XACML Authorization Delegate

This is an implementation of the authorization delegate API. This PEP compiles request and environment information into an authorization request that is passed to a XACML policy decision point (PDP).

- Requirements
- How to Map to XACML Policies
    - Policy Persistence
    - Finding the Effective Policy Set
    - No Applicable Policies
    - Implementation
- Local PDP
    - Cascading Deletes
    - XACML AuthZ Delegate
    - PolicyFinderModule (w/PolicyLocator for JBossPDP configuration)
    - ModeShapeResourceFinderModule
    - AttributeFinderModule(s)
        - ResourceAttributeFinderModule(s)
        - SubjectAttributeFinderModule
        - EnvironmentAttributeFinderModule
- XACML Role-Based Access Control

Requirements

- Authoring XACML policies is an involved technical process, with behavior hinging upon the total policy set. For this reason policies/sets will be centralized, named and reused as much as possible. (Less is more)
- Administrators may choose to enforce a different set of XACML policies at any point within the repository tree.
- Metadata, such as ACLs or rights statements, can be used to avoid authoring more XACML.
    - Resource properties can determine the relevant policy within a set and the outcome from within that policy.
    - Policies may depend upon an access role attribute.
    - Policies may reference any value obtained via a SPARQL query, relative to the content resource, but the query must be mapped to a XACML attribute in configuration.
- Policies (and/or sets of them) must be stored in the repository.
- Policies must be enforced on externally managed content, i.e. projected resources within a federated resource. (inc. filesystem connector)
- Must be able to authorize based on requesting I.P. address
- Must be able to authorize based on resource mixin types
- Must be able to authorize based on Hydra rightsMetadata datastream
- Must be able to authorize based on resource mimetype
- Must be possible to use same rules as defined in policies in the following contexts (except for #1, we only need to demonstrate/document the possibilities):
    1. calls to Fedora REST-API
    2. calls to Fedora Java classes
    3. calls to external Solr index
    4. calls to external triplestore

How to Map to XACML Policies

This includes how policies are stored in the repository and how they are linked with content resources.

Policy Persistence

Policy and Policy Set resources may be stored in any part of the repository tree at the discretion of the administrator.

Policies and Policy Sets are referenceable fcr:datastream resources that contain XACML XML.

Policy sets contain Fedora URI references to other policy sets and policies. Policy sets can define a tree structure of policies.

Policy URIs have the form `info:fedora/path/to/PolicyResource`. These URIs are the IDs of the Policies and PolicySets in the XACML datastream.

Finding the Effective Policy Set

One policy set in the workspace will be configured (in the XACML Policy Finder Module) as the default policy for the workspace. This is the same as saying that this default policy set is effective at the root of the Fedora resource tree and everywhere else unless overridden.

Any fcr:resource may set a property **policy** which makes a strong reference to a single policy or set resource. This overrides the effective XACML policy for itself and child resources. This action requires administrator levels of access, as determined by the effective policy, or by use of a login with the fedoraAdmin role.

The Fedora Policy Finder implements the JBoss XACML PolicyFinderModule interface. It retrieves the policy or set that is effective for a given context resource, and searches the tree for the closest parent with a policy property and returns that XACML. It also resolves internal URI references from policy sets at the request of the PDP, looking in the workspace by policy URI.

Note: When combining XACML policies in sets, you specify a combining algorithm of either permit-override or deny-override. For this reason the policy property is single-valued.

Here is an example repository tree:

- ROOT
  - collection A
    - policy property --> policy A
  - collection B (inherits default policy set from ROOT)
  - policies
    - default policy set (XACML contains links to B and C)
    - policy set A (XACML contains links to default and D)
    - policy B
    - policy C
    - policy D
  - collection X
    - policy set Z(XACML contains links to Y and W)
    - policy Y
    - policy W
    - My Documents
      - policy property --> policy set Z

No Applicable Policies

This situation can arise when the only policy set (or policy) for some content contains a *target* element or any other XACML construct that restricts it's applicability to authz requests. If the PDP can find no policy that targets the request, it returns a NOT_APPLICABLE result to the XACML AuthZ Delegate. The delegate will then return false to ModeShape, indicating that the action is not permitted.

Implementation

The delegate uses a JBoss XACML engine. PicketLink and PicketBox projects use the same XACML PDP, which is the Sun XACML implementation repackaged by JBoss. (PicketLink is a larger umbrella project of security services.)

The JBoss XACML engine has no significant runtime dependencies, outside of a PicketLink utilities jar. The other dependencies are the Java Servlet API and XML APIs.

Relevant APIs:

- org.jboss.security.xacml.sunxacml.finder.PolicyFinderModule is used to find a policy (or policy set) that matches the request evaluation context. Also used to lookup a policy that is referenced within a policy set by ID.
- org.jboss.security.xacml.sunxacml.finder.AttributeFinderModule is used to find attribute values when evaluating a policy.
- Constructing a policy set for the JBOSS engine:
  - see JCR 2.0 16.3 and JBossLDAPPolicyLocator as an example.

Sun XACML Javadoc: http://sunxacml.sourceforge.net/javadoc/

Local PDP

Is this better implemented as a remote or a local PDP service. The PDP can be used as a bean without the webapp runtime, or it can be configured as a separate service (SOAP). The trade-offs are identified in the table below.

Internal PDP (within ModeShape JVM)	External PDP (remote XACML service)
Minimal administrative overhead through dependency injection, etc..	Flexible, can be any XACML implementation
ModeShape cache will keep frequently used ACL metadata in memory. Removes the need for any additional cache.	Decent performance may require custom metadata caches.
No network overhead making connections or marshaling data.	Network latency, etc..
Decision and policy cache invalidation may be based on events.	Cache invalidation requires wiring JCR or Fedora JMS specifics into the chosen XACML service. Cache invalidation would be asynchronous.
Adds complexity to the runtime webapp – moving closer to a monolithic, coupled application.	

Cascading Deletes

When Modeshape checks for permission to remove a resource and the authz delegate returns true, there are no followup checks for removal of the child resources. The children (and their children, etc) are deleted along with the parent, but the AuthZ Delegate gets no permission callback for them.

The JBoss/Sun PDP has a notion of scope for an evaluation that can be EvaluationCtx.SCOPE_IMMEDIATE, EvaluationCtx.SCOPE_CHILDREN or EvaluationCtx.SCOPE_DESCENDANTS. If we are to use the scope feature, we need to implement another interface to retrieve children/descendants. This interface is a org.jboss.security.xacml.sunxacml.finder.ResourceFinder. When descendants are in the evaluation scope, then each is evaluated in turn by the PDP. The PDP traverses the descendants first, resolving applicable policies for each resource in turn. So this strategy should honor policy overrides with the scope of a tree delete action.

XACML AuthZ Delegate

This is the implementation of the AuthZ Delegate interface. The delegate functions as a Policy Enforcement Point (PEP) in the XACML orchestration. The delegate formulates requests for decisions and dispatch them to the JBossPDP. It interprets the results and return an appropriate response to the hasPermission() method. Most of the work for this component is in building a request context for the PDP.

PolicyFinderModule (w/PolicyLocator for JBossPDP configuration)

Fedora will need an implementation of this interface, originally part of Sun XACML. This finder module will deliver the correct policy set for the resource ID in a XACML request context according the rules for "Finding the Effective Policy Set" above. A similar kind of lookup, based on resource path, happens in the access roles provider.

This is admittedly a little abstruse, but the policy finder module becomes part of the PDP via an implementation of the PolicyLocator interface. An implementation must have a no argument constructor and received configuration via a setOptions() callback method. The PolicyLocator has a map where it will store its PolicyFinderModule for use by the JBossPDP. A simple example is the JBossPolicyLocator, which puts a list of pre-configured policies into a wrapper policy finder module. This wrapper policy finder module is added to the locator's internal map under the policy finder module key. To see how the PDP builds the list of policy finder modules at initialization time, see JBossPDP.boostrapPDP(), especially the createPolicyFinderModules() method. There you can see it build a list of finder modules from each configured policy locator.

The policy lookup operation involves the following steps:

1. finding the nearest real resource for a given the path, since access checks are sometimes performed on new resources before they are added.
2. Then you find the nearest parent with a policy property.
3. Then you read in and return the policy XACML. (There is a PolicyReader class you can use here)
4. It will also need to support requests for policies by resource URI, i.e. resolving policies that are linked from the first policy.

ModeShapeResourceFinderModule

Implements callback methods for finding child and descendant resources. This is used for delete and possibly move operations, which have cascading effect on the resource tree.

AttributeFinderModule(s)

Attribute finder modules do not work the same as policy finder modules. The AttributeLocator is an implementation of the AttributeFinderModule interface.

ResourceAttributeFinderModule(s)

These finder modules retrieve information about the resource which is being accessed. They should be implemented as several finder modules for clarity.

A triple finder module resolves attribute IDs (URIs) to RDF properties on fedora resources. Attributes are not configured in advance. XACML authors may reference any URI and if there is one or more triple with the correct subject resource and predicate, then the resource will be returned. It should match the data type expected in XACML, which is also part of the arguments passed to the finder module.

A SPARQL finder module retrieves data indirectly linked to the subject resource. This finder module will resolve attributes via a configured map of attribute IDs to SPARQL queries.

A common fedora finder module retrieves the standard attributes noted in the Fedora XACML Attributes page.

JBoss has an abstract class called StorageAttributeLocator which may be useful for formulating SPARQL queries that function much like prepared statements against a DB, where there is a replacement value. That pattern may be useful for a ResourceAttributeFinderModule.

SubjectAttributeFinderModule

EnvironmentAttributeFinderModule

XACML Role-Based Access Control

Fedora implements the XACML 2.0 Role-Based Access Control profile out-of-the-box. Under the profile, policies are divided into two types: **Roles** and **Permissions**. **Role** policies only define *Subjects*, then link to **Permission** policies. Permission policies only define permission sets: *Rules* that apply to *Resources* and *Actions*. Policies are grouped into *PolicySets*. Permission policy sets are never referenced directly, but only through Role policy sets.

This model makes it simple to assign multiple roles to intersecting sets of permissions, and to allow for hierarchical roles: more powerful roles can inherit the permissions of lesser roles, then extend them in their own permission policy. Thus, an admin role inherits the permissions of both reader and writer roles.

The Fedora default policies implement the roles **admin**, **writer**, and **reader**. See Fedora Basic Roles - ModeShape Permission for a mapping of roles to ModeShape actions and permissions..

Fedora XACML RBAC policies on github

Some notes on identifier conventions:

- *PolicySetId* identifiers will be URIs of the form `info:fedora/policies/PolicyResource`. This identifier will be resolvable to the path to the policy resource in the repository.
- Internal policy element identifiers for the XACML elements *PolicyId* and *RuleId* will have the urn prefix `fcrepo-xacml.`