Sun's XACML Implementation

Programmer's Guide for Version 1.2

Last Updated: July 11, 2004

Introduction

Welcome to the programmer's guide for Sun's XACML Implementation. This guide will introduce you to the APIs, the XACML (eXtensible Access Control Markup Language) standard, and how to build support for XACML into your applications. For more information about XACML, you should visit the OASIS XACML technical committee's web site. For a complete listing of the APIs, go check out the JavaDocs.

In a nutshell, XACML is a general-purpose access control policy language. This means that it provides a syntax (defined in XML) for managing authorization decisions. Sun's XACML Implementation is a set of Java[TM] classes that understand the XACML language, as well as the rules about how to process requests and how to manage attributes and other related data. With a relatively small amount of code, you can write applications that use XACML to manage their own policy or that hook into existing infrastructure components like LDAP or SAML. This implementation requires a 1.4.0 or later JDK to build or run.

A Brief Introduction to XACML

XACML is an OASIS standard that describes both a policy language and an access control decision request/response language (both encoded in XML). The policy language is used to describe general access control requirements, and has standard extension points for defining new functions, data types, combining logic, etc. The request/response language lets you form a query to ask whether or not a given action should be allowed, and interpret the result. The response always includes an answer about whether the request should be allowed using one of four values: Permit, Deny, Indeterminate (an error occurred or some required value was missing, so a decision cannot be made) or Not Applicable (the request can't be answered by this service).

The typical setup is that someone wants to take some action on a resource. They will make a request to whatever actually protects that resource (like a filesystem or a web server), which is called a Policy Enforcement Point (PEP). The PEP will form a request based on the requester's attributes, the resource in question, the action, and other information pertaining to the request. The PEP will then send this request to a Policy Decision Point (PDP), which will look at the request, find some policy that applies to the request, and come up with an answer about whether access should be granted. That answer is returned to the PEP, which can then allow or deny access to the requester. Note that the PEP and PDP might both be contained within a single application, or might be distributed across several servers. In addition to providing request/response and policy languages, XACML also provides the other pieces of this relationship, namely finding a policy that applies to a given request and evaluating the request against that policy to come up with a yes or no answer.

There are many existing proprietary and application-specific languages for doing this kind of thing but XACML has several points in its favor:

- It's standard. By using a standard language, you're using something that has been reviewed by a large community of experts and users, you don't need to roll your own system each time, and you don't need to think about all the tricky issues involved in designing a new language. Plus, as XACML becomes more widely deployed, it will be easier to interoperate with other applications using the same standard language.
- It's generic. This means that rather than trying to provide access control for a particular environment or a specific kind of resource, it can be used in any environment. One policy can be written which can then be used by many different kinds of applications, and when one common language is used, policy management becomes much easier.
- It's distributed. This means that a policy can be written which in turn refers to other policies kept in arbitrary locations. The result is that rather than having to manage a single monolithic policy, different people or groups can manage separate sub-policies as appropriate, and XACML knows how to correctly combine the results from these different policies into one decision.
- It's powerful. While there are many ways the base language can be extended, many environments will not need to do so. The standard language already supports a wide variety of data types, functions, and rules about combining the results of different policies. In addition to this, there are already standards groups working on extensions and profiles that will hook XACML into other standards like SAML and LDAP, which will increase the number of ways that XACML can be used.

To give you a better idea of how all these aspects fit together, what follows is a brief discussion of XACML, which will demonstrate many of its standard features. Note that XACML is a rich language, so only some of its features are shown here. You should look at the specification for more information on all of the features.

Top-Level Constructs: Policy and PolicySet

At the root of all XACML policies is a Policy or a PolicySet. A PolicySet is a container that can hold other Policies or PolicySets, as well as references to policies found in remote locations. A Policy represents a single access control policy, expressed through a set of Rules. Each XACML policy document contains exactly one Policy or PolicySet root XML tag.

Because a Policy or PolicySet may contain multiple policies or Rules, each of which may evaluate to different access control decisions, XACML needs some way of reconciling the decisions each makes. This is done through a collection of Combining Algorithms. Each algorithm represents a different way of combining multiple decisions into a single decision. There are Policy Combining Algorithms (used by PolicySet) and Rule Combining Algorithms (used by Policy). An example of these is the Deny Overrides Algorithm, which says that no matter what, if any evaluation returns Deny, or no evaluation permits, then the final result is also Deny. These Combining Algorithms are used to build up increasingly complex policies, and they are what allow XACML policies to be distributed and decentralized. While there are several standard algorithms, you can build your own to suit your needs.

Targets and Rules

Part of what an XACML PDP does is find policies that apply to a given request. To do this, XACML provides another feature called a Target. A Target is basically a set of simplified conditions for the Subject, Resource and Action that must be met for a PolicySet, Policy or Rule to apply to a given request. These use boolean functions (explained more in the next section) to compare values found in a request with those included in the Target. If all the conditions of a Target are met, then its associated PolicySet, Policy, or Rule applies to the request. In addition to being a way to check applicability, Target information also provides a way to index policies, which is useful if you need to store many policies and then quickly sift through them to find which ones apply. For instance, a Policy might contain a Target that only applies to requests on a specific service. When a request to access that service arrives, the PDP will know where to look for policies that might apply to this request because the policies are indexed based on their Target constraints. Note that a Target may also specify that it applies to any request.

Once a Policy is found, and verified as applicable to a request, its Rules are evaluated. A policy can have any number of Rules which contain the core logic of an XACML policy. The heart of most Rules is a Condition, which is a boolean function. If the Condition evaluates to true, then the Rule's Effect (Permit or Deny) is returned. If the Condition evaluates to false, the the Condition doesn't apply (NotApplicable). Evaluation of a Condition can also result in an error (Indeterminate). Conditions can be quite complex, built from an arbitrary nesting of functions and attributes branching from the top-level boolean function.

Attributes, Attribute Values, and Functions

The currency that XACML deals in is attributes. Attributes are named values of known types that may include an issuer identifier or an issue date and time. Specifically, attributes are characteristics of the Subject, Resource, Action, or Environment in which the access request is made. For example, A user's name, their group membership, a file they want to access, and the time of day are all attribute values. When a request is sent from a PEP to a PDP, that request is formed almost exclusively of attributes, and they will be compared to attribute values in a policy to make the access decisions.

A Policy resolves attribute values from a request or from some other source through two mechanisms: the AttributeDesignator and the AttributeSelector. An AttributeDesignator lets the policy specify an attribute with a given name and type, and optionally an issuer as well. The PDP looks for that value in the request, and failing that, can look in any other location (like a an LDAP service). There are four kinds of designators, one for each of the types of attributes in a request: Subject, Resource, Action, and Environment. Subject attributes can be broken into different categories to support the notion of multiple subjects making a request (eg, the user, the user's workstation, the user's network, etc.), so SubjectAttributeDesignators can also specify a category to look in. AttributeSelectors allow a policy to look for attribute values through an XPath query. A data type and an XPath expression are provided, and these can be used to resolve some set of values either in the request document or elsewhere (just as Designators do).

Both the AttributeDesignator and the AttributeSelector can return multiple values (since there might be multiple matches in a request or elsewhere), so XACML provides a special attribute type called a Bag. Bags are unordered collections that allow duplicates, and are always what designators and selectors return, even if only one value was matched. In the case that no matches were made, an empty bag is returned, although a designator or selector may set a flag that causes an error instead in this case. Bags are never encoded in XML or included in a policy or Request/Response.

Once some Bag of attribute values is retrieved, the values need to be compared to the expected values to make access decisions. This is done though a powerful system of functions. Functions can work on any combination of attribute values, and can return any kind of attribute value supported in the system. Functions can also be nested, so you can have functions that operate on the output of other functions, and this hierarchy can be arbitrarily complex. Custom functions can be written to provide an ever richer language for expressing access conditions.

One thing to note when building these hierarchies of functions is that most of the standard functions are defined to work on specific types (like strings or integers) while designators and selectors always return Bags of values. To handle this mis-match, XACML defines a collection of standard functions of the form [type]-one-and-only, which accept a bag of values of the specified type and return the single value if there is exactly one item in the bag, or an error if there are zero or multiple values in the bag. These are among the most common functions used in a Condition. Functions of the form [type]-one-and-only functions are not needed in Targets, however, since the PDP automatically applies the matching function to each element of a bag.

Requests and Responses

In addition to defining a standard format for policy, XACML defines a standard way of expressing Requests and Responses. A Request contains Attributes in each of the four categories: Subject, Resource, Action, and Environment (which is optional). There is exactly one collection of Resource and Action Attributes in a Request, and at most one collection of Environment Attributes. There may be multiple collections of Subject Attributes, and each collection is identified by a category URI (or, if no category is specified, then the default category is used). In addition to Attributes, the Resource section allows the inclusion of the content of the requested resource, which can be considered in policy evaluation through XPath expressions. The only Attribute that is required in a Request is the Resource identifier.

A Response consists of one or more Results, each of which represents the result of an evaluation. Multiple Results can only be caused by evaluation of a hierarchical resource (explained later), so typically there will only be one Result in a Response. Each Result contains a Decision (Permit, Deny, NotApplicable, or Indeterminate), some Status information (for instance, why evaluation failed), and optionally one or more Obligations (things that the PEP is obligated to do before granting or denying access). The Request and the Response provide a standard format for interacting with a PDP.

Putting it Together: An Example

This is a very simple example that shows most of the features discussed to this point. It is intended to be a concrete, valid example of XACML, though not an illustration of all the powerful features from the specification. Provided (in order) is a Request, a Policy, and the expected Response from evaluating the Request against the Policy. For sample code to let you do this evaluation, or for more complete examples, look in the samples package.

The scenario shows a user trying to access a web page. The request includes the Subject's identity as well as a group they belong to, the resource being accessed, and the action being taken. The policy applies to anyone taking any action on the resource, and contains a rule that applies to a specific action. The Condition in the Rule requires certain group membership to access the resource.

The Request:

```
<Request>
  <Subject>
    <Attribute AttributeId="urn:oasis:names:tc:xacml:1.0:subject:subject-id"
               DataType="urn:oasis:names:tc:xacml:1.0:data-type:rfc822Name">
      <AttributeValue>seth@users.example.com</AttributeValue>
    </Attribute>
    <Attribute AttributeId="group"
               DataType="http://www.w3.org/2001/XMLSchema#string"
               Issuer="admin@users.example.com">
      <AttributeValue>developers</AttributeValue>
    </Attribute>
  </Subject>
  <Resource>
    <Attribute AttributeId="urn:oasis:names:tc:xacml:1.0:resource:resource-id"
               DataType="http://www.w3.org/2001/XMLSchema#anyURI">
      <AttributeValue>http://server.example.com/code/docs/developer-guide.html</AttributeValue>
    </Attribute>
  </Resource>
  <Action>
    <Attribute AttributeId="urn:oasis:names:tc:xacml:1.0:action:action-id"
               DataType="http://www.w3.org/2001/XMLSchema#string">
      <AttributeValue>read</AttributeValue>
    </Attribute>
  </Action>
</Request>
```

The Policy:

```
<Policy PolicyId="ExamplePolicy"
        RuleCombiningAlgId="urn:oasis:names:tc:xacml:1.0:rule-combining-algorithm:permit-overrides
  <Target>
    <Subjects>
      <AnySubject/>
    </Subjects>
    <Resources>
      <Resource>
        <ResourceMatch MatchId="urn:oasis:names:tc:xacml:1.0:function:anyURI-equal">
          <AttributeValue DataType="http://www.w3.org/2001/XMLSchema#anyURI">http://server.example
          <ResourceAttributeDesignator DataType="http://www.w3.org/2001/XMLSchema#anyURI"
                                       AttributeId="urn:oasis:names:tc:xacml:1.0:resource:resource
        </ResourceMatch>
      </Resource>
    </Resources>
    <Actions>
      <AnyAction/>
    </Actions>
  </Target>
  <Rule RuleId="ReadRule" Effect="Permit">
    <Target>
      <Subjects>
        <AnySubject/>
      </Subjects>
      <Resources>
        <AnyResource/>
      </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</AttributeValu
            <ActionAttributeDesignator DataType="http://www.w3.org/2001/XMLSchema#string"
                                       AttributeId="urn:oasis:names:tc:xacml:1.0:action:action-id"
          </ActionMatch>
        </Action>
      </Actions>
    </Target>
    <Condition FunctionId="urn:oasis:names:tc:xacml:1.0:function:string-equal">
      <Apply FunctionId="urn:oasis:names:tc:xacml:1.0:function:string-one-and-only">
        <SubjectAttributeDesignator DataType="http://www.w3.org/2001/XMLSchema#string"
                                    AttributeId="group"/>
      </Apply>
      <AttributeValue DataType="http://www.w3.org/2001/XMLSchema#string">developers</AttributeValu
    </Condition>
  </Rule>
</Policy>
```

The Response:

```
<Response>
  <Result>
    <Decision>Permit</Decision>
    <Status>
      <StatusCode Value="urn:oasis:names:tc:xacml:1.0:status:ok"/>
    </Status>
  </Result>
</Response>
```

A Note About Time Ranges

Time ranges represent a common access model (eg, allow access only from 9am to 5pm). Indeed, the original version of this guide included an example policy with exactly this constraint. The example used the time-greater-than and time-less-than functions to define the range as this was the standard mechanism. At first glance, this looks like a viable solution, but in fact it doesn’t always work. The reason why is pretty complicated, but basically the problem is that the XACML specification references another specification for time values, and that specification requires normalization to GMT which can shift your time range across midnight, thus causing the greater-than or less-than test to fail. For more detail, look in the XACML TC mailing list archive.

Because supporting time ranges was an original goal in XACML, a quick solution has been proposed: a time-in-range function. This new function takes three time values (the moment, the lower-bound on the range, and the upper-bound on the range), and returns true if the moment falls between the two bounds. The semantics are such that the range can describe any period of time less than 24 hours in duration, and it handles time zones and normalization correctly. This problem was not identified in time for inclusion in the 1.1 specification, however, so it won’t be included until the 2.0 specification (though it has been proposed as an errata against 1.1, so it should be standardized sooner than XACML 2.0).

As a temporary measure, an implementation of this new function has been implemented, but not included in the core packages. Instead, it is included in the samples package. When this function is standardized, it will be moved into the core code. Because this is not a standard function yet, it is also not in the standard XACML namespace for functions. It is in a temporary namespace which also will be changed when the function is standardized. The bottom line is that if you need to express time ranges in your policies then you should use this new time-in-range function, even keeping in mind that it’s not standard functionality yet, since it gives the proper behavior in all time zones whereas the mechanisms in the current specification do not.

As a general rule of thumb, you should take care with using the time comparison functions. All time AttributeValues that don’t include a time-zone will be shifted to GMT. That may cause you to cross the midnight boundary and therefore fail the comparison test. Note that the included module implementation for supplying the current-time always includes the local time-zone, which makes things a little easier.

Using The XACML Implementation

Everything described in the XACML overview (and much more) is supported in this library. The rest of this guide will describe how to use the APIs to build and customize PDPs, PEPs, or any related pieces that fit into the XACML framework.

Overview Of The APIs

The APIs are broken into several packages:

- `com.sun.xacml` is the core package. It contains the logic for target matching, rule evaluation, policy and policy set handling, and other related features. It also contains the PDP class, which is the entry point for most code.
- `com.sun.xacml.attr` is the package that supports all the standard XACML attribute data types, as well as designators, selectors, and the factories used to create new attribute values. Standard interfaces and abstract classes are provided to define new attributes types.
- `com.sun.xacml.combine` contains all of the standard combining algorithms as well as the factory for accessing those algs. Standard interfaces are provided for creating new combining algorithms.
- `com.sun.xacml.cond` provides all of the condition and function logic, supporting all the standard functions (except the XPath functions which, for reasons explained in the SelectorModule section, are not included). Standard interfaces and classes are provided for defining new functions.
- `com.sun.xacml.ctx` represents all of the types defined in the context schema, ie. the request and response formats. All of these classes are both encodable and parsable, making it easy to build a PEP based on these classes.
- `com.sun.xacml.finder` provides support for retrieving things that the PDP needs. This is used for things like finding policies, retrieving attributes not provided in the request, resolving resource identifiers, or generating real-time values. A set of standard classes lets the application writer use different kinds of finder modules to suit their tasks.
- `com.sun.xacml.finder.impl` has a few simple finder modules that provide some basic functionality. Any of these can be pulled out and replaced with better or more fully featured implementations, but they provide enough functionality to get your PDP working without having to write any custom finder code.
- Also in the APIs are `com.sun.xacml.attr.proxy` and `com.sun.xacml.cond.cluster` which are really just convenience classes used by the configuration code. In practice these are rarely used by programmers.

This section gives an overview of how to use these APIs. For more in depth, runnable examples, look at the samples bundle. It contains several classes, including a command-line driven PDP app that’s useful if you’re just getting your feet wet or if you want a quick policy evaluator. It also contains several example policies and requests, and the time-in-range function. Between those samples and this guide, you should have all the examples you need to get started. If that’s not the case, mail the author and we’ll try to get it right next time.

Note that the core classes all do some logging using the standard `java.util.logging` interfaces. All classes that log messages register themselves under the name of their package and class, so (for example) if you want to see messages from from only the core classes, you can look for messages registered in the `com.sun.xacml` namespace. Logging is done using the standard logging levels, and is categorized as follows:

- **SEVERE:** Any errors that either halt or seriously impact the normal operation of the system (like a required config file being unavailable)
- **WARNING:** Messages that may reflect some problem, but don’t necessarily represent a problem in the system (like failure to load a policy because of an unavailable datatype)
- **INFO:** Common messages that are interesting from a security or management point of view even if they reflect normal operation (like not finding any applicable policies or finding too many applicable policies)
- **CONFIG:** Used only by startup routines (like the ConfigurationStore)
- **FINE:** Any message that’s useful to people who don’t know the details of the implementation (like the finder system being queried)
- **FINER:** Any message that doesn’t require detailed knowledge of the system, but may not be important (like Target match failing)
- **FINEST:** Any implementation-level details that people may actually want to consume

As you write your own code to extend the core SunXACML features you should keep these categories in mind and try to match your message levels to these guidelines.

Building a Basic PDP

One of the more common and simple uses for this system is building a PDP, which is a service that can handle access requests. There are two ways to approach this. One method is to programatically create and connect the compontents you want to use. This was the only option in the 1.0 and 1.1 releases, and is useful if you want to have strong control over your PDP's functionality or if your modules require a high degree of programmatic configuration. The other method is to use the run-time configuration system, introduced in the 1.2 release, which makes it easy to swap in new modules and factory configurations and doesn't require as much code. There are good reasons to use both approaches. The following text outlines the first method described above. More information about the run-time method is available in a separate guide. Regardless of the approach you take, you should read the rest of this section, since it explains the structure of modules used by the PDP.

In essence, a PDP is defined by the finder modules it contains. These modules are used to access policies, retrieve attribute values, and resolve resource hierarchies. A PDP may have any number of modules available to it, and this will often dictate exactly how the PDP functions. The rest of this section uses a few common modules that will get you started. In practice, you will probably need to write at least a few custom modules for your system based on factors like how you manage your policies, where attribute values are available, etc. For more on the finder module system, and how to use it to extend a PDP, see the final section of this guide.

To create and configure a PDP programatically, first you need to instantiate a set of finder modules for that PDP to use. Use of finder modules is optional, but because they provide access to policies (among other things), you always want to include at least one. A sample module (`FilePolicyModule`) is provided to access policies as files, but you can write more complex modules as needed (explained later). Note that this module is provided as a sample and as a quick way to get started. You should not use it any real system, and you should not count on it always being available in future releases. It is a convenient starting point, however, and you can add as many files to the provided module as you like.

```
FilePolicyModule policyModule = new FilePolicyModule();
policyModule.addPolicy("/path/to/policy.xml");
```

Note that this module will reject any policies that it can recognize as invalid. If schema validation is being used it will reject all invalid policies; otherwise it will catch some syntax errors, all cases where invalid function combinations are being used, and any data type mis-matches. Any policies that are recognized as invalid will not be available to the PDP.

A finder module you *should* add is the `CurrentEnvModule`, since it provides values for the current time, date, and dateTime. The XACML specification requires that this information always be available regardless of whether or not it's provided in the request. Of course, you can always use your own module to provide this information, but this functionality must always be available. This module will optionally ensure that current date/time values remain constant during a single evaluation (this is the default behavior).

```
CurrentEnvModule envModule = new CurrentEnvModule();
```

Note that a third module is provided in the `com.sun.xacml.finder.impl` package. It is the `SelectorModule`, and is provided for supporting AttributeSelectors (though only minimally). This is described in greater detail in the next section.

The two modules created above are provided to the PDP through finders:

```
PolicyFinder policyFinder = new PolicyFinder();
Set policyModules = new HashSet();
policyModules.add(policyModule);
policyFinder.setModules(policyModules);

AttributeFinder attrFinder = new AttributeFinder();
List attrModules = new ArrayList();
attrModules.add(envModule);
attrFinder.setModules(attrModules);
```

With these finders defined, we can create a new `PDP` and configure it with the modules using the `PDPConfig` class. Note that in this case we're not providing a `ResourceFinder`, so there will be no support for hierarchical resources (explained later), but this will be the common behavior for most applications.

```
PDP pdp = new PDP(new PDPConfig(attrFinder, policyFinder, null));
```

That's it. Now you can start processing requests, and as long as you provided the right policy/policies to the `FilePolicyModule`, the `PDP` will do the rest. The PDP can be queried through a `RequstCtx` (which represents an XACML Request), or by using an `EvaluationCtx`. The latter is what all the PDP-internal classes use, and effectively is a more efficient representation of both the Request and the evaluation context (ie, the Conext Handler described in the XACML specification). There is a default implementation, but you are free to write your own `EvaluationCtx`, if you like, and query the PDP with that. The samples package contains `SimplePDP`, a ready to run PDP example application, that shows these interfaces in action.

This distribution does not provide any kind of support for sending requests and responses over a network (e.g., to support an online PDP service), but it should be relatively clear how to do this. First, use the above code (or the method described in the run-time configuration guide) to build a PDP. Next, setup some kind of interface like a socket, for accepting Requests and returning Responses (note you could also invent your own wire format if you like, but it's probably easier to work with the standard XACML formats). Now, write a PEP (see below) that generates the Requests, and parses back in the Responses. Finally, plug that PEP code into your applications, and talk to an online PDP. This can be extended further by actually sending the data in some enveloping technology like SAML. There has been some experimental work done on this, and the SAML 2.0 release is planning to include a standard format for exactly this kind of interaction. For more information, send mail to the project team.

Building a Basic PEP

Another common use for this implementation is building a PEP, which interacts with a PDP by creating requests and interpreting responses. A PEP typically interacts in an application-specific manner with its native environment (like a server, filesystem, or other service), so if you're using XACML to provide access control for your system, you'll be writing a PEP. Because there is currently no standard way to send XACML Requests to an online PDP, you'll either need to include both the PDP and PEP in the same code, or you'll need to come up with your own way to pass

Requests and Responses. As explained above, future versions of SAML and XACML will address this issue.

Most of the code that you need for building a PEP is in the com.sun.xacml.ctx package (which represents the context schema). To start out, create a RequestCtx instance, which represents an XACML Request (note that the the samples package contains a full example of how to use these APIs). The Request constructor takes four inputs, which represent the four categories of Attributes in a Request (Subject, Resource, Action, Environment).

```
RequestCtx request = new RequestCtx(subjects, resourceAttrs,
                                    actionAttrs, environmentAttrs);
```

You now have a valid XACML Request that you can pass to your PDP. You can pass that `RequestCtx` instance directly to a `PDP` like the one created in the previous section

```
ResponseCtx response = pdp.evaluate(request);
```

or you can encode your Request and pass it to something else for evaluation

```
OutputStream ouput; // output to whatever uses the Request
request.encode(output);
```

Note the return type from the call to `evaluate`. At some point the Request gets evaluated by a PDP, and this results in a `ResponseCtx`, which represents an XACML Response. A Response is a collection of Results, though in practice you’ll usually only have one Result. A Result contains the Decision, a status code, and optionally a message or Obligations.

The `SampleRequestBuilder` class in the samples package has a complete example of how to use these APIs, and the `SimplePDP` class shows a Request being evaluated. Between these examples and the JavaDocs, it should be pretty easy to build a simple PEP.

Creating and Encoding Policies

Like Requests and Responses, Policies and PolicySets can also be created and encoded using this implementation. All policy-related classes are immutable, but they can be created either by their components using constructors, or through `getInstance` methods using the DOM node that defines the corresponding type. This lets a program build whole policies or just parts of policies as needed. Creating a Policy or PolicySet is somewhat more complex than creating a Request, so rather than walk through the example code here, a complete example is provided as the `SamplePolicyBuilder` class in the samples package. Given that, there are a few useful features to note.

One useful set of features are the `encode` methods, which can be found on any class that represents XML-encodable data. Like Requests and Responses, you can use the `encode` methods to encode a Policy or PolicySet to its XML form. This is useful if you’re writing policy authoring tools, code that generates policy in real time, or any number of related systems.

Another useful feature set is the accessor methods on all policy-related classes. This lets you parse a policy and then inspect it from within your code. This kind of functionality is especially useful for debuggers, visualizers, and other systems that require knowledge of how policies are structured (note that you may never need to use these interfaces if you’re only loading policies into a PDP and then evaluating Requests). These accessors are also useful if you write code that does dynamic policy generation based on existing policies. For instance, if you want to write code that creates a PolicySet that should have the same Target as some other Policy you’ve already loaded, that’s easy using the existing APIs. This example is actually quite common when you have PolicyFinderModules (explained in more detail later) that need to combine policies under a single PolicySet in real-time.

One final thing to note is the `PolicyTreeElement` interface. This is a general interface that lets you work with the various elements of a policy tree (Policy, PolicySet, PolicyReference, and Rule). It makes it easy to traverse the tree, doing applicability checking and evaluation. Programs that want to build policy sub-sets, test them, and then continue building the policies will find this interface particularly helpful.

Validating Policies and Requests

By default, schema validation is not done when a Request is handled, nor does the example `FilePolicyModule` finder module validate by default. This is intentional, since documents may be coming from a source that always generates valid documents (as in the above explanation of policy generation), or from a source that has already validated the documents. While some basic errors will be caught by the parsing routines even without schema checking, strict checking is not done (in the interest of processing time, allowing applications to build the XML in pieces, and for the reasons listed previously). So, if you are accepting unchecked requests or policies in your PDP, you should enable schema validation.

The first step in doing this is to make sure you have an XML parser package that implements the standard `jaxax.xml.parsers` interfaces, and that it supports validation. A popular example of this is the Xerces package from Apache. Once this is in your classpath, the standard way to set it as the XML parser to use is through the `javax.xml.parsers.DocumentBuilderFactory` property. In the case of Xerces, you would say:

```
javax.xml.parsers.DocumentBuilderFactory=org.apache.xerces.jaxp.DocumentBuilderFactoryImpl
```

Now that this is in place, the only thing left to do is tell the PDP to use these validating features. The easiest way to do this is by using the `com.sun.xacml.PolicySchema` (if you’re using the sample `FilePolicyModule` module) and `com.sun.xacml.ContextSchema` properties to set the schema file locations for the policy schema and the context schema respectively. If you want just context or just policy validation, then only set that one property.

Putting all of that together, your Java invocation might look like:

```
java -Djavax.xml.parsers.DocumentBuilderFactory=org.apache.xerces.jaxp.DocumentBuilderFactoryImp
     -Dcom.sun.xacml.PolicySchema=cs-xacml-schema-policy-01.xsd
     -Dcom.sun.xacml.ContextSchema=cs-xacml-schema-context-01.xsd
     ...
```

Now if any of the policy or request/response documents parsed are invalid, an error will be reported, and the document will be rejected. This does make parsing more expensive, but it also makes the PDP safer, and guarantees that you’re working with valid inputs.

Supporting AttributeSelectors

In addition to the two modules used in the PDP example, a third module called `SelectorModule` is provided for supporting the optional AttributeSelector type. Because AttributeSelectors are an optional part of the XACML language, you do not need to include this module to build a conformant PDP. If you do want to support selectors, however, you have two choices: include this module, or write your own.

The `SelectorModule` uses Xalan from Apache for XPath support, so if you're going to use this module, you'll have to download the Xalan package or use a JDK that supports these routines. Specifically, the module uses the `org.apache.xpath.XPathAPI` class. At present, there is no standard Java API for XPath queries, but when one becomes available, this module will be changed to use that interface.

A note about the provided `SelectorModule`: as the JavaDoc says, this class is provided as a separate module rather than core functionality (like the `AttributeDesignator`) expressly so that it's easy to pull out of a PDP. Because it's optional functionality, and because XPath queries are expensive operations, a PDP may want to exclude this functionality. However, this module is not particularly efficient, and it is tied to a particular XPath library, so for these reasons also you may want to exclude this module. The module will work perfectly well for simple cases, but for a more robust PDP you should consider writing your own module to support AttributeSelectors.

One final issue to consider is namespace handling. The `SelectorModule` tries to use the namespace definitions from the root of the policy document that contains the selector invoking the module. If namespaces are used in the XPath expression, then they will be mapped from the policy and used in the query. If no namespaces are used in the Request, then there should be no namespace mappings in the XPath expression. Beyond that, the implementation doesn't try to do anything fancier (like add default namespaces if none are used, or walk the tree looking for namespace mappings in nodes other than the root). In practice you shouldn't need this kind of functionality, but if you find that you do, then you should consider implementing your own module to support AttributeSelectors.

Getting and Building the Implementation

The distribution packages contain the code used to build that release. If you want to get the latest code, however, you'll need to get it from the CVS tree. The code in the CVS tree is always tested, and is usually at least as stable as the code included in each release, but it may contain changes that are incompatible with past releases. It may also conflict with the sample code and the explanations in this guide, so care should be taken when using the latest code. You can get the code using anonymous CVS (the CVS download page explains this). You can also grab nightly tarballs of the CVS tree.

The distribution packages also contain built jar files, but to keep up with the latest changes you may want to build the code yourself. Building the code included in the releases and building the code from the CVS tree works the same way, and is fairly simple. The build process uses Ant, a cross-platform build system. The default target builds the code into a `build` directory. There is also a `doc` target to build the JavaDocs, and a `jar` target to build a jar file of the code. Note that the code is built with debugging on, which is what most people who are building the code themselves want.

Testing the Implementation

The XACML TC has defined a set of conformance tests to help promote interoperability. The tests are defined as paired Requests and Policies, and the expected Response from evaluating those pairs. Some of the test cases are fairly basic, while others require custom functionality in the PDP and/or Context Handler. To run these test cases, there is a very simple test package included with this project. The test package's readme explains what you need to do to run the tests.

Note that there are currently no regression tests, input/output tests, or other formal tests available as part of this project. Mostly this is because no one has had the time to build these into the testing framework. If you're interested in helping to remedy this situation, please let us know!

Extending The XACML Implementation

Now that you've seen how to work with the basic components, the next sections will deal with how to extend the system. This covers most of the interfaces and factories, and talks about how to include new attributes, functions, combining algorithms, and finder modules.

Working With The Factories

As is explained in the next three sections, a simple system of factories is used to support all datatypes, functions, and combining algorithms. In the 1.0 and 1.1 releases the factory system was extremely simple: each factory was a singleton with static interfaces, and modifying the factory affected all users. Also, there was no mechanism for defining your own factory implementation. With the 1.2 release this has improved: a single static default instance is available, but you can swap in new default factories for the core code to use. Also, there are now generic factory instances, so you can define your own implementations of factories and use those in your code. The run-time configuration system supports this too.

Because the 1.0/1.1 factory system supported only a single factory instance for each kind of factory, there was no way to support different factories for different uses, and modifying a factory in one section of code affected all other code. In the 1.2 release this has improved, since you can now work with multiple factory instances. You stil define a single, global default factory that is used by the core code, however, so only half the problem is solved. The reason for this is that it will require changes to many interfaces to support the "right" functionality. Since the 2.0 release will require exactly these kinds of changes (because it will be supporting multiple versions of the XACML schema), the interfaces will be changed then to pair identifiable factories with PDPs. This will give the system full safety with the factories. Until then, care should still be taken when modifying the default factories.

A related problem in the 1.0 and 1.1 releases was, because there was only one instance of each kind of factory, you could modify the standard functionality and create non-standard factories. This has been fixed in the 1.2 release. Specifically, standard functionality is now supported by a specific instance of each kind of factory (for instance, the `com.sun.xacml.attr.StandardAttributeFactory`) and these standard factories are immutable. You cannot add to or remove from these factories (doing so will throw an exception). This ensures that you can ask for a factory supporting only standard functionality, and that it won't at some future time also support custom functionality.

Note that the system starts with standard factories as the default. This means that you cannot simply ask for the default factory and then add to it. If you want the standard functionality, but want to augment it with some new datatype, function, or algorithm, then you should first create a new factory instance supporting the same standard functionality. This is easy to do. For datatypes

```
StandardAttributeFactory standardFactory =
    StandardAttributeFactory.getFactory();
final BaseAttributeFactory newFactory =
    new BaseAttributeFactory(standardFactory.getStandardDatatypes());
...
// modify the contents of the new factory
...
AttributeFactory.setDefaultFactory(new AttributeFactoryProxy() {
```

```
            public AttributeFactory getFactory() {
                return newFactory;
            }
        });
```

and combining algorithms

```
    StandardCombiningAlgFactory standardFactory =
        StandardCombiningAlgFactory.getFactory();
    final BaseCombiningAlgFactory newFactory =
        new BaseCombiningAlgFactory(standardFactory.getStandardAlgorithms());
    ...
    // modify the contents of the new factory
    ...
    CombiningAlgFactory.setDefaultFactory(new CombiningAlgFactoryProxy() {
            public CombiningAlgFactory getFactory() {
                return newFactory;
            }
        });
```

it looks pretty much the same. Functions are handled differently, since they're more complex, so a convenience method is provided to help you out

```
    FunctionFactoryProxy proxy = StandardFunctionFactory.getNewFactoryProxy();
    ...
    // modify the contents of the new factory through the proxy
    ...
    FunctionFactory.setDefaultFactory(proxy);
```

In the following sections, the assumption is that the factories being modified have been setup in this manner.

With the addition of a generic interface for creating new factory implementations, new base implementations were added (along the lines of the `FunctionBase` class). These are basically convenience classes that implement the basic functionality that most people need for these factories. If you are implementing a new factory, consider using these base implementations as a starting point.

One final note about the factory interfaces: in the interest of keeping the interfaces as stable as possible from past releases to this release, all of the original static interfaces on `AttributeFactory`, `FunctionFactory`, and `CombiningAlgFactory` have been left intact and functional. They will still operate on the default factories. Many of these methods, however, have been deprecated, and they will be removed with the 2.0 release. They are also much slower in their new form. It is strongly recommended, therefore, that if you are currently using any of these interfaces you switch to using the new methods. Mostly, this involes asking for a factory instance (see the examples in the following three sections) and operating on it, rather than simply using the static intarfaces on the base factory classes. For more details, look at the javadocs for information about what has been deprecated and what methods are now recommended.

Adding New Attribute Types

XACML supports many common datatypes by default, including strings, integers, email addresses, dates, and URIs. Because all attributes in XACML specify their type, however, it's easy to define new types and include values of these types in standard policies and requests/responses. Adding support for a new datatype to your PDP is almost as easy, though it requires a little more work.

All attribute types are represented as subclasses of the abstract `AttributeValue`. To create a new type, therefore, you start by creating a new class that extends `AttributeValue`, you provide the type's identifier, and then implement the encode method as well as equals and hashCode (from `Object`). The other methods all have default behavior that will be correct for most types. Here is an example of an attribute that represents a simple string value:

```
    public class NewAttribute extends AttributeValue {

        public static final String TYPE = "newType";

        private String value;

        public NewAttribute(String value) throws URISyntaxException {
            super(new URI(TYPE));
            this.value = value;
        }

        public String encode() {
            return value;
        }

        public boolean equals(Object o) {
            return value.equals(o);
        }

        public int hashCode() {
            return value.hashCode();
        }

    }
```

Now that there is a class that supports the new type, it needs to be added to the `AttributeFactory` using an `AttributeProxy`. The proxy is called each time a new value is created. Typically you just use an anonymous class to support the new type, and then have that class call constructors or methods to get new instances of the new type:

```
AttributeFactory factory = AttributeFactory.getInstance();
factory.addDatatype("newType", new AttributeProxy() {
    public AttributeValue getInstance(Node root) throws Exception {
            // this method is not implemented in the previous example,
            // but would simply pull the data out of the node, validate
            // the data, and then create a new NewAttribute
            return NewAttribute.getInstance(root);
        }
        public AttributeValue getInstance(String value) throws Exception {
            return new NewAttribute(value);
        }
    });
```

Of course, if you wanted, you could make a more complex proxy, but this is really all that a proxy needs to do. At this point, if the `AttributeFactory` is asked to create a value of type "newType", it will call one of the two proxy methods (based on how the factory was invoked).

Adding New Functions

Just as with attributes, XACML includes standard support for a wide variety of functions from simple boolean operations (like integer-equal, and/or/not, date-less-than, etc.) to complex Set and Bag-related functions. There are even functions that let you do set mappings using other functions as transforms. Should this fairly large set of functions not be sufficient, however, you can write new functions and make them available for any policy to use.

All functions implement the `Function` interface, though a `FunctionBase` helper class, which implements some common functionality, is provided if you don't need to do anything too complex in your new function. The JavaDocs in `Function` spell out what is expected of any new function. Basically, a function can be evaluated against a set of inputs, can be asked if a set of inputs is acceptable, and provides information about its return type and identity.

The `FunctionBase` helper class may be useful if you're writing a function that has predefined input and output types and/or if it doesn't need to do any special checking when validating parameters (see the JavaDocs for more specifics). The `FunctionBase` implements most of the required methods, leaving a custom function needing to implement only the evaluation method. For instance, suppose you want to write a new function that takes a boolean and a string, and compares them to see if text of the string is equal to the value of the boolean. Using the `FunctionBase` helper class, this is a pretty easy task.

```
public class BoolTextCompare extends FunctionBase {

    // the name of the function, which will be used publicly
    public static final String NAME = "bool-text-compare";

    // the parameter types, in order, and whether or not they're bags
    private static final String params [] = { BooleanAttribute.identifier,
                                              StringAttribute.identifier };
    private static final boolean bagParams [] = { false, false };

    public BoolTextCompare() {
        // use the constructor that handles mixed argument types
        super(NAME, 0, params, bagParams, BooleanAttribute.identifier,
              false);
    }

    public EvaluationResult evaluate(List inputs, EvaluationCtx context) {
        // Evaluate the arguments using the helper method...this will
        // catch any errors, and return values that can be compared
        AttributeValue [] argValues = new AttributeValue[inputs.size()];
        EvaluationResult result = evalArgs(inputs, context, argValues);
        if (result != null)
            return result;

        // cast the resolved values into specific types
        BooleanAttribute bool = (BooleanAttribute)(argValues[0]);
        StringAttribute str = (StringAttribute)(argValues[1]);
        boolean evalResult;

        // now compare the values
        if (bool.getValue()) {
            // see if the string is "true"
            evalResult = str.getValue().equals("true");
        } else {
            // see if the string is "false"
            evalResult = str.getValue().equals("false");
        }

        // boolean returns are common, so there's a getInstance() for that
        return EvaluationResult.getInstance(evalResult);
    }
}
```

Now you just need to add this function into the factory. The `FunctionFactory` is split into three groups, providing different categories of functions (explained in detail in the JavaDocs). In this case, we've written a boolean function that should be available to any part of the system, so it gets added to the factory as a Target function (since it can be used in a Target, a Condition, or in any general situation). Most functions are singletons (since most don't need state so it saves time and memory), so to add the function, you instantiate it and pass it into the factory:

```
FunctionFactory factory = FunctionFactory.getTargetInstance();
```

```
factory.addFunction(new BoolTextCompare());
```

In addition to writing new functions from scratch, certain functions can be added when new attribute types are introduced. The standard Bag and Set functions, as well as the Equal function are all functions that can work on any datatype. When a new attribute type is created (as described in the previous section), it can be used with these standard functions quite easily. A name has to be chosen, and then the function is created and added to the factory. For instance, if you wanted to be able to do equality checking in a policy on the NewAttribute type created in the previous example, you would configure your factory:

```
FunctionFactory factory = FunctionFactory.getTargetInstance();
factory.addFunction(new EqualFunction("newAttribute-equal",
                                      NewAttribute.TYPE));
```

Now you can use the "newAttribute-equal" function in a policy. Likewise, the `BagFunction` and `SetFunction` classes provide several methods for creating new instances that use new attribute types. The [type]-one-and-only function is almost always a function that you want to include for your new type. To do this, you configure the factory the same way:

```
FunctionFactory factory = FunctionFactory.getGeneralInstance();
factory.addFunction(
    BagFunction.getOneAndOnlyInstance("newAttribute-one-and-only",
                                      NewAttribute.TYPE));
```

Now the PDP knows about the "newAttribute-one-and-only" function which can be used as a wrapper around an AttributeDesignator to get single values of the newAttribute type. Note that the higher order bag functions will also be able to use new attribute types, but because of how they work, you don’t need to explicitly install new instances for new attribute types.

One final kind of function is an abstract function. These are functions that keep some state and therefore can’t be completely constructed until they are provided some information from a policy. An example from the specification is the map function, which derives its return type from its parameters. Unlike regular functions, these are not singletons, and they are created as needed within a policy. The `FunctionBase` may be useful for abstract functions, but if the functions don’t know what types they accept or return (often the case for these functions), they’re better off being defined directly from the `Function` interface.

Like attributes, abstract functions are used by giving the `FunctionFactory` a proxy that is called each time a new instance is needed. The proxy’s `getInstance` method is passed a DOM node that contains the given function (typically an ApplyType), and returns a `Function` based on the data in that node. Because this is a somewhat expensive operation, these functions should be used only when a regular function cannot be used. The only standard function provided as an abstract function is the map function. For more on how to install and create these functions, see the JavaDocs for `FunctionFactory`.

Adding New Combining Algorithms

Combining Algorithms come in two flavors: Policy Combining Algorithms and Rule Combining Algorithms. Each of these is represented by an abstract class that is extended to define a new algorithm. While both classes have a single `combine` method, the two types of algorithms are expected to do somewhat different things.

To create a new Rule Combining Algorithm you need to extend `RuleCombiningAlgorithm` and implement its one method. The `combine` method takes the context and a list of rules (in order), and returns a single Result. All that a Rule Combining Algorithm is expected to do with the rules is evaluate each one, interpreting the result as appropriate. For example, you might write a "PermitOrDeny" rule that returns Permit if any Rule returns Permit, and otherwise always returns Deny. The code will evaluate each Rule, looking at the result, and then decide whether or not to keep going.

```
public class PermitOrDenyRuleAlg extends RuleCombiningAlgorithm {

    public PermitOrDenyRuleAlg() throws URISyntaxException {
        super(new URI("rule-combining-alg:permit-or-deny"));
    }

    public Result combine(EvaluationCtx context, List rules) {
        Iterator it = rules.iterator();

        while (it.hasNext()) {
            // get the next Rule, and evaluate it
            Rule rule = (Rule)(it.next());
            Result result = rule.evaluate(context);

            // if it returns Permit, then the alg returns Permit
            if (result.getDecision() == Result.DECISION_PERMIT)
                return result;
        }

        // if nothing returned Permit, then the alg returns Deny
        return new Result(Result.DECISION_DENY);
    }

}
```

To create a new Policy Combining Algorithm, you need to do a little more work. Start by extending `PolicyCombiningAlgorithm`, and again implement the single `combine` method. Now, in addition to evaluating each policy, you must first check that the policy applies (Rules check applicability when evaluated, but policies do not for various performance reasons). The algorithm must also handle Obligations correctly, ensuring that the correct obligations are returned. This issue is discussed in great detail in the XACML specification, and to a lesser degree, in the JavaDocs. The general rule is that you always return any Obligations that were considered during evaluation and that have the same Effect that the combining algorithm is returning. Taking the same "PermitOrDeny" example used above, the Policy Combining Algorithm will look similar with the additional steps described here.

```
public class PermitOrDenyPolicyAlg extends PolicyCombiningAlgorithm {

    public PermitOrDenyPolicyAlg() throws URISyntaxException {
        super(new URI("policy-combining-alg:permit-or-deny"));
    }

    public Result combine(EvaluationCtx context, List policies) {
        Set denyObligations = new HashSet();
        Iterator it = policies.iterator();

        while (it.hasNext()) {
            // get the next AbstractPolicy, and match it
            AbstractPolicy policy = (AbstractPolicy)(it.next());
            MatchResult match = policy.match(context);

            // see if the AbstractPolicy policy applies...
            if (match.getResult() == MatchResult.MATCH) {
                // ...and if it does, evaluate it
                Result result = policy.evaluate(context);

                // if it returns Permit, then the alg returns Permit
                if (result.getDecision() == Result.DECISION_PERMIT)
                    return result;

                // if it returns Deny, save the Obligations for later
                if (result.getDecision() == Result.DECISION_DENY)
                    denyObligations.addAll(result.getObligations());
            }
        }

        // if nothing matched and returned Permit, then the alg
        // returns Deny, including all Deny Obligations
        return new Result(Result.DECISION_DENY, denyObligations);
    }

}
```

Note that both of these examples are fairly simple, and don't look for Indeterminate returns from matching or evaluating. Often a Combining Algorithm will want to take special actions when these errors occur, either at the moment they happen, or if the algorithm makes it all the way to the end of the list of rules or policies. For instance, if an error occurred when trying to match a policy, then you can't really trust a final result of Deny, since had the matching operation succeeded, it might have a found a policy that returned Permit on evaluation, which would change the final result. These algorithms can be somewhat tricky, so before you try to write your own, you should probably look at the source code for the standard algorithms in the `combine` package for guidance.

The last step is to include the new algorithms in the `CombiningAlgFactory`:

```
CombiningAlgFactory factory = CombiningAlgFactory.getInstance();
factory.addAlgorithm(new PermitOrDenyRuleAlg());
factory.addAlgorithm(new PermitOrDenyPolicyAlg());
```

Like with Functions, there is only one instance of each Combining Algorithm since they don't keep any state.

Adding New Finder Modules

You've already seen how to create instances of finder modules, and how to include specific sets of finder modules in a PDP. Since you may want to write new modules, however, there are APIs to define new modules, and those modules are included in a PDP in the same manner that has already been discussed. There are three types of finder modules: attribute finder modules, policy finder modules, and resource finder modules. Note that these can get complex, especially given the rules laid down by the XACML specification, so you should read the JavaDocs carefully before trying to write one of these modules.

Attribute finder modules are used when an attribute value can't be found in the request. They are called from a designator or selector, and so there are two methods for finding values, one of which takes the fields used in a designator and the other which accepts an XPath expression. There are also methods used to specify which of those two retrieval methods (or both) are supported by a given module, and what types of attributes can be found using a designator. This information is used by the `AttributeFinder` to decide which modules to use for any given query.

Policy finder modules are used both to find Policies for a given request and to find Policies referenced from PolicySets. Like attribute modules, there are different methods for the different types of retrieval, and there are also methods that let the module specify which kinds of retrieval are supported. There are several rules about policy retrieval, most important of which is that a policy module may never find more than one matching policy. If it does, this is an error and must be reported. Decisions like whether or not to cache policies and whether to add support for signed policies are up to the implementor.

Resource finder modules are used by a feature of XACML called hierarchical resources. A request must name a resource, but that resource might be specified as the root of a hierarchy of other resources, like a filesystem or XML document. If the resource is specified as hierarchical, the `ResourceFinder` is used to get the complete list of resource identifiers associated with the root resource. Because of the potential for abuse of this feature, and because the specification doesn't include any specific requirements for what kinds of hierarchies must be supported, there are no resource modules included in this distribution.

Resource hierarchies are, perhaps, one of the most mis-understood features of XACML. Basically, they let you request access to more than one resource, as long as the resources are in some simple hierarchy. The resulting Response will contain a separate Result for each of the resources in the hierarchy. To make this happen, specify the root of the hierarchy in the standard resource-id attribute, and then include the standard scope attribute with a value of either "Children" or "Descendants" (for details, look at the javadocs for `ResourceFinder` and `EvaluationCtx`). If you supply a `ResourceFinderModule` that can handle the hierarchy, then it will be queried for all the resources under the root resource, and each resource will result in a separate Request to the PDP. This guide doesn't go into more detail on this topic since few people actually need this feature, and since XACML 2.0 will introduce a much clearer and more powerful system for handling this kind of fucntionality.

Finally, here is a simple example of how to write a custom `AttributeFinderModule` (writing other modules looks very similar, and the com.sun.xacml.finder.impl package provides more examples, as does the test code, which relies on custom finders to pass all the tests). In this example, a module is provided that can get the current processor load (a measure of how busy the CPU is right now), which might be useful for determining whether or not a new application is allowed to run on this host, or even if another intensive session (like an encrypted channel) should be opened. Note that this value isn't being returned to the PEP, so there's not much security risk in providing this module. It's simply providing another real-time piece of data that policies can use to make informed decisions. The module itself is pretty simple: it announces its capabilities, and when called to find a value it checks first to see that it's being called for the attribute it knows about.

```
public class LoadEnvModule extends AttributeFinderModule {

    // always return true, since this is a feature we always support
    public boolean isDesignatorSupported() {
        return true;
    }

    // return a single identifier that shows support for environment attrs
    public Set getSupportedDesignatorTypes() {
        Set set = new HashSet();
        set.add(new Integer(AttributeDesignator.ENVIRONMENT_TARGET));
        return set;
    }

    public EvaluationResult findAttribute(URI attributeType,
                                          URI attributeId,
                                          URI issuer, URI subjectCategory,
                                          EvaluationCtx context,
                                          int designatorType) {
        // make sure this is an Environment attribute
        if (designatorType != AttributeDesignator.ENVIRONMENT_TARGET)
            return new EvaluationResult(BagAttribute.
                                        createEmptyBag(attributeType));

        // make sure they're asking for our identifier
        if (! attributeId.toString().equals("processor-load"))
            return new EvaluationResult(BagAttribute.
                                        createEmptyBag(attributeType));

        // make sure they're asking for an integer return value
        if (! attributeType.toString().equals(IntegerAttribute.identifier))
            return new EvaluationResult(BagAttribute.
                                        createEmptyBag(attributeType));

        // now that everything checks out, get the load...
        long procLoad = someMethodToCalculateProcessorLoad();

        // ... and return that value
        Set set = new HashSet();
        set.add(new IntegerAttribute(procLoad));
        return new EvaluationResult(new BagAttribute(attributeType, set));
    }

}
```

The final step is to install this module as was explained back in the section on building a PDP. With this module in place, a policy can now include:

```
<EnvironmentAttributeSelector DataType="http://www.w3.org/2001/XMLSchema#integer"
                              AttributeId="processor-load"/>
```

Send comments, questions, and corrections to seth proctor.