A Comparison of Modeling Strategies in Defining XML-based Access Control Languages

Claudio Ardagna Sabrina De Capitani di Vimercati
Dip. di Tecnologie dell'Informazione
Università di Milano
26013 Crema, Italy
{ardagna,decapita}@dti.unimi.it

Abstract

One of the most important features of XML-based Web services is that they can be easily accessed over the Internet, but this makes them vulnerable to a series of security threats. What makes security for web services so challenging is their distributed and heterogeneous nature. Access control policy specification for controlling access to Web services is then becoming an emergent research area due to the rapid development of Web services in modern economy. Two relevant access control languages using XML are WS-Policy and XACML. The main conceptual difference between these two languages is that while XACML is based on a well-defined model that provides a formal representation of the access control security policy and its working, WS-Policy has been developed without taking into consideration this modeling phase.

In this paper, we critique WS-Policy pointing out some of its shortcomings. We then describe the architecture we implemented and that offers an interface for controlling access to Web services.

1 Introduction

Accessing information on the global Internet has become an essential requirement of the modern economy. Recently, the focus has shifted from access to traditional information stored in WWW sites to access to large *e-services* such as e-government services, remote banking, or airline reservation systems [10]. *Distributed Web service applications* are also coming of age, designed as custom applications exploiting single Web services already available on the Internet and integrating the results. The concept of Web service is similar to the concept of object in the object oriented programming paradigm: Web services provide some functionalities (services) to the clients through an interface well defined. The main difference between Web services and objects is that a Web service can be available on the Web and can be accessed by sending messages (e.g., SOAP messages), requesting specific actions, and receiving message responses (including fault indications). Calls to Web services, and in general e-services, are more easily modeled by *distributed object protocols* as *Remote Method Calls (RMCs)*, such as *CORBA* [19], *DCOM* [7], and *Java-RMI* [12], in which clients pass parameters to remote components and get some kind of result in return. However, these approaches exhibit two main problems that prevent their large-scale use on the Net: *verbosity* and

firewall traversal and user authentication. Many RMC-based protocols require considerable bandwidth due to their high service to data packets ratio and often many organizations are reluctant to enable RMC-based protocols. To avoid these problems, the Internet and Web communities have provided several proposals for the use of XML in *lightweight* network protocols and distributed applications: SOAP [18], WSDL [8], and UDDI [20] to name a few. One of the most important features of Web services is the possibility to use them for communicating and automatically exchanging information. Cross-enterprise exchange of information over the Internet is vital but may have security implications. There are three levels at which Web service security can be applied: *transport level*, *authentication level*, and *application level*. The transport level is probably one of the easiest ways to introduce security into a Web service architecture. The most widely used security mechanisms are HTTPS (i.e., the combination of HTTP and the Secure Sockets Layer protocol), and firewalls. The advantage of the transport layer security is that an application merely asks for a secure connection and no other application requirements are needed. However, this approach has two limitations. First, it allows for securing the connection only, not the data itself. This means that while the data is in transit, it will be secure. But, at the point it reaches the destination or any intermediary, there is nothing to protect the information. For Web services, protection at the destination is vital because the data could be processed by many intermediaries. The other limitation is that it is basically an all-or-nothing operation: it is not possible to selectively control what specific data is encrypted and who might be authorized to view it. The second level of security can principally provide two security mechanisms: a third party authentication mechanism which uses a a credential repository, and a certificate-based mechanism. The third level of security deals with securing the message itself. To this purpose, some of the available XML standards to help encrypt or sign the data can be used. XML digital signatures are typically used for data integrity, authentication, and non-repudiation. For instance, application level security can be used to validate that a given SOAP message cames from a particular party and that it has not been modified. XML Encryption adds the confidentially aspect, indicating whether the data can be viewed by the receiver. These XML security standards have been around for a while and are fairly stable for development projects. There are other Web services security standards that are still being defined by the industry: Security Assertion Markup Language (SAML) [17] and WS-Security [2]. SAML is an XML-based framework for exchanging security information defined by the OASIS organization. The SAML specification defines how to represent security credentials (*assertions* in SAML) using XML. SAML is designed to enable secure single sign-on to applications within organizations and across companies and supports ten different authentication mechanisms: the combination of username and password, tickets Kerberos, Secure Remote Password protocol (RFC 2945), token hardware, SSL (Secure Sockets Layer) client-side certificate, X.509 certificate, Pretty Good Privacy, Simple Public Key Infrastructure, XKMS (XML key management specification), and XML Digital Signature. After the subject authentication, the server SAML returns a particular security token to the client that makes the initial request. This security token has limited time validity and, in such cases, can offer only some kinds of access (read, write, or delete). WS-Security is being developed by IBM, Microsoft, and Verisign. WS-Security is a means of using XML to encrypt and digitally sign SOAP messages. It also provides a mechanism for passing security tokens for authentication and authorization for the SOAP messages. A typical example of security token is a user name and password token, in which a user name and password are included as text. SAML can provide a way to create the tokens used in WS-Security. The authentication mechanisms supported by WS-

Security are: the combination of username and password (a password can be sent unprotected or in hashed format), tickets Kerberos, and X.509 certificates. Element `Security` is used for including security information in a SOAP header message. Although WS-Security does not address other security issues, such as authorization or access control, WS-Security represents a useful initiative to support other security services.

Another important class of security languages that can be used at the application level are the *access control* (or *policy*) languages. Several proposals have been introduced for access control to distributed heterogeneous resources from multiple sources based on the use of attribute certificates [3, 9, 11, 13]. Two relevant access control languages using XML are WS-Policy [6] and XACML [15]. Based on the WS-Security, WS-Policy provides a grammar for expressing Web service policies. The WS-Policy includes a set of general messaging related assertions defined in WS-PolicyAssertions [4] and a set of security policy assertions related to supporting the WS-Security specification defined in WS-SecurityPolicy [21]. In addition to the WS-Policy, WS-PolicyAttachment [5] defines how to attach these policies to Web services or other subjects such as service locators. The eXtensible Access Control Markup Language (XACML) [15] is the result of a recent OASIS standardization effort proposing an XML-based language to express and interchange access control policies. XACML is designed to express authorization policies in XML against objects that are themselves identified in XML. The language can represent the functionalities of most policy representation mechanisms. An XACML policy consists of a set of rules whose main components are: a target, an effect, and a condition.[1] The target defines the set of resources, subjects, and actions to which the rule is intended to apply. The effect of the rule can be `permit` or `deny`. The condition represents a boolean expression that may further refine the applicability of the rule. A request consists of attributes associated with the requesting subject, the resource involved in the request, the action being performed and the environment. A response contains one of four decisions: `permit`, `deny`, `not applicable` (when no applicable policies or rules could be found), or `indeterminate` (when some errors occurred during the access control process). A request, a policy, and the corresponding response form the `XACML Context`.

The main conceptual difference between XACML and WS-Policy is that while XACML is based on a model that provides a *formal* representation of the access control security policy and its working, WS-Policy has been developed without taking into consideration this modeling phase. The result is an ambiguous language that is subject to different interpretations and uses. This means that given a set of policies expressed by using the syntax and semantics of WS-Policy, their evaluation may have a different result depending on how the ambiguities of the language has been resolved. This is obviously a serious problem especially in the access control area [1], where access decisions have to be deterministic. The contributions of this paper can be summarized as follows. First, we point out some of the WS-Policy shortcomings, showing how they can be fixed (see Section 3). Second, we describe the architecture we have implemented and that offers an interface for controlling access to Web services (see Section 4), and illustrate how the policy enforcement process works (see Section 5).

[1] We keep at a simplified level the description of the language and refer the reader to the OASIS proposal [15] for the complete specification.

Value	Meaning
`wsp:Required`	The assertion must be applied to the subject. If the assertion is not satisfy, a fault or error will occur.
`wsp:Rejected`	The assertion is not supported and if present will cause failure.
`wsp:Optional`	The assertion may be applied but it is not required.
`wsp:Observed`	The assertion will be applied and requestors of the service are informed that the policy will be applied.
`wsp:Ignored`	The assertion is processed, but ignored; no action will be taken as a result of it being specified. Subjects and requestors are informed that the policy will be ignored.

Figure 1: Values for attribute `Usage`

2 WS-Policy overview

Web Service Policy framework (WS-Policy) provides a generic model and a flexible and extensible grammar for describing and communicating the policies of a Web service [6]. Other specifications, such as *WS-PolicyAssertions* [4] and *WS-SecurityPolicy* [21], provide specific applications of this grammar for their domains. A policy is a collection of one or more *policy assertions* that represent an individual preference, requirement, capability, or other properties that have to be satisfied to access to the *policy subject* associated with the assertion. The XML representation of a policy assertion is called *policy expression*.[2] Element `wsp:Policy` is the container for a policy expression. Policy assertions are typed and can be *simple* or *complex*. A simple assertion can be compared to other assertions of the same type without any special consideration about the semantics' assertion. A complex assertion requires an assertion type-specific means of comparison. The assertion type can be defined in such a way that the assertion is parametrized. For instance, an assertion describing the maximum acceptable password size (number of chars) would likely accept an integer parameter indicating the maximum char count. In contrast, an assertion that simply indicates that a password is required does not need parameters; its presence is enough to convey the assertion. Every assertion is associated with an obligatory usage qualifier (attribute `Usage`) that specifies how the assertion should be processed. Figure 1 illustrates the five possible values for attribute `Usage` together with their meaning.

In cases where there are multiple choices for granting a given access (e.g., different authentication mechanisms), attribute `wsp:preference` can be used to establish an order among the different choices. Possible values for attribute `wsp:preference` are integers, where a higher number represents a higher preference. WS-Policy also provides an element, called `wsp:PolicyReference`, that can be used for sharing policy expressions between different policies. Conceptually, when a reference is present, it is replaced by the content of the referenced policy expression.

WS-Policy defines a set of *policy operators* for combining policy assertions. More precisely, WS-Policy provides three different types of policy operators:

- `wsp:All` requires that all of its child elements be satisfied;

[2]Note that using XML to represent policies facilitates interoperability between heterogeneous platforms and Web service infrastructures.

```
<wsp:Policy xmlns:wsp="..." xmlns:wsse="...">
  <wsp:ExactlyOne>
    <wsp:All wsp:Preference="100">
      <wsse:SecurityToken>
        <wsse:TokenType>wsse:Kerberosv5TGT</wsse:TokenType>
      </wsse:SecurityToken>
      <wsse:SecurityToken>
        <wsse:TokenType>wsse:UsernameToken</wsse:TokenType>
        <wsse:Username>Claudio</wsse:Username>
      </wsse:SecurityToken>
    </wsp:All>
    <wsp:All wsp:Preference="1">
      <wsse:SecurityToken>
        <wsse:TokenType>wsse:X509v3</wsse:TokenType>
      </wsse:SecurityToken>
      <wsse:SecurityToken>
        <wsse:TokenType>wsse:UsernameToken</wsse:TokenType>
        <wsse:Username>Claudio</wsse:Username>
      </wsse:SecurityToken>
    </wsp:All>
    <wsp:PolicyReference URI="#opts" />
  </wsp:ExactlyOne>
</wsp:Policy>
```

(a)

```
<wsp:Policy xmlns:wsse="..." xmlns:ns="...">
  <wsp:All wsu:Id="opts">
    <wsse:SecurityToken>
      <wsse:TokenType>wsse:X509v3</wsse:TokenType>
    </wsse:SecurityToken>
    <wsse:SecurityToken>
      <wsse:TokenType>wsse:UsernameToken</wsse:TokenType>
      <wsse:Username>Sabrina</wsse:Username>
    </wsse:SecurityToken>
  </wsp:All>
</wsp:Policy>
```

(b)

Figure 2: A simple example of policy (a) and the corresponding referred policy (b)

- `wsp:ExactlyOne` requires that exactly one of its child elements be satisfied;
- `wsp:OneOrMore` requires that at least one of its child elements be satisfied.

Lack to specify a policy operator is equivalent to specify the `wsp:All` operator. Figure 2(a) illustrates a simple example of policy stating that the access is granted if exactly one security token among the following is provided: *i)* a Kerberos certificate and a UsernameToken with Username `Claudio`, *ii)* an X509 certificate and a UsernameToken with Username `Claudio`, or *iii)* an X509

	Assertions			
Acceptable policy set	**X.509**	**Kerberos**	**3DES**	**AES**
Set 1	True	False	True	False
Set 2	False	True	False	True

Table 1: Tabular representation of Policy 1

	Assertions	
Acceptable type	**X.509**	**Kerberos**
Usage	`Required`	`Rejected`

(a)

	Assertions	
Acceptable policy set	**X.509**	**Kerberos**
Set1	True	False

(b)

Table 2: Assertions with their `Usage` (a) and their representation as boolean predicates (b)

certificate and a UsernameToken with Username `Sabrina`. The third option corresponds to the referred policy, called `opts`, illustrated in Figure 2(b).

2.1 Policy Assertion sets

A policy can be viewed as a table, where each column is an assertion and each row is a policy assertion set that verifies the policy. For instance, consider the following policy:

Policy. Either only a certificate X.509 and 3DES encryption or a certificate Kerberos and AES are required and all other combinations are invalid

Table 1 illustrates a tabular representation of `Policy` where the first row corresponds to the alternative "X.509 and 3DES are required, all other combinations are invalid" and the second row corresponds to the alternative "Kerberos and AES are required, all other combinations are invalid." A policy assertion set combines one or more assertions with their `Usage`. For instance, consider a policy with two assertion types: X.509 and Kerberos (each assertion has no additional parameters). The policy set illustrated in Table 2(a) states that X509 is `Required` and Kerberos is `Rejected`.

To simplify the analysis and processing of policy assertion sets, simple assertions can be viewed as Boolean predicates, with `Required` usage resulting in True and `Rejected` usage resulting in False. The example in Table 2(a) can then be described as shown in Table 2(b). It should be noted that complex assertion types require more sophisticated logic and must be handled in a type-specific manner. A simple assertion with `Optional` usage produces two boolean sets, where the first results in a True value and the second one results in a False value. For instance, if X509 assertion has been marked as `Optional` and Kerberos assertion as `Required`, the resulting sets are those illustrated in Table 3.

	Assertions	
Acceptable policy set	**X.509**	**Kerberos**
Set 1	True	True
Set 2	False	True

Table 3: Assertions as boolean predicates

```
<wsp:Policy xmlns:wsse="..." xmlns:wsp="...">
  <wsp:All>
    <wsse:SecurityToken wsp:Usage="wsp:Required">
      <wsse:TokenType>wsse:Kerberosv5TGT</wsse:TokenType>
      ...
    </wsse:SecurityToken>
    <wsse:SecurityToken wsp:Usage="wsp:Required">
      <wsse:TokenType>wsse:X509v3</wsse:TokenType>
      ...
    </wsse:SecurityToken>
    <Language Usage="Optional" Language="it"/>
  </wsp:All>
</wsp:Policy>
```

Figure 3: Example of an ambiguous combination between attribute `Usage` and policy operator `All`

3 A critique of WS-Policy

The development of an access control system is a multi-phase approach that involves a modeling phase [16]. This phase allows the definition of a formal model representing the policy and its working, making it possible to define and prove security properties that systems enforcing the model will enjoy [14]. Unfortunately, WS-Policy has been described only by means of XSD-Schemas and tutorials and therefore a formal representation is completely missing. This informal description is not enough to make WS-Policy a language without ambiguities and makes it interpretable and subject to different implementations and uses. In the following, we will describe the ambiguities that we have pointed out in WS-Policy, and we will propose and discuss possible solutions.

3.1 Ambiguous combination

Consider a policy where the policy operator `All` includes one or more assertions for which the attribute `Usage` has value `Optional`. Here, an ambiguity arises because it is not clear whether the `Optional` assertions should be considered or not. For instance, consider the policy illustrated in Figure 3. This policy can be interpreted in two different ways:

Interpretation 1. All assertions must be satisfied, including the optional one. This is equivalent to say that the semantics of the policy operator takes precedence over the semantics of attribute `Usage` with the `Optional` value.

Interpretation 2. Only the first two assertions must be satisfied while the language assertion is optional. This is equivalent to say that the semantics of attribute `Usage` with the `Optional` value takes precedence over the semantics of the policy operator.

Suppose now that a request includes a valid X509 certificate and a valid Kerberos certificate and does not provide a language. According to Interpretation 1, the access will be rejected; it will be granted, otherwise. This means that, the assertions marked `Optional` have to be controlled in accordance with the policy operator in which they are included. More precisely, we propose to resolve this ambiguity as follows.

- `All`: the optional assertions must be ignored.
  If the assertions marked `Required` are satisfied, it is not necessary to control the optional assertions. Otherwise, if at least one `Required` assertion evaluates to False, the operator `All` is not satisfied, and the optional assertions can be ignored.
- `ExactlyOne`: the `Optional` assertions must be checked if and only if the `Required` assertions are not satisfied.
- `OneOrMore`: the `Optional` assertions must be checked if and only if at least one of the `Required` assertions evaluate to False. In this case, the `Optional` assertions must be checked until one of them evaluates to True.

The same happens when the combination between operator `All` and attribute `Usage` with value `Ignored` is considered.

Another possible combination that seems meaningless is between element `usePassword` and attribute `Usage`. In principle, it is possible to define an assertion where element `usePassword` has an attribute `Usage` with value `Rejected`. In this case, the access is denied to all users that provide a password. It is clear that this combination is nonsense and should not be allowed.

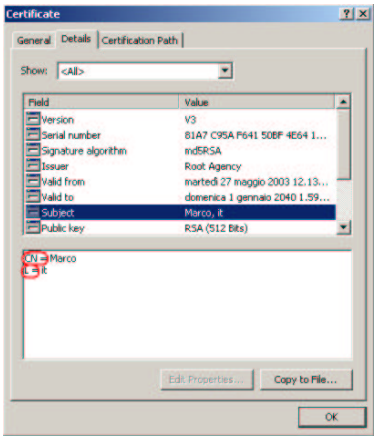
3.2 Ambiguous attributes

WS-Policy includes attributes that appear useless or whose meaning and semantics is ambiguous.

Usage="Observed". Assertions with value `Observed` for attribute `Usage` "will be applied to all subjects, and requesters of the service are informed that the policy will be applied". It is not clear for what reason a requester should be informed that a policy is applied. Without this feature, the value `Observed` has the same meaning of value `Required`.

Usage="Ignored". Assertions with value `Ignored` for attribute `Usage` must be processed and then the results must be ignored. This specification seems useless.

Preference. It has no any practical effect and it seems unnecessary especially in the cases where the assertions with which is associated are included in the `All` or `OneOrMore` operators.



(a)

```
<wsp:Policy Priority="...">
  <All>
    <MessageAge Usage="Required" Age="1000000"/>
    <SecurityToken TokenType="X509Security" Usage="Required"
      MatchType="Prefix">
      <Claim>
        <SubjectName> L=it </SubjectName>
      </Claim>
    </SecurityToken>
    <Language Usage="Required" Language="it"/>
  </All>
</wsp:Policy>
```

(b)

Figure 4: An example of an X.509 certificate (a) and an example of policy (b)

Match. This attribute can assume two values: `exact` and `prefix`.[3] Value `exact` means that the subject name of the certificate must be exactly the same of the value indicated in the assertion. Value `prefix` means that the value specified in the assertion must be a prefix of the subject name of the certificate. Consider now the simple certificate illustrated in Figure 4(a) and the policy depicted in Figure 4(b). According to the WS-Policy specification, when attribute `Match` has value `prefix`, the system has to control that the strings preceding the symbol "=" (equal) are the same. We decide instead to apply a semantics similar to the semantics of operator `like` in SQL: in our example, the `SubjectName` of the policy should then be a substring of the subject specified in the certificate.

3.3 Missing features

We decide to extend the WS-Policy specification by adding the following features.

Priority. We decide to reuse attribute `Priority` to efficiently manage the error messages communicated to the user. For instance, suppose that all policies applicable to a given access request evaluate to False. In this case, for each policy there is an exception message and the system has to decide what exception should be communicate to the client. Attribute `Priority` can then be used to establish an order between the exceptions raised by the policies: the exception associated with the policy with higest `Priority`'s value is selected. If more than one policy have the same highest value, then the selected exception is the exception raised by the the first policy that evaluates to False.

[3] Note that there are assertions for which a third value, `regexp`, is allowed.

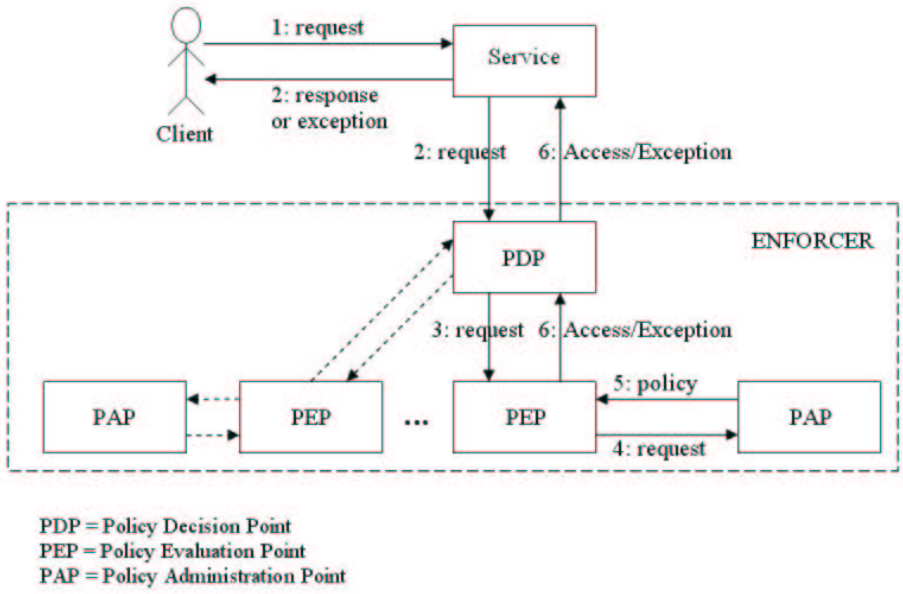


Figure 5: Architecture

Password. WS-Policy specification allows the specification of assertions stating that the access can be granted if a given username (token of type `UsernameToken`) is provided and if the authentication mechanism consists in providing a password (`UsePassword` element). We add the possibility to also restrict the access only if a given password is specified. To this purpose, we add element `Password` whose content is in encrypted form and corresponds to the given password. Element `Password` and `UsePassword` are mutually exclusive. An example of policy with element `Password` is the following.

```
<SecurityToken wsp:Preference="..." wsp:Usage="..." wsu:id="...">
  <TokenType>wsse:UsernameToken</TokenType>
  <Claims>
    <SubjectName MatchType="...">...</SubjectName>
    <Password wsp:Usage="..." Type="...">...</Password>
  </Claims>
</SecurityToken>
```

4 Specifications and Architectural Design

We describe the architecture we have developed in order to control access to Web services according to given policies. The proposed architecture satisfies the following seven requirements.

- *Modularity.* The architecture is realized as a set of independent modules. This allows for implementing and updating the different modules by different entities reducing, for example, the development time.

- *Policy language independent.* The architecture is independent from the specific access control language. This allows for re-using the architecture also if the policy language should change.
- *Extensibility.* The architecture can be easily extended. This is obtained by duplicating the architecture's modules thus resulting in additional levels of control.
- *Re-Usability.* The architecture's modules can be reused and shared among different entities; it makes the architecture generic and scalable.
- *High performance.* The architecture has a low time response to eventually support real-time applications.
- *Hardware and software independent.* The architecture allows for distributing different modules on different machines with different operating systems.
- *Programming language independent.* The modules' developers can use the preferred programming language without any restriction.

As shown in Figure 5, the architecture includes an *Enforcer* module that wraps up entirely the computation of access permissions to individual Web services, returning, for each request, the decision of whether the access should be granted or denied. Internally, the Enforcer is composed of three main modules implemented as Web services:

- The *Policy Decision Point* (PDP) module receives an access request and returns a "yes" or "no" decision.
- The *Policy Evaluation Point* (PEP) module interacts with the PAP that encapsulates the information needed to identify the applicable policies. It then evaluates the request against the applicable policies and returns the final decision to the PDP module.
- The *Policy Administration Point* (PAP) module retries the policies applicable to a given access request and returns them to the PEP module.

When a client submits an access request to a *service* (1), the service redirects the request to the PDP module (2). The request is then evaluated by the PEP (3) by first propagating the request to the PAP module (4) and then by enforcing the applicable policies returned by the PAP module (5). The PEP module returns the final decision to the PDP (6) that propagates it to the service. If the access is denied, the PDP returns an exception string to the service that can decide either to redirect the exception to the client as it is, or to process the exception and to retur to the client a message conforming to the communication specifications.

Note that since PDP, PEP, and PAP modules have been implemented as Web services, their use can be controlled by means of an instance of PDP-PEP-PAP. For instance, suppose that the PAP module requires the authentication of the PEP module. In this case, the PAP module has to instantiate a new PDP and at least a pair of PEP-PAP (see Figure 6).

In the remainder of this section, we briefly describe each of the main components of the Enforcer.

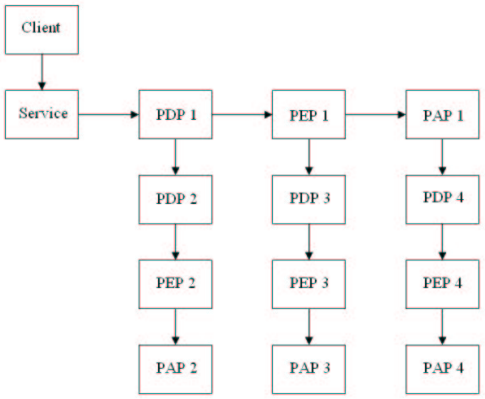


Figure 6: Recursive authentication schema

4.1 Policy Administration Point (PAP)

The PAP module is a policy repository that includes an administrative interface for inserting, updating, and deleting policies. This interface can be implemented both locally and via web by means of traditional components protocols or, also in this case, as a Web Service. Before inserting or updating a policy, the PAP module verifies if it is well formed and, if the policy does not have an external operator, the PAP module automatically adds the operator `All`. In this way, the semantics of the policy does not change and we avoid problems that may occur in the enforcement phase.

The main purpose of this module is to retrieve the policies applicable to a given access request. The PAP module then performs a search in the repository based on the received parameters (e.g., the service name and/or method name). The repository has been implemented as a relational database and the search is performed by means of a SQL query on the database. Note that, according to the WS-Policy specification, a policy may include references to external policies. In this case, the PAP module returns a policy where the references have been replaced by the corresponding policies.

4.2 Policy Evaluation Point (PEP)

The PEP module realizes the enforcement of the policies returned by the PAP module. The access request is granted if at least one policy is satisfied; the access is denied otherwise. In this latter case, the PEP module returns to the client an exception string indicating the error (see Section 5). More precisely, the PEP module works as follows. First, it creates a *SAXParser* for analyzing the policies and then iteratively enforces the policies. The enforcement phase can generate two possible events: *i)* a policy is satisfied, the enforcement process terminates, and the access request is granted; *ii)* a policy is not satisfied, an exception is raised and stored in a vector. After that all policies have been evaluated and none of them is satisfied, the PEP module selects the exception with the highest priority and sends it to the *service*.

```
<definitions xmlns:http="http://schemas.xmlsoap.org/wsdl/http/"
 xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/"
 xmlns:s="http://www.w3.org/2001/XMLSchema"
 xmlns:s0="http://tempuri.org/"
 xmlns:soapenc="http://schemas.xmlsoap.org/soap/encoding/"
 xmlns:tm="http://microsoft.com/wsdl/mime/textMatching/"
 xmlns:mime="http://schemas.xmlsoap.org/wsdl/mime/"
 targetNamespace="http://tempuri.org/" xmlns="http://schemas.xmlsoap.org/wsdl/">
<types>
 <s:schema elementFormDefault="qualified" targetNamespace="http://tempuri.org/">
   <s:element name="TemptureF">
     <s:complexType>
       <s:sequence>
         <s:element minOccurs="1" maxOccurs="1" name="intF" type="s:int"/>
       </s:sequence>
     </s:complexType>
   </s:element>
   <s:element name="TemptureFResponse">
     <s:complexType>
       <s:sequence>
         <s:element minOccurs="0" maxOccurs="1" name="TemptureFResult" type="s:string"/>
       </s:sequence>
     </s:complexType>
   </s:element>
 </s:schema>
</types>
```

Figure 7: part of WSDL of TempureF method

4.3 Policy Decision Point (PDP)

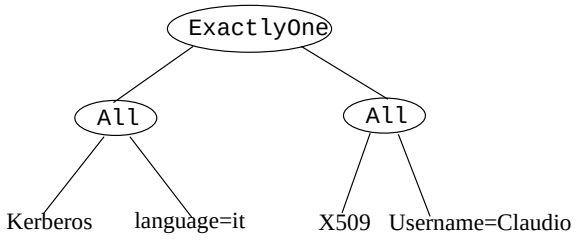
The PDP module is the interface between the service and the Enforcer. The body of a SOAP message is used for communicating the target service name and/or method name and the header can be used for specifying additional information (e.g., information for authenticating the service to the PDP). The PDP module instantiates one or more PEP module and each of them is based on different policy repositories (PAPs). The interaction between each pair PEP-PAP can return a different decision; the PDP module defines a policy for deciding how compute a final decision based on the responses of each PEP. Different decision criteria could be adopted, each applicable in specific situations. A natural and straightforward policy is the one stating that the PEP module with the highest priority wins, or a majority policy can be adopted.

5 Policy enforcement

The discussion in the previous section already makes clear how policy enforcement works. Given an access request, all applicable policies are evaluated and the request is granted if at least one policy is satisfied. More precisely, to verify if an access request satisfies a policy, all the assertions composing it have to be checked. To this purpose, we represent the set of all assertions composing a policy as an *evaluation tree*, where the leaves of the tree are the assertions and the internal nodes

```
<wsp:Policy xmlns:wsse="..." xmlns:wssx="...">
  <wsp:ExactlyOne>
    <wsp:All wsp:Usage="wsp:Required" wsp:Preference="100">
      <wsse:SecurityToken>
        <wsse:TokenType>wsse:Kerberosv5TGT</wsse:TokenType>
      </wsse:SecurityToken>
      <wssx:Language Usage="Required" Language="it"/>
    </wsp:All>
    <wsp:All wsp:Preference="1" wsp:Usage="wsp:Required">
      <wsse:SecurityToken>
        <wsse:TokenType>wsse:UsernameToken</wsse:TokenType>
        <wsse:Username>Claudio</wsse:Username>
      </wsse:SecurityToken>
      <wsse:SecurityToken>
        <wsse:TokenType>wsse:X509</wsse:TokenType>
      </wsse:SecurityToken>
    </wsp:All>
  </wsp:ExactlyOne>
</wsp:Policy>
```

(a)



(b)

Figure 8: An example of access policy (a) and the corresponding evaluation tree (b)

are the boolean operators (`All`, `OneOrMore`, `ExactlyOne` corresponding to and, or, and xor) used for combining the assertions. The Enforcer (the PEP module) simplifies the evaluation tree evaluating its leaves to true or false. Then, the evaluation tree is simplifies using the usual boolean laws for true and false. To fix ideas and make the discussion clear, consider a simple Web service whose purpose is to convert between degree Celsius and degree Fahrenheit scales. Figure 7 illustrates a portion of the WSDL for the method that takes a temperature expressed in degree Celsius as input (`intF`) and returns the temperature expressed in degree Fahrenheit (`TemptureFResponse`).

Consider also the access policy shown in Figure 8(a). The external operator `ExactlyOne` contains two operators `All` that in turn contain the following assertions: *i)* a Kerberos certificate and a specification of language; *ii)* an X509 certificate and an `Username Token` with `Username Claudio`. Figure 8(b) illustrates the corresponding evaluation tree. The header of the access request in Figure 9(a) contains an X509 certificate and a `Username Token` with `Username Claudio`. The evaluation tree is then simplified to true and the access is granted.

By contrast, Figure 9(b) illustrates an access request that does not satisfy the given policy. In this case the SOAP header includes both an X509 certificate and a Kerberos certificate. The

```
<?xml version="1.0" encoding="utf-8"?>
<S:Envelope xmlns:S="http://www.w3.org/2001/12/soap-envelope" xmlns:ds="http://www.w3.org/2000/09/xmldsig#">
  <S:Header>
    <wsse:Security xmlns:wsse= "http://schemas.xmlsoap.org/ws/2002/04/secext">
      <wsse:BinarySecurityToken ValueType="wsse:X509v3" EncodingType="wsse:Base64Binary">
        MIIEZzCCA9CgAwIBAgIQEmtJZc0...
      </wsse:BinarySecurityToken>
      <wsse:UsernameToken >
        <wsse:Username>Claudio</wsse:Username>
      </wsse:UsernameToken>
    </wsse:Security>
  </S:Header>
  <S:Body>
    <!-- access request go here -->
  </S:Body>
</S:Envelope>
```

(a)

```
<?xml version="1.0" encoding="utf-8"?>
<S:Envelope xmlns:S="http://www.w3.org/2001/12/soap-envelope" xmlns:ds="http://www.w3.org/2000/09/xmldsig#">
  <S:Header>
    <wsse:Security xmlns:wsse= "http://schemas.xmlsoap.org/ws/2002/04/secext">
      <wsse:BinarySecurityToken ValueType="wsse:X509v3" EncodingType="wsse:Base64Binary">
        MIIEZzCCA9CgAwIBAgIQEmtJZc0...
      </wsse:BinarySecurityToken>
      <wsse:BinarySecurityToken ValueType="wsse:Kerberosv5TGT" EncodingType="wsse:Base64Binary">
        JYTVjkvkjaOIJK76i7tuaeHJ...
      </wsse:BinarySecurityToken>
    </wsse:Security>
  </S:Header>
  <S:Body>
    <!-- access request go here -->
  </S:Body>
</S:Envelope>
```

(b)

Figure 9: Two examples of SOAP access requests

evaluation tree is simplified to false and the access is denied. The Enforcer has to report an error indicating the reason for which the access request has been rejected. Such a error message can specify, for example, that there is no a particular certificate or there is no a pair username-password. The error messages are in XML format and should be conform to the XSD schema shown in Figure 10. In our example, the Enforcer returns the error message depicted in Figure 11 stating that the access is rejected because the request does not specify a language associated with the Kerberos certificate or a Username Token associated with an X509 certificate.

```
<?xml version="1.0" ?>
<xs:schema xmlns:targetNamespace="http://seth/errors" xmlns:xs="http://www.w3.org/2001/XMLSchema">
  <xs:element name="All" type="Compositor"/>
  <xs:element name="ExactlyOne" type="Compositor"/>
  <xs:element name="OneOrMore" type="Compositor"/>
  <xs:element name="Error" type="xs:string"/>
  <xs:complexType name="Compositor">
    <xs:group ref="CompositorContent" maxOccurs="unbounded"/>
  </xs:complexType>
  <xs:group name="CompositorContent">
    <xs:choice>
      <xs:element ref="All"/>
      <xs:element ref="ExactlyOne"/>
      <xs:element ref="OneOrMore"/>
      <xs:element ref="Error" maxOccurs="unbounded"/>
    </xs:choice>
  </xs:group>
  <xs:group name="root">
    <xs:choice>
      <xs:element ref="All"/>
      <xs:element ref="ExactlyOne"/>
      <xs:element ref="OneOrMore"/>
    </xs:choice>
  </xs:group>
  <xs:element name="ErrorsReport" type="ErrorsReportExpression"/>
  <xs:complexType name="ErrorsReportExpression">
    <xs:group ref="root" minOccurs="0" maxOccurs="unbounded"/>
    <xs:anyAttribute namespace="##any" processContents="lax"/>
  </xs:complexType>
</xs:schema>
```

Figure 10: XSD Schema of message error

6 Conclusions

WS-Policy is a language for controlling access to Web services. Althought WS-Policy was designed for that purpose, its development has not follow the traditional and well-know multi-phase development process based on the definition of a security model that provides a formal representation of the access control security policy and its working. The result is a language that is subject to different interpretations and that presents some ambiguities. In this paper, we presented these shortcomings and proposed a solution. We then described the architecture we have implemented for controlling access to Web services.

We conclude by mentioning an interesting future direction for extending our work. Since the modules of the Enforcer have been implemented as Web services, it is possible to integrate our architecture with an UDDI register where the available PEP, PAP, and PDP can be registered. A trusted PAP module can, for example, use the UDDI register for searching a particular PEP satisfying given requirements.

```
<ErrorsReport>
  <ExactlyOne>
    <All>
      <Error>Needed Language associated with a Kerberos Certificate</Error>
    </All>
    <All>
      <Error>Needed UsernameToken associated with an X509 Certificate</Error>
    </All>
  </ExactlyOne>
</ErrorsReport>
```

Figure 11: Error string

Acknowledgments

We would like to thank Ernesto Damiani and Pierangela Samarati for helpful comments and suggestions. This work was supported in part by the European Union within the PRIME Project in the FP6/IST Programme under contract IST-2002-507591 and by the Italian MIUR within the KIWI and MAPS projects.

References

[1] R. Agrawal, J. Kiernan, R. Srikant, and Y. Xu. An XPath-based preference language for P3P. In *Proc. of the World Wide Web Conference*, Budapest, Hungary, May 2003.

[2] B. Atkinson and G. Della-Libera et al. Web services security (WS-Security). `http://msdn.microsoft.com/library/en-us/dnglobspec/html/ws-security.asp%`, April 2002.

[3] P. Bonatti and P. Samarati. A unified framework for regulating access and information release on the web. *Journal of Computer Security*, 10(3):241–272, 2002.

[4] D. Box et al. Web services policy assertions language (WS-PolicyAssertions) version 1.1. `http://msdn.microsoft.com/library/en-us/dnglobspec/html/ws-policyassert%ions.asp`, May 2003.

[5] D. Box et al. Web Services Policy Attachment (WS-PolicyAttachment) version 1.1. `http://msdn.microsoft.com/library/en-us/dnglobspec/html/ws-policyattach%ment.asp`, May 2003.

[6] D. Box et al. Web Services Policy Framework (WS-Policy) version 1.1. `http://msdn.microsoft.com/library/en-us/dnglobspec/html/ws-policy.asp`, May 2003.

[7] N. Brown and C. Kindel. Distributed Component Object Model Protocol, November 1996.

[8] R. Chinnici, M. Gudgin, J. Moreau, J. Schlimmer, and S. Weerawarana. Web services description language (wsdl) version 2.0 part 1: Core language. `http://www.w3.org/TR/wsdl12`, March 2004.

[9] E. Damiani, S. De Capitani di Vimercati, S. Paraboschi, and P. Samarati. Securing SOAP E-services. *International Journal of Information Security (IJIS)*, 1(2):100–115, February 2002.

[10] S. Feldman. The Changing Face of E-Commerce. *IEEE Internet Computing*, 4(3):82–84, May/June 2000.

[11] J.A. Hine, W. Yao, J. Bacon, and K. Moody. An architecture for distributed OASIS services. In *Proc. of the IFIP/ACM International Conference on Distributed Systems Platforms nd Open Distributed Processing*, Hudson River Valley, New York, USA, April 2000.

[12] Java Remote Method Invocation (RMI). `http://java.sun.com/j2se/1.3/docs/guide/rmi/index.html`.

[13] H. Koshutanski and F. Massacci. An access control framework for business processes for web services. In *Proc. of the 2003 ACM workshop on XML security*, Fairfax, Virginia, November 2003.

[14] C.E. Landwehr. Formal models for computer security. *ACM Computing Surveys*, 13(3):247–278, September 1981.

[15] OASIS eXtensible Access Control Markup Language (XACML) version 1.1. `http://www.oasis-open.org/committees/xacml/repository/cs-xacml-specific%ation-1.1.pdf`.

[16] P. Samarati and S. De Capitani di Vimercati. Access control: Policies, models, and mechanisms. In R. Focardi and R. Gorrieri, editors, *Foundations of Security Analysis and Design*, LNCS 2171. Springer-Verlag, 2001.

[17] Security assertion markup language (SAML) v1.1. http://www.oasis-open.org/committees/download.php/3400/oasis-sstc-saml-1.1-pdf-xsd.zip.

[18] Simple object access protocol (soap). `http://www.w3.org/TR/SOAP`, May 2000.

[19] The CORBA Security Service Specification (Revision 1.2). `ftp://ftp.omg.org/pub/docs/ptc/98-01-02.pdf`, January 1998.

[20] UDDI version 3.0.1. `http://uddi.org/pubs/uddi_v3.htm`.

[21] Web services security policy (WS-SecurityPolicy), December 2002. `http://www-106.ibm.com/developerworks/library/ws-secpol/`.