Regulating Access to XML documents

Alban Gabillon - Emmanuel Bruno

Laboratoire SIS - Equipe Informatique
Université de Toulon et du Var
83957 La Garde, France
{gabillon,bruno}@univ-tln.fr

Abstract

In this paper, our objective is to define a security model for regulating access to XML documents. Our model offers a security policy with a great expressive power. An XML document is represented by a tree. Nodes of this tree are of different type (element, attribute, text, comment…etc). The smallest protection granularity of our model is the node, that is, authorisation rules granting or denying access to a single node can be defined. The authorisation rules related to a specific XML document are first defined on a separate Authorisation sheet. This Authorisation sheet is then translated into an XSLT sheet. If a user requests access to the XML document then the XSLT processor uses the XSLT sheet to provide the user with a view of the XML document which is compatible with his rights.

Keywords: Security Model, Subject, Object, Authorisation Rule, Access Controls, XML, XPath, XSLT.

1. Introduction

XML is a mark-up language standardised by the World Wide Web Consortium (W3C) [1]. XML has become a standard for describing information distributed on Internet. Since Internet is a public network, internet applications need security mechanisms to protect sensitive data against unauthorised access. Standardisation activities for XML digital signature and element-wise encryption have already started [2]. However, a standardised authorisation mechanism for XML data still remains an open issue although some proposals have already been made [3][4][5].

In this paper, our objective is to define a security model for regulating access to XML documents. In [6], due to space limitations we only managed to present the basics of our model. In this paper we present the complete model. An XML document is represented by a tree. Nodes of this tree are of different type (element, attribute, text, comment…etc). The smallest protection granularity of our model is the node, that is, authorisation rules granting or denying access to a single node can be defined. The semantics of an authorisation rule is unique regardless of the node type. We show that our model allows us to define complex security policies including cover story management.

A prototype implementing our model is available online at `http://sis.univ-tln.fr/XML-secu`. Our prototype is based on XSLT [7]. The authorisation rules related to a specific XML document are first defined on a separate easy-to-read Authorisation sheet. This Authorisation sheet is then translated into an XSLT sheet. If a user requests access to the XML document then the XSLT processor uses the XSLT sheet to provide the user with a view of the XML document which is compatible with his rights.

Our model uses the XPath language. Therefore, section 2 of this paper briefly summarises the main characteristics of the XPath language. Section 3 presents our model. Section 4 sketches a straightforward implementation of our model using an XSLT processor. Section 5 discusses related work. Finally, section 6 concludes this paper.

2. XPath

XPath [8] is a language for addressing parts of an XML document. XPath models an XML document as a tree of nodes. The following XML document represents a file of medical records which contains only one record. This XML document can be represented by the tree of Figure 1.

```
<files>
   <record id="mrobert">
    <name>Martin Robert</name>
    <diagnosis>
        <item>Pneumonia</item>
    </diagnosis>
   </record>
</files>
```

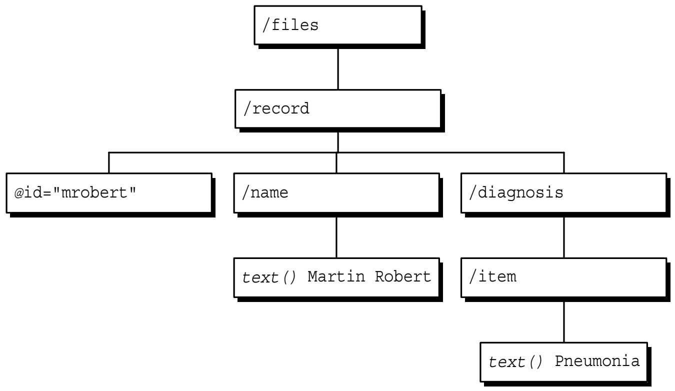

Figure 1 : Tree representation of an XML document

Nodes preceded by "/" are element nodes. Nodes preceded by "@" are attribute nodes and nodes preceded by "text()" are text nodes[1].

The following expressions are examples of *absolute `XPath` location paths* (see [8]):

- `/files/record`. This path addresses each record element which is a child of the files element.
- `/files//text()`.This path addresses each text node which is a descendant of the `files` element.

The following expressions are examples of *relative `XPath` location paths* (we assume the *context node* is the diagnosis element)(see [8]):

- `item/node()`.This path addresses all the child nodes (element or text node) of the `item` element.
- ../@*. This path addresses all the attribute nodes of the context node's parent element, that is, this path addresses all the attributes of the `record` element.

In addition to its use for addressing, `XPath` is also designed so that it has a natural subset that can be used for matching (testing whether or not a node matches a *pattern*). This use of `XPath` is made by `XSLT` [7]. A pattern specifies a set of conditions on a node. A node that satisfies the conditions matches the pattern. A node that does not satisfy the conditions does not match the pattern. Location paths that meet certain restrictions can be used as patterns (see [7]).

The following expressions are examples of patterns:

- `item`. Each `item` element matches this pattern.
- `item/text()`. Each text node of each `item` element matches this pattern.
- `record/@*`. Each attribute of each `record` element matches this pattern.

[1] The `XPath` data model includes three other types of node namely namespace node, processing instruction node and comment node. For a concise presentation, we do not include such nodes in our example.

- `record[1]/node() | record[1]/@*`. Each child node and attribute of the first `record` element matches this pattern ("|" is the *union* operator).
- `item[contains(text(),'Cancer')]`. Each `item` element containing the string `'Cancer'` matches this pattern.

3. Access Control Model

The development of an access control system requires the definition of *subjects* and *objects* for which *authorisation rules* must be specified and access controls must be enforced.

3.1 Subjects

A subject is a user. Each user has an identifier which can be the login name. Each user is the member of one or several user groups. For the sake of simplicity we assume that groups are disjoint but they can be nested.

The subject hierarchy is described in a separate *XML Subject Sheet* (`XSS`). The document below shows a simple example of such a sheet.

```
<subjects>
  <users>
    <member id="dupont">
      <name>Pierre Dupont</name>
    </member>
    <member id="durand">
      <name>Jacqueline Durand</name>
    </member>
    <member id="frobert">
      <name>Francine Robert</name>
    </member>
    <member id="mrobert">
      <name>Martin Robert</name>
    </member>
    <member id="beaufort">
      <name>Colette Beaufort</name>
    </member>
  </users>
  <groups>
    <Staff>
       <Secretary>
             <member idref="beaufort"/>
      </Secretary>
      <Doctor>
            <member idref="dupont"/>
      </Doctor>
      <Nurse>
             <member idref="durand"/>
       </Nurse>
    </Staff>
    <Patient>
             <member idref="mrobert"/>
    </Patient>
    <Family>
       <Robert>
             <member idref="frobert"/>
             <member idref="mrobert"/>
      </Robert>
    </Family>
  </groups>
</subjects>
```

This XSS document describes a subject hierarchy with four users. In this example, each user is member of at least one group. There are three groups: *Staff*, *Patient* and *Family*. The *Staff* group is subdivided into three subgroups: *Doctor*, *Nurse*, *Secretary*. The *Family* group contains only the *Robert* family sub-group.

Users registered in this XSS sheet are selected with location paths relative to the `subjects` element. If a location path addresses a node *n* then we say that all users who are referenced in the sub-tree of which *n* is the root are selected. Examples of such a path are the followings:

- `users`. This path selects all users.
- `users/member[@id='mrobert']`. This path selects user Martin Robert.
- `groups//Secretary`. This path selects all secretaries
- `groups/*[name()!='Staff']`. This path selects all the users who are not staff members.

The following document corresponds to the DTD for XSS sheets:

```
<!ELEMENT   subjects    (users,groups)     >
<!ELEMENT   users       (member*)          >
<!ELEMENT   member      (name?)            >
<!ELEMENT   name        #PCDATA            >
<!ATTLIST   member      id       ID
                        idref    IDREF     >
<!ELEMENT   groups ((ANY|member)*)         >
```

3.2 Objects

In section 3 we have seen that an XML document can be represented by an XPath tree. An object can be any node of an XPath tree.

3.3 Authorisation rules

An *authorisation rule* is a 4-tuple of the form:

<*set-of-subjects*, *set-of-objects*, *access, priority*>

We have seen in sections 3.1 and 3.2 that a subject is a user and an object is a node of an XPath tree.

set-of-objects is expressed with a pattern.

set-of-subjects is a location path relative to the element *subjects* of the XSS sheet (see section 3.1)

The value of *access* is either *grant* or *deny*.

priority is optional. It is used to fix the priority of the authorisation rule (see section 3.4 for more details). The default priority is 0.

In our model the semantics of an authorisation rule is unique regardless of the node type (element, attribute, text …):

If access to node *n* is granted to user *u* then *u* is permitted to see the sub-tree of which *n* is the root.

If access to node *n* is denied to user *u* then *u* is forbidden to see the sub-tree of which *n* is the root.

The Security Administrator writes the authorisation rules on an *XML Authorisation Sheet* (XAS). The following XML document is an example of XAS sheet. This sheet contains the rules which apply to the document described in section 2 and refers to the XSS sheet defined in section 3.1.

```
<!-- D E F A U L T   H O S P I T A L   P O L I C Y -->

<!-- Rule 1 -->
<xas DefaultPolicy="open" DefaultSubjectsFile="subjects.xss">

<!-- Rule 2 -->
<rule access="deny"  object="record" subject="groups/*[name()!='Staff']"/>

<!-- Rule 3 -->
<rule access="deny"  object="diagnosis" subject="groups//Secretary"/>

<!-- Rule 4 -->
<rule access="grant" object="record[@id=$user]"
      subject="users/member[@id=$user]"/>
</xas>
```

The first element of an XAS sheet determines whether the default policy is *open* or *closed*[9]. If no authorisation rule is specified regarding a user *u* and a node *n* then *u* is permitted to access to *n* in case of the open policy and is forbidden to access to *n* in case of the closed policy.

- Rule 1 says that the default policy is open.
- Rule 2 says that non staff members are forbidden to see the records.
- Rule 3 says that secretaries are forbidden to see the diagnosis of a patient.
- Rule 4 says that a patient is permitted to see his personal medical record.

$user is a variable which is instantiated with the *id* of the user accessing to the XML source document. Rule 4 overrides rules 2 when a user is accessing to his personal data.

None of the authorisation rules of our example includes the *priority* attribute. Therefore, the priority of each rule is set by default to 0.

The following document corresponds to the DTD for XAS sheets

```
<!ELEMENT  xas (rule*)   >
<!ATTLIST  xas
           DefaultPolicy        (open|closed)    #FIXED      "open"
           DefaultSubjectsFile  CDATA            #FIXED      "subjects.xss">
<!ELEMENT rule EMPTY    >
<!ATTLIST  rule
           object               CDATA            #REQUIRED
```

```
          subject              CDATA             #REQUIRED
          access               (grant|deny)      #REQUIRED
          priority             CDATA             "0"                        >
```

3.4 ComputeView Algorithm and Conflict Resolution Policy

If a user requests to see the XML source document then he has to be provided with the view of the document which is compatible with his rights. The aim of this section is to present the algorithm for computing such a view.

Before presenting the algorithm itself we have to perform the following preliminary task in order to obtain a *short* and *easy-to-read* algorithm[2].

We replace each grant rule of the form,

`<rule **access**="grant" **object**=n **subject**=u **priority**=p>,`

by the following three consecutive grant rules:

- `<rule access="grant" object=n          subject=u priority=p>`
- `<rule access="grant" object=n//node()  subject=u priority=p>`
- `<rule access="grant" object=n//@*      subject=u priority=p>`

The first rule is the same as the original rule. All the descendant nodes of[3] the node *n* match the object pattern of the second rule. All the attributes of *n* and the attributes of the descendant nodes of *n* match the object pattern of the third rule.

Note that this replacement does not change the security policy. Recall that granting access to node *n* to user *u* means that *u* is permitted to see the sub-tree of which *n* is the root (see previous section 3.3).

Finally,

if the default policy is closed then we insert the following rule:

- `<rule access="deny" object="/" subject= "users" priority="-1">`

and if the default policy is open then we insert the following rules:

- `<rule access="grant"  object="/"          subject="users" priority="-1">`
- `<rule access="grant"  object="//node()"  subject="users" priority="-1">`
- `<rule access="grant"  object="//@*"      subject="users" priority="-1">`

The rules implementing the default policy all have a negative priority. Path, `users`, selects all the users registered in the XSS sheet.

Consequently, our previous example of XAS sheet can be interpreted as the following list of rules:

[2] We could define the conflict resolution policy and write the algorithm without performing this preliminary task. However the definition of both the conflict resolution policy and the algorithm would be less straightforward (although the complexity would be unchanged).

[3] or, of each node of the set *n,* if *n* is a set of nodes (cf section 3.3)

```
<!-- D E F A U L T   H O S P I T A L   P O L I C Y -->

<!-- Rules 1a, 1b and 1c -->
<rule access="grant"   object="/"          subject="users"    priority="-1">
<rule access="grant"   object="//node()"   subject="users"    priority="-1">
<rule access="grant"   object="//@*"       subject="users"    priority="-1">

<!-- Rule 2 -->
<rule access  = "deny"                             object   = "record"
      subject = "groups/*[name()!='Staff']"        priority = "0"/>

<!-- Rule 3 -->
<rule access  = "deny"                object   = "diagnosis"
      subject = "groups//Secretary"   priority = "0"/>

<!-- Rules 4a, 4b and 4c -->
<rule access  = "grant"                   object = "record[@id=$user]"
      subject = "users/member[@id=$user]" priority = "0"/>
<rule access  = "grant"                   object="record[@id=$user]//node()"
      subject = "users/member[@id=$user]" priority = "0"/>
<rule access  = "grant"                   object="record[@id=$user]//@*"
      subject = "users/member[@id=$user]" priority = "0"/>
```

- Rule 1 has been replaced by three grant rules with a negative priority. The default policy has the lowest priority.
- Rule 4 has been replaced by three grant rules.
- Rules which were without a priority attribute in the original XAS sheet have now their priority set to 0.

There is a conflict between a deny rule and a grant rule for a node *n* and a user *u* if *n* matches the two *set-of-object* patterns and *u* is addressed by the two *set-of-subjects*.

In our example, rule 2 conflicts with rule 1b for each `record` element and each user who is not a staff member. Rule 3 conflicts with rule 1b for each `diagnosis` element and each secretary. Rule 4a conflicts with rule 2 for the current user's `record` element (this is in case the user is a patient).

The conflict resolution policy of our model is very simple:

1. If, for a node *n* and a user *u*, there is a conflict between a set of rules then the rules with the highest priority are selected.
2. If the selected rules are more than one then the last rule in the XAS sheet is elected.

Step 2 explains why rule 4a of our example overrides rules 2 in the conflicting cases.

Considering this policy, we can now present our algorithm for computing the views:

ComputeView Algorithm

```
Let U be the user for which the view has to be computed
Let L be an empty list of nodes
Insert the root element into L
Let R be an empty list of nodes
While L is not empty Do
      N ← the first node of L
      Select all the rules such as N matches the object pattern and U is
      selected by the subject path
      Apply the conflict resolution policy defined above
      If the elected rule is a deny rule then
            Remove N from L
      Else
            Append N to R
            Replace N into L by the attributes and child nodes of N
```

After the algorithm finishes R contains the pre-order list of the nodes which belong to the view. Using this algorithm we can easily compute the view for each user

View for *P. Dupont (Doctor)* and *J. Durand (Nurse)* and *M. Robert (Patient)*

```
<files>
   <record id="mrobert">
      <name>Martin Robert</name>
      <diagnosis>
         <item>Pneumonia</item>
      </diagnosis>
   </record>
</files>
```

Pierre Dupont and *Jacqueline Durand* are permitted to see everything. *Martin Robert* is permitted to see everything because the file contains only one record: his own record.

View for *C. Beaufort (Secretary)*

```
<files>
   <record id="mrobert">
      <name>Martin Robert</name>
   </record>
</files>
```

Colette Beaufort is forbidden to see the diagnosis element.

View for *F. Robert (Robert Family)*

```
<files/>
```

The default hospital policy says that non staff members are forbidden to see the medical records. This rule applies to family members. Therefore, *Francine Robert* is forbidden to see the medical records, including Robert's record.

As we have seen above, the conflict resolution policy of our model is based on priorities (whether they are implicit or explicit). Conflict resolution policies based on priorities are usually considered as difficult to manage and understand. We agree that in some cases it might be difficult for a human to figure out the output of conflicting rules. However, we have decided to use priorities for the following two reasons:

1. We do not see what are the advantages of using conflict resolution policies based on principles like "the most specific object takes precedence" or "the most specific subject takes precedence" (see [3] for instance). These policies which were first used in Object-Oriented environments are not well adapted to `XML`. Indeed, in many cases it is impossible for a human to predict which `XPath` expression is going to be the most specific. As a matter of facts let us consider the following two rules:

   <**rule** **access**="deny" **object**="diagnosis[item='Cancer' or item='Pneumonia'] **subject**="groups//Nurse"/>

   <**rule** **access**="grant" **object**="diagnosis[item='Pneumonia' or item='Ulcer'] **subject**="groups//Nurse"/>

   The first rule says that nurses are forbidden to see the trees the root of which is a diagnosis element which includes an item equal to Cancer or Pneumonia.

   The second rule says that nurses are permitted to see the trees the root of which is a diagnosis element which includes an item equal to Ulcer or Pneumonia.

   Clearly none of the `XPath` expressions for the `object` attribute is more specific than the other making difficult to predict what is going to happen if a nurse asks for an access to a diagnosis element which includes an item equal to Pneumonia.

   We could show many examples like this where a human cannot possibly predict which rule is going to preempt the others.

2. Our prototype of security processor is based on `XSLT` (see section 4). The conflict resolution policy of `XSLT` processors mainly uses priorities and has been proved to be efficient.

In any case, whatever the conflict resolution policy is, we think that the Security Administrator has to be provided with some policy debugging tools.

3.5 Highly Expressive Security Policy

The example that we used in the previous section is very simple and does not completely show the power of our model. The aim of this section is to show that our model allows us to express complex security policies easily. In particular we can define security policies supporting the concept of exception, the definition of content-based authorisation rules and the insertion of cover stories in the source document.

Consider our previous source document into which a new record has been inserted:

```
<files>
<record id="pfranck">
    <name>Patricia Frank</name>
    <diagnosis>
        <item>Cancer</item>
        <item coverstory="yes">Ulcer</item>
        <comments>life expectancy is limited to two years</comments>
    </diagnosis>
</record>
<record id="mrobert">
      ...
```

```
</record>
</files>
```

This `XML` document can be represented by the following `XPath` tree:

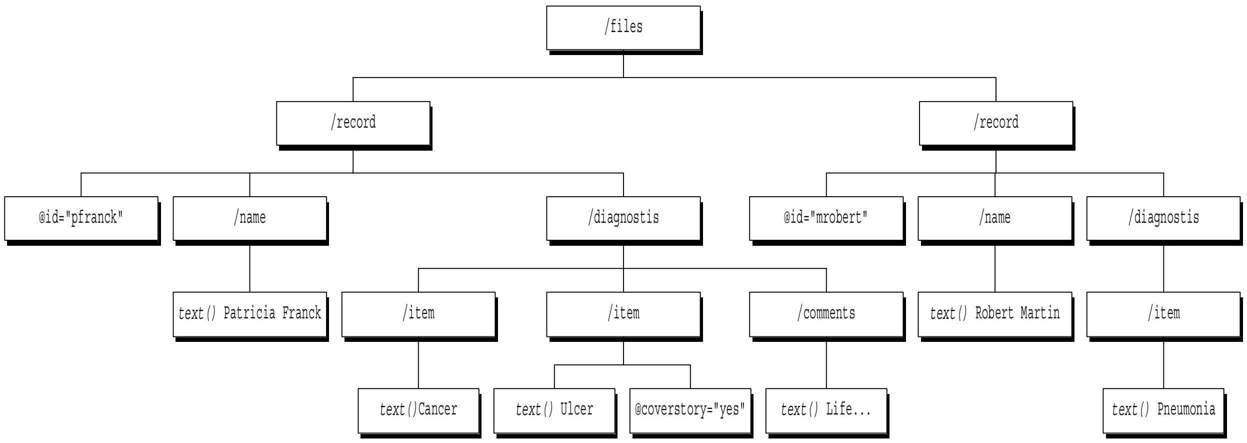

Figure 2 Tree representation of the source document

Patricia Franck is a new patient. She has a cancer. Her life expectancy is limited to two years. The item saying that she has an ulcer is a cover story. A cover story is a lie inserted in the source document in order to hide the existence of a sensitive information. Attribute `coverstory="yes"` informs users who are permitted to see everything that `ulcer` is a lie (see [10] for more information about cover stories and cover story management).

The `XSS` subject sheet is extended as follows:

```
<subjects>
  <users>
    ...
    <member id="pfranck"><name>Patricia Franck</name></member>
    <member id="gfranck"><name>Georges Franck</name></member>
    ...
  </users>
  <groups>
    ...
    <Patient>
            <member idref="pfranck"/>
            <member idref="mrobert"/>
    </Patient>
    <Family>
      ...
      <Franck>
            <member idref="gfranck"/>
            <member idref="pfranck"/>
      </Franck>
    </Family>
  </groups>
</subjects>
```

Patricia Franck and Georges Franck are new users. The *Franck* family is a new group.

The XAS sheet is extended as follows:

```
<!-- D E F A U L T   H O S P I T A L   P O L I C Y -->
...
<!-- P O L I C Y   S P E C I F I C   T O   P A T R I C I A   F R A N C K -->

<!-- Rule 5 -->
<rule access  = "grant" object = "record[@id='pfranck']"
      subject = "groups/Family/Franck"          />

<!-- Rule 6 -->
<rule access  = "deny"  object = "record[@id='pfranck']//comments"
      subject = "groups/Family/Franck"          />

<!-- Rule 7 -->
<rule access  = "deny"  object = "record[@id='pfranck']//comments/text()"
      subject = "groups//*[name()='Nurse']"     />

<!-- Rule 8 -->
<rule access  = "deny"  object = "item[contains(text(),'Cancer')]"
      subject = "users/member[@id='pfranck']"   />

<!-- Rule 9 -->
<rule access  = "grant" object = "item[@coverstory='yes']"
      subject = "users/member[@id='pfranck']"   />

<!-- Rule 10 -->
<rule access  = "deny"  object = "item/@coverstory"
      subject = "users/member[@id='pfranck']"   />
</xas>
```

- Rule 5 says that the Franck family is permitted to see the data of Patricia Franck.
- Rule 6 says that the Franck family (including Patricia) is forbidden to see the `comments` element of Patricia Franck's medical record.
- Rule 7 says that nurses are forbidden to see the text of the `comments` element of Patricia Franck's medical record. Note that rule 6 addresses the `comments` element itself. This means that the sub-tree of which the `comments` element is the root is protected. Rule 7 only protects the text of the `comments` elements.
- Rule 8 says that P. Franck is forbidden to see the `item` which says that she has a cancer. Rule 8 is a perfect example of a *content-based* authorisation rule
- Rule 9 says that P. Franck is permitted to see the `item` which is a cover story ...
- … but rule 10 says that P. Franck is forbidden to know that the `item` is a cover story

Rules 5 to 10 show that P. Franck is not considered as a standard patient. The default hospital policy does not apply to her. Doctors consider that for some psychological reasons, Patricia Franck must not know that she has a cancer. Therefore, doctors have decided to lie to Patricia Franck and to tell her that she has an ulcer. The family are told about this lie and are granted the permission to see the personal data of P. Franck. However the element saying that the life expectancy of P Franck is limited to two years must remain strictly confidential. Even nurses are forbidden to know this fact.

The views for some of the users are listed below:

View for *P. Dupont (Doctor)*

```
<files>
<record id="pfranck">
    <name>Patricia Frank</name>
    <diagnosis>
        <item>Cancer</item>
        <item coverstory="yes">Ulcer</item>
        <comments>life expectancy is limited to two years</comments>
    </diagnosis>
</record>
<record id="mrobert">
    <name>Martin Robert</name>
    <diagnosis>
        <item>Pneumonia</item>
    </diagnosis>
</record>
</files>
```

Pierre Dupont is a doctor. He can see everything.

View for *J. Durand (Nurse)*

```
<files>
<record id="pfranck">
    <name>Patricia Frank</name>
    <diagnosis>
        <item>Cancer</item>
        <item coverstory="yes">Ulcer</item>
        <comments/>
    </diagnosis>
</record>
<record id="mrobert">
    <name>Martin Robert</name>
    <diagnosis>
        <item>Pneumonia</item>
    </diagnosis>
</record>
</files>
```

Jacqueline Durand is a nurse. She can see everything except the content of the `comments` element.

View for *G. Franck (Franck Family)*

```
<files>
<record id="pfranck">
    <name>Patricia Frank</name>
    <diagnosis>
        <item>Cancer</item>
        <item coverstory="yes">Ulcer</item>
    </diagnosis>
</record>
</files>
```

Georges Franck has access to the medical file of Patricia Franck. He knows that doctors have decided to tell Patricia Franck a lie regarding her illness. He is not aware of the existence of the `comments` element.

View for *P. Franck (Patient)*

```
<files>
<record id="pfranck">
    <name>Patricia Frank</name>
    <diagnosis>
        <item>Ulcer</item>
    </diagnosis>
</record>
</files>
```

Patricia Franck believes that she has an ulcer.

This example has shown us that we have the possibility of defining default security policies which can be overridden by specific policies for some particular cases.

Finally note that, in [3], authors define the concept of *instance level* and *DTD level* authorisation sheet. An instance level authorisation sheet only applies to a specific `XML` document. A DTD level authorisation sheet applies to a set of documents which conform to a specific DTD. Our model allows us to define such a *DTD level* authorisation sheet. If both a DTD level `XAS` sheet and an instance level `XAS` sheet apply to a specific `XML` document then the instance level authorisation rules are appended to the DTD level authorisation rules in a *global* `XAS` sheet. In other words, the global `XAS` sheet contains the authorisation rules expressed at the DTD level followed by the rules expressed at the instance level. The global `XAS` sheet is then used to compute the different views. We can mention that choosing relevant priority levels allows us to define both DTD rules which can be overridden (low priority) at the instance level and mandatory DTD rules which cannot be overridden (high priority).

4. Sketch of implementation

We have developed an `XSLT`-based prototype of a security processor which implements our model. Our prototype is available online at `http://sis.univ-tln.fr/XML-secu`. Figure 3 describes the prototype.

Our prototype is integrated into the framework of an Apache Web server[4] which uses Tomcat[5] as a server for Java Servlets. Views of a source document are dynamically computed by using the Cocoon[6] publishing framework for `XML` data. In figure 3, the document to be protected is `Doc.XML`. `Doc.XAS` is the global `XAS` sheet. `Subjects.XSS` describes the subjects.

[4] `http://www.apache.org`

[5] `http://jakarta.apache.org/tomcat/index.html`

[6] `http://xml.apache.org/cocoon/index.html`

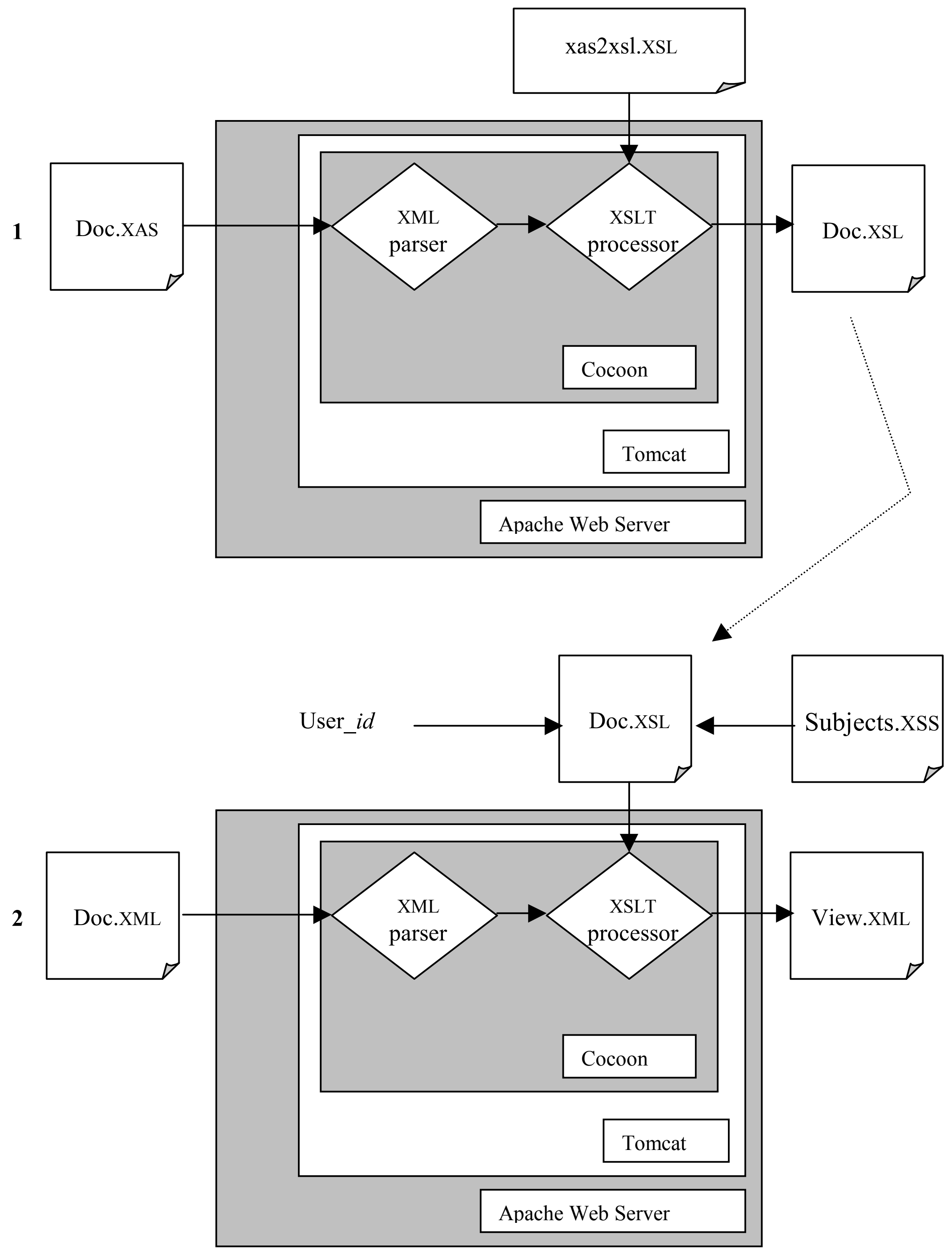

Figure 3 Prototype

1. Easy-to-read authorisation rules expressed in `Doc.XAS` are translated into `XSLT` rules (*templates*) producing an XSL sheet (`Doc.XSL`). This translation is performed by the `XSLT` processor by applying `xas2xsl.XSL`. This translation is performed *once*. After this operation, the `XAS` sheet is no longer used unless it is updated (in this case, it is automatically retranslated). `Doc.XSL` is saved into the cache.

2. If a user requests access to `Doc.XML` then he is provided with a view of the document which is compatible with his rights. This view is produced by the `XSLT` processor by applying `Doc.XSL`. Templates contained in `Doc.XSL` are used by the `XSLT` processor to compute the view according to the algorithm defined in section 3.4. `Doc.XSL` requires, as parameters, the user *id* (transmitted as a parameter by the Web server after the user has been authenticated) and `Subjects.XSS`. The view is also saved into the cache.

5. Related Work

Compared with other models, our model offers several advantages. Our model allows us to write security policies with a high expressive power since any node from an `XML` document can be independently protected (element, attribute, text …). The semantics of an authorisation rule is unique and precisely defined. Our conflict resolution policy is simple, efficient and adapted to an `XSLT` processor. Our model offers the possibility of defining content-based authorisation rules. Our proposal fully respects the W3C recommendation since we use the `XPath` language to address `XML` fragments and the `XSLT` language to compute the views. In [3], the semantics of an authorisation varies. An authorisation can be local (in this case it applies to an element and its attributes) or it can be recursive (in this case, it applies also to the sub-elements). The expressive power is limited since nodes like text nodes cannot be independently protected. Moreover, the conflict resolution policy is quite complex. In [4] the semantics of an authorisation is defined as *polymorphic*, that is, it varies according to the protected object. The model does not include the possibility of protecting all kinds of nodes. In [5], the model does not offer the possibility of protecting attribute or text nodes independently. None of these three models fully exploits the `XPath` language.

Our prototype also offers many advantages compared with other prototypes. We use a standard `XSLT` processor to compute the views. Therefore, we can choose among existing `XSLT` processors (Apache[7], Oracle[8], IBM) the one with the best performances. If there is a new W3C recommendation for the `XML`, `XPath`, and `XSLT` languages, then the only thing we have to do is to replace the `XML` parser and the `XSLT` processor with an `XML` parser and an `XSLT` processor which conform to the new recommendation. Integration of our prototype into the framework of an existing Web Server is also straightforward. In prototypes [11][12][13] which implement the models defined in [3][4][5], the views are produced by proprietary processors written in Java. These processors do not have the power and the performances of an `XSLT` processor. Moreover these processors need to be reprogrammed each time there is a new W3C recommendation.

[7] `http://xml.apache.org/xalan`

[8] `http://technet.oracle.com/tech/xml/`

6. Conclusion

In this paper, we have defined a model for regulating access to XML documents. We plan to extend this model in several directions:

- We are exploring the possibility of defining *provisional* authorisations. In [5] a provisional authorisation rule is defined as a rule which tells the user that his request will be authorised provided he or the system takes certain security actions.
- In the model described in this paper we implicitly assumed that users do not have access to the DTDs. Further versions of our model will include the possibility of protecting portions of DTDs or XML schemas.
- In this paper, we restricted ourselves to the *read* privilege. Indeed, the read privilege is the most important privilege to consider regarding documents which are published on the WEB. However, we plan to extend our model (and our prototype) with the *write* privilege.

References

[1] T. Bray et al. "Extensible Markup Language (XML) 1.0". World Wide Web Consortium (W3C). http://www.w3c.org/TR/REC-xml (October 2000).

[2] M. Bartel et al. "XML-Signature Syntax and Processing". W3C Candidate Recommendation. http://www.w3c.org/TR/xmldsig-core (October-2000).

[3] E. Damiani, S. De Capitani di Vimercati, S. Paraboschi, P. Samarati, "Securing XML Documents," in *Proc. of the 2000 International Conference on Extending Database Technology (EDBT2000),* Konstanz, Germany, March 27-31, 2000.

[4] E. Bertino, S. Castano, E. Ferrari and M. Mesiti. "Specifying and Enforcing Access Control Policies for XML Document Sources". World Wide Web Journal, vol. 3, n. 3, Baltzer Science Publishers.

[5] M. Kudo and S. Hada. "XML Document Security based on Provisional Authorisation". Proceedings of the 7th ACM conference on Computer and communications security. November, 2000, Athens Greece.

[6] A. Gabillon, E. Bruno. *A Filtering Model for XML documents*. WWW10 Conference Workshop on Information Filtering. Hong Kong, May 2001.

[7] J. Clark. "XSL Transformations (XSLT) Version 1.0". World Wide Web Consortium (W3C). http://www.w3c.org/TR/xslt (November 1999).

[8] J. Clark et al.. "XML Path Language (XPath) Version 1.0". World Wide Web Consortium (W3C). http://www.w3c.org/TR/xpath (November 1999).

[9] S. Jajodia, P. Samarati, V. Subrahmanian and E. Bertino. A Unified Framework for Enforcing Multiple Access Control Policies. Proc. of the 1997 ACM International SIGMOD Conference on Management of Data, Tucson, May 1997.

[10] F. Cuppens, A. Gabillon. *Cover Story Management*. Data and Knowledge Engineering Vol 37/2, 2001, pp 177-201.

[11] E. Damiani, S. De Capitani di Vimercati, S. Paraboschi, P. Samarati "XML Access Control Systems: A Component-Based Approach" in *Proc. IFIP WG11.3 Working Conference on Database Security*, Schoorl, The Netherlands, August 21-23, 2000.

[12] E. Bertino, M. Braun, S. Castano, E. Ferrari, M. Mesiti. "AuthorX: A Java-Based System for XML Data Protection". In Proc. of the 14th Annual IFIP WG 11.3 Working Conference on Database Security, Schoorl, The Netherlands, August 2000.

[13] AlphaWorks. XML Security Suite (xss4j). http://www.alphaWorks.ibm.com/tech/xmlsecuritysuite.