XML Security: Control information access with XACML

The objectives, architecture, and basic concepts of eXtensible Access Control Markup Language

Level: Intermediate

Manish Verma (mverma@secf.com)
Center Head & VP Delivery, Second Foundation
18 Oct 2004

Providing the right people with the right access to information is as important as (if not more important than) having the information in the first place. eXtensible Access Control Markup Language -- or XACML -- provides a mechanism to create policies and rules for controlling access to information. In this article, author Manish Verma continues his series on XML security issues by showing you how to incorporate XACML into your own applications.

In my last article in the series on XML security, I covered the Security Assertion Markup Language (SAML). This article on eXtensible Access Control Markup Language (XACML) continues from where that article left off. While SAML provides a mechanism for making authentication and authorization assertions and a mechanism for conveying them, XACML provides the language that defines the rules needed to make the necessary authorization decisions.

For example, consider the scenario where a subject makes a request to access a target resource. The Policy Enforcement Point (PEP) checks with the Policy Decision Point (PDP) before making a decision and releasing the target resource to the subject. Generation of the request to access the target resource and the subsequent response granting or denying access falls under the domain of SAML. XACML addresses the exchange of policy decisions between the PEP and the PDP.

Access control and XACML

XACML is an initiative to develop a standard for access control and authorization systems. Most of the current systems implement access control and authorization in a proprietary manner.

A typical access control and authorization scenario includes three main entities -- a **subject**, a **resource**, and an **action** -- and their attributes. A subject makes a request for permission to perform an action on a resource. For example, in the access request, "Allow the finance manager to create files in the invoice folder on the finance server," the subject is the "finance manager," the target resource is the "invoice folder on the finance server," and the action is "create files."

In proprietary access control systems, information about these entities and their attributes is kept in repositories. The repositories are called Access Control Lists (ACLs). Different proprietary systems have different mechanisms for implementing ACLs, making it difficult to exchange and share information between them.

XACML objectives

XACML aims to achieve the following:

- Create a portable and standard way of describing access control entities and their attributes.
- Provide a mechanism that offers much finer granular access control than simply denying or granting access -- that is, a mechanism that can enforce some before and after actions along with "permit" or "deny" permission.

XACML and SAML: How different and how similar?

XACML architecture is tightly intertwined with SAML architecture. They both share a lot of concepts and a domain -- the domain of authentication, authorization, and access control. However, the problems they address in the same domain are different. While SAML addresses authentication and provides a mechanism for transferring authentication and authorization decisions between cooperating entities, XACML focuses on the mechanism for arriving at those authorization decisions.

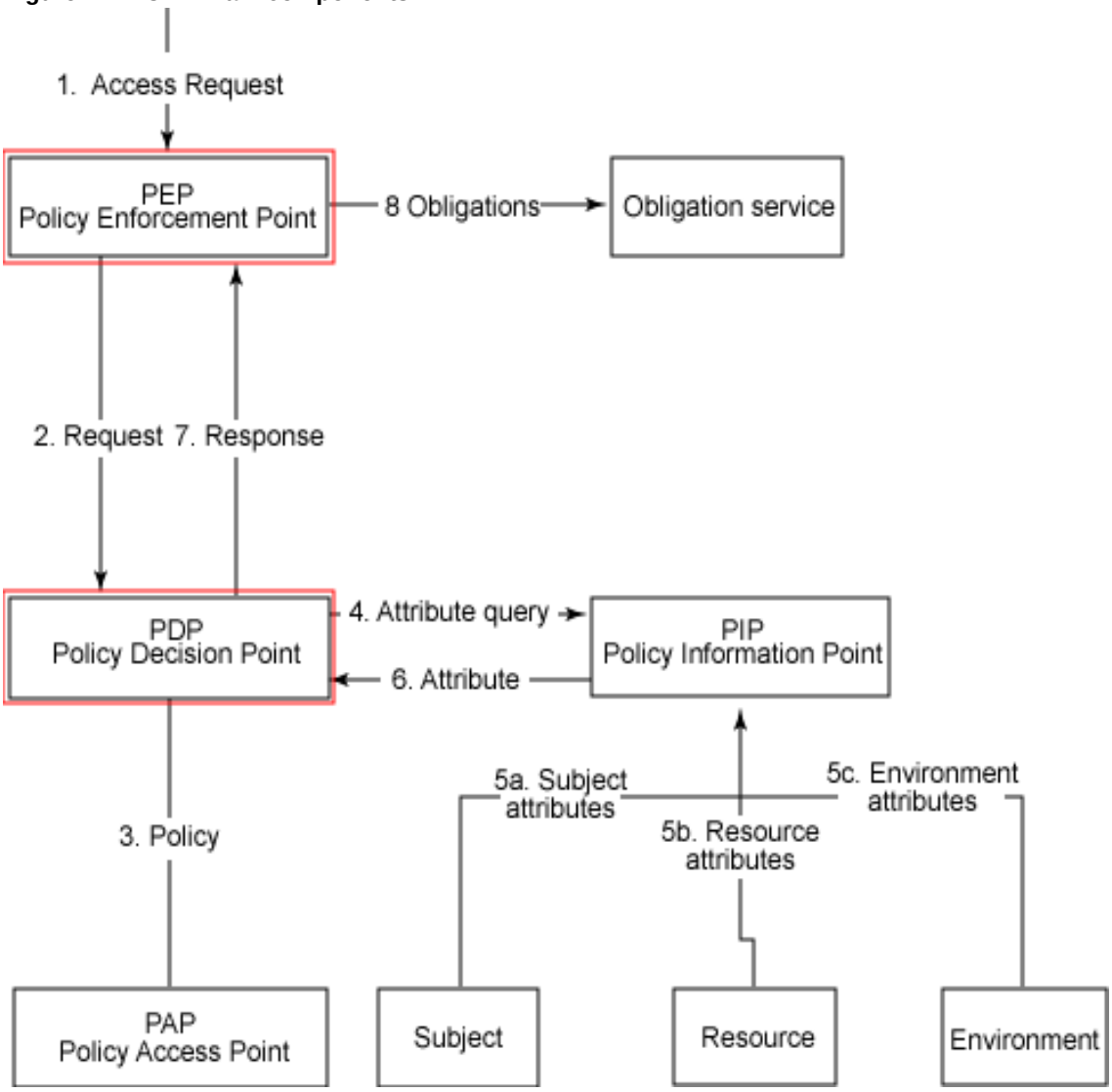
The SAML standard provides interfaces that allow third parties to send their requests for authentication and authorization. How these authorization requests are processed internally is addressed by XACML standards. XACML not only processes the authorization requests, but it defines the mechanism for creating the complete infrastructure of rules, policies, and policy sets to arrive at the authorization decisions. I will go over these concepts in more detail in XACML: Nuts and bolts.

Given that both SAML and XACML share the same domain, it is highly likely and desirable that these two specifications will eventually be merged into one.

XACML architecture

XACML is composed of many components described in Figure 1. Some of these components are also shared with SAML. The shared components are marked with a red border.

Figure 1. XACML main components



A request for authorization lands at the **Policy Enforcement Point (PEP)**. The PEP creates an XACML request and sends it to the **Policy Decision Point (PDP)**, which evaluates the request and sends back a response. The response can be either access permitted or denied, with the appropriate obligations. I will explain obligations later in this article.

The PDP arrives at a decision after evaluating the relevant policies and the rules within them. A number of policies may be available: The PDP does not evaluate all of them; only the relevant ones are chosen for evaluation, based on the policy target. The policy target contains information about the subject, the action, and other environmental properties. The complete process of how the policies are chosen for evaluation by a PDP is explained later in Policy target creation.

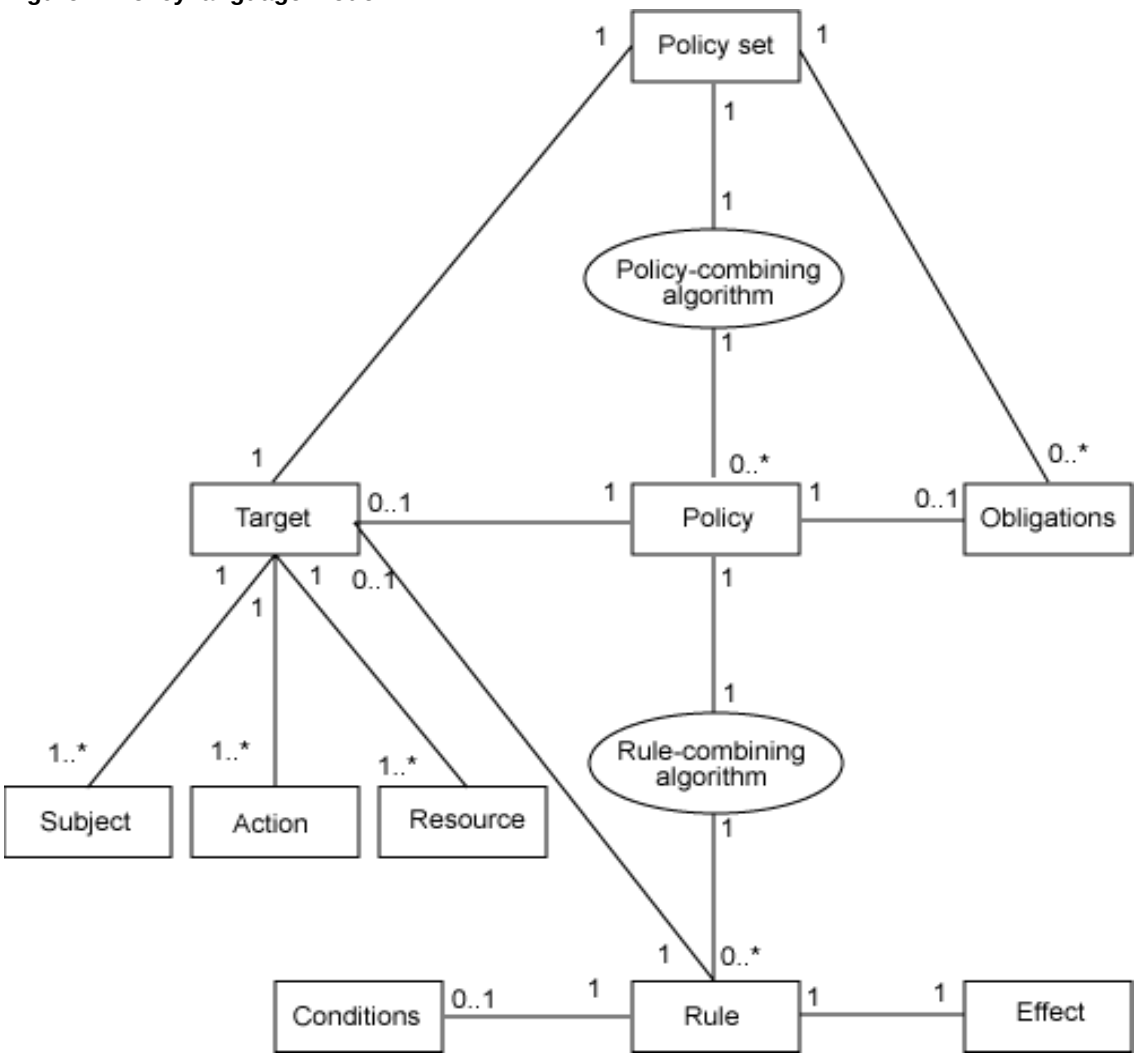
To get to the policies, the PDP uses the **Policy Access Point (PAP)**, which writes policies and policy sets, and makes them available to the PDP. The PDP may also invoke the **Policy Information Point (PIP)** service to retrieve the attribute values related to the subject, the resource, or the environment. The authorization decision arrived at by the PDP is sent to the PEP. The PEP fulfills the obligations and, based on the authorization decision sent by PDP, either permits or denies access.

XACML: Nuts and bolts

To illustrate the various XACML components, I'll show you how to take a specific access request and create all the XACML components required to process it. The access request is as follows: The subject -- `mverma@secf.com`, which belongs to the `owner` group (an attribute of the subject) -- is trying to perform an `open` action on the resource `file:///D:/Documents/Administrator/Desktop/Project Plan.html`. After you've created all of the required XACML components, you should get an authorization decision for this request.

Keep in mind that XACML has three top-level components: a policy, a PEP, and a PDP. The process of creating the XACML infrastructure for the defined request revolves around these three components only. The diagram in Figure 2 illustrates how these components are linked to each other:

Figure 2. Policy language model



The first thing you need to do is create a policy for handling the request.

XACML policy

A policy consists of: a set of rules, an identifier for rule-combining algorithms, a set of obligations, and a target. It is by far the most important aspect of XACML. Most of the action in XACML takes place in a policy.

I will show you how to create a policy that can address the request -- the scope of the policy needs to be broader than the scope of the request. The policy you'll create will read like this: Any user with an e-mail name in the `secf.com` namespace is allowed to perform any action on the resource `file:///D:/Documents/Administrator/Desktop/Project Plan.html`. Note that the policy is more generic than the request.

A policy is a combination of several subcomponents: target, rules, rule-combining algorithm, and obligations. Understanding these subcomponents is a must for understanding a policy. Let me explain the significance of each of these subcomponents:

- **Target:** Each policy has only one target. The target helps in determining whether the policy is relevant for the request. The policy's relevance to the request determines if the policy is to be evaluated for the request. This is achieved by defining attributes of three categories in the target -- subject, resource, and action -- along with their values. It is not mandatory to have attributes for all the three categories in a target. The values of these attributes are compared with the values of the same attributes in the request; if they match (after applying some function on them, which you will see in detail later in the article), then the policy is considered relevant to the request and is evaluated.
- **Rules** -- Multiple rules can be associated to a policy. Each rule is composed of a condition, an effect, and a target.
  - **Conditions** are statements about attributes that upon evaluation return either `True`, `False`, or `Indeterminate`.
  - **Effect** is the intended consequence of the satisfied rule. It can either take the value `Permit` or `Deny`.
  - **Target**, as in the case of a policy, helps in determining whether or not a rule is relevant for a request. The mechanism for achieving this is also similar to how it is done in the case of a target for a policy.

  The final outcome of the rule depends on the condition evaluation. If the condition returns `Indeterminate`, the rule also returns `Indeterminate`. If the condition returns `False`, the rule returns `NotApplicable`. If the condition returns `True`, the value of the `Effect` element is returned, which is either `Permit` or `Deny`.
- **Rule-combining algorithm:** As I explained in the point above, a policy can have multiple rules. It is possible for different rules to generate conflicting results. Rule-combining algorithms are responsible for resolving such conflicts to arrive at one outcome per policy per request. Only one rule-combining algorithm is applicable per policy. See Rule-combining algorithms for more details.
- **Obligations:** Remember, one of the objectives of XACML is to provide much finer-level access control than mere permit and deny decisions. Well, obligations are the mechanism for achieving this. Obligations are the actions that must be performed by the PEP in conjunction with the enforcement of an authorization decision. After a policy has been evaluated, specific obligations are sent to the PEP along with the authorization decision. In addition to enforcing the authorization decision, the PEP is responsible for executing the operations specified as obligations.

Rule-combining algorithms

Rule-combining algorithms combine the effects of all the rules in a policy to arrive at a final authorization decision. XACML defines the following rule-combining algorithms (you can define your own algorithms as well):

- **Deny-overrides:** If any rule evaluates to `Deny`, then the final authorization decision is also `Deny`.
- **Ordered-deny-overrides:** Same as deny-overrides, except the order in which relevant rules are evaluated is the same as the order in which they are added in the policy.
- **Permit-overrides:** If any rule evaluates to `Permit`, then the final authorization decision is also `Permit`.
- **Ordered-permit-overrides:** Same as permit-overrides, except the order in which relevant rules are evaluated is the same as the order in which they are added in the policy.
- **First-applicable:** The result of the first relevant rule encountered is the final authorization decision as well.

Policy creation

This section takes you deep into the policy creation code. You'll start by creating the `policy` object (see Listing 1) and the sub-components as required. Sun's XACML implementation is used for all the coding examples in this article (see Resources for the implementation.)

Listing 1. Create a policy

```
// Create policy identifier and policy description
URI policyId = new URI("ProjectPlanAccessPolicy");
String description =
"This AccessPolicy applies to any account at secf.com "
+ "accessing file:///D:/Documents/Administrator/Desktop/Project Plan.html.";

// Rule combining algorithm for the Policy

URI combiningAlgId = new URI(OrderedPermitOverridesRuleAlg.algId);

CombiningAlgFactory factory = CombiningAlgFactory.getInstance();

RuleCombiningAlgorithm combiningAlg =
  (RuleCombiningAlgorithm) (factory.createAlgorithm(combiningAlgId));


// Create the target for the policy
Target policyTarget = createPolicyTarget();

// Create the rules for the policy

List ruleList = createRules();

// Create the policy
Policy policy =
  new Policy(
    policyId,
    combiningAlg,
    description,
    policyTarget,
    ruleList);

// Display the policy on the std out
policy.encode(System.out, new Indenter());
```

Listing 1 demonstrates how the `policy` is created. But does it really? The meat of the `policy` creation is in the creation of its subcomponents. This listing merely demonstrates how those subcomponents are put together to make the `policy` object. But, it does demonstrate the overall structure of the `policy`. Here's how the policy is put together:

1. Choose the rule-combining algorithm for the policy. I have used the `ordered-permit-override` algorithm, so the rules are evaluated in the order in which they are specified in the policy.
2. Create the target for the policy. Target creation is a little involved -- it requires its own code listing and explanation to make sense. Listing 2 demonstrates target creation for the policy.
3. Create the rules to be associated with the policy. As with the target, rule creation is an involved process and requires its own listing. Listing 3 is devoted to explanation of rule creation.
4. After all the necessary policy subcomponents have been created, use them to create the `policy` object.
5. Store the `policy` object in a file for passing to the PDP later.

The next couple of sections demonstrate creation of the subcomponents for the policy. I start with the target creation and finish the policy creation discussion with the creation of the rule and the associated condition.

Policy target creation

Target creation is an important aspect of creating a policy. The target is the mechanism by which relevant policies are chosen for evaluation of a request. To create a policy target, you need to define attributes and their values that belong to one of three categories: subject, resource, or action. When the PDP evaluates the request, it looks for the policy whose target has an attribute with the same value as in the request.

Next, I explain how to achieve this programmatically. XACML provides a mechanism called `AttributeDesignator` that compares the attribute values in the request and in the policy target.

With `AttributeDesignator`, you specify an attribute by defining its name and type. In addition, you also specify the value of the attribute -- the attribute value that you specify in the target is compared with the attribute value in the request. To do this comparison, you choose one of the pre-configured functions. The function, along with the `AttributeDesignator`, is used to create a `TargetMatch` object. You can create multiple `TargetMatch` objects; one for each attribute. All of the `TargetMatch` objects that belong to a category are put in a list and passed to the `Target` object. This `Target` object is what you use to create the `Policy` object.

The policy should read like this: "Any user with an e-mail name in the `secf.com` namespace is allowed to perform any action on the

resource `file:///D:/Documents/Administrator/Desktop/Project Plan.html`." For this policy, you can create a target that has two attributes -- one for the subject and another for the resource. You don't need to create an attribute for the action, as the policy you are creating does not seek to restrict any action.

Listing 2 demonstrates the creation of the policy target. You want to compare the domain value of the subject's e-mail in the request with the domain specified in the target, so for the subject you can specify the attribute `urn:oasis:names:tc:xacml:1.0:data-type:rfc822Name` (the e-mail ID) in the target. To make it part of the target, you define the attribute type (`urn:oasis:names:tc:xacml:1.0:data-type:rfc822Name`), the name of the attribute (`urn:oasis:names:tc:xacml:1.0:subject:subject-id`), and the comparator function, which in this case is `urn:oasis:names:tc:xacml:1.0:function:rfc822Name-match`. This function compares the value of the attribute specified in the target with the value of the same attribute in the request. A similar process is followed for the attribute of the resource type `http://www.w3.org/2001/XMLSchema#anyURI`.

Listing 2. Create a policy target

```
public static Target createPolicyTarget() throws URISyntaxException {
  List subjects = new ArrayList();
  List resources = new ArrayList();

  // Attributes of Subject type
  // Multiple subject attributes can be specified. In this
  // case only one is being defined.

  List subject = new ArrayList();

  URI subjectDesignatorType =
    new URI("urn:oasis:names:tc:xacml:1.0:data-type:rfc822Name");
  URI subjectDesignatorId =
    new URI("urn:oasis:names:tc:xacml:1.0:subject:subject-id");
  // Match function for the subject-id attribute
  String subjectMatchId =
    "urn:oasis:names:tc:xacml:1.0:function:rfc822Name-match";
  AttributeDesignator subjectDesignator =
    new AttributeDesignator(
                AttributeDesignator.SUBJECT_TARGET,
                subjectDesignatorType,
                subjectDesignatorId,
                false);

  StringAttribute subjectValue = new
    StringAttribute("secf.com");


  // get the factory that handles Target functions
  FunctionFactory factory =
    FunctionFactory.getTargetInstance();

  // get an instance of the right function for matching
  // subject attributes
  Function subjectFunction =
    factory.createFunction(subjectMatchId);

  TargetMatch subjectMatch = new TargetMatch(
                        TargetMatch.SUBJECT,
                subjectFunction,
                subjectDesignator,
                subjectValue);

  subject.add(subjectMatch);


  // Attributes of resource type
  // Multiple resource attributes can be specified. In this
  // case only one is being defined.

  List resource = new ArrayList();

  URI resourceDesignatorType =
    new URI("http://www.w3.org/2001/XMLSchema#anyURI");
  URI resourceDesignatorId =
    new URI("urn:oasis:names:tc:xacml:1.0:resource:resource-id");

  // Match function for the resource-id attribute
  String resourceMatchId =
    "urn:oasis:names:tc:xacml:1.0:function:anyURI-equal";

  AttributeDesignator resourceDesignator =
    new AttributeDesignator(
                AttributeDesignator.RESOURCE_TARGET,
                resourceDesignatorType,
                resourceDesignatorId,
                false);

  AnyURIAttribute resourceValue =
    new AnyURIAttribute(
      new URI("file:///D:/Documents/Administrator/Desktop/Project Plan.html"));
```

```
  // Get an instance of the right function for matching
  // resource attribute
  Function resourceFunction =
    factory.createFunction(resourceMatchId);

  TargetMatch resourceMatch = new TargetMatch(
                              TargetMatch.RESOURCE,
                         resourceFunction,
                         resourceDesignator,
                         resourceValue);

  resource.add(resourceMatch);

  // Put the subject and resource sections into their lists
  subjects.add(subject);
  resources.add(resource);

  // Create and return the new target. No action type
  // attributes have been specified in the target
  return new Target(subjects, resources, null);
}
```

Rules creation

Rules are probably the most important subcomponent of a policy. Rules, as explained above, are essentially conditions that evaluate to `Permit`, `Deny`, `Indeterminate`, or `NotApplicable`.

To continue the process of creating the XACML components required to process the authorization request (to review, see the first paragraph of XACML: Nuts and bolts), you now create an appropriate rule. This essentially boils down to creating an appropriate condition. Listing 3 demonstrates rule creation. If you look at the code you will see three things happening:

- Creation of the rule target
- Definition of the effect
- Creation of the condition

Two of these three -- creating the rule target and defining the effect of the rule -- are straightforward. The third -- creating the condition -- is little more involved, and I describe that in more detail later.

The rule that you create to satisfy the request should look like this: "If the subject, `mverma@secf.com`, which belongs to the group `owner` tries to open the resource `file:///D:/Documents/Administrator/Desktop/Project Plan.html`, then permit access." To create such a rule, create the target such that the rule is evaluated for the request that you are processing. Creating the rule target is similar to the way that the target was created for the policy. You then define the `effect` of the rule as `Permit`. To have the rule return the value of the `effect`, the associated condition must return `True`. The code in Listing 4 illustrates creation of the condition.

Listing 3. Create rules

```
public static List createRules() throws URISyntaxException {
  // Step 1: Define the identifier for the rule
  URI ruleId = new URI("ProjectPlanAccessRule");
  String ruleDescription = “Rule for accessing project plan";

  // Step 2: Define the effect of the rule
  int effect = Result.DECISION_PERMIT;

  // Step 3: Get the target for the rule
  Target target = createRuleTarget();

  // Step 4: Get the condition for the rule
  Apply condition = createRuleCondition();

  // Step 5: Create the rule
  Rule openRule = new Rule(ruleId, effect,ruleDescription, target, condition);

  // Create a list for the rules and add the rule to it

  List ruleList = new ArrayList();
  ruleList.add(openRule);

  return ruleList;

}
```

Listing 4 demonstrates the creation of the conditions for a rule. In much the same way that you created `AttributeDesignator` for the policy target, you create the `AttributeDesignator` object to compare the value of the attribute in the request to the one specified in the condition. The attribute of interest here is the `group` to which the subject belongs. The `group` attribute must have the value `owner` for the condition to return `True`.

The only additional thing that you need to do for the condition is to use a function to extract one value from the multiple that are returned by the `AttributeDesignator`. In this case, you use the function `urn:oasis:names:tc:xacml:1.0:function:string-one-and-only`. See AttributeDesignator return values for more on handling multiple values from `AttributeDesignator`.

AttributeDesignator return values
`AttributeDesignator` can return multiple values for an attribute (since a request can contain multiple matches). To handle this, XACML has a special datatype called a **bag**. Bags are unordered collections that allow duplicates. `AttributeDesignator` always return a bag, even if only one value is to be returned. Once a bag is returned, the values in the bag are compared to the expected values to make the authorization decision. In the case of the target, you do not use any function to extract one value from the bag, whereas in the case of the condition you do. The reason is that with the target, the PDP automatically applies a matching function to each element that's returned by `AttributeDesignator`.

As you can see in Listing 4, you create the `AttributeDesignator` object to define an attribute whose value you want to compare with what you get in the request. The compare algorithm is used to do the comparison. Then you define the function to pick one value from those values that might be returned by the `AttributeDesignator`. Finally, you create the `Apply` object, which is similar to the `TargetMatch` object that you created in the target. The objective of the `Apply` object is to apply the compare function on the value picked from the bag (returned by `AttributeDesignator`) and compare it to the value specified in the condition.

Listing 4. Create conditions for a rule

```
public static Apply createRuleCondition() throws URISyntaxException {
  List conditionArgs = new ArrayList();

  // Define the name and type of the attribute
  // to be used in the condition
  URI designatorType = new URI("http://www.w3.org/2001/XMLSchema#string");
  URI designatorId = new URI("group");

  // Pick the function that the condition uses
  FunctionFactory factory = FunctionFactory.getConditionInstance();
  Function conditionFunction = null;
  try {
      conditionFunction =
          factory.createFunction(
              "urn:oasis:names:tc:xacml:1.0:function:" + "string-equal");
  } catch (Exception e) {
      return null;
  }

  // Choose the function to pick one of the
  // multiple values returned by AttributetDesignator

  List applyArgs = new ArrayList();

  factory = FunctionFactory.getGeneralInstance();
  Function applyFunction = null;
  try {
      applyFunction =
          factory.createFunction(
              "urn:oasis:names:tc:xacml:1.0:function:"
                  + "string-one-and-only");
  } catch (Exception e) {
      return null;
  }

  // Create the AttributeDesignator
  AttributeDesignator designator =
      new AttributeDesignator(
           AttributeDesignator.SUBJECT_TARGET,
           designatorType,
           designatorId,
           false
           null);
  applyArgs.add(designator);

  // Create the Apply object and pass it the
  // function and the AttributeDesignator. The function
  // picks up one of the multiple values returned by the
  // AttributeDesignator
  Apply apply = new Apply(applyFunction, applyArgs, false);

  // Add the new apply element to the list of inputs
  // to the condition along with the AttributeValue
  conditionArgs.add(apply);

  StringAttribute value = new StringAttribute("owner");
  conditionArgs.add(value);

  // Finally, create and return the condition
  return new Apply(conditionFunction, conditionArgs, true);
}
```

With this, the policy creation task is complete. Let me quickly recap what's involved in the policy creation process. First, you create the necessary policy subcomponents: the policy target, the rule, and the rule-combining algorithm. You don't create the obligations for the policy, as they are optional. Creation of the rule essentially means creating the rule target and the condition; these are also demonstrated with their own code listings. All these policy subcomponents are used here to create the policy.

The creation of the PEP is the next step in creating the XACML components required to process the authorization request. What does the PEP do? It creates the authorization request for which you are creating all these XACML components!

Policy Enforcement Point (PEP)

The PEP creates the request based on the requester's attributes, the resource in question, the action, and other information. Here, I demonstrate the mechanism for creating the request that you started with. For your convenience, I'll repeat the request: The subject, `mverma@secf.com` which belongs to `owner` group (attribute of the subject), is trying to perform an `open` action on the resource `file:///D:/Documents/Administrator/Desktop/Project Plan.html`. To create such a request, you need two subject attributes, one resource attribute and one action attribute. The two subject attributes are `rfc822Name` (e-mail ID) and the group to which the subject belongs. The one resource attribute is the URI of the resource, and the one action attribute is the open action on the resource. Listing 5 demonstrates the creation of the PEP with all of these attributes.

Up to this point, you have seen creation of the policy and the generation of the request by the PEP. The only XACML component that now remains for you to create is the PDP.

Policy Decision Point (PDP)

The PDP evaluates the request as against the policy and returns the response.

Since the XACML specification does not define any specific mechanism for passing the policy and the request to the PDP for the evaluation of the request, you can choose any mechanism that's convenient. In this case, the policy and the request are passed as command-line arguments to the PDP.

Note that the PDP does not change based on the type of request that needs to be serviced. It is a generic component that takes any request along with a set of policies, and evaluates the request with respect to the applicable policies. Listing 6 demonstrates how to create the PDP, and how it is used to evaluate the request from the PEP. The PDP creation and request evaluation process is described in detail after Listing 6.

Listing 6. Create a PDP

```
public static void main(String[] args) throws Exception {
  if (args.length < 2) {
      System.out.println("Usage: <request> <AccessPolicy> [policies]");
      System.exit(1);
  }

  // Step 1: Get the request and policy file from the command line
  String requestFile = null;

  requestFile = args[0];
  String[] policyFiles = new String[args.length - 1];

  for (int i = 1; i < args.length; i++)
      policyFiles[i - 1] = args[i];

  // Step 2: Create a PolicyFinderModule and initialize it
  // Use the sample FilePolicyModule, which
  // is configured using the policies from the command line

  FilePolicyModule filePolicyModule = new FilePolicyModule();
  for (int i = 0; i < policyFiles.length; i++)
      filePolicyModule.addPolicy(policyFiles[i]);

  // Step 3: Set up the PolicyFinder that this PDP will use
  //
  PolicyFinder policyFinder = new PolicyFinder();
  Set policyModules = new HashSet();
  policyModules.add(filePolicyModule);
  policyFinder.setModules(policyModules);

  // Step 4: Create the PDP
  PDP pdp = new PDP(new PDPConfig(null, policyFinder, null));

 //  Get the request send by the PEP
```

```
  RequestCtx request =
      RequestCtx.getInstance(new FileInputStream(requestFile));

  // Step 5: Evaluate the request. Generate the response.
  ResponseCtx response = pdp.evaluate(request);

  // Display the output on std out
  response.encode(System.out, new Indenter());
}
```

The following step-by-step description explains PDP creation and request evaluation in detail:

1. Get the request and policy files from the command line argument.
2. Create a `PolicyFinderModule` and initialize it using the `FilePolicyModule` object. `FilePolicyModule` is configured using the policies from the command line. `PolicyFinderModule` is one of the three finder modules:
    - `PolicyFinderModule`
    - `AttributeFinderModule`
    - `ResourceFinderModule`

    These modules are mechanisms used by Sun's implementation of XACML to find attributes, policies, and resources.
3. Set up the policy finder that the `PDP` will use.
4. Create the `PDP` by initializing the `PDP` class. The `PDPConfig` object, which is configured using the `FilepolicyModule`, is supplied to the `PDP` constructor.
5. Evaluate the request from the `PEP` by calling the `evaluate` method on the `PDP`. This returns the authorization decision. If you created all the XACML components as described in this article, the PDP will give the `Permit` authorization decision.

The authorization decision is conveyed back to the `PEP`. You can choose the mechanism for sending the decision back to the `PEP`.

And that completes the explanation and demonstration of the creation of the three major XACML components -- a policy, a PEP, and a PDP -- required for processing the request.

Conclusion

Access control is one domain that is used in almost all applications, large and small. XACML is an attempt to bring standardization to this domain. While XACML is a verbose language, once you grasp the basic concepts and the flow of the language, it is easy to build access control mechanisms.

In this article, I have taken you through the process of creating the essential XACML components:

- The policy, including the rules and the policy target
- The PEP
- The PDP

With this knowledge, you can adopt XACML to handle access control in all your current and future applications.

Resources

- Download the source code used in this article.
- Get all the documentation and information related to XACML standards at the OASIS site.
- Download Sun's XACML implementation used in this article.
- Read the previous articles in Manish Verma's *developerWorks* series on XML security:
    - "XML Security: Basic plumbing technologies" (October 2003)
    - "XML Security: Core technologies -- XML encryption and XML signature" (October 2003)
    - "XML Security: The XML Key Management Specification" (January 2004)
    - "XML Security: Ensure portable trust with SAML" (March 2004)
- Find more XML resources on the *developerWorks* XML zone.

- Browse a wide range of XML-related titles at the *developerWorks* Developer Bookstore.
- Find out how you can become an IBM Certified Developer in XML and related technologies.

Download

Name	Size	Download method
x-xacml.zip	5 KB	FTP

Information about download methods

About the author

Manish Verma heads Second Foundation's software development center in India. Manish has 12 years experience in all aspects of the software development lifecycle, and has designed integration strategies for client organizations running disparate systems. Manish's integration expertise is founded on his understanding of a host of technologies, including various legacy systems, .NET, Java technology, and the latest middleware. Prior to Second Foundation, Manish worked as a software architect and technical lead at Quark Inc., Hewlett Packard, Endura Software, and The Williams Company. You can contact Manish at mverma@secf.com.

Code e-mail it!

Rate this article

This content was helpful to me:

Strongly disagree (1) Disagree (2) Neutral (3) Agree (4) Strongly agree (5)

Comments?

Submit feedback

XML Security: Control information access with XACML

The objectives, architecture, and basic concepts of eXtensible Access Control Markup Language

Return to article

Listing 5. Creating a PEP

```
public static void main(String[] args) throws Exception {
  // Create the new request.
  // Environment hashset is empty
  RequestCtx request =
      new RequestCtx(
          setupSubjects(),
          setupResource(),
          setupAction(),
          new HashSet());

  // Encode the request and print it to standard out
  request.encode(System.out, new Indenter());
}


public static Set setupSubjects() throws URISyntaxException {
  HashSet attributes = new HashSet();

  // Set up the ID and value for the requesting subject
  URI subjectId = new
  URI("urn:oasis:names:tc:xacml:1.0:subject:subject-id");
  RFC822NameAttribute value = new RFC822NameAttribute("mverma@secf.com");

  // Create the subject section with two attributes, the first with
  // the subject's identity...
  attributes.add(new Attribute(subjectId, null, null, value));
  // ...and the second with the subject's group membership

  URI groupId = new URI("group");
  StringAttribute stringAttribValue = new StringAttribute("owner");

  attributes.add(new Attribute(groupId,null,null,stringAttribValue));

  // Bundle the attributes in a subject with the default category
  HashSet subjects = new HashSet();
  subjects.add(new Subject(attributes));

  return subjects;
}

public static Set setupResource() throws URISyntaxException {
  HashSet resource = new HashSet();

  // The resource being requested
  AnyURIAttribute value =
      new AnyURIAttribute(new URI("file:///D:/Documents/Administrator/Desktop/Project Plan.html"));

  // Create the resource using a standard, required identifier for
  // the resource being requested
  resource.add(
      new Attribute(
          new URI(EvaluationCtx.RESOURCE_ID),
          null,
          null,
          value));

  return resource;
}
public static Set setupAction() throws URISyntaxException {
  HashSet action = new HashSet();

  // This is a standard URI that can optionally be used to specify
  // the action being requested
```

```
  URI actionId = new URI("urn:oasis:names:tc:xacml:1.0:action:action-id");

  // Create the action
  action.add(
      new Attribute(actionId, null, null, new StringAttribute("open")));

  return action;
}
```

Return to article