# *XACML Policy Proposal*

The current syntax for policy in XACL is oriented around the 3-tuple {subject, object, action}. The *subject* primitive allows user IDs, groups, and/or role names. The *object* primitive allows granularity as fine as a single element within an XML document. The *action* primitive consists of four kinds of actions: read; write; create; and delete. Additionally, the concept of a provisional action has been included in the XACL work. The *provisional action* primitive allows the specification of provisions to be attached to the access decision (e.g., "Alice is allowed to read the salary field, provided that the access is logged").

This 3-tuple, or triplet, approach to policy language definition is simple and powerful for some environments, but is unfortunately limited – in at least three ways – for other environments. Firstly, the built-in assumption that the object to be protected is (some portion of) an XML document is clearly unsuitable for environments that wish to protect other kinds of documents, data, devices, and so on. The XACML TC should not restrict itself to the notion that the only things that need access control in any organization are XML documents.

Secondly, the matching of a subject to an object can be of limited usefulness; more generally what is of value is the outcome of a comparison between a particular privilege of a subject and a particular property of an object. To use the oft-quoted military example, should the subject with "Secret" clearance be granted access to the file with a "Top Secret" classification label? Similarly, should the subject with "Gold Card" status be granted access to the "Members Only" section of the Web site? The XACL proposal makes a step toward this generality by allowing the subject to be a role name (since a role is simply a named collection of specific privileges), but does not allow the full generality because this mechanism prevents the assignment of privileges directly to users (i.e., assignment can happen only indirectly, through roles). Furthermore, the concept of object properties (sometimes called "sensitivities") is not taken into account. The XACML TC should strive for the generality that will allow its output to be useful in many environments.

Thirdly, explicit inclusion of the action primitive within the policy language inherently limits the use of this language to actions that have been pre-defined by the standard-writers – in particular, read, write, create, and delete. In any real-world environment, however (especially within any given vertical market), there will be many other examples of actions that warrant access protection (e.g., "execute", "initiate", "approve", "send", "sign"). The primary purpose of standardization is interoperability; however, this cannot be achieved if the list of specified actions is incomplete and needs to be extended – in a proprietary fashion – within every environment. A preferred approach is to not specify the action primitive at all. This allows a policy to be written for a particular action (i.e., with a particular action in mind) and then processed by the access control decision function (sometimes called the Policy Decision Point) whenever an attempt is made to perform that action. [Conceptually, an access control policy is written for the "approve" method of the Purchase Order object. When another object tries to invoke that particular method, this policy is processed; a decision is made based upon a comparison of the requester's privileges, the method's sensitivities, and any other relevant variables (e.g., time of day, account balance); finally, the invocation of that method is either granted or denied.] Writing policies that include action primitives is inherently limiting; writing policies that only specify the comparisons to be made, and are linked to action primitives in the implementation environment, is much more general, flexible, and interoperable.

In order to address the above limitations in the XACL policy language, the following language is proposed for consideration by the XACML Technical Committee. The language is essentially identical to the PrivilegePolicy syntax contained in an informative annex in the International Standard X.509, translated into XML (both DTD and Schema versions are included). The syntax

- encompasses the generality of Boolean comparisons between privileges, sensitivities, and other relevant variables,
- leaves the linkage to action primitives to the implementation environment, and
- does not assume that the resource for which access control is desired is an XML document.

It is important to note that the XACL notion of provisional actions may still be utilized with this policy language; it is hoped that this important concept will be retained in the XACML TC.

```
<!ELEMENT privilegePolicy (ppe)>

<!ELEMENT ppe (ppPredicate | and | or | not | ordered)>

<!ELEMENT and (ppe , ppe+)>

<!ELEMENT or (ppe , ppe+)>

<!ELEMENT not (ppe)>

<!ELEMENT ordered (ppe+)>

<!ELEMENT ppPredicate (present | equality | greaterOrEqual |
      lessOrEqual | subsetOf | supersetOf | nonNullSetIntersection)>

<!ELEMENT present (referencedData)>

<!ELEMENT equality (referencedData , secondOperand)>

<!ELEMENT greaterOrEqual (referencedData , secondOperand)>

<!ELEMENT lessOrEqual (referencedData , secondOperand)>

<!ELEMENT subsetOf (referencedData , secondOperand+)>

<!ELEMENT supersetOf (referencedData , secondOperand+)>

<!ELEMENT nonNullSetIntersection (referencedData , secondOperand+)>

<!ELEMENT referencedData (#PCDATA)>

<!ELEMENT secondOperand (referencedData | hardcodedValue)>

<!ELEMENT hardcodedValue (#PCDATA)>
```

```
<?xml version = "1.0" encoding = "UTF-8"?>
<xsd:schema xmlns:xsd = "http://www.w3.org/2000/10/XMLSchema">
      <xsd:element name = "privilegePolicy">
            <xsd:complexType>
                  <xsd:sequence>
                        <xsd:element ref = "ppe"/>
                  </xsd:sequence>
            </xsd:complexType>
      </xsd:element>

      <xsd:element name = "ppe">
            <!-- privilegePolicyExpression -->
            <xsd:complexType>
                  <xsd:choice>
                        <xsd:element ref = "and"/>
                        <xsd:element ref = "or"/>
                        <xsd:element ref = "not"/>
                        <xsd:element ref = "ordered"/>
                        <xsd:element ref = "ppPredicate"/>
                  </xsd:choice>
            </xsd:complexType>
      </xsd:element>

      <xsd:element name = "and">
            <xsd:complexType>
                  <xsd:sequence>
                        <xsd:element ref = "ppe"
                              minOccurs = 2 maxOccurs = "unbounded"/>
                  </xsd:sequence>
            </xsd:complexType>
      </xsd:element>

      <xsd:element name = "or">
            <xsd:complexType>
                  <xsd:sequence>
                        <xsd:element ref = "ppe"
                              minOccurs = 2 maxOccurs = "unbounded"/>
                  </xsd:sequence>
            </xsd:complexType>
      </xsd:element>

      <xsd:element name = "not">
            <xsd:complexType>
                  <xsd:sequence>
                        <xsd:element ref = "ppe"/>
                  </xsd:sequence>
            </xsd:complexType>
      </xsd:element>
```

```
<xsd:element name = "ordered">
      <xsd:complexType>
            <xsd:sequence>
                  <xsd:element ref = "ppe"
                        maxOccurs = "unbounded"/>
            </xsd:sequence>
      </xsd:complexType>
</xsd:element>

<xsd:element name = "ppPredicate">
      <xsd:complexType>
            <xsd:choice>
                  <xsd:element ref = "present"/>
                  <xsd:element ref = "equality"/>
                  <xsd:element ref = "greaterOrEqual"/>
                  <xsd:element ref = "lessOrEqual"/>
                  <xsd:element ref = "subsetOf"/>
                  <xsd:element ref = "supersetOf"/>
                  <xsd:element ref = "nonNullSetIntersection"/>
            </xsd:choice>
      </xsd:complexType>
</xsd:element>

<xsd:element name = "present">
      <xsd:complexType>
            <xsd:sequence>
                  <xsd:element ref = "referencedData"/>
            </xsd:sequence>
      </xsd:complexType>
</xsd:element>

<xsd:element name = "equality">
      <xsd:complexType>
            <xsd:sequence>
                  <xsd:element ref = "referencedData"/>
                  <xsd:element ref = "secondOperand"/>
            </xsd:sequence>
      </xsd:complexType>
</xsd:element>

<xsd:element name = "greaterOrEqual">
      <xsd:complexType>
            <xsd:sequence>
                  <xsd:element ref = "referencedData"/>
                  <xsd:element ref = "secondOperand"/>
            </xsd:sequence>
      </xsd:complexType>
</xsd:element>
```

```
      <xsd:element name = "lessOrEqual">
            <xsd:complexType>
                  <xsd:sequence>
                        <xsd:element ref = "referencedData"/>
                        <xsd:element ref = "secondOperand"/>
                  </xsd:sequence>
            </xsd:complexType>
      </xsd:element>

      <xsd:element name = "subsetOf">
            <xsd:complexType>
                  <xsd:sequence>
                        <xsd:element ref = "referencedData"/>
                        <xsd:element ref = "secondOperand"
                              maxOccurs = "unbounded"/>
                  </xsd:sequence>
            </xsd:complexType>
      </xsd:element>

      <xsd:element name = "supersetOf">
            <xsd:complexType>
                  <xsd:sequence>
                        <xsd:element ref = "referencedData"/>
                        <xsd:element ref = "secondOperand"
                              maxOccurs = "unbounded"/>
                  </xsd:sequence>
            </xsd:complexType>
      </xsd:element>

      <xsd:element name = "nonNullSetIntersection">
            <xsd:complexType>
                  <xsd:sequence>
                        <xsd:element ref = "referencedData"/>
                        <xsd:element ref = "secondOperand"
                              maxOccurs = "unbounded"/>
                  </xsd:sequence>
            </xsd:complexType>
      </xsd:element>

      <xsd:element name = "referencedData" type = "xsd:string"/>
      <!-- an XPath expression, URL, OID, edifact field, etc., -->
      <!-- that points to a specific data value -->

      <xsd:element name = "secondOperand">
            <xsd:complexType>
                  <xsd:choice>
                        <xsd:element ref = "referencedData"/>
                        <xsd:element ref = "hardcodedValue"/>
                  </xsd:choice>
            </xsd:complexType>
      </xsd:element>

      <xsd:element name = "hardcodedValue" type = "xsd:string"/>
      <!-- data value explicitly included in the policy expression -->

</xsd:schema>
```

## *Example*

A concrete example may help to clarify the creation and use of the above **privilegePolicy** construct.

Consider the privilege to approve a salary increase. For simplicity, assume that the policy to be enforced states that only Senior Managers and above can approve increases, and that an approval can only be given for a position lower than your own (e.g., a Director can approve an increase for a Senior Manager, but not for a Vice President). For this example, assume that there are six possible ranks ("TechnicalStaff" = 0, "Manager" = 1, "Senior Manager" = 2, "Director" =3, "Vice President" = 4, "President" = 5).

Assume, furthermore, that the referencedData (the "privilege") corresponding to rank in a SAML attribute assertion is the XML element "Job Title" containing the data *xx*, and that the referencedData (the "sensitivity") corresponding to rank in the XML resource record whose salary field is to be modified is the element "Job Title" containing the data *yy* (these would of course be replaced by real values in an actual implementation). The following Boolean expression denotes the desired "salary approval" policy (codifying this in an XML **privilegePolicy** expression is a relatively straightforward task):

```
and   (
      not   (
            lessOrEqual ( xpath1 , xpath2 )
            )
      subsetOf ( xpath1 , {2, 3, 4, 5} )
      )
```

This policy encoding says that the rank of the approver, *xx*, (found by resolving XPath expression *xpath1*) must be strictly greater than (expressed as "**not lessOrEqual**") the rank of the approvee, *yy*, (found by resolving XPath expression *xpath2*) **and** that the rank of the approver must be one of {Senior Manager, ..., President} in order for this Boolean expression to evaluate to TRUE. The first privilege comparison is "by reference", comparing the values obtained by de-referencing the XPath expressions for "Job Title" for both entities involved. The second privilege comparison uses "hardcodedValue"; here the value corresponding to the XPath expression for the job title of the approver is compared with an explicitly-coded list of values.

The privilege verifier needs a construct that encodes the above policy (i.e., needs an XML document that is valid with respect to the privilegePolicy schema), along with two values – one associated with the approver and one associated with the approvee – that are obtained by de-referencing *xpath1* and *xpath2*. The approver's rank (which would be contained in an attribute assertion) may have the value {3} and the approvee's rank (which may be contained in an XML record) may also have the value {3}. Comparing the approver's rank with the approvee's rank results in a FALSE for the "**not lessOrEqual**" expression, and so the Director is denied the ability to approve a salary increase for another Director. On the other hand, if the approvee's value is {1}, the Director is granted the ability to approve the Manager's increase.

It is not difficult to conceive of useful additions to the above expression. For example, a third component of the **'and'** could be added saying that the environment variable "currentTime", read from the location *URL* must be within a particular span specified explicitly in the expression (i.e., **greaterOrEqual** *aa* **and lessOrEqual** *bb*). Thus, for example, salary updates may be permitted only if the above conditions are satisfied and the request takes place during business hours. Similarly, conditions may be stipulated regarding the location from which the request is made, other privileges of the requester, other sensitivities of the resource being updated, and so on.

This example illustrates the generality and power of what might be called the "comparison" model of policy construction, as opposed to the strict "ACL" model. The ACL model would encode the simple policy above as a set of subject/action/object triples:

| | | |
|---|---|---|
| { "senior manager" | "can approve salary increase for" | "technical staff" } |
| { "senior manager" | "can approve salary increase for" | "manager" } |
| { "director" | "can approve salary increase for" | "technical staff" } |
| { "director" | "can approve salary increase for" | "manager" } |
| { "director" | "can approve salary increase for" | "senior manager" } |
| { "vice president" | "can approve salary increase for" | "technical staff" } |
| { "vice president" | "can approve salary increase for" | "manager" } |
| { "vice president" | "can approve salary increase for" | "senior manager" } |
| { "vice president" | "can approve salary increase for" | "director" } |
| { "president" | "can approve salary increase for" | "technical staff" } |
| { "president" | "can approve salary increase for" | "manager" } |
| { "president" | "can approve salary increase for" | "senior manager" } |
| { "president" | "can approve salary increase for" | "director" } |
| { "president" | "can approve salary increase for" | "vice president" } |

While this may seem manageable for a small number of possible job titles, it is clearly cumbersome for policies that are larger or more complex. The comparison model can greatly simplify such rules, essentially replacing them all with one or two statements (i.e., "*this* must be greater than *that*", "*this* must be within *that* range", and so on).

Also, as noted above, the action "can approve salary increase for" may not fit neatly into the categories "read", "write", "create", and "delete" (at least in the eyes of a given application environment that has defined various actions with terminology that it deems suitable for its purposes). This highlights the value of a policy language that does not explicitly specify action, but instead is linked to the action through external means and processed whenever that particular object method is invoked.