CS437 XML Webdata Management

Semester Project: Secure XML Publishing

Phase 3 Report

Aaron Powers

Tom Guenther

Matt Nuckolls

Abstract

Secure Publishing of XML data to the web to provide access for multiple users is necessary. Certain users may need access to this data, while other users may not be allowed access. Additionally, publishers need the ability to update documents that have already been published. This paper proposes a method for achieving this, utilizing a publish/subscribe method and xpath querying for partial updates and user access level verification. The first section of this paper introduces the problem statement. Next, the problem is explored further by discussing XML and its security shortcomings. The third section describes the proposed solution to the problem, and work completed thus far to implement the solution. The fourth section describes potential data to user for testing as well as methods to verify that the model is functioning as intended. The final section will present a survey of other research that has been done in the same area.

Introduction

Review of Other Ideas

Currently, there is a large amount of work being done in the area of Secure XML Publishing. Mohamed Nabeel and Elisa Bertino published a paper titled “Secure Delta-Publishing of XML Content”. In this paper the authors propose a method of secure XML publishing that supports small incremental changes without the publisher needing to republish the entire paper. Their method also allows that the third party publishers do not need to be trusted due to encryption used by the original publisher.

Erwin Leonardi, Sourav S. Bhowmick, and Mizuho Iwaihara published a paper titled “Efficient Database-Driven Evaluation of Security Clearance for Federated Access Control of Dynamic XML Documents”. The authors of this paper propose a method of Secure XML distribution that utilizes a “policy enforcer” and “data provider”. Essentially, the data provider stores the XML documents as well as the access policy of those xml documents. A user can query the policy enforcer for data from the XML documents which the policy enforcer passes on to the data provider. The data provider runs the query then returns to the policy enforcer the documents found in which the user is allowed to access.

A sophisticated approach to the problem of secure xml publishing was presented in a paper by Mohammad Ashiqur Rahaman, Yves Roudier, and Andreas Schaad titled “A Publish/Subscribe Model for Secure Content Driven XML Dissemination”. This paper presents a very detailed example of a publish/subscribe model and how it could be used in a real world scenario between different organizations. This method ensures that the XML schema as well as the data itself are both confidential, ensures the integrity of transmitted data, and removes the coupling of publishers with subscribers by inducing a dissemination layer between the publishers and subscribers. The method also ensures the XML data cannot be read by the dissemination layer. In this model a publisher first establishes authorization policies that are enforced by the dissemination network then a user sends a subscription request along with his credentials to the dissemination network. When a publisher publishes his encrypted XML document he annotates it with information regarding the “concept” of the document and the dissemination layer forwards the document to the subscribed users. In this model, users only get portions of the document they are authorized for; they can get them automatically by distribution from the dissemination layer; there is confidence in the integrity of the data; and, the data is kept hidden from anyone that is not authorized to see it. This model seems to solve many of the problems surrounding secure XML publishing as well as providing efficiency within the dissemination network layer. However, it does not appear that this model supports incremental updates from publishers.

Another method has been proposed by B. Carminati, E. Ferrari, and E. Bertino in their paper “Secure Third Party Distribution of XML Data”. This is yet another method that provides confidentiality, integrity, and authenticity in secure XML data publishing; but, it also provides completeness. Completeness is provided by ensuring that the users are receiving every portion of the document they are allowed to receive based on their access control policy. Like the method proposed by Rahaman, Roudier, and Schaad, this method does not require that the third party publishers be trusted. However, data still remains confidential. This is achieved by the user receiving keys directly from the data publisher.

Problem Statement

Storing data in an XML format is convenient due to the fact that it does not follow a specific schema. The user has the responsibility of defining their own schema which makes data storage in XML very flexible. Data stored in an XML format can be published on the web for distribution purposes. This allows for a convenient method of distributing data to other individuals interested in the data. Publishing XML data on a network is a convenient way to distribute data, however security obviously is a concern. The person publishing may want to allow an individual, or individuals, to access some of their documents or even parts of their documents, while preventing everyone else on the network from accessing their data. A secure method of publishing XML data is needed to prevent access to all portions of a document that an individual is not allowed to access. Additionally, an individual should have the ability to subscribe to xml documents, or document topics. Subscribing to a document would allow an individual to be “pushed” updates as they are published so they do not have to periodically search for new or updated data. Additionally, a publisher may need to update a document that has already been published. A method of providing a partial update is needed to allow a publisher to update a portion of his document without republishing the entire document. Another feature that would be convenient is for newly published partial updates to be automatically pushed to all authorized subscribers to the document.

Analysis of the Problem

XML is a flexible markup language. It is intended to be both machine readable as well as human readable. Though it has a specific structure defined by tags, elements, attributes, etc. it has no specific format nor requirement for tag sets. This flexibility makes it versatile but also vulnerable. Since XML is simply a markup language, it has no intrinsic security mechanisms. Therefore, XML data storage and transmission security must be handled externally to the XML content. XML document encryption is necessary for secure storage and transmission. Furthermore, XML document access must also be handled externally. Again, XML has no intrinsic access controls. Since access and transmission control must be handled externally to the XML document (implicitly: the XML parser), a “3rd party” application may be required to authorize access over a secure transmission protocol such as SSL (secure socket layer).

Proposed Solution

To solve the problem of securely publishing XML data on the web to provide access to authorized users, we propose to implement a simple publisher/subscriber model and an interface by which users can access the system. We intend to store data externally using SQL Lite. Stored data will consist of published XML data, user access level information, and user subscription information. By maintaining this data, newly published or updated documents can be pushed to all authorized subscribed users automatically. When a user accesses the system for the first time, any topic he subscribes to can be stored, as well as all access level information.

To provide a better understanding of the proposed solution, we now describe the layers that it is broken into. The first layer (and the only layer visible to the user) is a simple interface. This can be thought of as a simple web page that allows a user to publish documents, update already published documents, or subscribe to documents. The only purpose of this layer is to pass data and query requests through to the next layer. The second layer of the system is the security, or access control, layer. This layer is responsible for passing published data, user subscriptions, and access levels through to the next layer. When a publisher publishes a document, he also publishes what users or access levels are authorized to access the document. It is the security layer’s responsibility to store this information, while the information that it doesn’t need to store (e.g. the actual data and user subscriptions) is passed through to the next layer. It is also the security layer’s responsibility to verify user access level for results retrieved from the third layer, before returning them to the user. The third and final layer can be seen as a Database Processing Layer. This layer has the responsibility of storing published data into a database as well as storing user subscriptions. This layer also has the responsibility of retrieving stored data and returning it to the security layer. It can be viewed as a layer that handles database input and output. A diagram detailing the model is shown below.

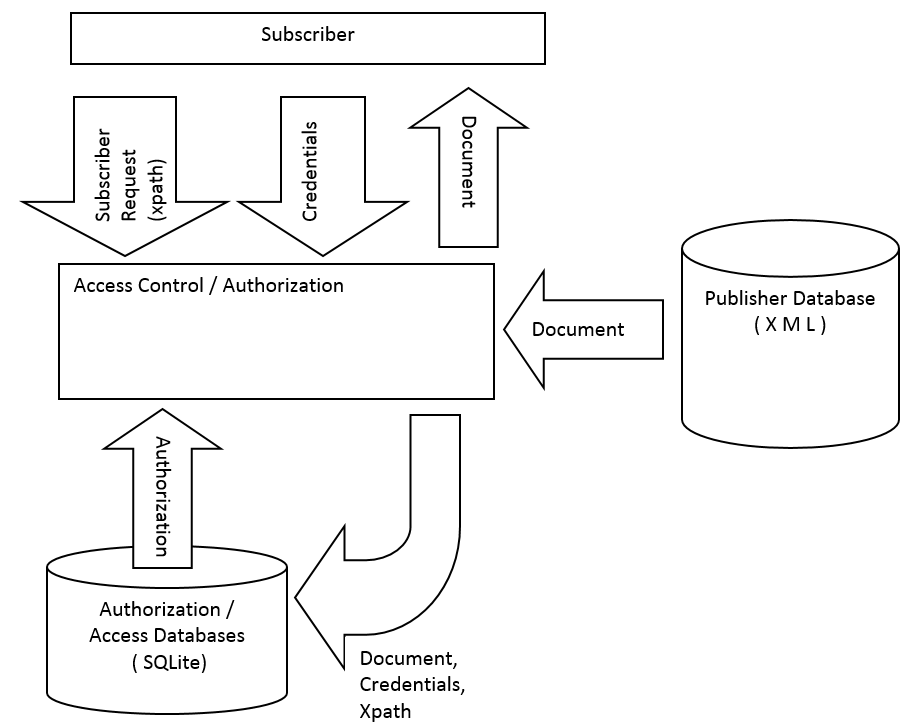
 As described above, the first layer in our proposed solution is an interface to allow a user to publish and subscribe to data. This layer is currently implemented as a simple HTML page in the python language. Running the python program creates the webpage on the local host on port 8025, so simply opening an internet browser and browsing to the local host on that port displays the interface. While not intended for actual use, this simple interface provides a user-friendly, convenient way to test the lower layers of the model which is where the actual functionality takes place.

Figure 1: Secure Publish/Subscribe Model

The responsibility of the second layer of our system is to handle the security aspect of the model. When a publisher publishes data, this layer passes the published data onto the third layer and also stores any users that the publisher provides as an authorized user of the document, or a portion of the document. We have not implemented this aspect yet but plan to require the publisher to provide user names for each document that he has published to indicate which users can access that document. Additionally, the publisher must provide and xpath query. This xpath query defines the specific portions of a document that a user can access. For instance, if a user should be allowed to access multiple parts of a document but not the entire document, the publisher can provide that user’s name along with an xpath query that finds all these parts and nothing else, or that publisher can provide that user’s name with an xpath query more than once (each time with different xpath query) to specify what that user can access. If a user should be allowed access to the entire document, this can also be specified by the publisher. The user names, document ids, and xpath queries are stored in an “authorization database” within this layer. When a user requests a document or topic, the authorization database is scanned to determine what documents and portions of documents the user can access, then this layer requests those documents from the third layer. The authorization layer finally returns the results back to the interface layer. We have also discussed a method of storing the authorization data within each xml published xml document as metadata. This option could potentially simplify the overall model, but the details of the implementation would be more complex to achieve this, so it is unlikely this method will be fully explored at this time.

The third layer in our solution has the responsibility of storing and retrieving the xml data that has been published. As stated earlier this is stored using SQL Lite to keep persistent data even when the program that runs the interface is not running. The XML data is stored along with a topic and a document id to allow partial updates of individual documents as well as searches based on document topic. Partial document updates are possible by utilizing xpath. Similar to providing a way to verify user authorization of partial documents, xpath also provides a way for the publisher to update a portion of a document. In this case, the publisher will need to provide the document id to be updated, the new/modified data, and an xpath query that will return only those portions to be updated from the original document. When this is provided, this third layer retrieves the original document, utilizes the xpath query to find a portion of it, replaces that portion of the document with the new data, and finally replaces the original document in the database with the newly modified document. Of course in the case of updating an existing document, authorization must be checked by the authorization layer to verify that the user is allowed to update the document. It goes without saying that the original publisher can update a document, but it is conceivable that a publisher would want to allow other users to update his document as well. This could be achieved by storing a list of “authorized modifiers” in the authorization database for each document, but this concept has not been explored to date. A diagram of the current proposed partial update method is provided below.

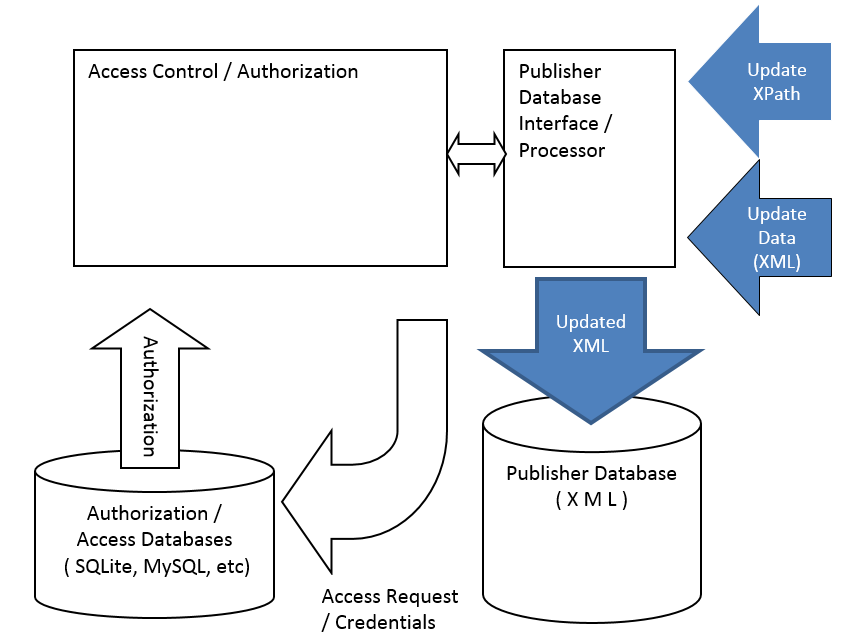


Figure 2: Partial Document Update Method

Algorithms and Discussion

**Secure XML**

Our security model contains two layers. The first layer is external to our model but is assumed and required for our second layer to function. The second layer is internal to our model. It is on this second layer where the focus of this discussion will lie. However, this discussion will not be complete without at least mentioning some details of the first layer.

We implemented a security model that requires a layer of authentication around the XML. In our test implementation, the security model assumes the concept of a “user”. The user could be either a publisher or a subscriber. The inner security model assumes that the identity of the user has already been verified. To authenticate a user, we leveraged the Twisted library’s HTTP authentication. Once the user has been authenticated by the Twisted library, the identity is assumed to be valid for the sake of passing to the secondary security layer. The outer layer could have been any mechanism for determining valid user identity.

The next layer in our security model used the authenticated user (again, publisher or subscriber) identity to select access controls from a database. Authorization is based on an Xpath query defined at the time an XML document is published. Upon publication, the publishing user grants access to any user known by the system. Then when a user subscribes to a document topic, only those portions of the documents under that topic for which the user has been granted access are provided to the subscriber. Furthermore, a user can only edit those portions of the document to which that user has been granted access.

As mentioned, the first security layer could be implanted in any of a number of ways. The layer need only provide for some method to authenticate / validate a user. For the sake of our testing, we could have eliminated this layer altogether and simply used hard-coded user names that were already assumed to be authenticated. But, for the sake of a better demonstration, we implemented an HTML based session/realm scheme that leveraged functionality in the Twisted library.

The second security layser is implemented as an SQLite database table. The table maps a user identity to the xpath that defines the elements to which the user has access. There is at least one entry in the database for each document. And, the document identifier is used to compose the table’s primary key.

**XML Partial Document Update**

For the partial document update, we used xpath queries to select out the portion of the document to be updated. The user (assumed to be authenticated) can modify the contents of any tag returned by the Xpath query. The original document is then updated with the modified tags and saved back into the database.

Implementation

Our implementation was done using the Python programming language. To demonstrate the concepts we implemented a web-based application to provide an interface. The interface was initially intended to simply wrap the security and partial update algorithms. However, due to lack of time and direction on this project, we ended up embedding most of the logic in the interface’s implementation.

The interface is implemented on top of the Twisted [REFERENCE] framework. Twisted provides networking implementation that was outside the scope of the project; and, therefore would have been unnecessary burden for us to implement ourselves. We leveraged the Twisted framework to provide logging as well as to handle transport of the HTML used to display the interface pages.

The interface is arranged in a series of web pages. Each page provides access to a portion of the functionality provided by our project. The pages are stored as static HTML with some dynamic elements. The dynamic elements are updated either when the project system starts or at runtime as the user interacts with the system. The interface was originally meant to only be a portal to access and demonstrate the underlying functionality. However, some work is still needed to separate the concepts from the interface.

Upon initialization the system establishes a list of authorized users. This list is currently stored as plain text in a flat file. However, the list could be stored in any of a number of secure ways. We chose an unencrypted flat file for simplicity.

Additionally, the system initializes the databases to be utilized. Each database is stored in an SQLite format. There is a separate database for the publisher data, the subscriber data, and the security data.

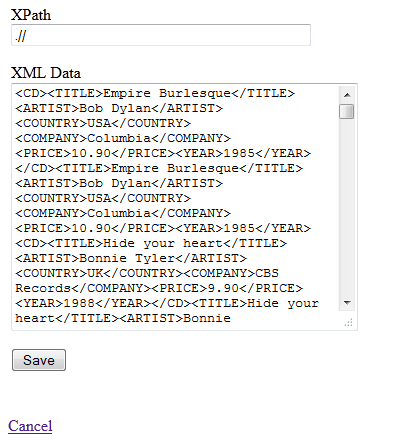
The publisher database stores the published data. Only a single table is needed for the publisher database. This single table consists of rows of document identifiers, document topics, the documents’ text, and the identity of the publishing user. The subscriber database stores the users’ subscriptions. Again, only a single table is needed. Each row in the table contains a user identifier and the topic to which that user has subscribed. Finally, the security database is also implemented as a single table. The security table contains the document identifier, the user identifier, and the xpath to which the user has access in the given document. Several python modules have been implemented to provide our with functionality to create and manipulate the various databases.

It is worth mentioning without going into too many details that the Twisted realm and session are used to establish and authenticate the user’s identity. This is done for convenience and fit well with the web-portal based interface design. However, any method for validating the user identity could have been used instead.

As mentioned previously, the interface is broken into a series of pages. The pages are organized into a tree hierarchy. There is a root page, a publisher page, subscriber page, an editor page, and a page that provides various functionality for manipulating the databases. We have called this latter page the Admin page. However, in future implementations we may remove or rename it.

The static contents of the root page are saved in the file main.html [REFERENCE]. This file also contains several dynamic components and the java script necessary to update those components. Two lists are built at run time on the main page, one for the published documents and one for the current user’s subscriptions. Both of these lists are built by java script stored in the main html. The lists are populated and updated at run time by a method of long polling. The long poll requests a publisher documents page and a subscriber documents page from the server on a periodic interval. Each such page is built dynamically by pub\_docs.py [REFERENCE] and sub\_docs.py [REFERENCE].

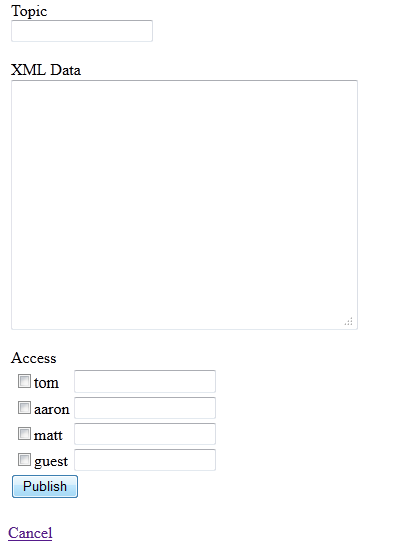
The dynamic content consists of a table of links. Each link refers to either an editor page (as in the case of the published documents) or a display page (as in the case of the subscribed documents). The functions of these links will be discussed further in subsequent paragraphs.



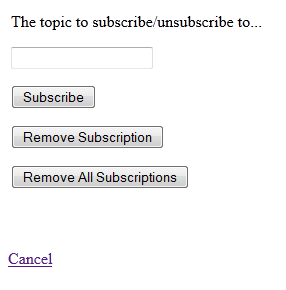
The editor page and the display page or nearly identical with the exception that the fields on the display page is read only. Due to the similarity and the lack of any actual functionality in the display page, only the editor page will be discussed in further detail. Both pages display an Xpath and the portion of the XML document found along that path. The editor page, as the name suggests allows the user to edit published XML data. The Xpath and document data portions of the page are populated from data passed in the URL. This data is added when the links are built. The reader should keep in mind that all of these implementation details are not fundamental to the system at this point. However, due to the close integration between the security and editor models and the interface, it is necessary to explain the interface in detail.

The next two pages in the hierarchy are the publisher page and the subscriber page. Both can be accessed from links on the root / main page. The most complex and interesting of the two is the publisher page.

The publisher page contains data fields for entering the document topic, document XML data, and the document access controls. The topic can be any arbitrary text used to categorize the data. Multiple documents falling under the same topic will be grouped in the database. The XML data field is where the user will enter the XML data. Any arbitrary XML data can be entered into this field. The access control inputs are dynamically built from the list of users known to the system. For each known user, there is a check box beside the name of the user and an input text field. The text field is used to input the xpath defining the access to grant to the corresponding user. Only users authorized in this way will have access. Even the publisher must be sure to grant himself access.



The Subscriber page contains a single text input for the topic to which the user wants to subscribe. The user can also remove subscriptions from this page as well.



Given the close coupling and integration between the interface and the underlying model implementation, the interface closely resembles the system. However, this is simply an artifact of the design decisions that were made to expedite the implementation for this project. Ideally, the system would be distinct from the interface. However, since this is not the case, some explanation should be given to the working of the Twisted library so as to help the reader understand the source code. The following paragraphs describe how Twisted glues the pages together.

First, one should understand that the Twisted.web class is being utilized. This class integrates a web / HTTP server so that any application using Twisted.web can easily implement a web portal / application. To implement a Twisted.web portal an application need only implement a class for each page. The classes must inherit from Twisted.web.resource and provide a specific set of methods. Specifically, the class must provide a render\_GET and render\_POST method to be called when the web browser requests the page via the GET or POST method [HTTP REFERENCE]. Additionally, the class can provide a set of child pages. The method getChild will be called when the browser requests a child page. The getChild method should return an instance of the class implementing the child page. The children pages are enumerated in a dictionary called ‘children’. Each entry in the dictionary maps the HTTP path to the instance of the child class that implements the page along that path. For example, the root page has the following children:

children = { 'pub' : PublisherPage(self),

'sub' : SubscribePage(self),

'pub\_docs' : PubDocs(self),

'sub\_docs' : SubDocs(self),

'edit\_xml' : EditorPage(self),

'show\_xml' : DisplayPage(self),

'admin' : AdminPage(self)

};

This bit of python code declares a python dictionary mapping a text string to instances of the page that implements the page along each path. So, assuming that the root page is along the following path:

<http://localhost:8025/>

Then, the children would lie along the paths:

<http://localhost:8025/pub/>

<http://localhost:8025/sub/>

<http://localhost:8025/pub_docs/>

<http://localhost:8025/sub_docs/>

etc…

Likewise, each of the pages in our implementation will be defined. If a page has no children, then the class implementing that page should set it’s ‘children’ data member to an empty dictionary, {}.

Test Data and Methodologies

We are currently considering molecular data from a protein database for testing our model. Utilizing this type of data provides a number of benefits. One of which is that there is plenty of real world data, readily accessible on the internet. Another benefit is that this data can provide us with a potential real-world use case scenario to test our model. A real-world scenario where this could be used is sharing protein data between scientists in a lab. Certain scientists may only be allowed to view certain portions of data or certain molecules. This could be based on a number of things such as the scientist’s job title or the level that they are. The molecular data from the protein database comes in a “PDB” file. These files define the contents of the XML document and will be stored in the publisher database in our model.

There are three potential test cases that should prove that the model is working as intended by allowing users to publish and subscribe to data and to do so securely. These three tests cases are publish, subscribe, and update. The publish test case should prove that the publisher database is being populated. The use case is that a user “logs in" to the system and publishes data. The data is then pushed to the publisher database and the XML data is stored as a document in the database. The second test case is to prove that a user that subscribes, gets data that they subscribed to. In this test case, the user logs into the system and subscribes to a topic. The system verifies the user’s access level and then pushes all data that the user is authorized for to the user. Additionally, new data and data changes are automatically pushed periodically to the subscribed user. The third test case is to show that a publisher can do a partial update to a document that has already been published. In this case, the user logs in to the system, and publishes a document that already exists in the database along with an update xpath query. The system first will verify that the user is authorized to make updates to the document in question, then it will retrieve the document and use the provided xpath query to replace the data in the document with the data provided by the publisher. The newly modified document is then stored in the database overwriting the original document.

The interface for running the test cases is as described above. The model will be tested utilized the simple web interface previously described. Users will log in to the web interface to publish and subscribe to documents. Additionally users will provide partial updates via this interface as well. Both publishers and subscribers will need to log into the system so that their name can be recorded. This is utilized to verify authorization for subscribers and to store the publishers name for future document updates. The web page also provides the interface by which users will receive periodic updates for newly published and modified data. At this time, the interface is mainly used to prove that the inner workings of the model are working as intended.

Conclusions

References