An Access Control Model for Tree Data Structures

Alban Gabillon, Manuel Munier, Jean Jacques Bascou, Laurent Gallon, Emmanuel Bruno

An Access Control Model for Tree Data Structures

Alban Gabillon[1], Manuel Munier[1], Jean-Jacques Bascou[1],
Laurent Gallon[1], Emmanuel Bruno[2]

[1]LIUPPA/CSySEC. Université de Pau. IUT de Mont de Marsan, 40 000 Mont de Marsan, France.
http://csysec.univ-pau.fr
{gabillon,munier,bascou,gallon}@univ-pau.fr
[2]SIS-Equipe Informatique. Université de Toulon, 83 000 Toulon, France
bruno@univ-tln.fr

Abstract. Trees are very often used to structure data. For instance, file systems are structured into trees and XML documents can be represented by trees. There are literally as many access control schemes as there are tree data structures. Consequently, an access control model which has been defined for a particular kind of tree cannot be easily adapted to another kind of tree. In this paper, we propose an access control model for *generic* tree data structures. This model can then be applied to any specific typed tree data structure.

Keywords: Authorization Rule, Permission, Security Policy, Access Control, Tree Data Structures.

1. Introduction

Trees are very often used to structure data. For instance, file systems are structured into trees and XML documents [Bray00] can be represented by trees. There are literally as many access control schemes as there are tree data structures:

- many access control models for XML have recently been proposed [GB01][Ber00a] [Dam00a] [KH00]
- SNMPv3 [WPM99] proposes a View Access Control Model (VACM) for MIB trees (Management Information Base)
- the security model of UNIX is an access control model for hierarchical file structures
- LDAP [OL] includes an access control model for LDAP tree directories [Sto00]
- etc.

Historically, applications like the SNMP protocol, the LDAP protocol, WEB servers or file systems were initially developed without integrating access control facilities. For the developers of these applications, the most important was to organize the data and to define primitives for manipulating them. Later on, some access control

mechanisms were added to these applications. However most of these access control mechanisms rely on models which are not clearly defined and which are specific to a particular kind of tree data structure. Very often these models are even based on the semantics of the data.

In this paper, we propose an access control model for *generic* tree data structures. This model can then be applied to any specific typed tree data structure.

The development of an access control model requires the definition of *subjects* and *objects* for which *authorization rules* must be specified and *access controls* must be enforced. An authorization rule grants or denies a *privilege* on an object to a subject. The *security policy* consists of a set of authorization rules. Access controls enforce the security policy.

The model we define in this paper provides us with the possibility of writing security policies with a great expressive power:

- The *granularity* of the objects is very fine
- The set of privileges includes the *read* privilege but also various types of *write* privileges

Moreover, our model provides us with a *security policy control language*. We use this language to administrate the security policy. It is based on the classical `GRANT/REVOKE` scheme.

The remainder of this paper is organized as follows:

In section 2 we define the concept of *tree*. In this section we also suggest a language for addressing the *nodes* of a tree. In section 3, we present our model. We give the definition of a subject and an object. We present the privileges which are supported by our model. We show how we can define authorization rules by labeling the source tree with some *authorization attributes*. Finally, we present the core of our security policy control language. In section 4, we show how we can use our model to protect `XML` data structures. Section 5 compares our model with some related works. Section 6 concludes this paper.

2. Tree Data Structure

2.1 Definition

A tree is a type of data structure in which each datum is called a *node*. Each node is the parent of zero or more *child* nodes. Each node has one and only one parent except one fundamental node which has no parent and which is called the *root* node (or simply the root). Trees are often called *inverted trees* because they are normally drawn with the root at the top.

In figure 1, `v1, v2, v3` etc. are nodes. `v1` is the root. `v6`, `v5` and `v3` are nodes without a child. They are called *leaf* nodes. `v2` and `v3` are the child nodes of `v1`. `v2`

is the parent node of v4 and v5. v4, v5, v6 are the *descendant* nodes of v2. v1 is the *ancestor* of all the other nodes etc.

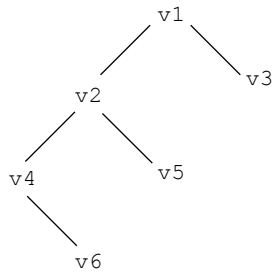

Fig. 1. Example of a Tree (1)

2.2 A language for addressing nodes

The purpose of this section is to suggest a language for addressing nodes. Our language can be seen as a simplified version of XPath [CD99]. Simplification comes from the fact that in our language each node is a non typed datum whereas in XPath, each node is of a specific type (element, attribute, text etc.). Due to space limitations, we cannot formally and completely define our language here. However, the readers who know XPath can understand expressions which are written in our language without any problem. They can also predict what is going to be the expression for addressing this or that set of nodes. In this section, we only give an intuitive overview of our language by showing some examples of expressions for addressing nodes. This presentation is sufficient to understand the paper.

Nodes are individually or collectively addressed by the means of *path* expressions. Examples of path are the followings (these paths apply to the tree depicted in figure 1):

- /v1/v2. This path addresses the node v2.
- /v1/v2/*. This path addresses the child nodes of v2.
- /v1//*. This path addresses the descendant nodes of v1.
- /v1/v2//* | /v1/v2. This path addresses the node v2 and all its descendants (| is the union operator).

Let us now consider the tree of figure 2. Examples of path expressions which apply to this tree are the followings:

- /v1/v2[position()=1]. This path addresses the first node v2 starting from the left.
- /v1/v2/v4. This path addresses the two nodes v4 which are childs of a v2 node.
- /v1/*/v4. This path addresses the three nodes v4.

– `/v1/v6[v4]`. This path addresses the node `v6` which has the node `v4` as a child.

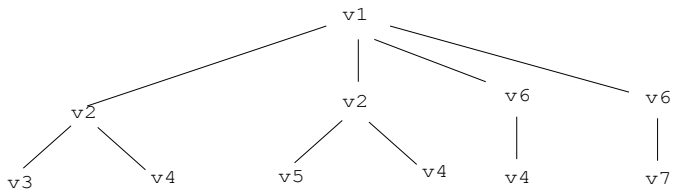

Fig. 2. Example of a Tree (2)

3. Access Control Model

In this section, we define an access control model for tree data structures. This section is organized as follows:

In section 3.1, we briefly define the concept of subject and object. In section 3.2, we first introduce the privileges which are supported by our model. We then show how we can define a security policy by labeling the source tree with some authorization attributes. In section 3.3, we define some access control algorithms which we use to enforce the security policy. Finally, in section 3.4, we suggest a security policy control language which we use to efficiently administrate the security policy.

3.1 Subjects and Objects

In our model, a subject is a user. It is not the purpose of this paper to give detailed information on how these subjects are organized. Let us mention that we could easily extend our model to take into account the concepts of user groups and/or roles.

In our model, an object is a node. As we are going to see in section 3.2, each node is associated with its own *access control list*. The node is the smallest granule of information which we can protect.

3.2 Security Policy

In our model, we define the security policy by labeling the nodes of the source tree with some authorization attributes.

An authorization attribute refers to a unique user and a unique privilege. We define an authorization attribute as follows:

An authorization attribute is a pair `<subject, privilege>`

- `subject` identifies a user.
- `privilege` takes its value from the following set: {`read`, `delete`, `insert`, `update`}

Authorization attributes are associated with nodes. An authorization attribute may be associated with several nodes and a node may be associated with several authorization attributes. The association of an authorization attribute with an object makes a *permission*. The set of authorization attributes associated with a node makes an access control list.

Let `v` be a node. Let <`s`,`p`> be an authorization attribute. If node `v` is associated with <`s`,`p`> then `s` is granted the privilege `p` on node `v`.

- if `p = read` then `s` is granted the right to see `v`.
- if `p = delete` then `s` is granted the right to delete the sub-tree whose root is `v`.
- if `p = insert` then `s` is granted the right to add a new sub-tree to `v`.
- if `p = update` then `s` is granted the right to update `v` (i.e. replace `v` by another datum).

Privileges should not be confused with *operations*. Operations need privileges to complete. Operations depend on the application. For example, in one application, we might need an *append* operation which always performs the insertion of a sub-tree at the last position (the rightmost position), whereas in another application we might need an insert operation which is able to perform the insertion of a sub-tree at any position. Both operations need the `insert` privilege to complete.

In our model what is not permitted is forbidden. If node `v` is *not* associated with <`s`,`p`> then `s` is *denied* the privilege `p` on `v`. In other words, our model always enforces the *closed* policy [Jaj97]. Note that, some other access control models (see [Dam00a] or [GB01] for instance) offer the possibility to insert explicit prohibitions in the security policy. In these models, one single authorization rule (permission or prohibition) generally addresses several subjects and/or objects. This allows the Security Administrator to *override* such an authorization rule with another more specific rule which addresses a smaller subset of subjects and/or objects. In our model, a permission is always "as specific as possible" since it addresses a unique object, a unique subject and a unique privilege.

For the sake of simplicity, we have assumed that the security policy applies to a single tree. Thanks to this assumption, we do not need to consider any kind of `create-tree` privilege. If we face a situation of having to define a security policy for a set of trees, then we just need to consider that these trees are actually all sub-trees of a virtual root. Under this assumption, creating a new tree means adding a new sub-tree to the virtual root.

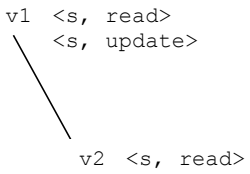

Fig. 3. Labeled Tree

Labeling the nodes with authorization attributes allows us to define security policies which are dynamic and which are updated automatically whenever the source tree is modified. For example, let us consider the tree of figure 3. In our model, nodes are labeled with authorization attributes. The security policy is dynamic and it continues to apply even if `v1` is updated and replaced by `v3`.

Another way of defining the previous security policy would be to write authorization rules separately as it is shown on figure 4.

Fig. 4. Unlabeled Tree

With this approach the security policy would not be dynamic: if `v1` is updated and replaced by `v3` then the three rules above become invalid since their `object` field has become invalid.

In the remainder of this paper we shall assume that the pseudo `write` privilege refers indifferently to the `delete`, `insert` or `update` privilege.

As we mentioned at the beginning of this section, defining the security policy is done by labeling the source tree with some authorization attributes. There is no constraint which should be respected when labeling the tree. However, we add the following two rules to *every* security policy:

- **Integrity Rule 1**: A subject is forbidden to perform a write operation on a node which he is not permitted to see (even if he has been granted the corresponding `write` privilege on this node by the labeling process).
- **Integrity Rule 2**: The view of the source tree a subject is permitted to see has to be a *pruned* version of the original source tree.

Following comments can be made about these rules:

- These two rules may conflict with some permissions which are defined by the labeling process. However, the priority of these two rules is always higher than the priority of other rules.

– Integrity Rule 1 means that we forbid *blind writes*.
– Integrity Rule 2 means that a subject is forbidden to see a node if he is not permitted to see all the ancestors of this node. We want Integrity Rule 2 to be enforced because we want users to be provided with consistent views of the source tree.

3.3 Access Control Algorithms

In this section we propose two access control algorithms. The first algorithm, which we call `View Control` algorithm, computes the view of the source tree a given user is permitted to see. The second algorithm which we call `Write Control` algorithm is able to determine whether a given user is allowed to perform a write operation on a particular node.

`View Control Algorithm`

```
1.  Let S be the user for which the view has to be computed
2.  Let L be an empty list of nodes
3.  Insert the root node into L
4.  Let R be an empty list of nodes
5.  While L is not empty Do
6.    N ← the first node of L
7.    If N is associated with the attribute <S,Read> then
8.          Append N to R
9.          Replace N in L with all the child nodes of N
10.   Else
11.         Remove N from L
```

This algorithm traverses the tree in pre-order. After the algorithm finishes, `R` contains the pre-order list of the nodes which belong to the view.

Regarding this algorithm, we can make the following comments:

– Lines 7, 8 and 9: The `if` statement is true when `N` is associated with the attribute `<S,Read>`. In this case, node `N` has to appear in the view. Therefore, node `N` is appended to `R`. Node `N` is also replaced in `L` with all its child nodes so that the while loop can start a new iteration with the first child node of `N`.
– Lines 10 and 11: The `else` statement is true when `N` is not associated with the attribute `<S,Read>`. In this case, node `N` should not appear in the final view. Therefore, node `N` is removed from `L` and is not appended to `R`. Note that `N` is *not* replaced in `L` with its child nodes. Therefore, the child nodes of `N` (and consequently all the descendant nodes of `N`) do not appear in the view. Authorization attributes which are associated with the descendant nodes of `N` are not even checked. Consequently, Integrity Rule 2 is enforced. Finally, if `L` is not empty then the `while` loop can start a new iteration with the new first node in `L`.

Let us apply the `View Control` algorithm to the sample source tree in figure 5:

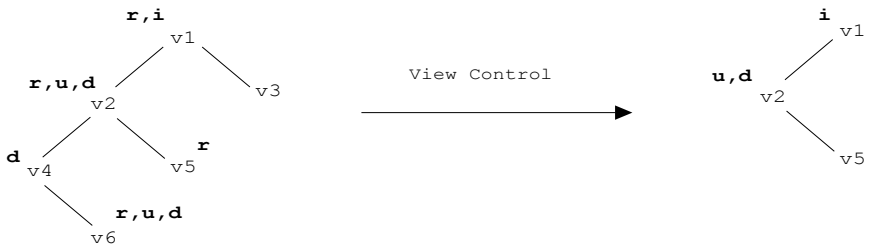

Fig. 5. View Control Algorithm

Let s be a user. Symbols **r** (respectively **i, d, u**) represent the attributes which grant read (respectively insert, delete, update) privileges to s. We can see that the sub-tree, whose root is v4, does not appear in the final view although s has been granted the privilege to read v6. Accordingly with Integrity Rule 2, the tree which is computed by the View Control algorithm is a pruned version of the source tree.

Let us now present the Write Control Algorithm.

In order to answer the question "Is user s permitted to perform an operation which requires the delete|insert|update privilege on node n ?", we apply the following algorithm:

Write (Insert,Delete,Update) Control Algorithm

```
1. Let S be the user performing the operation which requires
   the WRITE[1]  privilege on node N.
2. Use the View Control Algorithm to compute the view of the
   source tree for user S
3. If N does not appear in the view then
4.   S is forbidden to perform the operation (node unknown)
5. Else
6.   If N is associated with the attribute <S,WRITE> then
7.        S is permitted to perform the operation
8.   Else
9.        S is forbidden to perform the operation
```

Regarding this algorithm, we can make the following comment:

– Lines 2, 3 and 4 mean that we consider only the nodes the user is permitted to see. Consequently, Integrity Rule 1 is enforced. Regarding line 4, the answer user S is provided with, should not be something like "access denied" because such an answer would reveal the existence of node N to user S. The answer should rather be "node unknown" since node N does not belong to the view user S is permitted to see.

Let us consider the previous example and use the Write Control algorithm to answer the following questions:

1 Recall that WRITE stands for either INSERT, DELETE or UPDATE

– Is user s permitted to update v2 ?
  Yes. s has been granted the privilege to update v2 and v2 belongs to the view.
– Is user s permitted to update v6 ?
  No. v6 does not belong to the view.
– Is user s permitted to add a new sub-tree to v1 ?
  Yes. s has been granted the insert privilege on v1 and v1 belongs to the view. Section 3.4 explains how a newly inserted sub-tree is labeled.
– Is user s permitted to delete the sub-tree of which v2 is the root ?
  Yes. s has been granted the privilege to delete the sub-tree of which v2 is the root.

However, regarding the delete privilege, we have to comment on the following two points:

1. If user s deletes the sub-tree whose root is v2 then nodes v4 and v6, which the user is not permitted to see, are deleted.
2. If user s deletes the sub-tree whose root is v2, then node v5 is deleted although s does not hold the delete privilege on it.

Regarding point 1, it does not contradict Integrity Rule 1 which says that blind writes are forbidden. Indeed, recall that user s performs the delete operation on node v2 which belongs to the view. Now, the fact that user s indirectly deletes some data which he is not aware of, might be seen as an Integrity violation. Therefore, we could define the following Integrity Rule 3:

– **Integrity Rule 3**: A subject is forbidden to perform a delete operation on a node if this operation leads to the deletion of data he is not permitted to see.

With such a rule, s would be forbidden to delete the sub-tree whose root is v2 since this operation would lead to the deletion of v4 and v6. However, this would reveal to user s the existence of data he is not permitted to see. In other words, enforcing Integrity Rule 3 would lead to a Confidentiality violation.

Our previous delete Control algorithm emphasizes the Confidentiality and does not enforce Integrity Rule 3.

However, in some cases, we might need to enforce Integrity rule 3. Therefore we propose a second version of the delete Control algorithm which enforces Integrity Rule 3.

Delete Control Algorithm (enforces Integrity Rule 3)

```
1. Let S be the user performing the operation which requires
   the DELETE privilege on node N
2. Let T be the tree whose root is N
3. If T contains at least one node which is not associated with
   the attribute <S,READ> then
4.           S is forbidden to perform the operation
5. Else
6.           S is permitted to perform the operation
```

Regarding point 2, recall that user `s` performs the delete operation on node `v2` where user `s` holds the `delete` privilege. Now, the fact that user `s` indirectly deletes some data on which he does not hold the `delete` privilege might be seen as an Integrity violation. Therefore, we can define the following Integrity Rule 4:

– **Integrity Rule 4**: A subject is forbidden to perform a delete operation on a node if this operation leads to the deletion of data he is permitted to see but not permitted to delete.

With such a rule, `s` would be forbidden to delete the sub-tree whose root is `v2` since this operation would lead to the deletion of `v5`. In some cases, it would perfectly be acceptable to introduce Integrity Rule 4 in the security policy. Therefore, we propose a third version of the `delete Control Algorithm`:

Delete Control Algorithm (enforces Integrity Rule 4)

```
1. Let S be the user performing the operation which requires
   the DELETE privilege on node N
2. Let T be an empty tree
3. Use the View Control Algorithm to compute the view of the
   source tree for user S
4. If N does not appear in the view then
5. S is forbidden to perform the operation (node unknown)
6. Else
7.      T ← the View of the sub-tree whose root is N
8.      If T contains at least one node which is not associated
   with the  attribute <S,DELETE> then
9.           S is forbidden to perform the operation
10.     Else
11.          S is permitted to perform the operation
```

The following fourth version of the delete control algorithm enforces both Integrity rules 3 and 4. If the security policy particularly emphasizes the Integrity then this algorithm might be needed.

Delete Control Algorithm (enforces Integrity Rules 3 and 4)

```
1. Let S be the user performing the operation which requires
   the DELETE privilege on node N
2. Let T be the tree whose root is N
3. If T contains at least one node which is not associated with
   the attribute <S,READ> then
4.      S is forbidden to perform the operation
5. Else
6.      Use the View Control Algorithm to compute the view of
   the source tree for user S
7.      T ← the View of the sub-tree whose root is N
8.      If T contains at least one node which is not associated
   with the attribute <S,DELETE> then
9.           S is forbidden to perform the operation
10.     Else
11.     S is permitted to perform the operation
```

3.4 Security Policy Control Language

In this section, we define the core of a security policy control language which we can use for administrating the authorization attributes. Before presenting this language, we need to introduce a new special privilege which we call the `owner` privilege.

Let `v` be a node. Let `<s,owner>` be an authorization attribute. If node `v` is associated with `<s,owner>` then `s` holds the `owner` privilege on `v`. This means

that s is automatically granted the four other privileges (read, insert, delete and update) on v. This also means that s is allowed to grant/revoke these four privileges on v to/from other users. The owner privilege cannot be transferred.

Given a node, the user who holds the owner privilege is the user who created the node. If a user inserts a new sub-tree into the source tree then this user is granted the owner privilege (and consequently all the other privileges) on all the nodes of this new sub-tree. This defines the default labeling of a newly inserted sub-tree.

From the fact that the owner privilege is held by the user who has created the node and from the fact that the owner privilege cannot be transferred, we can easily deduce that each node has one and only one owner.

The fact that the owner privilege cannot be transferred may be seen as a limitation of our model. However, this has the advantage of providing us with a simple *delegation scheme* and minimizing the so-called *safety problem* [HRU76]. Nevertheless, in our future work, we might extend the delegation scheme of our model with the privilege of granting/revoking the "privilege to transfer privileges".

Our security policy control language includes the GRANT command and the REVOKE command. The syntax for these two commands is the following:

- GRANT <set of privileges> ON <set of nodes> TO <set of subjects>
- REVOKE <set of privileges> ON <set of nodes> FROM <set of subjects>

<set of privileges> is a subset of {read, insert, delete, update}.

<set of nodes> addresses nodes which belong to the source tree. It is an expression which is written with the language we describe in section 2.2. <set of nodes> may include nodes on which the user does not hold the owner privilege, but for such nodes, the GRANT and REVOKE commands have no effect.

<set of subjects> addresses users. In our future work, we plan to include the concept of *role* (see [San98] for a definition of role) in our model.

Let us consider the labeled source tree in figure 6:

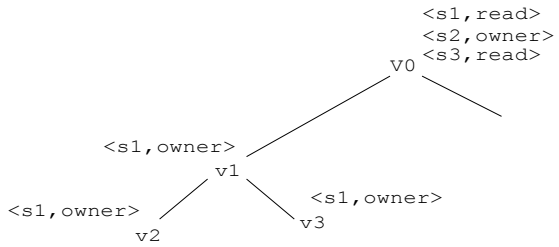

Fig. 6. Owner Privilege

User s1 is the owner of the sub-tree of which v1 is the root. Let us now consider some examples of commands which user s1 may invoke:

- GRANT read ON /v0/v1 TO s2
- GRANT read,update ON /v0/v1 | /v0/v1//* TO s3

– REVOKE update ON /v0/v1 FROM s3

With the first command, user s1 grants the privilege to read node v2 to s2. With the second command, user s1 grants the privilege to read and update node v1 and all its descendant nodes to s3. With the last command, user s1 revokes the privilege to update v1 from s3. Figure 7 shows the resulting labeled tree .

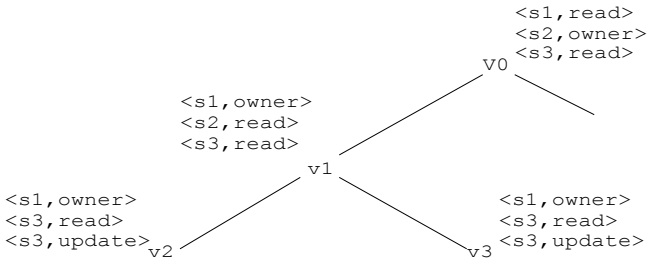

Fig. 7. Grant/Revoke Commands

4. An Access Control Model for XML Data Structures

In this section, we show through an example that we can easily apply our model to specific XML data structures.

The language we describe in section 2.2, allows us to write path expressions which can address non typed nodes of generic data structures. Now, what is important to understand is that each time we apply our model to specific data structures, we need to adapt our language so that it takes into account the fact that nodes are typed. Therefore, in this section we use the XPath language which has been specifically designed for addressing nodes of XML data structures.

Let us consider the following XML document which contains two medical files:

```
<database>
       <record id="Robert">
              <doctor>d1</doctor>
              <diagnosis>Pneumonia</diagnosis>
       </record>
       <record id="Franck">
              <doctor>d2</doctor>
              <diagnosis>Ulcer</diagnosis>
              <comment>serious case</comment>
       </record>
</database>
```

Markups like <database></database> or <record></record> are elements. Elements may have some attributes associated with them. For example the record elements which are in the document above are respectively associated with the

attribute Id="Robert" and id="Franck". Strings like d1, Pneumonia are character data. More information about XML can be found in [Bray00].

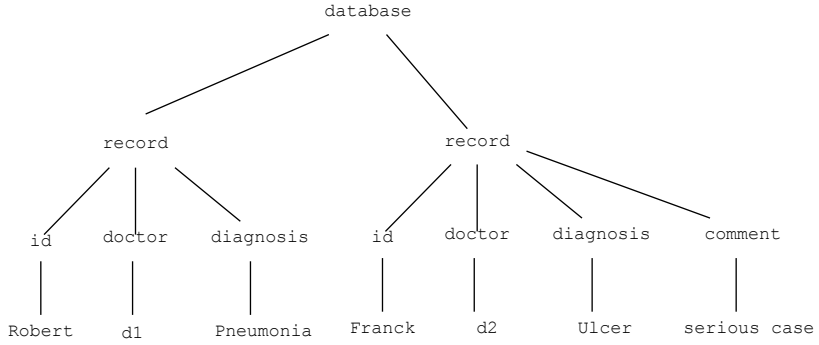

Fig. 8. Tree Representation of an XML Document (1)

There are several possible tree representations of the above XML document. Figure 8 shows one representation. Figure 9 shows another representation.

In the tree in figure 8, the granularity level is very small whereas in the tree in figure 9, the granularity level is the medical file as a whole.

Intuitively, we can choose the tree representation of our document according to the security policy which applies to the document. If the security policy addresses small portions of medical files then we definitely need to adopt the tree representation in figure 8. Now, the tree representation in figure 9 may be more convenient to define a security policy which treats each medical file as a whole.

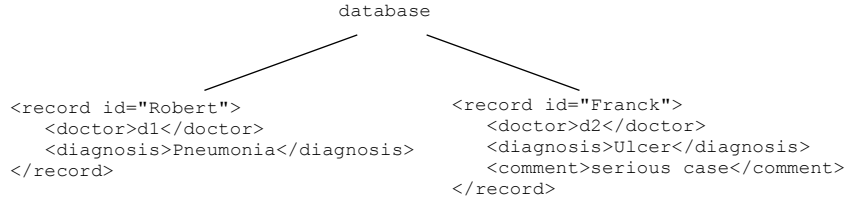

Fig. 9. Tree Representation of an XML Document (2)

The approach which consists of choosing the tree representation which best adapts to the security policy, gives flexibility to the model. However, as for XML, we do not follow this approach. We always use the XPath data model to represent an XML document, regardless of the security policy which applies to the document. Figure 10 shows the XPath tree which corresponds to our previous document. This tree is slightly different from the tree in figure 8. In the XPath tree in figure 10, each attribute and its value make a single node whereas in the tree in figure 8, each attribute and its value make two distinct nodes. Therefore, the XPath tree in figure

10 provides us with a granularity level which is almost the same as the granularity level we could obtain from the tree in figure 8.

It might look as a disadvantage to always adopt the XPath data model to represent XML documents. We reduce the flexibility of the model and we cannot define a security policy which separately addresses an attribute and its value. The main advantage of using the XPath data model actually resides in the fact that XPath has been designed to be used by XSLT [Clark99]. XSLT is a language for transforming XML documents into other XML documents. Therefore, we can use an XSLT processor for computing different views of the source tree and for modifying the source tree. We are currently designing a security processor for XML documents which implements our model. Due to space limitations, we cannot describe this security processor in this paper. However, let us mention that the core of this security processor is made of an XSLT processor.

Let us assume that doctor d1 is the owner of Robert's medical file and doctor d2 is the owner of Franck's medical file. Administrator a is the owner of the root node. In particular, administrator a is in charge of granting to other doctors the privilege to insert medical files.

In the tree in figure 10, symbols o,r,i,d,u respectively stand for owner, read, insert, delete, update. Figure 10 shows the initial labeling of this tree.

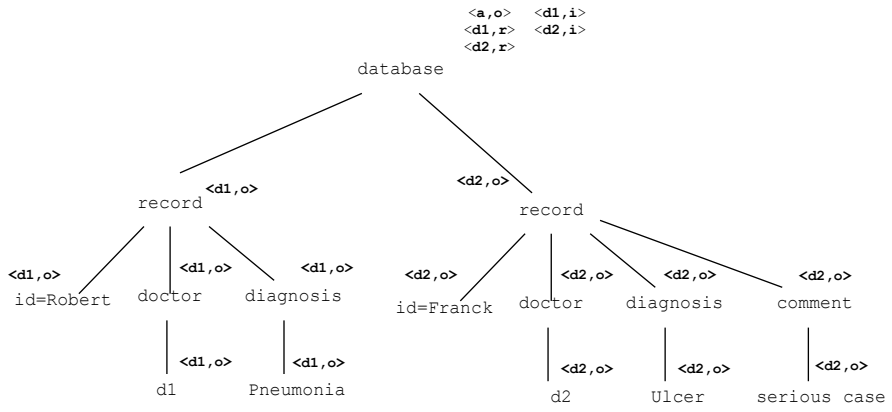

Fig. 10. Labeled XPath Tree

Let us now consider the following set of commands. Each command is prefixed with the reference of the user who types the command. We can use the XPath language to address the nodes.

d3 is another doctor and s is the secretary of doctor d2.

```
a:  GRANT read,insert ON /database TO d3

d2: GRANT read ON /database/record[doctor='d2']/descendant-or-
    self::node() TO d3,s
```

```
d2: REVOKE read ON
    /database/record[doctor='d2']/diagnosis/text() FROM s

d2: GRANT read ON /database/record[id='Franck']/descendant-or-
    self::node() TO d1

d2: GRANT update ON /database/record[id='Franck']/comment/text()
    TO d1
```

The first command says that administrator a grants to doctor d3 the privilege to insert medical files. Note that, granting the sole privilege insert would not be enough. According to Integrity Rule 1, a also needs to grant the read privilege on the root node to doctor d3.

The next command says that doctor d2 grants to both doctor d3 and secretary s the privilege to see all medical files such as doctor = 'd2' that is, all the files doctor d2 is in charge of.

The next command says that doctor d2 revokes from secretary s the privilege to see the content of the diagnosis element from all the files doctor d2 is in charge of.

The next command says that doctor d2 grants to doctor d1 the privilege to see Franck's medical file.

The last command says that doctor d2 grants to doctor d1 the privilege to update the content of the comment element of Franck's medical file.

This simple example has shown that we can easily apply our model to XML data structures. Due to space limitations, we cannot show how we apply our model to other specific tree data structures like the hierarchical file system structure. Let us however briefly mention the two following points:

- We need to slightly adapt the language we describe in section 2.2 so that it can differentiate between the nodes which are simple files and the nodes which are folders
- For implementing our model into a UNIX system, we need to patch the kernel

5. Comparison with Related Works

There are literally as many access control schemes as there are tree data structures. Historically, applications like the SNMP protocol, the LDAP protocol, WEB servers or file systems were initially developed without integrating access control facilities. For the developers of these applications, the most important was to organize the data and to define primitives for manipulating them. Later on, some access control mechanisms were added to these applications. However most of these access control mechanisms rely on models which are not clearly defined and which are specific to a particular kind of tree data structure. Very often these models are even based on the semantics of the data.

Regarding XML, the situation is almost the same. Many access control models for XML have been defined but these models are all very recent. However, XML data are

said to be *semi-structured*. This means in particular that the structure of XML trees is weakly constrained. Therefore, access control models which have been proposed for XML are certainly the most sophisticated ones and the most interesting to compare with our model.

The purpose of this section is to give an overview of all these models and to make a comparison between them and our model. However since in the previous section we have applied our model to XML data, we particularly investigate the models for XML.

View-based Access Control Model (VACM) for the SNMP Protocol

The View-based Access Control Model (VACM) [WPM99] allows the security administrator to regulate the access to the Management Information Base (MIB). The MIB is a fixed tree data structure. The privileges are not associated with the nodes of the MIB but with all its possible views. Each view is a pruned tree of the MIB tree. VACM uses a kind of Access Matrix [HRU76] to grant privileges on the views to subjects. Only the read and write privileges are supported. The write privilege corresponds to our update privilege.

Lightweight Directory Access Protocol (LDAP)

There are as many access control schemes as there are LDAP servers. The OpenLDAP [OL] server keeps the access control lists in the configuration file and uses regular expressions for the comparison of objects and subjects while Netscape and IBM keep the access control information in the directory tree as an attribute of the entries. However, the basic ideas are similar across server implementations [Sto00]. Basically, the access control lists can control access to the entire directory tree or portions of it. There is usually a default access mode, and the access control lists are used to override that default. Privileges which are supported are more or less equivalent to the privileges of our model (including the owner privilege). However, their semantics strongly depends on the nodes' type.

Web servers

Web servers organize the data into a hierarchical file system structure. Web servers like the Apache server [APA] allow us to regulate the access to some portions of the tree structure. This can be done by using some special .htaccess files. An .htaccess file is simply a text file containing Apache directives. Those directives apply to the documents in the directory where the .htaccess file is located and to all subdirectories under it as well. Other .htaccess files in subdirectories may change or nullify the effects of those in parent directories. Since the purpose of Web servers is to publish the information, basically, only the read privilege is supported.

Note that, the goal of the emerging WebDAV [WW98] (Web-based Distributed Authoring and Versioning) protocol is to leverage the success of HTTP in being a standard access layer for a wide range of storage repositories. HTTP gives them read access, while DAV gives them write access. See [WG99] for more information on WebDAV.

File Systems

Regarding file systems, it is also very difficult to talk about one model. Each file system has its own way of regulating the access to the data. Moreover, the semantics of privileges varies, depending on whether they apply to directories or to files. For example, the execute privilege of UNIX grants either the right to *enter* (?) a directory or to execute a file.

Regarding the write privilege, if it applies to a directory, then it generally encompasses our insert privilege on the directory itself (right to add some entries in the directory), our delete privilege on the entries (right to delete the entries) and our update privilege on the entries (right to rename the entries).

In most cases, each node (file or directory) is labeled with its own access control list.

Security Models for Object-Oriented Databases

There has been a considerable amount of work which has been done in the area of access control models for object-oriented databases (see [FGS94][Rab91] for instance). It would be too long to detailed information about each of them. Let us however mention that many of these models use inheritance of rights along tree hierarchies. We could show that some of these models have some similarities with our model when it is applied to object-oriented data.

Access Control Models for XML documents

Recently, several access control models for XML documents have been proposed:

- In [Dam00a], Damiani et al. propose an access control model which deals with the read privilege. An authorization 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 level of granularity is not as fine as it is in the model we propose in section 4. Indeed, a single granule of information includes an element, its content (text node) and its attributes. Conflicts between authorization rules are solved by a conflict security policy which applies principles like "the most specific object takes precedence" or "the most specific subject takes precedence". We believe that these policies which were first used in Object-Oriented environments are not well adapted to XML.

- In [Ber00a], Bertino et al. propose an access control model which deals with the `read` and the `write` privileges (the model includes also a `navigate` privilege. This is a kind of `read` privilege for objects which are targeted by links). Depending on the targeted object, the semantics of an authorization varies making sometimes difficult to understand the security policy. The `write` privilege is more or less equivalent to the `update` privilege we introduce in this paper.
- In [KH00], Kudo and Hada propose an access control model which deals with the `read`, `write`, `create` and `delete` privileges. These privileges are more or less equivalent to our `read`, `write`, `insert` and `delete` privileges. However, the level of granularity is not as fine as it is in the model we propose in section 4. Indeed, attributes or text nodes cannot be considered as granules. Therefore, they cannot be independently protected. The model offers the possibility of inserting *provisional authorizations* into the security policy. A provisional authorization grants a privilege to a subject provided he or the system completes an *obligation* which is specified by the authorization rule.
- In [GB01], Gabillon and Bruno propose an access control model which deals with the `read` privilege. The level of granularity is exactly the same as it is in the model we describe in section 4. Each node can be independently protected (element node, attribute node, text node etc.). The conflict resolution policy is based on priorities.

None of the above models associate the nodes of the source tree with access control lists. Authorization rules match the basic pattern `<subjects, objects, privilege>`.

In all these models, conflicts between the rules make sometimes difficult to predict what is going to be the output of the security policy. In the model we present in this paper, authorization attributes cannot conflict with each other and neither can Integrity Rules. Only Integrity Rules may conflict with authorization attributes. However, these conflicts are always solved in favor of the Integrity Rules. This very simple conflict resolution policy makes easy to predict what is going to be the output of the security policy.

6. Conclusion

In this paper, we propose an access control model for generic tree data structures. Our model does not rely on any assumption regarding the structure of the trees. The labeling process allows us to define dynamic security policies supporting various types of `write` privileges. The `owner` privilege along with the `grant/revoke` commands provide us with a distributed security management scheme.

We plan to extend our work in several directions:

- Access control lists does not provide a user with a clear vision of the security policy which currently applies to the nodes he is the owner. Therefore, we shall propose a solution to translate the security policy which is expressed within the

access control lists into a set of authorization rules matching the basic pattern <subjects, privileges, objects>. At any time, each user should have the possibility of being provided with the translation of the access control lists he owns. Note that a translation which is made at time t might not be valid at time t+1 since the owner may invoke new grant/revoke commands, and since the structure of the tree may change.

- We shall include the concept of provisional authorizations which Kudo and Hada define in [KH00]. A provisional authorization grants a privilege to a subject provided he or the system completes an *obligation* which is specified by the authorization rule.
- We shall include the concept of role in our model.

References

[APA] The Apache Software Foundation. http://www.apache.org/

[Ber00a] 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. 2000.

[Ber00b] 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.

[Bray00] T. Bray et al. "Extensible Markup Language (XML) 1.0". World Wide Web Consortium (W3C). http://www.w3c.org/TR/REC-xml (October 2000).

[CD99] J. Clark and Steve DeRose . "XML Path Language (XPath) Version 1.0". World Wide Web Consortium (W3C). http://www.w3c.org/TR/xpath (November 1999).

[Dam00a] 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.

[Dam00b] 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.

[FGS94] Eduardo B. Fernández, Ehud Gudes, Haiyan Song: A Model for Evaluation and Administration of Security in Object-Oriented Databases. TKDE 6(2): 275-292(1994)

[GB01] Alban Gabillon and Emmanuel Bruno. "Regulating Access to XML documents". Fifteenth Annual IFIP WG 11.3 Working Conference on

Database Security. Niagara on the Lake, Ontario, Canada July 15-18, 2001.

[HRU76] Harrison, M.H., Ruzzo, W.L. and Ullman, J.D. Protection in Operating Systems. Communications of ACM 19(8):461-471 (1976).

[Jaj97] 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.

[KH00] 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.

[OL] OpenLDAP. http://www.openldap.org/

[Rab91] F.Rabitti, E.Bertino, W. Kim and D. Woelk."A model of authorization for Next Generation Database Systems.". ACM TODS vol 16, n°1. Mar 1991.

[San98] R. Sandhu. "Role-Based Access Control". Advances in Computers. Vol 48. Academic Press. 1998.

[Sto00] E. Stokes, D. Byrne, B. Blakley, and P. Behera. "Access Control Requirements for the Lightweight Directory Access Protocol". RFC 2820, May 2000.

[WG99] Jim Whitehead, and Yaron Y. Goland. "WebDAV: A network protocol for remote collaborative authoring on the Web". European Computer Supported Cooperative Work (ECSCW'99) conference.

[WPM99] B. Wijnen, R. Presuhn, and K. McCloghrie. "View-based Access Control Model for the Simple Network Management Protocol". RFC 2575, April 1999.

[WW98] E. James Whitehead Jr, and Meredith Wiggins. "WebDAV: IETF Standard for Collaborative Authoring on the Web". IEEE Internet Computing, September/October 1998.

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