Issues in Information Systems
Volume 14, Issue 1, pp.224-233, 2013

MAINTAINING CONFIDENTIALITY IN MULTILEVEL XML

Lawrence Kerr, University of Idaho, lawrence.kerr@vandals.uidaho.edu,
Jim Alves-Foss, University of Idah , jimaf@uidaho.edu

ABSTRACT

Using XML in environments with multiple levels of security presents a number of challenges. From a confidentiality standpoint, users must be prevented from accessing data elements above their clearance, while at the same time be prevented from leaking data to lower levels through write operations. This work employs a security tagging approach to ensure each element in a document is appropriately labeled for a mandatory access control environment. Operations include not only read, but insert, update and delete, all with sensitivity to potentially polyinstantiated or dominating elements.

Keywords: XML, mandatory access control, schema, polyinstantiation

INTRODUCTION

XML is a W3C standard [1] that has become a leading standard for the storage and exchange of data [2]. The extensible nature of XML lends it to a multitude of uses ranging from simple configuration files, to the underlying structure for a variety of file formats for popular office application suites. Office Open XML is one such format used with Microsoft Office products, based on the Ecma-376 Standard [3]. It defines various schemas, elements, and attributes that make up a word processing, spreadsheet, or presentation file. Adherence to the standard allows the creation of these types of files with the hopes that they will interoperate with other applications.

One scenario in which XML might be employed is a mandatory access control (MAC) environment in which users of various levels access data stored at various sensitivities. Here a user may want to search for information in a single document or collection of documents. A user presents a clearance and is allowed to operate on only a set of the elements among the data source based on the relative sensitivity of each element. In this way, an element containing a variety of sensitivities among its children can be shared between a low clearance user and a high clearance user, with each user only receiving a view of allowed elements.

This paper presents an approach to providing access to XML data in a MAC environment while maintaining confidentiality across read, insert, update, and delete operations. The balance of the paper is organized as follows: the next section provides a quick overview of the structure of XML and XML Schema Definitions, followed by a summary of related work. Next is a discussion of mandatory access controls including the principle of polyinstantiation and how it might affect XML. Later sections include a description of when each operation is granted or denied, brief discussion of a prototype system implementing each of these operations, and concluding remarks.

XML AND SCHEMAS

The basic unit of XML is the element, which serves to distinguish various data items from each other. An element consists of a starting and ending tag, and may contain data or other elements. Attributes serve to provide further information, specific to an element and appear within that element's starting tag. An XML document consists of a number of elements, nested as descendants of a root element. A well-formed XML document is one consisting of only properly nested tags. This document is typically considered a tree, starting at the root element, with other elements being nested within the root. The elements without any children are the leaves.

```
<?xml version= “1.0"?>
<missions>
    <mission id= “123">
        <starship>Enterprise</starship>
        <target>Vulcan</target>
        <class>UNCLASSIFIED</class>
        <task>Research</task>
    </mission>
    <mission id= “125">
        <starship>Reliant</starship>
        <target>Romulan</target>
        <class>TOPSECRET</class>
        <task>Intelligence</task>
    </mission>
    <mission id= “126">
        <starship>Enterprise</starship>
        <target>Vulcan</target>
        <class>SECRET</class>
        <task>Diplomatic</task>
    </mission>
</missions>
```

Figure 1. Example XML

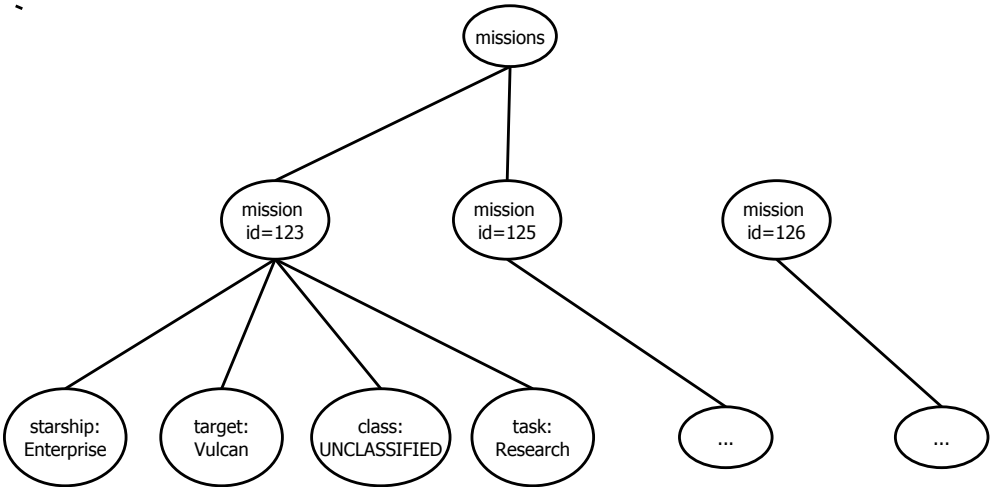

Figure 2. XML tree structure

In
, the <missions> tag is the root of the XML tree. All other elements are nested within the <missions> opening tag and </missions> closing tag. Each <mission> element bears an id attribute and a collection of child elements. These child elements have their own child elements, and represent the leaves of the tree (Figure 2).

XML allows a great deal of flexibility by giving the document creator the ability to define his or her own element and attribute types. Varying degrees of order can be imposed on the structure and type of these tags by following a schema. A schema defines the allowed elements for a document, as well as their structure, order or allowed number of occurrences. A number of schema formats are available for XML, our work employs XML Schema Definition (XSD), a W3C standard for defining XML schemas. An XSD document is itself written in XML as a collection of elements. With an XSD schema, the schema defines not only what types of elements can be allowed in a document, but user-defined types may be declared, domains on type values may be defined, and values can be strongly typed. Figure 3 shows an example of an XSD schema for the XML instance in

```
<?xml version= “1.0"?>
<xs:schema xmlns:xs=“http://www.w3.org/2001/XMLSchema">
    <xs:element name= “missions">
        <xs:complexType>
            <xs:sequence>
                <xs:element maxOccurs= “unbounded" name= “mission">
                    <xs:complexType>
                        <xs:sequence>
                            <xs:element name= “starship" type= “xs:string" />
                            <xs:element name= “target" type= “xs:string" />
                            <xs:element name= “class" type= “xs:string" />
                            <xs:element name= “task" type= “xs:string" />
                        </xs:sequence>
                        <xs:attribute name= “id" type= “xs:integer" use=“required" />
                    </xs:complexType>
                </xs:element>
            </xs:sequence>
        </xs:complexType>
    </xs:element>
</xs:schema>
```

Figure 3. Example XSD based schema

RELATED WORK

A great deal of work has been conducted on access control strategies for XML data. Kuper, Massacci, and Rassadko propose an approach based on views [4]. A view is described by a schema that identifies those portions of a document are to be in a users’s view. Bertino et al. [5] focus on the parent-child nature of the XML tree, with unlabeled elements or attributes inheriting security information from the parent element. This is concept is extended to the inheritance of security information from a schema, although instances of XML based on this schema are allowed to override the schema defined labels.

For dissemination of XML data while preserving confidentiality, Bertino and Ferrari [6] encrypt different sections of the document using different keys based on the sensitivity of the section. A particular user will have the key to only certain portions of the document and be unable to read other portions.

Jo and Chung propose an XML Access Control System (XACS) with a focus on web usage of XML data [7]. A user searches through an XML document, and the resulting transmission includes only allowed portions of the underlying XML data source. Mahfoud and Imine cite a deficiency in the literature of projects attempting to address any type of access to XML data beyond reading, and propose a solution to updates based on XQuery and DTD schemas [8].

Zhang et al. describe a role-based approach to securing XML data, and make a critical observation [10] . In previous research, a Document Type Definition (DTD) style schema is used, which introduces a number of weaknesses including requiring an additional parser as the DTD schema is not itself XML, lack of ability to place constraints on values, and inability to define types. They prefer instead to employ XSD style schemas, which have been shown to be more expressive and feature-rich, a finding that is echoed in the XML literature [9].

Reviews these and other of schemes for securing XML documents in the literature we found that some propose a schema based approach, but the vast majority of employ DTD based schemas. Our work introduces an approach that instead uses the more flexible and expressive XML Schema specification, imposing mandatory labeling at the element level. Of those that have attempted the use of XML Schema, none deal with MLS environments with mandatory security policies, which we have addressed.

MANDATORY SECURITY

This work employs a security policy for XML similar to the Bell-LaPadula security model [10] in a mandatory access control (MAC) environment. In a MAC environment, each user is assigned to one of a number of totally ordered security clearances and some number of compartments. Data items are labeled and compartmentalized in a similar manner. Access by a user to a particular data item is granted or denied based on the relation between the user's assigned clearance (including all compartments) and that of the target data item. Although the examples here are based on a military style multi-level security, it is straightforward to define other security models including corporate, health care and academic; each with security policies and regulations that require protection of data confidentiality and integrity.

The totally ordered set of clearances used here has been arbitrarily defined, though resembles clearances one might find in military or government policies. With the total ordering, the lowest level is U (unclassified), increasing to C (confidential), S (secret), and finally the highest clearance TS (top secret). Compartments have no ordering, and thus no comparison is possible between two compartments. Rather, the compartments are treated as a set, with comparisons made between entities based on the relative contents of sets. For this work, we adopt a system including only three compartments: red, green, and blue. A user or data item may be assigned to zero or more compartments in any combination. For a particular entity, we define a function *class(K)* which returns the full security description of the entity K. A user K_u, for example, might have a secret clearance and belong to the red and blue compartments. The result of *class(K_u)* then is <S,{red, blue}>.

The "dominates" relation describes the difference between the classifications of two entities within this system. A classification K_1 is said to dominate a classification K_2 if K_1 meets two conditions: 1) the label or clearance associated with K_1 is greater than or equal to that of K_2 and 2) the set of compartments K_1 belongs to is a superset of the compartments of K_2. If both of these conditions are met, we say K_1 dominates K_2, denoting it as such:

$$class(K_2) \sqsubseteq class(K_1)$$

Another measure of dominance, "strictly dominates" imposes a stronger condition on the relation between the clearances of the compared entities. For K_1 to strictly dominate K_2, either K_1 holds a clearance that exceeds K_2 and belongs at least to all the compartments of K_2, or K_1 holds an identical clearance to K_2 while belonging to a true superset of compartments of K_2. For example, <S,{red, blue}> strictly dominates each of <S,{}>, <S,{red}>, and <S,{blue}>, but only dominates <S,{red, blue}>. Also, <S,{red, blue, green}> and <TS,{red, blue}> both strictly dominate <S,{red, blue}>.

There may arise situations in which two entities have classifications that cannot be readily compared using the dominates relationship. These entities are called non-comparable. Non-comparability arises when each entity belongs to a set of compartments that contains some item exclusive to that entity. In this situation, we cannot identify either entity as possessing a superset of the other's compartments, and thus cannot state that one dominates the other. An example of non-comparable classifications is <S,{red, blue}> and <S,{red, green}>. Here neither classification dominates the other as no superset of compartments is found in either entity.

The Bell-La Padula security model uses the concept of the dominates relationship for subjects (users) and objects (data items), defining two properties that must be maintained. The first is the simple security property, which states that in order for a subject to be granted read access to an object, the classification of the subject must dominate the classification of the object. In other words, a particular user is allowed to read anything bearing a classification no higher than that associated with the user. This prevents a user from viewing information that is more sensitive than he or she is allowed.

The second property, the *-property, deals with write access to objects. Here, to avoid a user writing potentially sensitive information to a lower level, writes to an object are typically only allowed if the classification of the object dominates that of the requesting subject. This prevents a downward flow of sensitive information – a user can only write to objects at least as sensitive as his or her classification. This work adopts a stronger *-property, where writes are only granted when the classification of the subject matches that of the object - subject K_u can write to object K_o if and only if:

$$class(K_u) \sqsubseteq class\ (K_o) \wedge class(K_o) \sqsubseteq class(K_u)$$

Imposing this model on XML, each element in a document will bear an attribute representing a security label, with another attribute representing the set of compartments. Each user will be assigned a clearance and a set of compartments and will be granted or denied various operations based on the *dominates* relationship between the user and the element. Each element will have a classification that dominates its ancestors, so data can only become increasingly sensitive (if at all) as one traverses from the root of the tree to the leaves. A third attribute is employed here, the *preserve* attribute, which is used during write operations that might impact elements that strictly dominate the operating user. The presence of these attributes in an XML document is imposed by an XML Schema document that each data document must conform to. This schema will determine the structure and content of various elements in the document, as well as ensure that the necessary attributes to support the security policy are always included.

An example schema is shown in **Figure 4**. Here the *securitLbl* type is defined as a string value, but is restricted to one of four values "U", "C", "S", or "TS." This ensures when this type is used in an XML instance, it will bear only one a valid label. Any attempt at using this type with a different value outside these four will fail. This type is employed as an attribute for the ship complex type. Each of the three attributes label, compartment, and preserve (compartment and preserve are defined similarly to the label) is required by the schema. Thus any XML instance based off this XSD must include a valid label, compartment, and preserve element each time a ship type is used. Be validating a document against the schema, we can ensure that the labeling of elements occurs as desired.

```
<xs:simpleType name="securityLbl">
    <xs:restriction base="xs:string">
        <xs:enumeration value="U"/>
        <xs:enumeration value="C"/>
        <xs:enumeration value="S"/>
        <xs:enumeration value="TS"/>
    </xs:restriction>
</xs:simpleType>
...
<xs:complexType name ="ship">
    <xs:simpleContent>
        <xs:extension base="xs:string"/>
            <xs:attribute name="label" type="securityLbl" use="required"/>
            <xs:attribute name="compartment" type="securityComp" use="required"/>
            <xs:attribute name="preserve" type="preserve" use="required"/>
        </xs:extension>
    </xs:simpleContent>
</xs:complexType>
```

Figure 4. Tagging XSD excerpt

Polyinstantiation in XML

The term *polyinstantiation* was coined in research developing multilevel secure relational databases [11]. Polyinstantiation describes a situation in which information related to a particular primary key exists in the database under multiple security classifications. An example follows in Figure 5. Each field in the relation is assigned a classification. The *starship* attribute is the apparent key – the attribute which appears to the user to be the primary key. However, in order to allow the second tuple, an effective key is generated by combining the apparent key with the highest classification of a field in the tuple, the tuple classification. Creating and using this as our key allows for the existence of both tuples. Figure 5 shows an example with two tuples. Here the apparent key is the starship name. A low clearance user (at the U level) views this relation as containing only a single tuple, tuple 1.

Tuple	*Starship*	*Mission*	*Target*	*Tuple class*
1	Enterprise U	Explore U	Talos U	U
2	Enterprise U	Explore U	Rigel S	S

Figure 5. Polyinstantiated tuple

Polyinstantiation in XML relies not on any concept of an apparent key, as in relational databases, but rather the parent of the element and the distinguishing security classification. Polyinstantiated elements that appear will have the same parent, and thus a similar path from root to parent. What distinguishes one from the other is the security classification of each element. We define polyinstantiation here in terms of XML to maintain as much of such relationships between elements in the presence of data modification. With no consideration of polyinstantiation (or in more general terms, any other strictly dominating descendants), lower level users are presented an opportunity to manipulate higher level data without knowing of its existence or the effects of their actions. If an insertion is allowed without regard for higher level polyinstantiated elements, we stand to lose these items through the update operation. To avoid this we impose on data manipulation operations the necessity to preserve higher level data that may exist in descendants of the operation's target element.

```
<mission id= "0" label= "C" preserve="PRESENT" compartment= "RED">
    <target label= "C" preserve="PRESENT" compartment= "RED">Ceti Alpha VI</target>
    <starship label= "C" preserve="PRESENT" compartment= "RED">Reliant</starship>
    <type label= "C" preserve="PRESENT" compartment= "RED">Scientific</type>
    <type label= "C" preserve="PRESENT" compartment= "REDBLUE">Recovery</type>
    <type label= "S" preserve="PRESENT" compartment= "RED">Spying</type>
</mission>
```

Figure 6. Polyinstantiated child elements

Using this tagging scheme, we describe four operations that are allowed on XML data: read, insert, delete and update. All operations rely on the path from the root element to the location in the tree where the operation is to take place. Figure 6 shows the polyinstantiation security labels for an example system.

OPERATIONS ON MULTILEVEL XML

Our model allows a range of read, write (including insertion of new elements and updating of element content), and element or tree removal operations. While read access, is straightforward on its own, data modification operations introduce a number of concerns, most notably we must deal with not only the simple security policy to ensure that the operations deal only with allowed elements, but also the *-property as the changes must not write information to lower classifications.

Of considerable importance to these operations is the classification of the path between an element and the root of the XML tree, or some other ancestor. For all types of access, a user must dominate the classification of each element along the path from the root to the target element. This ensures that the user only operates on elements in the user's view. Elements that strictly dominate the user are not in the user's view, and thus should not be allowed as targets of operations – under the simple security property the user should not know that these elements exist. In determining if an operation on an element *e* is granted, a function *maxp* is used. This function determines the highest classification along a path from *e* to the root element:

$$maxp(e) = \begin{cases} class(e) & \text{if } e \text{ is the root} \\ max\left(class(e), maxp\left(parent(e)\right)\right) & \text{otherwise} \end{cases}$$

Here the function *max* determines the dominating of two classifications, and the *parent* function returns the immediate ancestor of a given element. While *maxp* follows a single path up the XML tree to the root element, the *maxd* function accepts an element, and returns the maximum classification found in the sub tree rooted at the element. This function becomes critical when changes to existing data are performed, or new elements are inserted.

Read operation

The read operation reflects the simple security property – for a read operation to be granted, a subject must possess a classification that dominates not only the element being read, but the entire path from the root to that element. This ensures that the data is read only when a direct ancestral line to the root exists. If this dominates relationship is violated, we know that either the element or some ancestor dominates the classification of the requesting user. In either case, we must not grant the read operation. This leads to a *canRead* function as such, where *u* is the classification of the requesting user and *e* is the element to be read:

$$\boldsymbol{canRead(u,e) = maxp(e) \sqsubseteq class(u)}$$

While this suffices in read-only scenarios, the addition of further operations complicates read. Due to the possible existence of child elements at a higher sensitivity, we must consider the preserve attribute as well. The preserve attribute will only affect the outcome of the read operation when the classification of both the requesting user and the element to be read match. Expanding the *canRead* function to account for these scenarios results in:

$$canRead(u,e) = \big(maxp(e) \sqsubseteq class(u)\big) \wedge \neg\Big(\big(class(u) = class(e)\big) \wedge preserve(e)\Big)$$

In this way, the value of the preserve element only affects the granting or denying of a read operation when the classifications of the user and element are identical. Any other combination of user and element classifications will ignore the preserve element, and the granting of the operation lies exclusively on the classifications, not the preserve element.

Insert operation

The insert operation encompasses not only the simple security policy, but the *-property as well. The insertion of an element only makes sense when the user has read access to where the insertion is desired. The content of the inserted data must also be taken into consideration. A user must not be allowed to make the insertion if the new element introduces data containing incompatible classifications. To ensure no new classifications are introduced, the insertion candidate element *n* is examined for any classifications that do not match the classification of the requesting user *u*. If any are discovered in the new element, the insertion is denied.

$$canInsert(u,e,n) = canRead(u,e) \wedge \big(class(u) = maxd(n)\big)$$

Thus a user may insert a child element to any readable

, a <C,{}> user would be granted an insert operation with the root as the target. Any other element in this tree will not be allowed as the user is not granted read access to any element but the root. Provided the new element the user attempts to insert passes the *maxd* test, the insertion proceeds and a new child is inserted under the root element.
, a <C,{}> user would be granted an insert operation with the root as the target. Any other element in this tree will not be allowed as the user is not granted read access to any element but the root. Provided the new element the user attempts to insert passes the *maxd* test, the insertion proceeds and a new child is inserted under the root element.

Deletion operation

The deletion operation selects a single element and deletes it and any sub tree rooted at that element. Of consideration here is the need to preserve elements that are among the descendants of the selected element but are of a higher classification.

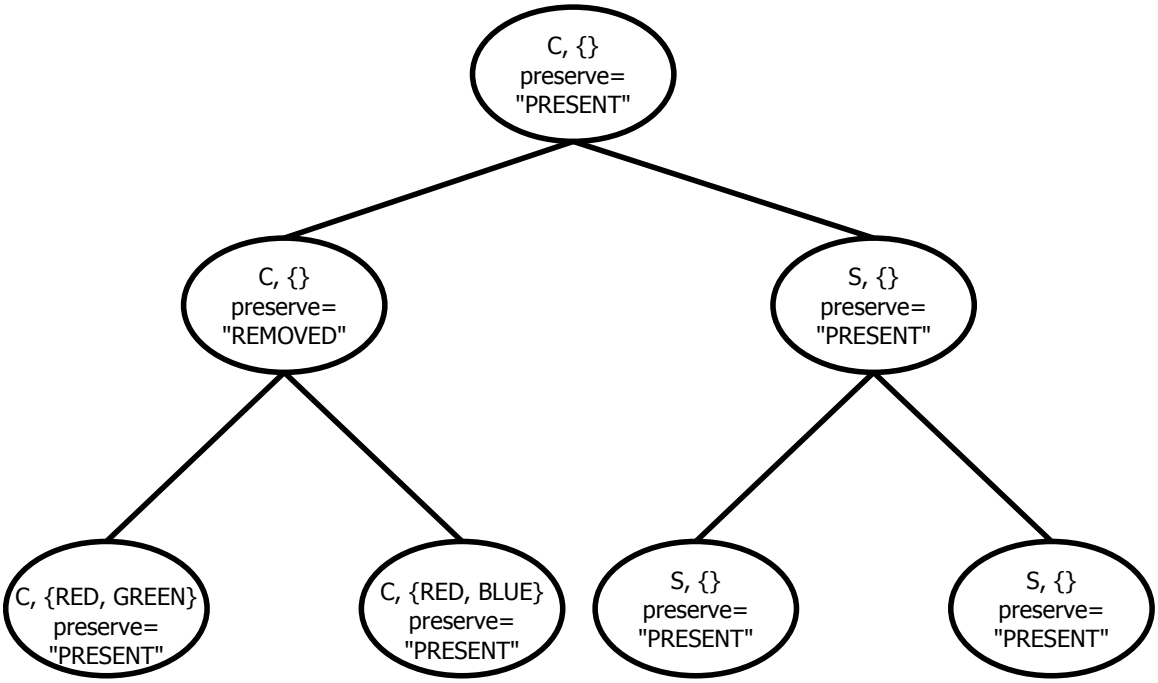

Figure 7. Read by <C,{}> only reads root element

Allowing a deletion operation is relatively simple. Only two checks are required: 1) the classifications of the user and the target element are equal and 2) a "present" preserve attribute is at the target element. If both of these conditions are met, the deletion is allowed. This will ensure that no user is able to remove items of a higher or lower level, but also previously removed elements.

$$canDelete(u,e) = \big(class(u) = class(e)\big) \wedge preserve(e)$$

$$\text{where } preserve(e) = \begin{cases} true & \text{if preseve atrribute} = \text{"present"} \\ false & \text{otherwise} \end{cases}$$

On a deletion the sub tree rooted at the target element is

1. There are no descendants of the target element. The element is simply removed.
2. The entire sub tree contains elements that all bear the same classification as the requesting user. In this case, the entire sub tree is removed.
3. The sub tree contains elements whose classifications are strictly comparable to each other. In this case, the element and sub tree are not removed, rather the classification of each element matching the user is upgraded to the lowest strictly dominating classification. Should any sub tree contain only a matching classification it can be removed. In this manner, the target element is removed from the user's view, giving the impression of deletion.
4. The sub tree rooted at the target element contains descendant elements of non-comparable classifications. In cases such as this, a single classification to upgrade to is not available, and we must consider an alternative to simply upgrading. Here the preserve flag is set to "removed" on the target element, and any descendants that share the target's classification. This removes the sub tree from the user's read view, while still preserving the higher level descendants.

once again, we see that a deletion operation has already taken place. The left child of the root has already been deleted by a <C,{}> user. Due to the existence of the non-comparable children of this node, however, we are not

able to remove it, so the preserve element was changed to remove it from the user's view. This user has the option of performing a delete on the root element, but as in the prior deletion, it will result only in a changing of the preserve attribute because <S,{}> is not comparable to either <C,{RED, GREEN}> or <C,{RED,BLUE}> so a single upgrading classification cannot be determined.

once again, we see that a deletion operation has already taken place. The left child of the root has already been deleted by a <C,{}> user. Due to the existence of the non-comparable children of this node, however, we are not able to remove it, so the preserve element was changed to remove it from the user's view. This user has the option of performing a delete on the root element, but as in the prior deletion, it will result only in a changing of the preserve attribute because <S,{}> is not comparable to either <C,{RED, GREEN}> or <C,{RED,BLUE}> so a single upgrading classification cannot be determined.

Update operation

For simplicity, the update operation is treated as a deletion of the original content followed by an insertion of the changes. In determining if an update access is granted, we simply combine the rules *canInsert* and *canDelete*:

$$canUpdate(u, e, n) = canInsert(u, e, n) \wedge canDelete(u, e)$$

This operation first performs any deletion operations that may be necessary. Children that may be lost during the update can be removed or upgraded as per a deletion operation, followed by an insertion of the "updated" element content as a new child.

Prototype

A prototype system has been developed that follows this security model. The prototype uses a server attached to a file store that houses an XML document containing mission information labeled at a variety of levels, similar to that found in Figure 6. Users access the server by first presenting their clearance and compartments. The user enters an operation to be completed, along with an XPath expression that describes which elements are of interest. The server evaluated the XPath expression, returning only those portions of the result which the user is allowed under the security model.

Testing of the prototype includes four types of data: 1) data that strictly dominates the user, 2) data that is strictly dominated by the user, 3) data that matches the users clearance and compartments, and 4) non-comparable data. Each operation (read, insert, update, delete) was tested in the presence of each of these types. Testing consisted of submitting various XPath expressions to different sets of data, and comparing the result with the expectation of what should be allowed or denied based on the security model. No leaking of information from higher levels to lower were observed in testing. Data changing operations worked as expected from the perspective of the user and successfully preserved dominating descendants where necessary.

CONCLUSIONS

In addition to providing a security mechanisms using a more expressive and flexible scheme, a further contribution of our work is the sensitivity to the concept of polyinstantiation. While it is a prevalent subject in the MLS relational database literature, it is not addressed in the context of XML security, a gap that we feel needed to be filled.

Maintaining of confidentiality in XML in a MAC environment presents a number of challenges. Among these challenges, preserving higher level data while allowing lower level users to perform data modifying operations is critical as it allows a truly multilevel document as an element can safely allow for children of varying sensitivity.

Continuing work in this area expands into introducing these access control concepts into some of the other use cases for XML, such as office suite document formats. But leveraging this scheme against existing formats, we allow a range of users to work with sensitive information through a familiar interface while ensuring confidentiality requirements are met.

ACKNOWLEDGEMENTS

This material is based upon work supported by the National Science Foundation under Grant No. DUE-1027409. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the National Science Foundation.

REFERENCES

1. T. Bray, J. Paoli, C. M. Sperberg-McQueen, E. Maler, and F. Yergeau, "Extensible Markup Language (XML) 1.0 (Fifth Edition)." [Online]. Available: http://www.w3.org/TR/REC-xml/. [Accessed: 27-Oct-2011].
2. E. Damiani, M. Fansi, A. Gabillon, and S. Marrara, "A general approach to securely querying XML," *Comput. Stand. Interfaces*, vol. 30, no. 6, pp. 379–389, 2008.
3. "Standard ECMA-376." [Online]. Available: http://www.ecma-international.org/publications/standards/Ecma-376.htm. [Accessed: 30-Jun-2013].
4. G. Kuper, F. Massacci, and N. Rassadko, "Generalized XML security views," *Int. J. Inf. Secur.*, vol. 8, no. 3, pp. 173–203, 2009.
5. E. Bertino, S. Castano, E. Ferrari, and M. Mesiti, "Specifying and enforcing access control policies for XML document sources," *World Wide Web*, vol. 3, no. 3, pp. 139–151, 2000.
6. E. Bertino and E. Ferrari, "Secure and selective dissemination of XML documents," *Acm Trans. Inf. Syst. Secur. Tissec*, vol. 5, no. 3, pp. 290–331, 2002.
7. S.-M. Jo and K.-Y. Chung, "Access Control Mechanism for XML Document," in *Proceedings of the International Conference on IT Convergence and Security 2011*, 2012, pp. 81–90.
8. H. Mahfoud and A. Imine, "A General Approach for Securely Updating XML Data," in *International Workshop on the Web and Databases (WebDB 2012)*, 2012.
9. D. Lee and W. W. Chu, "Comparative analysis of six XML schema languages," *Acm Sigmod Rec.*, vol. 29, no. 3, pp. 76–87, 2000.
10. D. E. Bell and L. J. La Padula, "Secure computer system: Unified exposition and Multics interpretation," 1976.
11. D. E. Denning and T. F. Lunt, "The SeaView security model," 1988.