is registered,* `deregister` *method ensures that only R_i can be deregister itself from L_K.*

Proof Sketch: `deregister` implementation (see algorithm 2) ensures this by checking the attribute certificate (line 7).

Theorem 1 (Safety of the exclusive access:)**.** *Given an XACML policy P for exclusive access to an available resource R and multiple concurrent access requests from subjects $S_i, i \in [1, n]$ (i.e.,* `exclusiveAccess(`R, S_i`)` *), only one request from the above set is authorized by P.*

Proof: From lemma 1, the resource R is registered to a single lock manager, say L_K, and from lemma 2, there is a single lock, say $lock.R$, maintained by L_K (in, say $global.lock$ data structure) corresponding to resource R. Now implementation of `exclusiveAccess` (algorithm 3) ensures – (a) $global.lock$ is locked (line 8) and (b) $lock.R$ is available (empty `lock.R.subject` string) (lines 16-19). The first step ensures only a single access request gains access to $lock.R$ and the second ensures that the resource is available. Finally, (line 26) updates `lock.R.subject` string which precludes any other access request from acquiring the same lock.

Theorem 2 (Safety of dynamic constraints:)**.** *Given an XACML policy P for dynamic constraint for access to a resource and multiple conflicting access requests* (`accessResource(`R, S_i`)`), *then P authorizes only one request.*

Proof: Similar to that of Theorem 1.

Theorem 3 (Liveliness1:)**.** *Given a resource R and an exclusive use access request by subject S* `exclusiveAccess(R,S)` *), then policy evaluation will release locks to all internal resources irrespective of the access control decision.*

Proof Sketch: The proof is straightforward from the implementation of `exclusiveAccess`, because the implementation releases all locks before the method terminates.

Theorem 4 (Liveliness2:)**.** *Given resources x, y and an exclusive use access requests by subject S* `exclusiveAccess(x,S)` *followed by* `exclusiveAccess(y,S)` *) and concurrent exclusive use access requests by subject T* `exclusiveAccess(y,T)` *followed by* `exclusiveAccess(x,T)` *), then at-least one of the exclusive access requests is denied by policy evaluation and all locks acquired for that policy evaluation are released.*

Proof Sketch: The proof is straightforward from the implementation of `exclusiveAccess`, because the implementation releases all locks before the method terminates.

Liveliness2 is achieved by the absence of `while loops` in `acquire lock` method. This ensures that deadlocks for exclusive access don't occur.

8 Conclusion

The current XACML framework has emerged as the default access control specification language and enforcement mechanism for the applications supported by the World Wide Web [1, 2, 18]. But XACML does not currently support three types of access control use cases, *viz.*, ensuring exclusive access to globally available resources, preventing access to a resource given a concurrent conflicting use of another resource (DSoD constraints), and preventing access to a resource given a history of conflicting access (Chinese Wall constraints). In this paper we extend XACML syntax for supporting the above-mentioned use cases. We enhance its framework with a new module, called the *lock manager*, to realize the additional use cases, and also informally argue that safety and liveliness properties are ensured by our implementation.

References

[1] Entrust. http://www.entrust.com/.

[2] Vordel. http://www.vordel.com/.

[3] B. Benatallah, F. Casasti, F. Toumani, and R. Hamadi. Conceptual modeling of web service conversations. Technical Report HPL-2003-60, HP Laboratories Palo Alto, March 2003.

[4] R. Bhatti, E. Bertino, and A. Ghafoor. A trust-based context-aware access control model for web services. In *2nd IEEE International Conference on Web Services (ICWS)*, July 2004.

[5] R. Bhatti, J. B. D. Joshi, E. Bertino, and A. Ghafoor. Access control in dynamic XML-based web services using X-RBAC. In *First International Conference on Web Services (ICWS)*, June 2003.

[6] R. Bhatti, J. B. D. Joshi, E. Bertino, and A. Ghafoor. X-GTRBAC admin: A decentralized administration model for enterprise-wide access control. In *9th ACM Symposium on Access Control Models and Technologies (SACMAT)*, June 2005.

[7] R. Bhatti, J. B. D. Joshi, E. Bertino, and A. Ghafoor. X-GTRBAC:an xml-based policy specification framework and architecture for enterprise-wide access control. *ACM Transactions on Information and System Security (TISSEC)*, 8(2), May 2005.

[8] D. Clark and D. Wilson. A comparison of commercial and military computer security policies. In *IEEE Symposium on Security and Privacy*, pages 184–194, Oakland, April 1987.

[9] D. Clark and D. Wilson. Evolution of a model for computer integrity. In *Eleventh National Computer Security Conference*, Baltimore, October 1988.

[10] S. Farrell and R. Housley. RFC 3281- an internet attribute certificate, April 2002.

[11] D. F. Ferraiolo, R. Sandhu, S. Gavrila, D. R. Kuhn, and R. Chandramouli. Proposed nist standard for role-based access control. *ACM Transactions on Information and System Security*, 4(3):224–274, August 2001.

[12] S. Haddad, P. Moreaux, and S. Rampacek. Client synthesis for Web Services by way of a timed semantics. In *8th International Conference on Enterprise Information Systems*, May 2006. ICEIS 06.

[13] J. B. Joshi, E. Bertino, U. Latif, and A. Ghafoor. A generalized temporal role-based access control model. *IEEE Transaction on Knowledge and Data Engineering*, 17(1), January 2005.

[14] R. Lepro. Cardea: Dynamic access control in distributed systems. Technical Report NAS-03-020, NASA Advanced Supercomputing (NAS) Division, NASA Ames Research Center, Moffet Field, CA, Nov 2003.

[15] OASIS. Business process execution language for web services, May 2003.

[16] OASIS. Core and hierarchical role based access control (rbac) profile of xacml v2.0. http://docs.oasis-open.org/xacml/2.0/access_control-xacml-2.0-rbac-profile1-spec-os.pdf, Feb 2005.

[17] OASIS. Extensible access control markup language, Feb 2005.

[18] RFC 2753. A framework for policy-based admission control.

[19] R. S. Sandhu. A lattice interpretation of the chinese wall policy. In *Proc. 15th NIST-NCSC National Computer Security Conference*, pages 329–339, 1992.

[20] A. S. Tanenbaum and M. v. Steen. *Distributed Systems: Principles and Paradigms*. Prentice Hall, 2002.

[21] A. S. Tannenbaum. *Modern operating systems*. Prentice-Hall Inc., Englewood Cliffs, NJ, 1992.

[22] F. Tartanoglu, V. Issarny, N. Levy, and A. Romanovsky. Dependability in the web service architecture. In *ICSE Workshop on Architecting Dependable Systems*, Orlando, FL, May 2002.