

# XML Document Security based on Provisional Authorization

Michiharu Kudo

IBM Research, Tokyo Research Laboratory  
1623-14 Shimotsuruma Yamato-shi,  
Kanagawa-ken, 242-8502, Japan  
+81-46-215-4642

kudo@jp.ibm.com

Satoshi Hada

IBM Research, Tokyo Research Laboratory  
1623-14 Shimotsuruma Yamato-shi,  
Kanagawa-ken, 242-8502, Japan  
+81-46-215-4281

satoshih@jp.ibm.com

## ABSTRACT

The extensible markup language (XML) is a promising standard for describing semi-structured information and contents on the Internet. When XML comes to be a widespread data encoding format for Web applications, safeguarding the accuracy of the information represented in XML documents will be indispensable. In this paper, we propose a provisional authorization model that provides XML with sophisticated access control mechanism. The well-recognized need for such a system has only recently been addressed. Based on this authorization model, we present an XML access control language (XACL) that integrates security features such as authorization, non-repudiation, confidentiality, and an audit trail for XML documents. We describe our implementation, which can be used as an extension of a Web server for e-Business applications.

## Keywords

Access Control, XML, Provisional Authorization, Security Transcoding

## 1. INTRODUCTION

The eXtensible Markup Language (XML) [1] is a promising standard for describing the structure of information and content on the Internet. Some of the well-recognized benefits of using XML as data container are its simplicity, richness of the data structure, and excellent handling of international characters [2]. However there are often cases where the data must be protected from possible threats since, for example, the data may contain confidential information or it is necessary to prevent repudiation. The goal of our research is to combine practical data representation like XML with various security features.

In this paper, we enhance XML with a sophisticated access control mechanism that enables the client not only to securely browse XML documents but also to securely update each document element. Moreover, we propose an XML access control language (XACL) that integrates security features such as authorization, non-repudiation, confidentiality, and audit trail for XML documents. In other words, our approach extends XML documents in terms of

security. The basic idea for securing XML document security is the notion of provisional authorization that adds an extended semantics to a traditional authorization model, and the notion of security transcoding that protects data from possible threats by transforming the original data into another encoding format.

The World Wide Web Consortium (W3C) is working on XML standards. Standardization activities for XML digital signature [3] and element-wise encryption have just begun but implementations are already available [4]. An authorization mechanism for XML data still remains an open issue.

### 1.1 Related Works

Early authorization models have assumed that the system either authorizes the access request or denies it [5], [6]. Recent work by [7], [8], [9] aims at providing a general framework that is capable of supporting flexible and multiple access control policies such as *sub-subject overrides policy*. All these models, however, assume that the system either authorizes the access request or denies it. The proposed model enables the authorization system to make more flexible authorization decisions by incorporating the notion of provisional actions into traditional authorization semantics.

Damiani, et al. proposed an access control model for XML documents and schema definition [10]. Although their intension in providing XML documents with an access control mechanism is the same as ours, they lay stress more on the semantics for reading XML documents. We deal with not only with the *read* function (“action” as defined below) but also with *write*, *create*, and *delete* capabilities. We also describe an access control specification language defined in XML while they only present a behavioral model, not a language.

PolicyMaker [11] and KeyNote [12] are unified approaches to specifying and interpreting security policies, credentials, and relationships. The access control rules in KeyNote are termed *assertion* that consist of an issuer, subjects to whom trust is delegated, and a predicate which specifies the class of actions delegated, together with conditions under which the delegations applies. The KeyNote system returns a text string, called a compliance value, which are similar to the notion of the provision in this paper. However the model semantics and assumptions are different. First of all, our approach has more comprehensive semantics since we applied the syntax of the traditional access control language that consists of an object, a subject, and an action, while in KeyNote there are no distinct primitives except for subject. Second, in the formalized model in this paper, the semantics of the provision is defined within the authorization model. On the contrary the KeyNote system has no relation with the semantics of the

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

CCS'00, Athens, Greece.

Copyright 2000 ACM 1-58113-203-4/00/0011...\$5.00.

compliance value and only sends it to each application that in turn handles it independently.

CORBA defines several security services for such as authentication and authorization [13], [14]. Their approach is similar to ours in defining a provisional authorization model (as in Section 3 of this article) in the sense that the precise authorization semantics are determined in an implementation, not in the model. However, the access decision is still binary, “yes” or “no”, while we provide an authorization model with the notion of tentative authorization of provisional actions.

Policy Director [15] is designed based on the notion of a centralized authorization framework, which is similar to our objective. Although they provide relatively simple authorization semantics compared with our XACL language, their framework is so extensible that our transcoding system could in theory be combined with their system.

Our proposed provisional actions can be viewed as triggering actions in the sense that they are triggered by access requests. Although they have already been investigated on the context of intrusion detection systems [19] and some Database products (DB2, Oracle, and others), it has not been explored in the access control domain.

## 1.2 Outline of the Paper

This paper is organized as follows. We describe our authorization architecture in Section 2. In Section 3, we present the formal definition of the provisional authorization model that is the primary component of this architecture. Section 4 describes the request execution module and the semantics of the security transcoding by using examples. We briefly describe the specification of XML and XPath in this section. Section 5 presents the syntax and semantics of the XML access control language (XACL). Section 6 addresses our implementation, which can be an extension of a web server for e-Business applications.

## 2. AUTHORIZATION ARCHITECTURE

Figure 1 shows the architecture of our proposed authorization system. First an authorization request is passed on to the provisional authorization module (PAM). Each authorization request involves permission to execute some action on some object with some parameters such as current time. Using authorization information such as object and subject information, the PAM makes a set of authorization decisions based on the authorization policies such as “You may perform the desired access provided you cause a condition to be satisfied,” which has extended semantics compared with the traditional authorization policies. The PAM makes a set of authorization decisions and calls the request execution module (REM). The REM executes object transformation operations according to the authorization decisions. Finally, the REM updates one or more portions of the target XML document and/or returns an authorization result that corresponds to the securely *transcoded* target document.

The authorization architecture assumes the existence of the authentication module that verifies the identities of the users and checks their permitted roles. However the authentication module is beyond the scope of this paper. Existing authentication mechanisms [16], [17] are adequate for the purposes of our system. Once a client has been authenticated and the current role assigned, the client may make authorization requests.

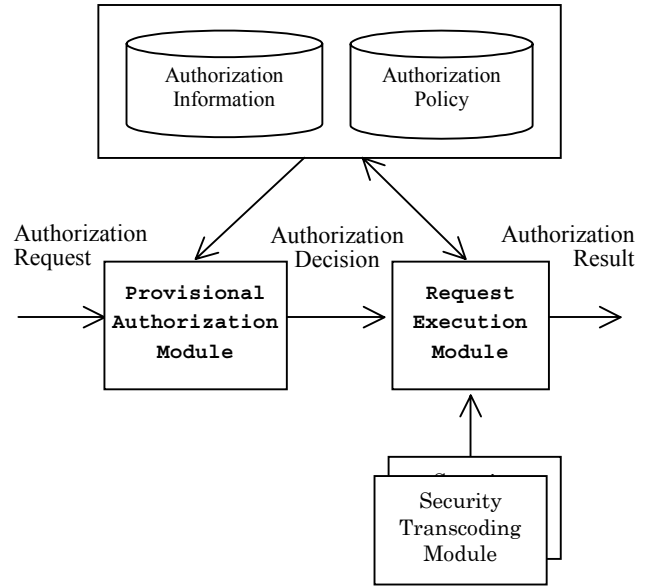


Figure 1. Provisional Authorization Architecture

Basically, this architecture conforms to the Access Control Framework standard [18]. The PAM, the REM, and the authorization policy correspond to the Access Control Decision Function (ADF), Access Control Enforcement Function (AEF), and Access Control Policy Information (ACPI), respectively.

## 3. PROVISIONAL AUTHORIZATION

Almost all studies in access control and authorization systems have assumed the following model: “A user makes an access request of a system in some context, and the system either authorizes the access request or denies it.” Here we propose the notion of a *provisional authorization* which tells the user that his request will be authorized provided he (and/or the system) takes certain security actions such as signing his statement prior to authorization of his request. The following are examples:

- You are allowed to access confidential information, but the access must be logged
- You are allowed to read sensitive information, but you must sign a terms and conditions statement first
- If unauthorized access is detected, a warning message must be sent to an administrator

We formalize the provisional authorization model here by listing the component primitives and by defining the syntax and semantics of the model components. Then we show the differences from the traditional authorization model through a simple example. While we intend to describe the model as generically as possible, several primitives depend on XML-based implementation features such as action and provisional action primitives.

The following elements are component primitives.

- SUBJECT (SBJ): Initiator information such as user ID that is an output from an authentication processor. While any information regarding the subject such as group name and role name can be used in this model, for brevity we only address user IDs and role names.

- OBJECT (OBJ): Resource pointer of target object such as a directory path expression. Since we deal with XML documents in this paper, we use XPath [20] to specify the target objects.
- ACTION (ACT): Action is an access mode permitted on the target object. We define the following generic actions permitted on XML document: *read*, *write*, *create*, and *delete*. The *read* is defined as the subject can read any element-related data such as tag name, attribute type/value pairs, and the contents of the elements. The *write* is defined as allowing subject to update the content of the elements or the values of the attributes. The *create* is defined as allowing the subject to create a new tree structure under the target element. The *delete* action is defined as permitting the subject to delete the target element and any sub-tree below it, or the attribute type/value pairs.
- CONTEXT (CXT): Context is any data such as the time at which the request occurred, the location from which the request was initiated, and any arguments for the specific action.
- PROVISIONAL\_ACTION (PRV): Provisional action is a set of functions used for adding extended semantics on the authorization policy. This primitive has not been introduced in past authorization models. We use the following functions in this paper as a initial set of provisional actions: *log*, *verify*, *encrypt*, *transform*, *write*, *create*, and *delete*. We describe the semantics of each provisional action in the next section.

The following elements are primary component.

- REQUEST (REQ):  $OBJ \times SBJ \times ACT \times CXT$   
REQ refers to a query as to whether or not the subject of SBJ is permitted to perform the action of ACT on the object of OBJ under the context of CXT.
- PERMISSION (PMS): {grant, deny}. PMS is a flag indicating whether access can be granted or denied. The semantics for determining the authorization decision from a set of the permission, the action, and the provisional action is defined in Definition 3.
- FORMULA (FML): A formula expression that returns a binary value, true or false. This expression is evaluated in the Evaluate function that selects authorization rule that fits the authorization request. The formula is defined as logical conjunction of equalities and/or inequalities.

We define an evaluation function using above components.

- Evaluate:  $REQ \Rightarrow PMS \times PRV$   
This function takes the authorization request as input, evaluates authorization rules with the formula expression and returns the access permission of an appropriate authorization rule, along with any required provisional actions. In this paper, we only define the interface instead of defining the specific authorization rule selection algorithm (how to select the appropriate authorization rule through conflict resolution process) because there have been many proposals for such selection algorithms [21], [9]. We will describe our implementation of this function including hierarchy propagation and conflict resolution for the XACL language processor in Section 5.

We define the provisional authorization model based on the above model components.

**Definition 1:** Authorization Policy is a 7-tuple  $\langle obj, sbj, act, pms, prv, cxt, fml \rangle$ . This means that if the access flag *pms* is 'grant' and the *fml* holds, then the initiator *sbj* can perform the *act* on the *obj* under the *cxt*, provided that all of the *prvs* are executed. If the *pms* is 'deny' and the *fml* holds, then the *sbj* cannot perform the *act* on the *obj* under the *cxt*, however all of the *prvs* must still be executed.

**Definition 2:** Authorization Request is 4-tuple  $\langle obj, sbj, act, cxt \rangle$ . This is already defined as the REQ primitive above.

**Definition 3:** Authorization Decision is 6-tuple  $\langle obj, sbj, act, pms, prv, cxt \rangle$ . The *pms* and the *prv* are the output of the Evaluate(*q*) function where *q* is an authorization request. This means if the *pms* is 'grant' and the *prv* is not empty, the *sbj* is permitted to perform the *act* on the *obj* under the *cxt*, provided all of the *prvs* are executed. If the *pms* is 'deny' and the *prv* is not empty, the *sbj* is denied to perform the *act* on the *obj* with the *cxt* but all of the *prvs* must still be executed.

Consider we have two authorization policies:

$\langle t\_and\_c, Manager, read, grant, \_, \_, \_ \rangle$   
 $\langle contractor, RegisteredClient, read, deny, log, today\ is\ holiday, \_ \rangle$

The first rule says that *Manager* can read element *t\_and\_c* and the second says that if today is holiday, the *Registered Client* cannot read the *contractor* element but the access must be logged. When the authorization request of  $\langle t\_and\_c, Manager, read, \_ \rangle$  is submitted, the PAM returns  $\langle t\_and\_c, Manager, read, grant, \_, \_ \rangle$  that says the *Manager* is permitted to read element *t\_and\_c* if the Evaluate function selects the first rule as an appropriate authorization policy. When the authorization request of  $\langle contractor, Registered\ Client, read, \_ \rangle$  is submitted and if the time at which the request occurred is a holiday, the PAM returns  $\langle contractor, Registered\ Client, read, log, \_, \_ \rangle$  that says the *Registered Client* is not allowed to read the *contractor* element but the access must be logged for auditing purposes if the Evaluate function selects the second rule as an appropriate policy. The latter example shows that PAM works differently from the traditional authorization model. However, note that the PAM is identical with traditional authorization in cases without provisional actions.

## 4. SECURITY TRANSCODING

In this section, we describe the REM module that protects target data by transforming the original data into a secure representation. For example, if an authorization decision generated by the PAM says that the initiator is permitted to read a contract document except for the identity of the contractor, the REM generates a contract document without contractor's identity field. Another example is that the REM transforms a content of the element from plain-text format to cipher-text by applying some cryptographic operation and stores it in the authorization information repository in order to satisfy confidentiality. We call these transformational operations *security transcoding*. Basically, the notion of the security transcoding implies two types of transformations: the secure data retrieval from the repository; and the secure data modification in the repository. Table 1 shows the semantics of the actions that are used for handling XML documents. The *read*

action corresponds to the former transformation that generates an authorization result data structure, which only contains permitted target object information. The semantics of the *write*, *create*, and *delete* actions corresponds to the latter. For example, the *write* action updates the associated target object stored in the authorization information repository. These do not generate any authorization result data structure.

**Table 1. Semantics of the actions**

Action	Semantics
read	Generate an authorization result that contains the associated target object (in case of XML, the tag, text content, and attribute type/value pair) retrieved from the authorization information
write	Update the associated target object (text content or attribute value), stored in the authorization information, with the parameter specified in the authorization request
create	Create a new data structure (element structure) under the associated target object (tag) stored in the authorization information
delete	Delete the associated target object (element structure) from the authorization information

Table 2 shows the semantics of the provisional actions that are used for handling XML documents. The *log* provision invokes the log transcoding module that appends an authorization decision to the associated status section. The *verify* and *encrypt* provision invokes cryptographic utilities such as XML digital signature and element-wise encryption module. We assume that the algorithm name and key information are specified in the authorization policy or the parameter from the authorization request. The *transform* provision is used with the read action and it transforms the authorization result from one format (e.g. XML) to another format (e.g. HTML). XSLT [22] is a typical module for the transform provision. The *write*, *create*, and *delete* provisions have the same semantics with those of the actions.

**Table 2. Semantics of the provisional actions**

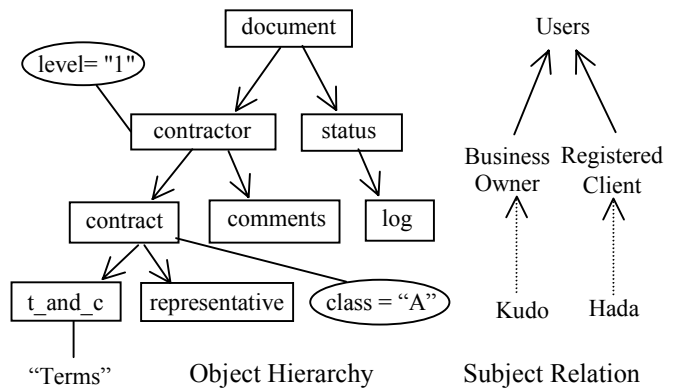
Provisional Action	Semantics
log	Log the authorization decision
verify	Verify a digitally signed parameter specified in the authorization request. Return true if it is verified successfully.
encrypt	Encrypt the parameter specified in the authorization request.
transform	Transform the authorization result according to the transformation parameter.
write, create, delete	The same semantics as specified in Table 1

Note that the instances of the security transcoding modules are not limited to those described here but extensible by plugging application-specific modules into the REM.

We describe the semantics of each security transcoding by using an online contracting scenario as an example application. Consider that there is an electronic contract document represented in XML and there are two roles: business owner who offers business to clients; and the registered client who makes the contract with the

business owner. Figure 2 illustrates the target XML contract document and the subject relation, which are stored in the authorization information repository. In the following examples, for brevity we use an element name for referring the target object.

For background, we briefly describe the specification of XML and XPath here. An XML document has a rooted tree structure. An element that is specified by a tag  $\langle \dots \rangle$  is called a node. Each element can have attributes that consist of types and the associated values, and can have some content. In Figure 2, a rectangle indicates an element and an ellipse indicates an attribute type/value pair. The text string at the leaf indicates the contents of the element. Thus “contract” is the element name, “class” is the attribute type and “A” is the attribute value both attached to the “contract” element, and “Terms” indicates the text content of the “t\_and\_c” element. The structure of an XML document is defined by a Document Type Definition (DTD) that uses a notation similar to BNF. XPath represents path expression in a document tree by a sequence of element names or predefined functions separated by the “/” character. For example,  $\text{/document/contractor/contract/t\_and\_c}$  denotes the t\_and\_c element that is a child of the contract element that is a child of the contractor element.



**Figure 2. Object and subject hierarchy**

#### 4.1 Read Transcoding

Consider the following set of authorization decisions coming from the PAM. We assume that these are generated in response to the authorization request of  $\langle \text{document}, \text{Registered Client}, \text{read}, \_ \rangle$  with the authorization policy of  $\langle \text{contract}, \text{Registered Client}, \text{read}, \text{grant}, \_ \rangle$  and a default-policy of denial.

1.  $\langle \text{document}, \text{Registered Client}, \text{read}, \text{deny}, \_ \rangle$
2.  $\langle \text{contractor}, \text{Registered Client}, \text{read}, \text{deny}, \_ \rangle$
3.  $\langle \text{status}, \text{Registered Client}, \text{read}, \text{deny}, \_ \rangle$
4.  $\langle \text{log}, \text{Registered Client}, \text{read}, \text{deny}, \_ \rangle$
5.  $\langle \text{comments}, \text{Registered Client}, \text{read}, \text{deny}, \_ \rangle$
6.  $\langle \text{contract}, \text{Registered Client}, \text{read}, \text{grant}, \_ \rangle$
7.  $\langle \text{t\_and\_c}, \text{Registered Client}, \text{read}, \text{grant}, \_ \rangle$
8.  $\langle \text{requester}, \text{Registered Client}, \text{read}, \text{grant}, \_ \rangle$

The first authorization decision says that *Registered Client* is not allowed to read the *document* element, while the sixth says that *Registered Client* is permitted to read the *contract* element.

The read transcoding module receives the above authorization decisions and generates corresponding set of elements according to the type of permissions. If the permission is *grant*, it generates an element with any related information such as attribute type/value pair and the text string. If the permission is *deny*, it generates only an element name. As a result, tree structure below the *status* and *comments* elements and the attribute type/value pair of “*level = 1*” are not generated in the authorization result data structure. Figure 3 illustrates the authorization result for the initiator.

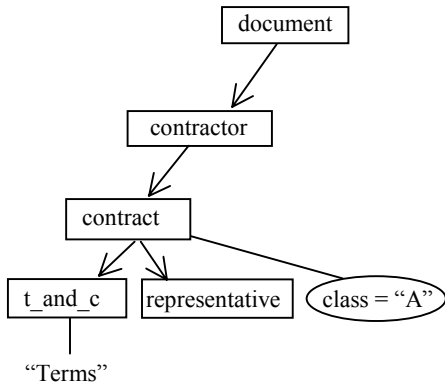


Figure 3. Authorization result of permitted objects

### 4.2 Update and Log Transcoding

Consider the following authorization decision.

*<t\_and\_c, Business Owner, write, grant, log, “Purchase of \$1M over one year”>*

This says that *Business Owner* is allowed to update the *t\_and\_c* element with the value of “*Purchase of \$1M over one year.*” provided the access is logged. We assume that this is generated in response to the authorization request of *<t\_and\_c, Business Owner, write, “Purchase of \$1M over one year”>* with the authorization policy of *<t\_and\_c, Business Owner, write, grant,*

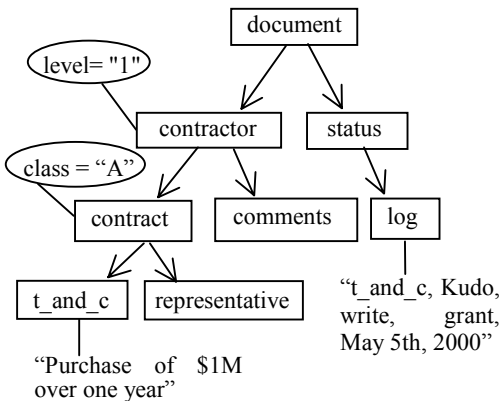


Figure 4. Updated XML document

*log, \_, the comments field is empty>*. First, the *log* transcoding module adds a new log entry under the *status* element of the target XML document. If it succeeds, the *write* transcoding module updates the *t\_and\_c* element. The sequence of calling transcoding modules conforms to the semantics of the provisional

authorization defined in the previous section. Figure 4 illustrates the target XML document.

### 4.3 Encryption Transcoding

Consider the following authorization decision.

*<t\_and\_c, guest, read, grant, encrypt, key1234>*

This says that *guest* user is allowed to read the *t\_and\_c* element, provided the value is encrypted with the cryptographic key of *key1234*. This means that the guest who knows the value of the *key1234* can understand the *t\_and\_c* element. We assume that this is generated in response to the authorization request of *<t\_and\_c, guest, read, >* with the authorization policy of *<t\_and\_c, guest, read, grant, encrypt, key1234>*. First the *encrypt* transcoding module is called and encrypt the *t\_and\_c*. Next the *read* transcoding module generates a corresponding element and its content as an authorization result. Figure 5 illustrates the final authorization result.

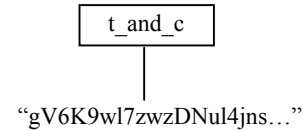


Figure 5. Authorization result of encrypted message

### 4.4 Signature Verification Transcoding

We think that it is reasonable that the contractor is required to put his/her signature on the statements if s/he makes the contract. Here we assume that the statement and the signature value are sent from the client as the context parameters. Consider the following authorization decision.

*<comments, Registered Client, write, grant, log&verify, “We accept the contract”>*

This says that *Registered Client* is allowed to write “*We accept the contract*” in the *comments* element, provided the value is logged then and successfully verified using the signer’s public key. First the *log* transcoding module is called and next the *signature verification* module is called to verify the client’s signature. If both succeed, the *write* transcoding module updates the *comments* element with the verified statement of “*We accept the contract.*” Figure 6 illustrates the updated target XML object.

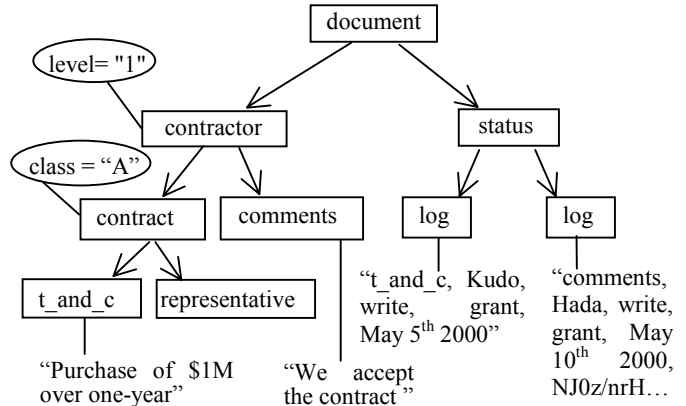


Figure 6. Final state of XML document

## 5. XACL

XACL is an access control language based on the provisional authorization model described in Section 3. In this section, we specify the syntax and semantics of XACL. Specifically, we define an element with an identifier `<policy>`. The primary purpose is to enable security policy programmers to write flexible authorization policies in XML. Another advantage of using the XML format for policy specification is that the XACL language can easily implement the policy management authorization rules. In other words, authorization rules on the authorization policy itself can be defined by meta-rules also described in XACL.

In this paper, we do not specify how to bind a set of policies written in XACL with target documents. There are two fundamental approaches. One is at the schema definition (e.g. DTD) level and the other is at the level of each specific document. In the former approach, a set of policies is bound to all documents valid according to a DTD. Therefore, one needs to maintain the mapping between a particular DTD and the associated policy. In the latter, a policy is bound to each specific document. In this case, an associated policy, which is encoded as a `<policy>` element, may be an element contained within the target document. While the XACL can be applied to both approaches, the example in the Appendix shows the latter case.

### 5.1 Syntax

Figure 7 shows a simple authorization policy that says the authenticated user Alice, who is assigned an Employee role, has write privilege on contractor elements below the document, provided the access is logged.

```
<policy>
  <xacl>
    <object href="/document/contractor"/>
      <rule><acl>
        <subject><uid>Alice</uid>
        <roles><role>Employee</role></roles></subject>
        <action name="write" permission="grant">
          <provisional_action timing="before" name="log"/>
        </action>
      </acl></rule>
    </xacl>
  </policy>
```

Figure 7. Authorization Policy specified in XACL

A subject is specified by a triple: uid, role-set and group-set, where role-set and group-set are a set of role names and group names, respectively. Semantically, it represents a user who has the specified identity, performs all specified roles and belongs to all specified groups. Both roles and groups have a hierarchical structure. When a user belongs to a group, he is a member of all its super groups. The same is true of roles. Therefore, all authorizations of a group and a role propagate to all its subgroups and sub roles, respectively.

```
<!ELEMENT subject (uid?,role*,group*)>
<!ELEMENT uid (#PCDATA)>
<!ELEMENT role (#PCDATA)>
<!ELEMENT group (#PCDATA)>
```

### 5.1.2 Objects

An object represents an element or a set of elements in a target XML document. We identify it by a single XPath expression [20]. The "href" attribute is used to specify it. Objects have an element-based hierarchical structure. This means that in a tree structure of XML documents, all nodes except for the element nodes are ignored with respect to hierarchy. In other words, authorizations specified for an element are intended to be applicable to all its nodes (attributes, texts, etc) but not to any sub-elements. However, policies can propagate on this hierarchy.

```
<!ELEMENT object EMPTY>
<!ATTLIST object href CDATA #REQUIRED>
```

### 5.1.3 Actions

An action is specified as an `<action>` element. XACL supports both grants and denials. The "permission" attribute is used to indicate whether it is a grant or a denial. The "name" attribute is used to specify the name of the action: read, write, create or delete (See Table 2).

We can specify provisional actions associated with an action. The "name" attribute is used to specify the name of a provisional action and the "timing" attribute is used to specify the timing constraints on when the provisional action is executed, i.e., before or after the specified action is executed. A provisional action may succeed or fail. Provisional actions may take zero or more input parameters as predicates and functions (See the next section.) Provisional actions are application-specific.

```
<!ELEMENT action (parameter*,provision*)>
<!ATTLIST action name (read|write|create|delete) #REQUIRED
               permission (grant|deny) #REQUIRED>
<!ELEMENT provisional_action (parameter*)>
<!ATTLIST provisional_action name CDATA #REQUIRED
                             timing (before|after) "after">
```

### 5.1.4 Conditions

A condition corresponds to FMT in the PAM, but we do not specify the details here. Refer to the example in Appendix.

```
<!ELEMENT condition ANY>
```

### 5.1.5 Policy

A policy consists of multiple XACLs, an `<xacl>` element consists of multiple `<rule>` elements and a `<rule>` consists of multiple `<acl>` elements. The target object is specified for each `<xacl>` element. This means you can specify multiple rules and acls for the same target element.

In an `<acl>` element, zero or more subjects, one or more actions and an optional condition are specified. An `<acl>` says that all specified subjects are to be authorized to execute all specified actions (although they may be associated with provisional actions) on all specified sub-elements under `<object>` elements if

```

<!ELEMENT policy (property?, xacl)*>
<!ELEMENT xacl (object+,rule+)>
<!ELEMENT rule (acl)+>
<!ELEMENT acl (subject*, privilege+, condition?)>
<!ELEMENT property (propagation?, conflict-resolution?,
default?)>
<!ELEMENT propagation EMPTY>
<ATTLIST propagation read (no|up|down) "down"
write (no|up|down) "down"
create (no|up|down) "down"
delete (no|up|down) "up">
<!ELEMENT conflict-resolution EMPTY>
<ATTLIST conflict-resolution read (dtp|ptp|ntp) "dtp"
write (dtp|ptp|ntp) "dtp"
create (dtp|ptp|ntp) "dtp"
delete (dtp|ptp|ntp) "dtp">
<!ELEMENT default EMPTY>
<ATTLIST default read (grant|denial) "denial"
write (grant|denial) "denial"
create (grant|denial) "denial"
delete (grant|denial) "denial">

```

the specified condition is satisfied. A policy also may have a single <property> element. The property element is used to specify propagation, conflict resolution and default policies. As described before, objects have an element-based hierarchical structure. A policy may propagate to them. The <propagation> element is used to specify a propagation policy for each action. We have three types of propagation policies and Table 3 shows their semantics.

**Table 3. The semantics of hierarchy propagation policy**

Type	Semantics
no	No propagation: Authorizations are not propagated.
up	Propagation up: An authorization for an element is propagated to all its parent elements.
down	Propagation down: An authorization of an element is propagated to all its sub-elements.

Due to the fact that XACL allows us to specify both grants and denials, it is the possibility that some access is simultaneously authorized and denied. The <conflict-resolution> element is used to specify three types of conflict resolution policies for each action. Table 4 describes the semantics of conflict resolution supported in the XACL language.

When there is no authorization rule (grant or denial) for an authorization request or when the conflict resolution policy "ntp" is enforced, we need to make a decision according to the specified default policies. We can specify it in the <default> element for each action. When any one of the <propagation>, <conflict-resolution> or <default> elements are omitted or if the

<property> element is omitted, the default policies specified in the DTD are applied for each action.

**Table 4. The semantics of conflict resolution policy**

Name	Semantics
dtp	Denials take precedence.
ptp	Permissions take precedence.
ntp	Nothing takes precedence. We neither authorize nor deny an access when there is a conflict. Instead, we will make a decision by referring to the default policies.

### 5.1.6 Policy Evaluation

We provide a policy evaluation algorithm that takes as input an authorization request and outputs an authorization decision including provisional actions.

The following is a basic matching algorithm. This algorithm doesn't deal with propagation and conflict resolution, but simply finds all matching ACLs.

**Input:** An authorization request

**Output:** A decision list, which may contain multiple decisions

**Step 1 [Object-Check]** Search the associated policy for <xacl> descriptions such that a set of elements expressed by its <object> element contains the object specified in the authorization request.

**Step 2 [Subject-Check]** For each <xacl>, check if the subject of the requester and the requested action is semantically equal to a corresponding requester and action as specified in the <xacl>. In particular, the requester must belong to all specified groups and perform all the specified roles. If not, ignore it and remove the <xacl> from further consideration.

**Step 3 [Condition-Check]** For each of the remaining <xacl> elements, check if the specified condition holds. If not, ignore it and remove the <xacl> from further consideration.

**Step 4 [Decision-Record]** Make a decision for each of the remaining <xacl> elements, where a decision includes the requested object, the subject of the requester and the action specified in the <xacl>, and append all decisions to the authorization decision list.

The following is the policy evaluation algorithm using the above basic matching algorithm. In this algorithm, both propagation and conflict-resolution are dealt with. We note that this algorithm outputs exactly one authorization decision.

**Input:** An authorization request

**Output:** A decision list with only one decision

**Step 1: Propagation Processing**

**1-1** Call the Matching algorithm for the input authorization request.

**1-2** If the propagation policy for the requested action is "no", go to Step 2.

**1-3** If it is "down", create an empty decision list and an authorization request for the parent element, and recursively call this algorithm for these inputs. Append the resulting decision to the original decision list as decisions for the original target element.

1-4 If it is "up", for each of all its sub elements, create an empty decision list and an authorization request, and recursively call this algorithm for these inputs. Append each of the resulting decisions to the original decision list as decisions for the original target element.

**Step 2: Conflict Resolution**

2-1 If there is no authorization decision in the list, make a decision according to the default policy for the requested action and append it to the list. In this case, there is no provisional action associated with the decision.

2-2 If there is a conflict, resolve it according to the resolution policy for the requested action. If the conflict resolution "ntp" is applied, we make a decision according to the default policy for the requested action. In this case, there is no provisional action associated with the decision.

**Step 3: Select only one decision**

3-1 At this point, the list contains at least one decision (obtained via the propagation), but may contain more than one decisions.

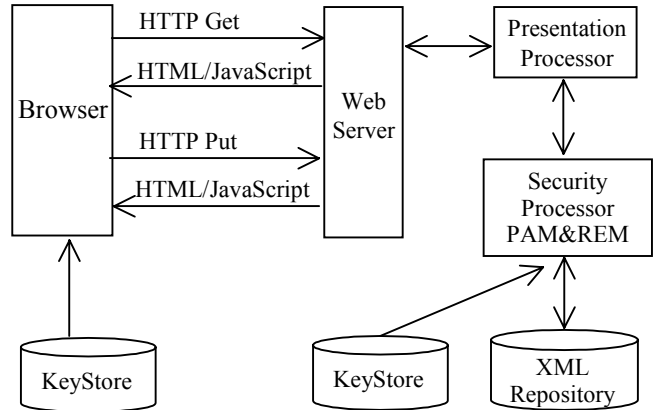
3-2 If there is more than one decisions, eliminate any decision that was obtained via propagation (up or down). If the list still contains more than one decisions, select one from them in an arbitrary way.

Appendix shows the complete set of target XML data, authorization policies, and access logs based on the online-contracting scenario described in the section 4. It also contains concrete signature parameters that conform to the XML digital signature standard [3].

**6. WEB APPLICATIONS**

We have implemented the PAM and the REM described in the previous sections. Figure 8 shows how the security processors are integrated in the web application environment. Since a Web application is assumed to be using the HTTP protocol, the client's browser initiates the communication and the server responds to it. The sequence for events for requesting a web page and receiving the response is described below:

1. The browser requests a Web page from the Web server. After the client is authenticated, the HTTP parameters containing the XML file name and the target element name are sent to the presentation processor that constructs an authorization request for the PAM. Then the REM generates securely the transcoded authorization result. The presentation processor generates dynamic HTML and/or a JavaScript Web page by calling, for example, an XSLT processor [22] that converts XML to HTML and/or JavaScript.
2. After viewing the HTML page sent by the Web server, the client sends data that may contain input values and/or a signature value to the Web server. When the browser signs the input data, the KeyStore is used to store the client's private key.
3. After receiving the input parameters, the presentation processor makes an authorization request for a write action for specific elements. The REM may verify the signature and/or update the target element. The server's KeyStore is used to retrieve the signer's public key or to sign the input value using the server's private key.



**Figure 8. Web Application based on XML Document Security**

For example, in Step 1, a securely transcoded cyber-catalogue in HTML format is returned to the client, although the original cyber-catalogue that may contain information of higher security class is represented in XML format. Using the HTTP Put mode is, a purchase order document stored in the Web server in XML format is securely updated with purchase order parameters submitted in HTML format. The advantages of this approach are the security requirements are easily integrated in this architecture without any server-side programming. For example, the system understands whether or not proof of the client's inability to repudiate the proposed contract is necessary before making it. The system understands whether or not credit card numbers should be encrypted with a specific cryptographic key. The signature transcoding module can be used to issue a receipt in response to the client's request that is signed with the server's private key. The XACL language is capable of specifying Web applications like this in integrated manner. Note that our approach does not depend on any specific transportation protocol such as HTTP. Thus it is easy to implement the system in other ways, for example, where the signed receipt is transmitted as e-mail using the SMTP protocol.

**7. CONCLUSION**

We have presented a provisional authorization model that provides XML with element-wise access control mechanism. We have formalized a provisional authorization model that adds extended semantics to traditional authorization models. The proposed model integrates several security features such as authorization and non-repudiation in unified XML documents and enables the authorization initiator not only to securely browse XML documents but also to securely update each document element. We have developed an XML access control language (XACL) based on the proposed model. We also described how our authorization system can be integrated into a conventional Web application.

**8. RERERENCES**

[1] T. Bray et al. "Extensible Markup Language (XML) 1.0," World Wide Web Consortium (W3C), <http://www.w3.org/TR/REC-xml> (February, 1998).

[2] H. Maruyama, K. Tamura, and N. Uramoto, "XML and Java, Developing Web Applications," Addison Wesley (May, 1999).

[3] M. Bartel et al. "XML-Signature Syntax and Processing," <http://www.w3.org/TR/xmlsig-core>, W3C Working Draft (Feb, 2000).

[4] alphaWorks, "XML Security Suite," <http://www.alphaWorks.com/tech/xmlsecuritysuite> (1999).

[5] D. E. Denning, "Cryptography and Data Security," Addison-Wesley, Reading, MA (1983).

[6] S. Castano, M. Fugini, G. Martella, and P. Samarati, "Database Security," Addison-Wesley, Reading, MA (1994).

[7] T. Y. C. Woo and S. S. Lam, "A Framework for Distributed Authorization," *1<sup>st</sup> ACM Conference on Computer and Communications Security* (November 1993).

[8] S. Jajodia, P. Samarati, and V. S. Subrahmanian, "A Logical Language for Expressing Authorizations," *Proc. 1997 IEEE Symposium on Security and Privacy*, 31-42 (May 1997).

[9] S. Jajodia, P. Samarati, and V. S. Subrahmanian, and E. Bertino, "A Unified Framework for Enforcing Multiple Access Control Policies," *Proc. ACM SIGMOD International Conference on Management of Data*, 474-485 (May 1997).

[10] E. Damiani, S. D. C Vimercati, S. Paraboschi, and P. Samarati, "Securing XML Documents," *Proc. EDBT 2000*, Lecture Notes in Computer Science vol. 1777, 121-135, (2000).

[11] M. Blaze, J. Feigenbaum and J. Lacy, "Decentralized Trust management," *Proc. 1996 IEEE Symposium on Security and Privacy*, 164-173 (May 1996).

[12] M. Blaze, J. Feigenbaum, and A. Keromytis, "KeyNote: Trust Management for Public-Key Infrastructures," *Proc. 1998 Cambridge Security Protocols International Workshop*, LNCS, vol. 1550, 59-63 (1999).

[13] Object Management Group, "Security Service Specification," In CORBA services: Common Object Services Specification, Chapter 15 (November 1997).

[14] G. Karjoth, "Authorization in CORBA Security," *Proc. 5th European Symposium on Research in Computer Security*, 143-158 (Sep. 1998).

[15] SecureWay Policy Director, <http://www-4.ibm.com/software/security/policy/>

[16] J. G. Stener, B. C. Neuan, and J. I. Schiller, "Kerberos: An Authentication Service for Open Network Systems," *Proc. USENIX Conference* (Feb. 1998).

[17] A. Herzberg, Y. Mass, J. Mihaeli, D. Naor, Y. Ravid, "Access Control Meets Public Key Infrastructure, Or: Assigning Roles to Strangers," *Proc. IEEE Symposium on Security and Privacy* (May, 2000).

[18] ISO/IEC 10181-3, "The Access Control Framework"

[19] D. E. Denning, "An Intrusion Detection Model," *IEE Transactions on Software Engineering*, vol. SE-13, No. 2 (February, 1987).

[20] World Wide Web Consortium (W3C), "XML Path Language (XPath) Version 1.0," <http://www.w3.org/TR/PR-xpath19991008>, (October 1999).

[21] E. Bertino, F. Buccafurri, D. Ferrari, and P. Rullo, "An Authorization Model and Its Formal Semantics," *Proc. 5th European Symposium on Research in Computer Security*, 127-142 (September 1998).

[22] J. Clark, "XSL Transformations (XSLT) Version 1.0," W3C Recommendation, <http://www.w3.org/TR/xslt> (November, 1999).

## Appendix: Online-Contracting Example

We describe the complete and practical set of XML data for online-contracting example. This XML consists of three portions: <contractor>, <policy>, and <status> portion. The <contractor> portion is the XML document that is an electronic contract document of Figure 2. The <policy> portion is provisional authorization policy specified in XACL that consists of three authorization rules. The <status> portion is used for storing the access log.

```
<?xml version="1.0"?>
<!DOCTYPE document SYSTEM "contract.dtd">
<document>
<!--=====
<!-- *** T A R G E T   O B J E C T ***-->
<contractor level="1">
<contract class="A">
  <t_and_c>Purchase of $1M over one year</t_and_c>
  <representative/>
</contract>
<comments>We accept the contract</comments>
</contractor>
<!--=====
<!-- *** A U H O R I Z A T I O N   P O L I C Y ***-->
<policy>
<property>
  <propagation read="down" write="down"/>
  <conflict-resolution read="dtp" write="dtp"/>
  <default read="denial" write="denial"/>
</property>
<!--=====
<!-- First xacl says that Business Owner can write t_and_c if
t_and_c field is empty provided the access is logged-->
<xacl>
  <object href="/document/contractor/contract/t_and_c"/>
  <rule>
    <acl>
      <subject><roles><role>Business
Owner</role></roles></subject>
      <action name="write" permission="grant">
        <provisional_action timing="before" name="log"/>
      </action>
      <condition operation="and">
        <predicate name="compareStr">
          <parameter>eq</parameter>
          <parameter><function name="get_field"/></parameter>
          <parameter>t_and_c</parameter>
          <parameter/>
        </predicate>
      </condition>
    </acl>
  </rule>
</xacl>
<!--=====
<!-- Second xacl says that the Registered Client can write
comments element if t_and_c element is not empty and comments
```

```

element is empty, provided access is logged and the signature on
the comments sent from the client is verified successfully -->
<xacl>
  <object href="/document/contractor/comments"/>
  <rule>
  <acl>
    <subject><roles><role>Registered
Client</role></roles></subject>
    <action name="write" permission="grant">
    <provisional_action timing="before" name="log"/>
    <provisional_action timing="before" name="verify">
    <parameter>
    <SignedInfo>
    <CanonicalizationMethod
Algorithm="http://www.w3.org/TR/1999/WD-xml-c14n-19991115"/>
    <SignatureMethod
Algorithm="http://www.w3.org/2000/01/xmldsig/rsa-sha1"/>
    <Reference>
    <Transforms>
    <Transform
Algorithm="http://www.w3.org/TR/1999/WD-xml-c14n-19991115"/>
    </Transforms>
    <DigestMethod
Algorithm="http://www.w3.org/2000/01/xmldsig/sha1"/>
    </Reference>
    </SignedInfo>
    </parameter>
    </provisional_action>
    </action>
    <condition operation="and">
    <predicate name="compareStr">
    <parameter>neq</parameter>
    <parameter><function name="get_field"/></parameter>
    <parameter>t_and_c</parameter>
    <parameter/>
    </predicate>
    <predicate name="compareStr">
    <parameter>eq</parameter>
    <parameter><function name="get_field"/></parameter>
    <parameter>comments</parameter>
    <parameter/>
    </predicate>
    </condition>
    </acl>
  </rule>
</xacl>
<!-- Third xacl says that the Registered Client can read contractor
subtree. -->
<xacl>
  <object href="/document/contractor"/>
  <rule>
  <acl>
    <subject><roles><role>Registered
Client</role></roles></subject>
    <action name="read" permission="grant">
    </acl>
  </rule>
</xacl>
</policy>
<!-- ***** S T A T U S ***** -->
<status>
<!-- *** Log #1: Kudo updated t_and_c ***-->
<log href="/document/contractor/contract/t_and_c" time="5/5/00">
  <subject>
  <uid>CN=Michiharu Kudo, OU=TRL, O=IBM, C=JP</uid>
  <roles><role>Business Owner</role></roles>
  </subject>
  <action permission="grant" name="write">
  <parameter>Purchase of $1M over one-year</parameter>
  </action>

```

```

</log>
<!-- ***** S T A T U S ***** -->
<!-- *** Log #2: Hada updated comments ***-->
<log href="/document/contractor/comments" time="5/10/00">
  <subject>
  <uid>CN=Satoshi Hada, OU=TRL, O=IBM, C=JP</uid>
  <roles><role>Registered Client</role></roles>
  </subject>
  <action permission="grant" name="write">
  <parameter>
  <Signature xmlns="http://www.w3.org/2000/01/xmldsig"/>
  <SignedInfo>
  <CanonicalizationMethod
Algorithm="http://www.w3.org/TR/1999/WD-xml-c14n-19991115"/>
  <SignatureMethod
Algorithm="http://www.w3.org/2000/01/xmldsig/rsa-sha1"/>
  <Reference IDREF="Res0">
  <Transforms>
  <Transform
Algorithm="http://www.w3.org/TR/1999/WD-xml-c14n-19991115"/>
  </Transforms>
  <DigestMethod
Algorithm="http://www.w3.org/2000/01/xmldsig/sha1"/>
  <DigestValue
Encoding="http://www.w3.org/2000/01/xmldsig/base64">
WjDP4Pbe1KGFHhPHI967w4SA=
  </DigestValue>
  </Reference>
  </SignedInfo>
  <SignatureValue>NJ0z/nrH0MXy5XQW...</SignatureValue>
  <KeyInfo>
  <X509Data>
  <509Name>CN=Satoshi Hada, OU=TRL, O=IBM,
C=JP</X509Name>
  <X509Certificate>MIIB7TCCA...</X509Certificate>
  </X509Data>
  </KeyInfo>
  <dsig:Object Id="Res0" xmlns="
xmlns:dsig="http://www.w3.org/2000/01/xmldsig"/>
  <parameter>We accept the contract</parameter>
  </dsig:Object>
  </Signature>
  </parameter>
  <provisional_action timing="before" name="log"/>
  <provisional_action timing="before" name="verify">
  <parameter>
  <SignedInfo>
  <CanonicalizationMethod
Algorithm="http://www.w3.org/TR/1999/WD-xml-c14n-19991115"/>
  <SignatureMethod
Algorithm="http://www.w3.org/2000/01/xmldsig/rsa-sha1"/>
  <Reference>
  <Transforms>
  <Transform
Algorithm="http://www.w3.org/TR/1999/WD-xml-c14n-19991115"/>
  </Transforms>
  <DigestMethod
Algorithm="http://www.w3.org/2000/01/xmldsig/sha1"/>
  </Reference>
  </SignedInfo>
  </parameter>
  </provisional_action>
  </action>
</log>
</status>
</document>

```