Purpose of the document:

This document includes an introduction to the security system developed by UMass Boston for NatureServe, and a detailed configuration and development guide for it. It can be used by developers who are interested in integrating this system into their project or extending it with its predefined interface. It also can serve system administrators who want to change the behavior through the configuration file.
Table of Contents

UMASS-BOSTON ACCESS CONTROL SYSTEM FOR DISTRIBUTED XML DATA VERSION 1.0........ I

1 OVERVIEW ................................................................................................................................. I

1.1 INTRODUCTION .................................................................................................................. I
1.2 PURPOSE ............................................................................................................................. I
1.3 SCOPE .................................................................................................................................. 2
1.4 ARCHITECTURE .................................................................................................................. 2
1.5 SYSTEM LIMITATIONS ....................................................................................................... 4

2 DEPLOYMENT GUIDE .................................................................................................................. 6

2.1 PREREQUISITE SOFTWARE ............................................................................................... 6
2.2 LIBRARY REQUIREMENTS ................................................................................................. 6
2.3 DEPLOYMENT ENVIRONMENT ............................................................................................ 6
2.4 INSTALLING THE SYSTEM ................................................................................................. 6
2.5 DEPLOYING THE SYSTEM ON A SINGLE HOST ................................................................. 6
2.6 DISTRIBUTED DEPLOYMENT ............................................................................................. 7
  2.6.1 Substitution in the Apache Ant buildfile................................................................. 7
2.7 DEPLOYMENT OF THE AUTHENTICATION SERVICE .................................................... 8
2.8 DEPLOYMENT OF A PDP ........................................... ................................................................ 9
2.9 DEPLOYMENT OF A PEP ................................................................................................. 9
2.10 CONFIGURATION IN TDS ............................................................................................. 10

3 CONFIGURATION GUIDE ......................................................................................................... 11

3.1 GENERAL CONFIGURATION STEP BY STEP .................................................................. 11
3.2 AUTHENTICATION SERVICE .............................................................................................. 11
  3.2.1 Backend authentication provider .............................................................................. 11
  3.2.2 Certificate store ........................................................................................................ 11
  3.2.3 Tokens ....................................................................................................................... 12
    3.2.3.1 Token content .................................................................................................. 12
    3.2.3.2 Token verification ......................................................................................... 13
3.3 PEP (POLICY ENFORCEMENT POINT) ........................................................................... 13
  3.3.1 Schema locator ....................................................................................................... 13
    3.3.1.1 Schema annotation .................................................................................. 13
    3.3.1.2 Schema annotation in Release 1.0 ................................................................ 14
  3.3.2 Filter generation .................................................................................................... 15
  3.3.3 Filter caching ......................................................................................................... 15
  3.3.4 PDP access point .................................................................................................. 16
  3.3.5 Format of XPath in generated filters .................................................................. 16
3.4 PDP (POLICY DECISION POINT) .................................................................................... 17
    3.4.1 Policy Discovery ................................................................................................. 17

4 WRITING ACCESS CONTROL POLICIES ........................................................................... 18

4.1 THE STRUCTURE OF XACML POLICIES ........................................................................ 18

5 INTEGRATION GUIDE ............................................................................................................... 19

5.1 HOW TO BUILD THE SYSTEM FROM SOURCES ............................................................ 19
    5.1.1 Authentication ..................................................................................................... 19
    5.1.2 Data retrieval ..................................................................................................... 20
    5.1.3 Filter generation by the PEP .......................................................................... 21
    5.1.4 Application of the filter .................................................................................. 22
    5.1.5 PDP interfaces .................................................................................................. 23
5.2 REQUIREMENTS FOR A TRUSTED DATA SERVICE(TDS) .............................................. 19

6 DEVELOPER GUIDE .................................................................................................................. 25
1 Overview

1.1 Introduction

This document describes an access control (AC) subsystem of substantial generality designed for, and deployed in, the Internet Data Delivery (IDD) project directed by NatureServe for distributed service of biodiversity observation data served by the 75 partners of the NatureServe Natural Heritage Network. The AC subsystem was written in the Computer Science Department, University of Massachusetts at Boston. Throughout the document, this system is designated as UMB-AC. It is designed to provide fine-grained access control of data transported between network data providers and IDD applications using the eXtensible Markup Language (XML). The principal use of UMB-AC is in furtherance of NatureServe’s goal to present the widest possible public access to the Natural Heritage data, while permitting sensitive data to be offered only to individuals, programs, or computers reliably authorized for such access. Important examples include the protection of the exact location of individuals of rare or endangered species, and the protection of the privacy of land owners on whose property biotic surveys have been carried out. The architecture of UMB-AC makes it particularly appropriate to use in circumstances where the overwhelming fraction of the data are not sensitive, but where the sensitive data must be either repressed or have an appropriate substitution made for it, such as the replacement of a geocoding by a containing place name (e.g. a county).

⚠️ Throughout this document we carry several warning signs:

⚠️ denotes a difficult or critical point

⚠️ denotes material special to deployment as SOAP-based Web Services

⚠️ denotes material special to developers who wish to reimplement some system functionality. Readers deploying and configuring the distributed implementation can usually skip this material.

⚠️ If the roadsign appears on a section heading, it should be taken to apply to the entire section.

1.2 Purpose

The main purpose of the UMB-AC system is to provide a role-based access control framework that can be applied in a distributed data provision environment, which means a data service may serve data from different data providers and every data provider may have its own authentication and authorization service. An access control decision may depend on information spreading among the service nodes, which requires that control policies from different nodes should be understandable across the network and client credentials from different nodes should be honored.
1.3 Scope

As detailed below, UMB-AC is based on the eXtensible Access Control Markup Language (XACML)\(^1\). XACML provides a very general framework for access control, including the specification of access control policies. Some of the components of UMB-AC may well be usable for management of data beyond that required to offer data in response to distributed read-queries, but such applications are not encompassed in the design goals of the system. For example, no consideration has been given to the need to maintain security and integrity on the part of applications which accept update queries. Nor does UMB-AC offer any assistance to the database designer or the access policy author, on whom jointly rest the responsibility to guard against internal correlations between controlled and uncontrolled information items that would cause the former to be deduced by datamining techniques against the latter. The classic example of this is the assignment of, and access to, sequential record identifiers on geocoded observation data. Long sequences of such records likely represent observations at the same location, and controlling the geolocation of one will not suffice if its predecessor and successor records are at the same location as one another.

While the system provides for secure, certificate-based authentication of users to their roles, and hence to their access privilege, it does not explicitly provide for the secure transmission of data between the components enforcing access policies. These must be provided, e.g. by communication over secure channels such as the Transport Layer Security protocol (TLS)\(^2\) or its predecessor the Secure Socket Layer (SSL)\(^3\) of TCP/IP, or simply by operation of all the components inside a single otherwise secure network as in the current NatureServe implementation.

1.4 Architecture

Following the XACML framework, the UMB-AC security system is composed of three kinds of subsystems: authentication services, Policy Decision Points (PDPs), and Policy Enforcement Points (PEPs). By design, each of these may be located at one or more of the data providers in a fully distributed system, with suitable local configuration. However, in the late 2006 deployment at NatureServe, there is a single provider aggregating the data of all the NatureServe partners, and a single instance of each of the principal components.

This system provides a security token service implementation, which will depend on a backend authentication provider that implements a predefined interface to verify the password of the end user and get back the corresponding roles. The token is signed with the private key of the issuer. Based on the trust relationship defined by the certificate chain, every node honors the token issued by trusted parties. Through the token service interface, the service invoker can add extra information (such as the IP address of the client) into the token, which can be used later to verify a token holder.

The current deployment provides a PDP service front end, which depends on a back end repository that implements a predefined interface to store and serve policies. The policies shared

---

\(^1\) http://www.oasis-open.org/committees/xacml
\(^2\) http://www.ietf.org/html.charters/tls-charter.html
\(^3\) http://wp.netscape.com/eng/ssl3/
across the network follow the XACML standard. Policy conflicts are resolved by the policy combination algorithms defined in XACML.\(^4\)

This system provides a PEP service implementation, which depends on a schema repository to serve XML schemas constraining the query returns and brief related schema annotations describing which schema elements contain sensitive information. The current system uses the file system to store schemas, and a Spring\(^5\) configuration file to provide schema annotation in the form of property-value pairs as well as naming the Java classes which provide the implementation of certain interfaces defining the system.

Because the PEP only returns a filter to a client trusted by the data provider (a “Trusted Data Service”, or TDS), it will rely on the TDS to do the real enforcement. The filter can be an XSLT\(^6\) template or an STX\(^7\) template with the corresponding setting in the configuration file.

The diagram below is a high level picture of the work flow of the security system. A more detailed data flow as reflected in the deployment at NatureServe at this writing is found in Appendix G.

\(^4\) In research at UMASS-Boston we have explored combination algorithms extending those supported by XACML, based on the RuleML language for rule-based applications.
\(^5\) http://www.springframework.org
\(^6\) http: www.w3.org/TR/xslt
\(^7\) http://stx.sourceforge.net
1.5 System Limitations

Data flow between applications and data sources takes place in the XML. The data served by a source must conform to an XML-Schema\(^8\) which can be determined authoritatively by the PEP. Access control filters are expressed in either the XSLT or STX transformation language, depending on configuration, in the form of templates expressed with XPath\(^9\) expressions based on the schema. The XPaths are selected from an enumeration of all the possible XPaths through the schema, together with information from policies, and a small amount of schema annotation (Section 3.3.1.1). This architecture allows us to delegate policy conflict resolution to the PDP and to cache filters, which results in performance linear in the size of data instance documents.

In the remainder of this subsection, we discuss some limitations of UMB-AC arising mainly from the implementation at NatureServe in late 2006.

In general the system can accept recursion in the data provider’s XML-Schema, but the current implementation requires that either the TDS passes to the filter the maximum depth in the instance document, or there is provided in the configuration file a maximum tree depth that will occur in any presented query response, or, more precisely, a maximum depth below which there is no sensitive information. If the schema is detected to be non-recursive, the software computes this maximum depth. A wrong maximum depth of instance documents may cause a security breach because it will limit the depth of the XPath that can appear in the filter in the absence of a directly assigned parameter of document depth when PEP is called.

The current implementation of filter generation is controlled by annotation of the schema, provided in the Spring configuration file. Using the annotations, the Schema, and the XACML policy representations, the implementation decides the XPaths that can appear in the filter corresponding to a given set of policies determined by the role of the individual presenting a query. A misconfiguration may lead to leakage of sensitive data in ways we discuss in Section 7.

As described in Section 9.1, the XPaths permitted in the rules of a policy from a proper subset of XPath 1.0. One reason is that the STX streaming process does not allow looking forward or branching. The other is that the current implementation of filter generation, in the case of recursive Schemas, uses the Schema and corresponding annotation to enumerate the possible controlled XPaths.

There are two approaches that can be used to generate filters: the first is simple translation, which directly translates policies into a filter; the second is schema path enumeration, which constructs filters depending on the decision of the PDP on every possible XPath enumerated based on the schema and the schema’s annotation. We implemented that latter on performance grounds discussed below.

Simple translation has big advantages for the generation process of filters: it has no restriction on the XPath expression; its time consumption is polynomial only in the policy tree size, which is typically small; it has no concern about the data schema. On the other hand, our implementation of path enumeration has some limitations on the generation process of filters: for the enumerated XPaths, it only supports the expression of attributes whose value can be enumerated by the PEP;\(^10\) it also needs a maximum depth of any submitted instance document if the schema has

---

\(^8\)http://www.w3.org/XML/Schema
\(^9\)www.w3.org/TR/xpath
\(^10\)For example, it is difficult for us to express access control filtering based on the value of a numeric data attribute.
recursive elements; its time consumption is polynomial to the policy tree size multiplied by the size of the expanded tree formed in the process of XPath enumeration; it needs both the schema and its annotation.

However, based as it is only on policies, the simple translation approach lacks information at filter generation time about the syntactic (resp. semantic) constraints on instance documents provided by the schema and its annotations. Therefore, it must delay rule combination operations to filter application time, which will leave a complex script (for depth first traversal of the policy tree) in the filter. What makes it worse is that the traversal has to run for every element in the instance document, which can be avoided with the enumeration approach. So the running time of the filter generated by this approach is polynomial in the policy tree size multiplied by the size of the instance document tree. In contrast, schema enumeration can exploit those constraints at filter generation time, so the rule combination can be accomplished then. As a result, the running time of the filter generated by schema enumeration is polynomial only in the size of the instance document tree.

Considering the size of policies and huge instance documents, benchmarking convinced us that the running time of filter applications crucial to this project. In addition, we can use a caching mechanism to ease the problem of the long filter generation time for the schema enumeration approach. However, as discussed in detail in Section 6 (Developer Guide), the system is generally sufficiently modular as to support a different filter generation implementation.

For complete description of the filter generation, see our paper *Schema-Driven Security Filter Generation For Distributed Data Integration*11.

---

2 Deployment guide

⚠️ Most of the security system is known to work with JDK 1.5. However, extensive testing in-situ has been done only with JDK 1.4, and this guide should be used with care for any other JDK version.

2.1 Prerequisite software

JDK 1.4 or above
Tomcat 5.0 or above
Axis 1.2 or above

2.2 Library requirements

Each of the jar files below is provided with the distributed system which has been tested only with the versions given.
Xalan-2.6.0.jar Release: 2.6.0
Opensaml-1.1.jar Release: 1.1
Jcs-1.2.7.jar Release: 1.2.7
Joost.jar Release: 20061015
Sunxacml.jar Release: 1.2
Spring.jar Release: 1.2.8
Concurrent.jar Release: 1.3.4

2.3 Deployment environment

Apache Ant\textsuperscript{12}

2.4 Installing the system

Install Apache tomcat and axis.
Check the axis installation using happyaxis.jsp under the axis directory
If you are using JDK 1.4, copy the jar files in the endorsed folder of the project into the endorsed folder of tomcat.
In the file build.xml, modify the properties sever.deploypoint, sever.deploy.libdir, and sever.deploy.classdir according to the configuration of your machine.

2.5 Deploying the system on a single host

After installation as described in Section 2.4, run “ant deploy”.

\textsuperscript{12} http://ant.apache.org/
Copy files under the *samples* folder to the axis directory. Modify the server address in those .jsp files to that of your machine and test the installation using MyJsp2.jsp, e.g. by pointing a web browser at http://localhost:8080/axis/MyJsp2.jsp.

If you are deploying on a single host, you may safely skip to Section 3., “Configuration guide”, to adjust the default configuration.

### 2.6 Distributed deployment

Configuration and deployment of the UMB security system is managed in two places. Principally, configuration is in a Spring\(^\text{13}\) configuration file `security_spring.xml`, but this file in turn has a small number of tokens of the form `#name#` where `<name>` is a string whose value is provided by the Apache Ant `build.xml`. At this writing, the build file must be edited by hand to provide locally appropriate addresses for web service endpoints in a distributed environment.

This section assumes familiarity with SOAP-based Web Services and their configuration requirements, as well as the Apache Axis Web Service container.

#### 2.6.1 Substitution in the Apache Ant buildfile

⚠️ The current installation package does not provide for less than full installation, but this section describes how to configure a host for separate *deployment* of the pieces, and how to configure them for collaboration.

Determine which components you intend to run as web services from this host. The names of the services can be obtained by running

```
   ant showServiceNames
```

At this writing these are

- `authenticationservice` the authentication service discussed in Section 2.7
- `pdp` the PDP described in Section 2.8
- `pep` the PEP described in Section 2.9

⚠️ There are three tokens in `security_spring.xml` that will be replaced one time only, by the initial invocation of Apache Ant controlled by `build.xml`. This is accomplished by the `<replace>` tasks in the build file. In those tasks, “#”-delimited tokens signal what tokens in `security_spring.xml` get changed. Thus, `#schemafilepath#` represents the folder where the system can find all the related schemas and it will be replaced with the `classes` directory of the local axis server where you install the security system. `#policyfilepath#` represents the path of the policy file that is used by bean `filemodule` and it will also be replaced with the `classes` directory of the local axis server where you install the security system. `#endpoint#` represents the access point of PDP web service that is used by bean `pdprequestorremote` and it will be replaced with the URL of the local axis server where you install the security system. WARNING: if you subsequently change the targets of these in `build.xml` the changes will not propagate to `security_spring.xml` and you need to edit that file by hand to correspond to your changes. Comments in `security_spring.xml` containing the character “#” help identify where are the points which need addressing.

\(^{13}\) [http://springframework.org](http://springframework.org)
Independently of the above, you may be motivated to change one or another of those beans in **security_spring.xml** to use your own implementation of one or another security modules. In such a case, bear in mind that you have decoupled the build process from the Spring file with respect to the “#”-delimited bean properties, and thereby find yourself exactly in the situation above if you then subsequently change the values of those resources and redeploy with Ant.

After you have configured **build.xml** you must deploy the system before fully configuring it.

To install for deployment as web services run

```
    ant deploy -Dservices= [servicename]*
```  

Here [servicename]* denotes one or more service names separated by spaces. The quotation marks are required if you have more than one service in your list, otherwise not.

### 2.7 Deployment of the authentication service.

If you are only deploying authentication as a web service, you should have run

```
    ant deploy -Dservices=authenticationservice
```  

You may be deploying with several other services, in which case the more general form of Ant invocation is required as shown in Section 2.6.1. Remember, as described there, that if you wish to change your deployment, you may have to re-insert the “#” delimited variables into **security_spring.xml**.

This section assumes that you have an authentication service that implements the **IAuthenticationServiceProvider** interface. In the file **security_spring.xml**, if necessary change the implementation class of verifier bean to the one that implements the interface in your package.

Using the **keytool** application from the JDK, create a java key store for token issuance. Import the certificate of this service into this key store. Make note of the alias you attach during this import, as it is used in subsequent steps. Also import certificates that are trusted by this authentication server.14

Change the constructor arguments (the objects in the `<constructor-arg>` XML element) for the **keystore** bean in **security_spring.xml**. The first one should be the name of the key store you just created and the second one should be the password you set for your key store file.

Change only the second constructor argument for the **tokenissuer** bean. This argument is composed of four properties of which only the first three normally need configuration. The first property is the password for key store. The second is the private key alias that you assigned earlier to the certificate when you imported it for this service to provide with tokens. The third property is the issuer name that will appear in the security token. The fourth determines the time in milliseconds after issuance for token expiration and normally need not be changed unless you want shorter or longer certificate life. The default configuration is 86400000 ms, which is one day.

---

14 More information about keytool and java key store can be found at [http://java.sun.com/j2se/1.4.2/docs/tooldocs/windows/keytool.htm](http://java.sun.com/j2se/1.4.2/docs/tooldocs/windows/keytool.htm)
2.8 Deployment of a PDP

If you are only deploying authentication as a web service, you should have run

```
ant deploy -Dservices=pdp
```

You may be deploying with several other services, in which case the more general form of Ant invocation is required as shown in Section 2.6.1.

Remember, as described Section 2.6.1, that if you wish to change your deployment, you may have to re-insert the “#” delimited variables into `security_spring.xml`.

The system installation described in Section 2.4 provides a class `SimpleFilePolicyFinder` that extends the class `FilePolicyModule` of the SUN XACML distribution.\(^{15}\) `SimpleFilePolicyFinder` uses a file as the repository of policies. If you implement your own policy repository not based on a local file, or implement another file-based policy repository, it must directly or indirectly extend the abstract class `PolicyFinderModule` of SUN XACML. In this case, in `security_spring.xml` you must change the implementation class of bean `policyrepository` to the corresponding class in your package.

2.9 Deployment of a PEP

If you are only deploying authentication as a web service, you should have run

```
ant deploy -Dservices=pep
```

You may be deploying with several other services, in which case the more general form of Ant invocation is required as shown in Section 2.6.1. In this case, you will need to know their SOAP endpoints.

Remember, as described Section 2.6.1, that if you wish to change your deployment, you may have to re-insert the “#” delimited variables into `security_spring.xml`.

There are several likely scenarios if you are running the PEP as a service. One is that the PDP required by the PEP is running as a remote service. Another is that it is running on the same host as the PEP. Because, at this writing, installation and deployment will have left the PDP classes in `security.jar` these will be available for local use.

In any case you must:

- Copy all the XML schemas that you will use into a file folder
- For every schema, repeat the following process.
  - Give it an identifier name such as ‘EOData’, which will be used as the resource identifier in a PEP request.
  - In the `security_spring.xml`, add the following properties to ‘props’ of the bean `schemalocator`, where you should replace ‘EOData’ with the identifier you specify in previous step. The meaning of those properties is given in the Schema annotation configuration reference, Section 3.3.1.1.
    - `pep.EOData.sensitiveAttributeNames`

\(^{15}\) [http://sunxacml.sourceforge.net](http://sunxacml.sourceforge.net)
In `security_spring.xml`, the implementation class of bean ‘transformer’ determines what kind of filters are generated. Selecting ‘edu.umb.cs.xmlsecurity.pep.XsltTransformer’ provides XSLT\textsuperscript{16}-based filters, whereas ‘edu.umb.cs.xmlsecurity.pep.STXTransformer’ provides STX\textsuperscript{17}-based filters. In general, XSLT consumes more memory but is faster than STX and may be suitable only when most instance documents are small. See Appendix B ("Comparison of STX filter and XSLT filter languages").

In addition, if the PDP is a remote SOAP service, then you must also:

- In `security_spring.xml`, change the property ‘pdpRequestor’ of the bean ‘filterGenerator’ into `pdprequestorremote`.
- In `security_spring.xml`, change the property ‘endpoint’ of the bean `pdprequestorremote` to the web service access point of the PDP SOAP service that will be invoked by this PEP.

### 2.10 Configuration in TDS

The UMB-AC system does not provide TDS implementations. Please see the Nature Serve documentation for their TDS architecture and description. If you are implementing a TDS, it must conform to the interfaces described in Section 5.2, “Requirements for a Trusted Data Service(TDS).”

\[\text{http://www.w3.org/TR/xslt}\]
\[\text{http://stx.sourceforge.net/}\]
3 Configuration guide

3.1 General configuration step by step

The main configuration parameters for the security module are in `security_spring.xml` except those for filter caching, which are in `cache.ccf`. Some of those parameters normally need to be adjusted after the installation and the others do not unless you want to replace some default beans with your own implementations. The former include:

- For the bean `verifier`, the parameter to indicate the class that implements the `AuthenticationServiceProvider` interface. It is discussed in Section 3.2.1.
- For the bean `tokenissuer`, parameters for issuer’s name, expiration limit, the password for the key store, etc. They are discussed in Section 3.2.3.1
- for the bean `tokenverifier`, parameters for the password of its key store, the alias of the verifier’s certificate in its key store, etc. They are discussed in Section 3.2.3.2
- For the bean `filtergenerator`, the parameter for PDP requestor (local or a web service) and the parameter for the access point if it uses a PDP web service. They are discussed in Section 0
- For the bean `schemalocator`, the schema annotations of every schema you are using. They are discussed in Section 3.3.1.1.

Parameters in `cache.ccf` are for the filter caching. They are discussed in Section 3.3.3

3.2 Authentication service

3.2.1 Backend authentication provider

In the file `security_spring.xml`, the bean `verifier` has a default implementation that is based on a simple xml file. Most installations need a more sophisticated and secure authentication implementation. This entails providing an implementation of the interface `IAuthenticationServiceProvider` and replacing the default one by changing the implementation class of the bean `verifier`. An introduction about this interface can be found in Section 6.1, “How to write an Authentication provider”. Note that wrapping the authentication provider with an implementation of `IAuthenticationServiceProvider` is required even if authentication will be provided by the implementation as a SOAP service. (See Section 2.7 “Deployment of the authentication service.”)

3.2.2 Certificate store

An authentication service provides two functionalities: token issuance and token verification. The java key store created by the procedure in Section 2.7 is used to hold two corresponding resources: the private key of the token issuer that is used to sign the token issued by this issuer, and certificates of all the trusted third parties, whose signatures on tokens are trusted by the authentication service.

To establish these trust relationships you need follow steps below:
- Use the `keytool` application included in the JDK\(^{18}\) to generate the private and public key pair in the key store. You may use `keytool` to generate a certificate request for that key pair and then get a certificate from a Certificate Authority, which can be imported into the key store to replace the automatically generated certificate. In this process you will be asked for the file name for the key store, the password that protects the key store and the alias for the key entry that you generated. Record all those information for use in security system configuration files, detailed in section 3.2.3

- Import all the certificates of issuers trusted by the authentication service with `keytool`.

### 3.2.3 Tokens

#### 3.2.3.1 Token content

A security token that is securely digitally signed by the token issuer provides confidence in the user (human or software agent) identity and roles across the system. It can be gotten through an authentication service with the user name and the password, or by other means such as biometric identification and digital smart cards. As distributed, the UMB security system requires that the token includes the name and signature of the issuer, the name of the user, role-jurisdiction\(^{19}\) pairs of the user, and the token expiration time. In `security_spring.xml`, the following token issuance parameters must be edited in the `tokenissuer` bean.

<table>
<thead>
<tr>
<th>Property of bean <code>tokenissuer</code> in <code>security_spring.xml</code></th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>tokenIssuer.keyStorePassword</td>
<td>password used to access key store</td>
</tr>
<tr>
<td>tokenIssuer.keyAlias</td>
<td>alias of the private key entry in key store</td>
</tr>
<tr>
<td>tokenIssuer.duration</td>
<td>lifetime of the token</td>
</tr>
<tr>
<td>tokenIssuer.name</td>
<td>name of the token issuer, which must be the same as the alias of the private key entry in the issuer’s key store as well as the alias of its certificate entry in the key stores of any services wishing to verify the tokens of this issuer.</td>
</tr>
</tbody>
</table>

\(^{18}\) More information about `keytool` and java key store can be found at [http://java.sun.com/j2se/1.4.2/docs/tooldocs/windows/keytool.html](http://java.sun.com/j2se/1.4.2/docs/tooldocs/windows/keytool.html)

\(^{19}\) *jurisdiction* is a notion particular to the NatureServe deployment. In practice, it simply identifies the data provider. In reality, one can understand that deployment as though roles themselves had two substrings, one called “jurisdiction” and the other the name of a role typically common to all the data providers. Indeed, the jurisdiction and role are concatenated in the distributed PDP implementation, and policies typically exploit string regular-expression matches on role names. See Section 4, “Writing access control policies”.
3.2.3.2 Token verification

An authentication service accepts tokens that are issued by itself or a trusted third party. It identifies the origin of a token by verifying the signature on the token. In `security_spring.xml`, the token verification parameters must be edited in the `tokenverifier` bean.

<table>
<thead>
<tr>
<th>Property of bean <code>tokenverifier</code> in <code>security_spring.xml</code></th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>tokenVerifier.keyStorePassword</td>
<td>password used to access key store</td>
</tr>
<tr>
<td>tokenVerifier.keyAlias</td>
<td>name of the local token issuer</td>
</tr>
</tbody>
</table>

3.3 PEP (Policy Enforcement Point)

The UMB access control system accomplishes independence of the data XML-Schema by accepting that Schema as an argument to a number of facilities for policy enforcement. This insulates the access control system from changes to the schema. However, in a number of cases, information is cached based on the schema (and policies and the cache implementation), so a deployment which encounters many, many different schemas rapidly may have poor performance in the present implementation.20

The PEP addresses a number of concerns, whose configuration is described below. These include: discovering the schema location from its name, locating the PDP for a decision about access control for incoming queries, and caching of the resulting access control filters for performance improvement.

3.3.1 Schema locator

3.3.1.1 Schema annotation

In a PEP it is assumed that instance data returned from a data provider is an XML document based on a locatable XML Schema. XML-Schema itself is not expressive enough for the generation of all the constraints required to be enforced by the PEP in the access control filters it generates. Therefore, in addition to locating the Schema, the PEP must rely on additional annotation of the schema to represent input to this generation. These annotations specify particular attribute properties—expressed relative to XPaths in the Schema— that may be restricted by security policies in the PDP. This information is provided by the schema location facility, conforming to a Java interface named `ISchemaLocator`.

The major resources controlled by the schema annotations address issues of fine-grained XML access control. The annotations express which data attributes may represent sensitive data, and values of that attribute which signal what policies impact the element containing this attribute. In particular, the representations of policies express a value corresponding to the annotation value for the determination of to which elements the particular policy applies. For details on the syntax and semantics of policies, see Section 4, “Writing access control policies”.

20 At this writing, that is the Apache Java Caching System (JCS). http://jakarta.apache.org/jcs/
Based on the annotations, the PEP generates XPaths to be passed to the PDP, and the PEP generates XSLT or STX templates based on the XPaths and the response from the PDP. These templates are applied to enforce the policy.

⚠️ In the distributed implementation, schema annotations are statically defined as properties in the `schemalocator` bean, defined in `security_spring.xml`. However, the underlying architecture does not depend on this, but rather only on the Java interface. Consequently, fully dynamic schema annotation requires the provision of a different implementation of that interface. How one can provide that is documented in Section 6.4 (“How to write a Schema locator”).

### 3.3.1.2 Schema annotation in Release 1.0

Because in this release schema annotations are defined statically, we specify them as properties in the `schemalocator` bean, defined in `security_spring.xml`.

To configure this bean, parameters are organized according to a string providing a resource identifier. In the table below “pep.<resource>.sensitiveAttributeName” should be read as “the attribute name used to indicate sensitiveness in the schema for the resource identified by <resource>”. The string “<resource>” should be replaced by a real resource identifier in your system. The brackets “<“ and “>” should be omitted. The parameters in this table are for the bean `schemalocator` in `security_spring.xml`.

<table>
<thead>
<tr>
<th>Property of bean schemalocator in security_spring.xml</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>pep.&lt;resource&gt;</td>
<td>path for the schema file</td>
</tr>
<tr>
<td>pep.&lt;resource&gt;.sensitiveAttributeNames</td>
<td>name of the attribute in the XML document, whose value indicate the sensitiveness of the element that it belongs to. It can be a list of names with space as the delimiter. Only these and enumeratedAttributeNames should be referenced in policies. Other attribute names are ignored by the PEP.</td>
</tr>
<tr>
<td>pep.&lt;resource&gt;.sensitiveAttributeValue</td>
<td>corresponding sensitive values of attributes defined above. Every attribute can only have one value defined here, which means, if the attribute has this value, the element enclosing it is sensitive otherwise not. Values for different attributes are delimited by space and follow the same order as attribute names defined above. The values may contain regular expressions excluding spaces. Take care to maintain this correspondence.</td>
</tr>
<tr>
<td>Property of bean schemalocator in security_spring.xml</td>
<td>Value</td>
</tr>
<tr>
<td>-----------------------------------------------------</td>
<td>-------</td>
</tr>
<tr>
<td>pep.&lt;resource&gt;.rootElement</td>
<td>root element of the document</td>
</tr>
<tr>
<td>pep.&lt;resource&gt;.enumeratedAttributeNames</td>
<td>name of the attribute of elements in the XML document that need to enumerate their values for reference in policies. Only these and sensitiveAttributeNames should be referenced in policies. Other attribute names are ignored by the PEP.</td>
</tr>
<tr>
<td>pep. &lt;resource&gt;.enumeratedAttributeValues</td>
<td>corresponding values of the attributes above. Every attribute can have more than one value that is separated by coma and values for different attribute is delimited by space and following the same order as attributes are defined above.</td>
</tr>
</tbody>
</table>

### 3.3.2 Filter generation

The filter generation process is triggered by a TDS’ invocation of the method `getFilter(...)` of the PEP, which is implemented by the class `FilterGenerator` in UMB-AC. The PEP will call the schema locator to fetch the schema and its annotations through a resource identifier passed in by TDS. Then, based on element and element type definitions in the schema, the PEP will traverse all possible XPaths of a document complying with this schema. In the traversal process, schema annotations are used to construct attribute expressions for the condition parts of those XPaths. Each of those XPaths together with the user information, resource identifier and action will be passed to PDP to get the authorization decision, which will cause the PEP to generate a copy or truncation template for elements in instance documents complying with this XPath. If the schema has recursively defined elements, to decide when to stop the otherwise infinite traversal process, the PEP need the TDS to pass in maximal element depth of the instance document or it will use a default one that is defined in the `security_spring.xml` for bean `filtergenerator`.

### 3.3.3 Filter caching

To improve the performance of the filter generation in the PEP, the system caches the filters. Settings are contained in cache.ccf, which is a standard configuration file for Apache JCS. The two main properties are:

<table>
<thead>
<tr>
<th>Property name in cache.ccf</th>
<th>value</th>
</tr>
</thead>
<tbody>
<tr>
<td>jcs.region.filterCache.cacheattributes.MaxObjects</td>
<td>max. num. of cache objects</td>
</tr>
<tr>
<td>jcs.region.filterCache.cacheattributes.MemoryCacheName</td>
<td>algorithm for replacement</td>
</tr>
</tbody>
</table>

Further details can be found in the JCS documentation at http://jakarta.apache.org/jcs/.
JCS implements a hash table whose keys we construct from the roles of the user presenting a query, a policy ID, a resource identifier, the action that the user wants to exercise and the maximum depth of an instance document. This hashkey mechanism reflects that the filters are generated by considering policy combination rules available to the PDP invoked by the PEP based on the set of roles presented to it with a query.\textsuperscript{21} This represents a performance compromise in the cache: it will be most effective if many users have the same set of roles, and there are only a small number of roles. Thus, the cache enforces semantics of a pure Role-Based Access Control system in which the enforcement is not against a single authenticated role, but to a particular list of authenticated roles. One way to use the system in a more per-user access control fashion is to make PolicyIDs specific to a user for those policies which are indeed user-specific. Nevertheless, the distributed cache will have a high miss-rate if there are many users, with many user-defined policies.

### 3.3.4 PDP access point

The PEP can be configured to invoke local or remote PDP service. The distributed default is local, which means that the PDP will be accessed through a Java interface, not a SOAP service. These cases are represented respectively by two beans in security_spring.xml: pdprequestor, pdprequestorremote. The former will specify a local, and the latter will specify a remote service. A third bean, filtergenerator, specifies the name of the bean that will be invoked.\textsuperscript{22}

<table>
<thead>
<tr>
<th>Bean name given in bean filtergenerator in security_spring.xml</th>
<th>purpose</th>
</tr>
</thead>
<tbody>
<tr>
<td>pdprequestor</td>
<td>specify local PDP access</td>
</tr>
<tr>
<td>pdprequestorremote</td>
<td>specify remote PDP access through web service</td>
</tr>
</tbody>
</table>

### 3.3.5 Format of XPath in generated filters

The PEP can generate two types of filter: XSLT templates or STX templates. Comparison between them can be found in Appendix B. To select the type of the filter generated by the PEP, you need change the implementation class of bean transformer in security_spring.xml:

<table>
<thead>
<tr>
<th>Implementation class of bean transformer in security_spring.xml</th>
<th>purpose</th>
</tr>
</thead>
<tbody>
<tr>
<td>edu.umb.cs.xmlsecurity.pep.XsltTransformer</td>
<td>generate XSLT templates</td>
</tr>
<tr>
<td>edu.umb.cs.xmlsecurity.pep.STXTransformer</td>
<td>generate STX templates</td>
</tr>
</tbody>
</table>

\textsuperscript{21} This is done because it is very time consuming to apply policy combination rules at the time of application of the filters, rather than at construction. See the last paragraphs of Section 1.5.

\textsuperscript{22} In fact, you can name and provide your own PDP invocation bean in the filtergenerator bean, as long as it provides an implementation of the same interface which pdprequestor and pdprequestorremote do, namely IPDPRequestor.
3.4 PDP (Policy Decision Point)

3.4.1 Policy Discovery

In security_spring.xml, for the bean pdp you need to provide a reference to a bean as constructor parameter, which is used for policy location. This bean should be a class derived from the abstract class PolicyFinderModule of SUN XACML. The implementation included in this project is based on a simple XACML file policy_multiple_roles.xml
4 Writing access control policies

4.1 The structure of XACML policies

The root element of a policy file is `Policy` or `PolicySet`. A `PolicySet` `PS` contains a sequence of `Policy` and `PolicySet` elements, whose decisions on a PDP request will be reconciled by the combining algorithm specified by the attribute `policyCombingAlgorithm` of `PS`. Normally, the value of this attribute is a URI denoting one of XACML’s standard policy combination rules, but the Sun XACML package supports the provision of new combining algorithms and the Java classes that implement them. A `Policy` is composed of a sequence of `Rule` elements, whose decisions on a PDP request will be reconciled with the combining algorithm specified by the attribute `ruleCombingAlgorithm` of this `Policy`. A rule has an attribute called `effect` to define the authorization decision if it can be applied to PDP request. `Rule`, `Policy` and `PolicySet` elements all have a sub element `Target`, which is used to decide whether they are suitable for a request. A `Target` defines three criteria, which are represented by elements `Resources`, `Actions` and `Subjects`. Those elements in turn contain elements `Resource`, `Action` and `Subject` respectively, all of which specify one “match” function, also given by a URI which allows it to be located. Normally match functions compare values defined in policies to those in requests. In addition “designator” functions (also specified as a URI) are used to reference specific parts in a request. There are predefined match functions in the specification, which you can find in the reference document\(^{23}\) for the XACML standard. You can add new match functions, in which case you need to add their implementation classes in a location suitable for discovery by the Sun XACML implementation package correspondingly. For further information about this extension mechanism, you can reference the SUN XACML implementation site.\(^{24}\)

In UMS-AC, a new match function is added to support XPath comparison. The URI of this match function is "urn:oasis:names:tc:xacml:1.0:function:generalxpathmatch". In general, you can take the PDP processing as two steps: first, it is a depth-first traversal of the `PolicySet` to find rules that are applicable to the request based on the `Target` elements along the traversal path; second, it is a bottom-up process to combine the effects of rules and in turn the effects of `Policy` objects based on their respective combing algorithms.

The formal definition of each of the XACML elements and attributes, as well as their semantics, can be found in the XACML specification. A detailed example is given in Appendix D.

In practice it would be unusual to write policies directly in XML. Instead, an installation would build an application for generating policy XACML with a friendlier interface.

\(^{23}\) http://www.oasis-open.org/committees/download.php/2406/oasis-xacml-1.0.pdf

\(^{24}\) http://sunxacml.sourceforge.net/guide.html
5 Integration Guide

5.1 How to build the system from sources

The Apache Ant build.xml file is the starting point. There are tasks such as compile, jar, deployment and unit tests for compiling, deployment and testing.

⚠️ run ant javadoc

to build the javadocs directory src/production/docs. These sometimes provide further detail of the interfaces and classes than that provided here, where the focus is on the semantics of the arguments to the java methods discussed.

5.2 Requirements for a Trusted Data Service (TDS)

For a TDS, there are four main tasks: authenticate the user, retrieve the data, query the PEP to get a filter and execute the filter. A TDS author writes calls against the PEP interface and the authentication interface, either of which may be a Java API or a SOAP service. Following accepted notion of Service Oriented Architecture, we will consider both cases to be services, of which the TDS is a client. The general flow of control between these software agents is detailed in Schema-Driven Security Filter Generation For Distributed Data Integration[^25], but we reiterate it in Figure 2 below.

In what follows, we describe what needs to be done by a TDS if the service interfaces of the distributed PEP and authentication service are used. Any other architecture is beyond the scope of this document. If you decide on different implementations with the same interface, this document will apply.

5.2.1 Authentication

To authenticate a user, there are two different cases:

1. If the user presents the username-password pair, you can just pass the pair to an authentication service to receive a token in case it passes the validation. This is done by calling one of the `getToken(...)` methods in the interface `IAuthenticationService`. Alternatively you can write the TDS as a SOAP client of a Web Service implementing that interface. If you want a separate SOAP authentication service, you might be better off writing to the Java API and wrapping SOAP service calls in a proxy.

2. Your TDS or its clients might obtain tokens entirely outside the UMB security system. In such a case, a TDS can pass it to the authentication service to validate it. These tokens are passed through the `verifyToken(...)` method in the `IAuthentication` interface. If `verifyToken()` returns “true”, you can use present the token to a PEP with your client’s query.
5.2.2 Data retrieval

If authentication succeeds, the TDS must make a query to the trusting data sources for unfiltered data. This data must conform to the XML-Schema that will be found by the SchemaLocator based on the identifier associated by the TDS with the desired resources. For example, at this writing, in the NatureServe deployment, these include “EOSummary”, “EOREport”, and “CompleteSpecies” and the SchemaLocator uses these names to find the appropriate XML-Schema.

⚠️ It is the responsibility of the TDS to acquire documents valid for the expected Schema before it applies filters. The filter generator does not see the instance document and is driven only by the Schema and the schema annotation. Filters applied to Schema-invalid documents may result in the release of sensitive data. Whether the TDS trusts the provider to generate valid data or itself checks it (perhaps at a performance penalty), is not within the scope of the security system.

5.2.3 Filter generation by the PEP

Figure 3. Actions of the PEP during filter generation.

For querying the PEP, you invoke one of the `getFilter(...)` methods in the interface `IPEP`, passing the resource id and security token described above, the maximum depth of any possible
instance document, and the action the user want to apply to the resource, which normally is “read”. Note, however, that one invocation of `getFilter(...)` requires no instance depth, and reads a property `maxDepht` from the Spring configuration file. See Section 3.3.2 for details. In fact, this property is needed only for the support of recursive schemas.

⚠️ The action names passed to the PEP must correspond to those which will be found in access control policies. See Section 4 for details. Should the names passed to the PEP not be found in policies corresponding to the roles listed in the security token for the authenticated user, then the PEP will receive “Not Applicable” from the PDP and will generate a filter that denies access to all the information requested.

### 5.2.4 Application of the filter

Depending on the Spring configuration, the filter that a TDS receives from the PEP is either an XSLT template or an STX template for streaming transformation. Invoking the method `applyFilter(...)` in interface `IXMLTransformer` will return a filtered document. The determination of which kind of filter will be used is made at startup time in the Spring configuration. See Section 3.3.5 “Format of XPath in generated filters”. The implementation class of the Spring bean named “transformer” must be appropriate to the type of filter returned. The distribution includes jarfiles of Apache Xalan for XSLT and JOOST\(^{26}\) for streaming STX processing.

\(^{26}\) [http://joost.sourceforge.net](http://joost.sourceforge.net)
5.2.5 PDP interfaces

Although not last in data flow, we mention the PDP here because logically, a TDS is not concerned with the PDP, which is invoked only by the PEP to receive a decision about the query, given the role and user, based upon policies found in the policy store. Unless you are implementing your own PEP, you don’t need to be concerned about PDP interfaces.

There are two methods in the PDP interface IPDP. The first, getPolicyVersion(...), is used in caching filter. The second, evaluate(...), is used to get a PDP decision. It has only one parameter, a string which should follow the format of a XACML request as specified in the XACML 1.0 Context Schema (http://www.oasis-open.org/committees/download.php/919/cs-xacml-schema-context-01.xsd). You can use the helper class PDPRestoratorBuilder to construct the request string.
5.3 Logging

For all components, we use the Apache Jakarta Commons Logging framework\(^{27}\), which elegantly separates the logging interface and the implementation. So the logging implementation used at runtime will depend on the configuration of the logging system.

When the LogFactory of the framework is invoked, it will try to find the logging implementation through the value of the property `org.apache.commons.logging.Log` of the file `commons-logging.properties` in the application’s Java classpath. If it can not find the property value in that file, it will look for a system property named `org.apache.commons.logging.Log`. If it still can not find the implementation and the Log4J logging system is available in the application class path, it will use the corresponding wrapper class `Log4JLogger`\(^{28}\). We don’t explicitly define a `logging.properties` file in the current deployment, but do include a log4j logging system. So normally based on the discovery process introduced above, the security system will use log4j to log information.

With Log4j, the properties of the system are defined in a file `log4j.properties` in the classpath. When invoking the default deployment, the distributed Ant build file copies suitable copies of this and related log4j configuration and jar files to the appropriate tomcat directories. You may need to configure both the location and content of these files for your local configuration. Manuals can be found online,\(^{29,30}\)

\(^{27}\) http://jakarta.apache.org/commons/logging
\(^{28}\) http://jakarta.apache.org/commons/logging/apidocs/org/apache/commons/logging/impl/Log4JLogger.html
\(^{29}\) http://jakarta.apache.org/commons/logging/guide.html#Configuration
\(^{30}\) http://logging.apache.org/log4j/docs/documentation.html
6 Developer Guide

Section 6 is a brief supplement to the in Section 5. It adds material you need to know if you want to re-implement portions of the security system.

6.1 How to write an Authentication provider

The Authentication provider is the real user verifier when user using name-password to login. It also need provide the user profile for authentication service. It should implement IAuthenticationServiceProvider interface, which include verify(...) for validating the name-password pair and getRoles(...) for retrieving roles of a user, which will be contained in the token and used in authorization. The user role contains two parts, one is the jurisdiction and the other is the name of the role. See Footnote 19, page 2.

The distributed implementation is a toy, implemented with a simple file-based Java class edu.umb.cs.xmlsecurity.authentication.AuthenticationServiceProviderImpl.

6.2 How to write a Certificate provider

Certificate provider is the core of the trust chain of data provider. Only the token that is signed by trusted issuers will be honored in the token verification. In this project, there is an implementation using java key store.

If you want to implement your own provider, you need implement IIDDKeyStore. It contains three methods: getCertificate is used for retrieving the certificate using an alias parameter; getKey is used to get the private key; isCertificateEntry is used to check the existence of certificate with a specific name.

The distributed implementation is a simple Java class edu.umb.cs.xmlsecurity.authentication.IDDKeyStoreImpl

6.3 How to write a Token service

A token service has two main functionalities: one is to issue token; the other is to verify token. Normally it will query the backend authentication provider do the user verification work. Correspondingly in the IAuthenticationService that need to be implemented by every token service has two principal methods verifyToken(...) and getToken(...)

The distribution provides an implementation edu.umb.cs.xmlsecurity.authentication.AuthenticationServiceImpl based on SAML\(^{31}\) tokens.

In this implementation a token contains two parts: assertions and the signature of the issuer. In the assertion part it includes an assertion about the user name and its corresponding roles, and other additional assertions, such as the client IP address. You fetch those contents through the IToken interface. The method getUser(...) returns user name and roles; the method

\(^{31}\) http://www.oasis-open.org/committees/tc_home.php?wg_abbrev=security
getAttributeValue(...) returns other attribute-value pairs. See Section 7.2 (Replay threat) for further details.

6.4 How to write a Schema locator

The Schema locator is the service by which the PEP retrieves the schema and related annotation. The interface is ISchemaLocator which includes four methods:

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>getSchema</td>
<td>Retrieve the schema</td>
</tr>
<tr>
<td>getSensitiveAttributesForSchema</td>
<td>Retrieve the list of sensitive attributes</td>
</tr>
<tr>
<td>getRootElementName</td>
<td>Get root element of the schema</td>
</tr>
<tr>
<td>getRegularAttributesForSchema</td>
<td>Retrieve the list of other attributes</td>
</tr>
</tbody>
</table>

The distributed implementation is a simple file-based Java class edu.umb.cs.xmlsecurity.pep.SchemaLocator

6.5 How to write a Policy store

You can implement a PolicyFinderModule to replace the default file-based implementation. It is a class defined in Sun XACML implementation\(^{32}\).

The distributed implementation is a simple file-based Java class edu.umb.cs.xmlsecurity.pdp.SimpleFilePolicyFinder

\(^{32}\) http://sunxacml.sourceforge.net
7 Security threats and countermeasures

General discussions about Web Services security can be found in Securing Web Services with WS-Security by Jothy Rosenberg and David L. Remy., SAMS Publishing, 2004. Here we mention some standard threats, how they might arise, and how you should mitigate them.

7.1 Man in the middle

The UMB Access Control framework doesn’t directly provide any mechanism to prevent man-in-the-middle attacks. In order to prevent this kind of attack, some bilateral authentication is required. We suggest using HTTP over TLS/SSL to counter this problem.

7.2 Replay threat

Token Stealing can pose a potential security threat to this system, although a short token lifetime setting may ease the problem. Weak client or server security can expose the system to token stealing, but SAML provides two standard methods to help deal with this problem by providing syntax for the support of additional verification of the identity of the token holder: one is called Holder-of-Key, which means the token holder needs to sign the message with its private key to prove it is the rightful owner; another one is called Sender-Vouches, which means the request message (including the SAML token) is signed by a third party authority and the service provider has a trust relationship with that authority. For the first case, one has to maintain a key management system for user certificates, which may introduce a big maintenance challenge. For the second case, although the key management is not a big problem because of the relatively small number of providers, it may infeasible for clients to ask a voucher provider to forward data queries along with vouchers. Beside those standard methods, it is possible to take the IP address of the token holder (or other information that the server and client agree is reliable ) as the confirmation information. This method is still subject to an IP spoofing attack combined with token stealing. It also commits the token holder to an IP address that does not change between the time of token acquisition and the time of presentation of the query.

To support additional information in the token beyond that required by SAML, we add two new methods for token processing. In the ITokenIssuer interface, we provide a method called getToken(…) with an attributes parameter which is a set of attribute-value pairs. Through this method, the invoker can pass arbitrary attribute-value pairs that will appear in the generated token. Correspondingly, we also introduce a method called verifyToken(…) with a similar parameter, which is used by the verification service for comparison with the attribute values in the token.

7.3 Instance document depth overrun

Filter generation in UMB/AC depends on enumerating paths in the data schema. Because a schema may have recursively defined XML element types, the path enumeration may have an infinite loop. To solve this problem, a TDS may provide the depth of the instance document when invoking getFilter(…). If the TDS doesn’t provide that parameter, the class
FilterGenerator will use a default one defined in the Spring configuration file. Should the TDS provide a depth less than is actually presented in an instance document, then data which should be inaccessible will not be filtered out if it is deeper than the depth provided by the TDS. Similarly, should FilterGenerator use a default depth that is less than the actual one, deeper data will also be revealed even if the filter would not otherwise. In both cases, path enumeration having stopped at the given depth, the filters never see any elements below that depth.

### 7.4 Keystore password exposure

The distributed implementation of the authentication service uses the Sun JDK KeyStore class to manage an encrypted keystore (See Section 2.7, “Deployment of the authentication service.”). Access to keys through KeyStore.getKey(...) requires a cleartext password that was provided when the keystore was generated. This password is given in the Spring configuration file XREF and so is accessible to anyone with access to that file. In particular, the ability to regenerate the system will usually require such access. This means that anyone with access to both the configuration file and the keystore can extract arbitrary keys from the keystore. In general, simply providing operating system protection to the encrypted keystore alone will not defend against an exploit launched from a running system based on the injection with java reflection of a class implementing the java clients of KeyStore.getKey(...). Such an exploit would succeed launched by anyone with the password as found in the Spring configuration file.
8 Further reading

8.1 XACML reference

http://www.oasis-open.org/committees/tc_home.php?wg_abbrev=xacml

8.2 Spring reference

http://www.springframework.org/

8.3 SAML reference

9 Best Practices

9.1 Filter limitations

1. Access control policies have some restrictions on XPath attribute conditions. Section 9.2 has a detailed description.
2. For recursive schemas, the TDS must provide instance-document maximum depth at which sensitive information will be found, or such a default depth must be provided acceptable for all documents.
3. Filter creation performance will depend on controlled attribute combinations in the schema annotation, and on maximum instance tree depth.
4. We use a filter cache mechanism, and so application runtime performance is linear with instance document size.
5. Policy conflicts are handled using XACML combination mechanisms.33
6. A small amount of schema annotation (Section 3.3.1.1) is required to improve the path enumeration process.

In general, the schema path enumeration approach we have chosen presents some limitations in policy resource expression, and increased time for filter generation. We presently mitigate this with a filter cache.

9.2 XPath limitations in policies

1. Any restrictions imposed by the deployed transformation engine (e.g. Xalan for XSLT, Joost for STX, etc.) will apply to UMB/AC. For example, Xalan does not support XPath 2.0.
2. Only the following axes are supported in XPath axis traversal: child, descendent, and self.
3. Wildcards are supported along any supported axes.
4. Regular expression match between policy and enumerated path is not supported (see Example 1 below).
5. Noderset functions are not supported.
6. Only elements, not attributes, are permitted in the result node set of the XPath.

In fact, the restrictions above on the path enumeration approach can be relaxed if the conditions expressed below on nodese sets can be met. Failure to meet the conditions below can result in revelation of controlled data. Sometimes it is difficult to verify whether these conditions are met, but the above guarantee it.

In general, the current restrictions can be stated in terms of the nodese t E corresponding to the XPaths in the enumeration and the nodeset R of nodes corresponding to the XPaths in the policy rules. Our current implementation only considers two cases: R \( \leq \) E, and R \( \cap \) E is empty. In the

---

latter case all rules are inapplicable. In the former case (the rules only describe paths that are in the enumeration), the combination rules of the policies can all be meaningfully applied. In all other cases, because the combination rules apply only on \( R \cap E \), that intersection must be computed. There are two problems with this: (a) It is quite difficult to translate this intersection of nodesets into a set of XPaths which can be put into a filter and (b) the computation is exponential in the number of rules.

**Example 1:** XPath expression exp1 in policy:

\[
//B[@jurisdiction='NY[.].4']
\]

and enumerated XPath exp2:

\[
//B[@jurisdiction='N[..].4']
\]

The first of these has a nodeset that is a subset of the second and might be acceptable, but computing this in the face of arbitrary regular expressions has the difficulty mentioned above, and we exclude this in rule 4.34

**Example 2:** An expression like 

\[
//Element[not(jurisdiction = 'NY')]
\]

is supported in policies, and so are other boolean functions as well as string functions.

### 9.3 Best practices for performance and security: policies

1. Reduce the number of rules through combining similar rules

   If rules have similar structure and the same effect, they normally can be combined into one rule. In this way, we can effectively reduce the number of XPath evaluations, which consume the biggest part of PDP execution time.

   **Example 1:**
   - Rule 1: (a/b/c, deny)
   - Rule 2: (a/b/d, deny)
   - Combined rule: (a/b/*[fn:name(.) = 'c' or fn:name(.) = 'd'], deny)

   **Example 2:**
   - Rule 1: (a/c/b, deny)
   - Rule 2: (a/d/b, deny)
   - Combined rule: (a/*[fn:name(.) = 'c' or fn:name(.) = 'd']/b, deny)

   **Example 3:**
   - Rule 1: (a/c/b[@f = 'x'], deny)
   - Rule 2: (a/c/b[@f = 'y'], deny)
   - Combined rule: (a/c/b[@f = 'x' or @f = 'y'], deny)

2. Group similar rules

   If the rule has a similar structure but different effects, they can be grouped into one rule set sharing the same group filtering criteria.

   **Rule 1:** (a/b/c, deny)
   **Rule 2:** (a/b/d, permit)

---

34 This and others of the limitations would not be necessary with an architecture that generates filters entirely from the policies. But see Section 1.5 for the reasons we did not choose this architecture.
Policy set 1: (a/b/* [fn:name(.) = 'c' or fn:name(.) = 'd'])
Rule 1.1: (a/b/c, deny)
Rule 1.2: (a/b/d, permit)

Although such grouping may increase the number of XPath evaluations for any XPath that matches policy set group filtering, normally this prefiltering against the policies will significantly reduce the unnecessary evaluation for XPaths that would be filtered out by the policy set. A realistic and lengthy example which groups rules is found in Appendix D.
Appendix A  Some possible distributed deployments and their motivation

In the table below, boldface indicates that the functionality is configured as a SOAP-based Web Service. Otherwise, it is accessed by its clients through Java interfaces running on the same host.

<table>
<thead>
<tr>
<th>App Host</th>
<th>Auth. Host</th>
<th>PEP Host</th>
<th>PDP Host</th>
<th>Policy Store Host</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>A</td>
<td>A</td>
<td>A</td>
<td>A</td>
<td>This is the NatureServe configuration. In this case, all functionality is accessed as Java classes, none as web services.</td>
</tr>
<tr>
<td>A</td>
<td>B</td>
<td>C</td>
<td>D</td>
<td>D</td>
<td>Fully distributed. B, C, and D must be configured as SOAP services with trusted communication between them. It invokes only the Auth Host and the PEP host, which in turn accesses the PDP Host. The Policy Store Host is by default the PDP Host, but the Spring configuration allows an arbitrary class to implement the policy store. Such an implementation could be a proxy for another host, which must have secure access by the PDP host.</td>
</tr>
<tr>
<td>A</td>
<td>B</td>
<td>B</td>
<td>B</td>
<td>B</td>
<td>Only the Authentication and PEP services are exposed as SOAP services. Both are accessed by the Application Host. The PDP and policy store are accessed by host B through Java interfaces.</td>
</tr>
<tr>
<td>A</td>
<td>B</td>
<td>C</td>
<td>C</td>
<td>C</td>
<td>The Authentication Host and PEP host are accessed by the Application host as SOAP services over a secure communication channel. The PDP is only accessed by the PEP, so its communication is via Java interfaces with the PDP.</td>
</tr>
<tr>
<td>A</td>
<td>A</td>
<td>B</td>
<td>B</td>
<td>B</td>
<td>The Application Host provides its own authentication. It invokes the PEP host through a SOAP service, but the PEP invokes the other components through Java interfaces. It is unclear that this is a useful configuration.</td>
</tr>
</tbody>
</table>
## Appendix B  Comparison of STX filter and XSLT filter languages

<table>
<thead>
<tr>
<th>XSLT Filter</th>
<th>STX</th>
</tr>
</thead>
<tbody>
<tr>
<td>We support versions of XSLT derived from XPath 1.0.</td>
<td>XPath is restricted to STX Spec. In particular:</td>
</tr>
<tr>
<td>Because the path expression function <em>match</em> is defined only in XPath 2.0,</td>
<td>Only / and // axis is supported, so axis names like ‘child’ or ‘following’ etc. are not supported. Generally operations needing to look forward or look backward are not supported because STX processes the document in one pass. See Example 2.1 It is not possible for the engine to decide what to memorize given an arbitrary path expression. The STX filter scales well, having nearly fixed memory usage for big XML documents (about 34mb memory for a 512MB document)</td>
</tr>
<tr>
<td>we define and use an extension of the Xalan XSLT engine version 2.7 to do the regular expression match. This is not practical for transforming big documents (will get an out of memory exception on a 256M JVM while dealing with a 100mb+ document)</td>
<td></td>
</tr>
</tbody>
</table>


Appendix C  Summary Spring configuration. System Version 1.0.

The tables that follow in this Appendix assume familiarity with Section 1.4, “Architecture.” It summarizes information discussed in more detail in Section 3, “Configuration guide.”

The security module is organized based on XML configuration files which are used by the runtime library of the Spring Framework to maintain the relationship between components (beans, in the terminology of Spring). Version 1.0. of the system uses a single Spring file, `security_spring.xml`, for PEP, PDP and authentication, but a later version may have separate files.

Usually you change the behavior of the system by modifying properties of beans. In general, if a bean, say BeanB references BeanA, then the corresponding Java code for the implementation of BeanB must invoke the BeanA Java code through an interface implemented by BeanA. You can change the implementation class of a bean without breaking the system as long as you don’t change its interface. If the bean properties mentioned below are not specified in the Spring configuration file, a runtime exception—typically a null pointer exception—will be thrown when other clients of the security system try to use the functionality provided by it.
**Bean:** filterGenerator  
**Implementation Class:** pep.FilterGenerator

<table>
<thead>
<tr>
<th>Property</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>pdpRequestor</td>
<td>Spring <code>&lt;ref&gt;</code> element naming a bean implementing <code>IPDPRequestor</code> interface. In the distributed configuration file, it points to the bean <code>pdprequestor</code>.</td>
</tr>
<tr>
<td>schemaLocator</td>
<td>Spring <code>&lt;ref&gt;</code> element naming a bean implementing <code>ISchemaLocator</code> interface. In the distributed configuration file, it points to the bean <code>schemalocator</code>.</td>
</tr>
<tr>
<td>calculator</td>
<td>Spring <code>&lt;ref&gt;</code> element naming a bean whose implementation class inherits from <code>PathCalculatorAbstract</code> class. It is used to generate XPath expressions in the process of schema path enumeration used in the Version 1.0. filter generation.. In the distributed configuration file, it points to the bean <code>pathcalculatorequal</code>.</td>
</tr>
<tr>
<td>sensitiveControl</td>
<td>Boolean used to optimize path enumeration. If it is set, PEP will generate filters only controlling instance document elements whose sensitiveness attributes (i.e. those listed in the value of the property <code>pepID.sensitiveAttributeNames</code> of bean <code>schemalocator</code>) have value equal to that which signals sensitivity, as given in the values in property <code>pepID.sensitiveAttributeValues</code> corresponding to the names specified by <code>pepID.sensitiveAttributeNames</code>.</td>
</tr>
<tr>
<td>maxDepth</td>
<td>integer used in the schema enumeration when TDS doesn’t indicate the maximum depth of the instance document in its invocation of the PEP. It is used only when the schema of the XML document has recursive elements, in which case the PEP needs know when to stop traversing the possible XPaths. See section 7.3</td>
</tr>
</tbody>
</table>
| transformer      | Spring `<ref>` element naming a bean whose implementation inherits from the `AbstractTransformer` class. It corresponds to the kind of filter you want to use in a TDS. In the distributed system, there are two candidates for this bean: `STXTransformer` and `XsltTransformer`, which generate STX templates and XSLT templates respectively. N.B.: The value of this reference must be a bean defined elsewhere in the configuration file. In the distributed configuration, this is also named “transformer”, so the entire property element is  
  ```xml
  <property name="transformer">
    <ref local="transformer"/>
  </property>
  ```                                                                                                                                                                                                                                                                                                                                 |

**Bean:** transformer  
**Implementation Class:** pep.XsltTransformer or pep.STXTransformer  
(see filterGenerator.transformer, above)

<table>
<thead>
<tr>
<th>Property</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>version</td>
<td>standard version number of the template that will appear in the filter.</td>
</tr>
</tbody>
</table>
### Bean: schemalocator

**Implementation Class:** pep.SchemaLocator

<table>
<thead>
<tr>
<th>Property</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>pep.&lt;resource&gt;</td>
<td>path to the schema file for the resource for distributed pep.SchemaLocator, which is file based. In the Spring file, “&lt;resource&gt;” should be replaced by the name of the resource whose schema file is given by the value of this property.</td>
</tr>
<tr>
<td>pep.&lt;resource&gt;.sensitiveAttributeNames</td>
<td>name of the attribute in the XML document, whose value indicates the sensitiveness of the element to which it belongs. It can be a list of names with space as the delimiter. Only these and those specified by the property “pep.enumeratedAttributeNames” should be referenced in policies. Other attribute names are ignored by the PEP.</td>
</tr>
<tr>
<td>pep.&lt;resource&gt;.sensitiveAttributeValues</td>
<td>corresponding sensitive values of attributes defined above. Every attribute can only have one value defined here, whose meaning is this: if the attribute has this value, the element enclosing it is sensitive otherwise not. Values for different attributes are delimited by space and follow the same order as attribute names defined in pep.&lt;resource&gt; sensitiveAttributeNames above.</td>
</tr>
<tr>
<td>pep.&lt;resource&gt;.rootElement</td>
<td>root element of the document</td>
</tr>
<tr>
<td>pep.&lt;resource&gt;.enumeratedAttributeNames</td>
<td>name of the attribute of elements in the XML document that need to enumerate their values for reference in policies. Only these and sensitiveAttributeNames should be referenced in policies. Other attribute names are ignored by the PEP.</td>
</tr>
<tr>
<td>pep.&lt;resource&gt;.enumeratedAttributeValues</td>
<td>corresponding values of the attributes above. Every attribute can have more than one value. Values are separated by comma and the comma separated values for different attributes are delimited by space. These must follow the same order as attributes are defined in enumeratedAttributeNames above.</td>
</tr>
</tbody>
</table>

### Bean: pdprequestor

**Implementation Class:** pdp.PDPRequestor

<table>
<thead>
<tr>
<th>Property</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>pdp</td>
<td>Spring <code>&lt;ref&gt;</code> element naming a bean implementing <code>IPDP</code>.</td>
</tr>
</tbody>
</table>

---

35 The main difference between sensitive attribute and other attribute is the former support regular expression match for attribute.
<table>
<thead>
<tr>
<th>Bean: pdprequestorremote</th>
<th>Implementation Class: pdp.PDPRemoteRequestor</th>
</tr>
</thead>
<tbody>
<tr>
<td>Property</td>
<td>Value</td>
</tr>
<tr>
<td>endpoint</td>
<td>access point of the PDP web service</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Bean: pdp</th>
<th>Implementation Class: pdp.SimplePDP</th>
</tr>
</thead>
<tbody>
<tr>
<td>Property</td>
<td>Value</td>
</tr>
<tr>
<td>constructor-arg</td>
<td>Spring &lt;ref&gt; element naming a bean that implements PolicyFinderModule class. It is where the PDP find policies.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Bean: pathcalculatorequal</th>
<th>Implementation Class: pep.PathCalculatorWithPathAttributeEqual</th>
</tr>
</thead>
<tbody>
<tr>
<td>Property</td>
<td>Value</td>
</tr>
<tr>
<td>None</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Bean: filemodule</th>
<th>Implementation Class: pdp.SimpleFilePolicyFinder</th>
</tr>
</thead>
<tbody>
<tr>
<td>Property</td>
<td>Value</td>
</tr>
<tr>
<td>constructor-arg</td>
<td>full path to policy file</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Bean: authenticationService</th>
<th>Implementation Class: authentication.AuthenticationServiceImpl</th>
</tr>
</thead>
<tbody>
<tr>
<td>Property</td>
<td>Value</td>
</tr>
<tr>
<td>vf</td>
<td>backend authentication provider, which implements IAuthenticationProvider interface. It will check the validity of the user name and password, and return roles of the user in this system.</td>
</tr>
<tr>
<td>tokenVerifier</td>
<td>Spring &lt;ref&gt; element naming a bean used to check whether the token is issued by a trusted source and is still valid. It implements ITokenVerifier interface</td>
</tr>
<tr>
<td>tokenIssuer</td>
<td>Spring &lt;ref&gt; element naming a bean that implements interface ITokenIssuer</td>
</tr>
<tr>
<td>publicRole</td>
<td>name of a role if the user can not pass verification. If not supplied, code throws an AuthenticationFailureException. If supplied in Spring file, but there is no such role in any policy, then PEP gets “Not Applicable” and returns “Deny”.</td>
</tr>
</tbody>
</table>
### Bean: verifier
**Implementation Class:** authentication.AuthenticationServiceProviderImpl

<table>
<thead>
<tr>
<th>Property</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>None</td>
<td></td>
</tr>
</tbody>
</table>

### Bean: tokenIssuer
**Implementation Class:** authentication.TokenIssuerImpl

<table>
<thead>
<tr>
<th>Property</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>constructor-arg</td>
<td>Spring &lt;ref&gt; containing name of bean that describes key store for retrieving certificates to accompany token issuance.</td>
</tr>
<tr>
<td>tokenIssuer.keyStorePassword</td>
<td>password used to access key store as stored in the encrypted keystore when it was created. ▶️ The password is in clear text. See Section 7.4, “Keystore password exposure”.</td>
</tr>
<tr>
<td>tokenIssuer.keyAlias</td>
<td>alias of the private key entry in key store</td>
</tr>
<tr>
<td>tokenIssuer.duration</td>
<td>lifetime of the token</td>
</tr>
<tr>
<td>tokenIssuer.name</td>
<td>name of the token issuer, which must be the same as the alias of the private key entry in the issuer’s key store as well as the alias of its certificate entry in the key stores of any services wishing to verify the tokens of this issuer.</td>
</tr>
</tbody>
</table>

### Bean: tokenVerifier
**Implementation Class:** authentication.TokenVerifierImpl

<table>
<thead>
<tr>
<th>Property</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>constructor-arg</td>
<td>Spring &lt;ref&gt; containing name of bean that describes key store for retrieving trusted certificates used for token verification</td>
</tr>
<tr>
<td>tokenVerifier.keyStorePassword</td>
<td>password used to access key store as stored in keystore. ▶️ The password is in cleartext. See Section 7.4, “Keystore password exposure”.</td>
</tr>
<tr>
<td>tokenVerifier.keyAlias</td>
<td>name of the local token issuer</td>
</tr>
</tbody>
</table>
**Bean:** keyStore (in distributed Spring File)  
**Implementation Class:** authentication.IDDKeyStoreImpl

<table>
<thead>
<tr>
<th>Property</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>constructor-arg</td>
<td>filename of the key store used in Version 1.0 for both tokenIssuer and tokenVerifier key stores</td>
</tr>
</tbody>
</table>
| constructor-arg| The password of the key store. ! The password is in clear text.  
See Section 7.4, “Keystore password exposure”. |
Appendix D  XACML policy example

The example below applies to any subject (e.g., any role). According to line 5, policy rules are combined in order, but if a “Deny” rule appears, it over-rides all other rules. In fact, the Sun XACML processor would, upon reaching this rule, simply stop and return a decision of “Deny”.

The Policy controls two resources. The first, “theResource” (line 25-27) specifies that the Policy has target “theResource”. The semantics of <Target> is that this policy is only applicable to theResource and no further processing occurs if a request for a PDP decision presents some other Resource in its arguments. Assuming this is the desired Target, the other Resource controls access based on XML paths in the XML returned from the data provider. In particular, XPaths matching // geoLocation, (i.e., an element named “geoLocation” anywhere in the data (line 64-66)).

Beginning at line 44 is a rule whose effect is to deny read access (line 76-84) to such an “geoLocation” element for anyone authenticated to a global:public role, i.e. whose role string has a regular-expression match to any string containing “global:public” (line 49-57). Note that there is no mention of any action except “read” (line 78-79), so a PDP decision request for, say, “write”, will return “NotApplicable”. Note that several different data providers could have global:public roles, differing from one another only in their initial and final strings. The regular expression match here allows a single policy to be applied—deny read access to “geoLocation”—to anyone authenticated to a global:public role, and no matter what provider is offering “theResource”.

Finally, note that lines 90-141 are essentially default rules for any target and for action “read” and “write”. These rules could, in fact, be simplified if we wished to assume that there is no “write” action ever specified in any policy. In that case, a simple default rule of the form

  <Rule RuleID="defaultRule" Effect="Permit"/>

would suffice, because the context would never generate “write”.

Note that this example would only be realistic in case the XML returned by accessing theResource actually contains elements named geoLocation. However, that is not logically necessary: a data provider might offer several resources each with different sensitive XPaths, needing read-access denial to global:public roles. In that case, this example could be extended to apply to several resources and several XPath’s. Elsewhere (Section 9.3), we suggest exploiting such combinations of policies in order to improve the performance of filter generation and application.
<?xml version="1.0" encoding="UTF-8"?>
<Policy xmlns="urn:oasis:names:tc:xacml:1.0:policy"
xmlns:xsi=http://www.w3.org/2001/XMLSchema-instance
PolicyId="GeneratedPolicy" RuleCombiningAlgId="urn:oasis:names:tc:xacml:1.0:rule-combining-algorithm:ordered-deny-overrides">
<!-- Target element specifies subjects, resources, and actions used to decide if rules or policies are applicable to a certain request. You can assign a mapping function with the attribute MatchId. SubjectAttributeDesignator, ResourceAttributeDesignator and ActionAttributeDesignator are used to indicate which part of data in a PDP request to be used in the mapping process. Policy has an attribute "RuleCombiningAlgId" to specify how to solve conflicts if several rules have different effects on the same request. -->
<Target>
<Subjects>
<AnySubject/>
</Subjects>
<Resources>
<!-- this policy is for the resource type "theResource"-->
<Resource>
<ResourceMatch MatchId="urn:oasis:names:tc:xacml:1.0:function:anyURI-equal">
<!-- specify "theResource" as first argument of ResourceMatch -->
<AttributeValue
DataType="http://www.w3.org/2001/XMLSchema#anyURI">theResource</AttributeValue>
</ResourceMatch>
</Resource>
</Resources>
<Actions>
<AnyAction/>
</Actions>
</Target>
<!-- this rule means: 1. the global "public" role, not have the right to "read" "//geoLocation" that is under a sensitive element. -->
<Rule RuleId="ReadRule" Effect="Deny">
<Target>
<Subjects>
<Subject>
<!-- this rule is for the "public" role -->
<SubjectMatch MatchId="urn:oasis:names:tc:xacml:1.0:function:regexp-string-match">
<AttributeValue DataType="http://www.w3.org/2001/XMLSchema#string">.*global:public.*</AttributeValue>

<SubjectAttributeDesignator DataType="http://www.w3.org/2001/XMLSchema#string" AttributeId = "role"/>

</SubjectMatch>

</Subject>
</Subjects>
<Resources>
  <Resource>
    <ResourceMatch MatchId="urn:oasis:names:tc:xacml:1.0:function:generalxpathmatch">
      <AttributeValue DataType="http://www.w3.org/2001/XMLSchema#string">//geoLocation</AttributeValue>
      <ResourceAttributeDesignator DataType="http://www.w3.org/2001/XMLSchema#string" AttributeId="urn:oasis:names:tc:xacml:1.0:resource:resource-path"/>
    </ResourceMatch>
  </Resource>
</Resources>
<Actions>
  <Action>
    <ActionMatch MatchId="urn:oasis:names:tc:xacml:1.0:function:string-equal">
      <AttributeValue DataType="http://www.w3.org/2001/XMLSchema#string">read</AttributeValue>
      <ActionAttributeDesignator DataType="http://www.w3.org/2001/XMLSchema#string" AttributeId="urn:oasis:names:tc:xacml:1.0:action:action-id"/>
    </ActionMatch>
  </Action>
</Actions>
</Target>
</Rule>

<Rule RuleId="DefaultReadRule" Effect="Permit">
  <Target>
    <Subjects>
      <AnySubject/>
    </Subjects>
    <Resources>
      <Resource>
        <AnyResource/>
      </Resource>
    </Resources>
    <Actions>
      </Actions>
  </Target>
</Rule>
<Action>
  <ActionMatch MatchId="urn:oasis:names:tc:xacml:1.0:function:string-equal">
    <AttributeValue DataType="http://www.w3.org/2001/XMLSchema#string">read</AttributeValue>
    <ActionAttributeDesignator DataType="http://www.w3.org/2001/XMLSchema#string" AttributeId="urn:oasis:names:tc:xacml:1.0:action:action-id/>
  </ActionMatch>
</Action>
</Target>
</Rule>

<Rule RuleId="DefaultWriteRule" Effect="Deny">
  <Target>
    <Subjects>
      <AnySubject/>
    </Subjects>
    <Resources>
      <Resource>
        <AnyResource/>
      </Resource>
    </Resources>
    <Actions>
      <Action>
        <ActionMatch MatchId="urn:oasis:names:tc:xacml:1.0:function:string-equal">
          <AttributeValue DataType="http://www.w3.org/2001/XMLSchema#string">write</AttributeValue>
          <ActionAttributeDesignator DataType="http://www.w3.org/2001/XMLSchema#string" AttributeId="urn:oasis:names:tc:xacml:1.0:action:action-id"/>
        </ActionMatch>
      </Action>
    </Actions>
  </Target>
</Rule>
# Appendix E  Known defects

This appendix documents defects in the distributed system known as of February 3, 2007, as recorded in the UMASS-Boston Bugzilla. None are deemed critical.

<table>
<thead>
<tr>
<th>ID</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>397</td>
<td>All text should be from property files in support of internationalization.</td>
</tr>
<tr>
<td>398</td>
<td>Schema annotation needs more formal description and implementation. Maybe an XML-Schema for schema annotation?</td>
</tr>
<tr>
<td>399</td>
<td>The components of the Spring configuration file should each be in their own Spring file for ease of documenting and maintaining a fully distributed deployment.</td>
</tr>
<tr>
<td>400</td>
<td>In v 1.1.2.6 of AuthenticationServiceImpl, the public role is created by the code if there is a Spring entry for defaultRole. Probably this should not be the responsibility of the code.</td>
</tr>
<tr>
<td>401</td>
<td>There should be a dynamic mechanism for choosing which kind of transformer to use. Ideally, an instance document would arrive with trusted metadata telling which transformer to use. It might even be that a trusted implementation class of the transformation interface IXMLTransformer is dynamically loaded when the TDS gets the filter.</td>
</tr>
<tr>
<td>404</td>
<td>In the example JSP files, URLs are hard coded and need change for deployment. However, the current URL and other context in tomcat are detectable. If this were done, then a standard deployment not invoking a SOAP service for those calls would simply work. It would be better to do that and also fix the Configuration Document at Section 2.5 &quot;Deploying the system on a single host&quot; to reflect that these changes are needed only in a few cases.</td>
</tr>
<tr>
<td>405</td>
<td>There should be a simple, single JSP page that has a good chance of indicating that installation and deployment are correct. This could be tricky for less than a completely local deployment.</td>
</tr>
<tr>
<td>409</td>
<td>The alternative to schema path enumeration, simple direct filter generation from policies should be supported for environments that have small simple policies and small documents.</td>
</tr>
<tr>
<td>410</td>
<td>The test page MyJsp2.jsp should by default be relative to the deployment server, at least on a single-host deployment (and it should be renamed HappyACL.jsp in honor of HappyAxis.jsp.).</td>
</tr>
<tr>
<td>411</td>
<td>Possibly there should be deployment unit tests for each component similar to the current MyJsp2.jsp e.g. HappyPDP.jsp etc. with static test data that would serve to show any deployment problems. See also #410</td>
</tr>
<tr>
<td>412</td>
<td>security_spring.xml and any other configuration files should be (re)refactored into separate files for each component to better promote fully distributed deployment.</td>
</tr>
<tr>
<td>ID</td>
<td>Description</td>
</tr>
<tr>
<td>-----</td>
<td>-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------</td>
</tr>
<tr>
<td>413</td>
<td>The NatureServe notion of &quot;jurisdiction&quot; as a first class concept is not general. In point of fact, the way they use it, it is really part of a role, and is implementing what amounts to a role hierarchy, perhaps of the form jurisdiction.rolename. Indeed, jurisdiction is a notion particular to the NatureServe IDD deployment. In practice, it simply identifies the data provider. In reality, one can understand that deployment as though roles themselves had two substrings, one called “jurisdiction” and the other the name of a role typically common to all the data providers. Indeed, the jurisdiction and role are concatenated in the distributed PDP implementation, and policies typically exploit string regular-expression matches on role names. Possibly a urn scheme should be defined for roles in some hierarchical way.</td>
</tr>
<tr>
<td>414</td>
<td>The ontology-driven policy combination work in our paper should be extensively implemented.</td>
</tr>
<tr>
<td>415</td>
<td>There should be greater ability to defend the keystore passwords. Perhaps at least in their own file? Anyone with access to both the configuration file and the keystore can extract arbitrary keys from the keystore. In general, simply providing operating system protection to the encrypted keystore alone will not defend against an exploit launched from a running system based on the injection with java reflection of a class implementing the java clients of KeyStore.getKey().</td>
</tr>
<tr>
<td>417</td>
<td>The keyStore Spring bean specification implies that there is only one keystore, despite the fact that the PEP beans could support two keystores, one for the tokenIssuer and one for the tokenVerifier with possibly different passwords. This seems to imply that parts of the system are factored</td>
</tr>
<tr>
<td>420</td>
<td>The filter cache construction in and access FilterGenerator should be isolated in its own class and controlled by Spring so that a cache congenial to a particular usage pattern can be injected at system startup, perhaps even one that is a wrapper around several cache classes, one of which is a startup argument to the system.</td>
</tr>
<tr>
<td>421</td>
<td>It's a research project to decide whether any of the system lends itself to datamining exploits against internal correlations in the data. In particular, can there be hooks for the support of inadvertent addition of internal correlation between sensitive and insensitive data in the face of write access rights.</td>
</tr>
</tbody>
</table>
Appendix F  Future work

There are no outstanding reported debilitating bugs, but we include in Appendix E the current open defect list desired from our Bugzilla database. Currently there are only outstanding suggested enhancements on that list, of which only two are consequential research and may qualify as major future work. The others are relatively minor and will proceed as minor releases are made in the future.

The major projects are:

- A more formal description of schema annotation is necessary, and annotation applications should be driven by this formalism. In particular, dynamic schema annotation should be supported as part of the schema location service interface.
- Research on more sophisticated policy combinations based formal logic will continue, based on our manuscript Rule based Security Policy management for Web Service Integration 36
- More thorough investigations of write access, especially defense against inadvertently increasing the internal correlation of sensitive to insensitive data that might raise the risk of data mining exploits.

The minor enhancements include:

- There is early version code that implemented the “simple” filter generation, depending on policies only. This will be reintegrated into the code base with Spring-configurable deployment.
- Keystore passwords should be separated from the Spring configuration file and if possible have greater protection. See Section 7.4. “Keystore password exposure”.
- All generated text will be put in external files to support internationalization.
- The system as a whole is highly modularized, and so should be the Spring file, as it was in early versions.
- Role syntax should be more formalized and role-aware interfaces should be driven by a formal syntax. This will remove the NatureServe “jurisdiction” concept which intrudes in the code in very minor ways.
- Refactoring the filter cache mechanisms for greater flexibility.

Appendix G  Special considerations for NatureServe IDD

Data flow

Figure 5. Detailed dataflow in NatureServe 2007 implementation. Courtesy of Dan Shoutis, NatureServe.