[OpenAz] Version 94 and "sketch of tutorial" and rough cut (smoke test only) of "download, build, and run"

Rich.Levinson rich.levinson at oracle.com
Thu Jun 10 02:13:48 EDT 2010

- Previous message: [OpenAz] OpenAz Conference Call - Thursday June 10, 1 PM ET
- Next message: [OpenAz] Slides on Multi-Decision Processing for Today's Meeting
- **Messages sorted by:** [date] [thread] [subject] [author]

To OpenAz maillist,

Version 94 has been released, which includes minor cosmetic changes that have been part of ongoing testing. Also, note that version 93 was also a minor
update that added one file for the AzAttributeFinder javadoc.

At this point in the project, all the core functionality that was originally planned
is pretty much included with the addition of query and attributefinder in recent
releases. Therefore, the current effort will temporarily focus on firming up the existing code and javadoc, so that it is easy to use and understand.

This effort will consist of two focal points:

1. We will attempt to make the project easy to download, build, and run. It is not there yet, but an initial set of instructions are provided below
   which interested parties may attempt to execute, and hopefully report any issues that are found. What is currently provided is what "in theory
   should work", but there is no evidence at this point in time aside from
   my own work area that it will.
2. We are beginning to prepare a "tutorial", which will attempt to explain
   the following:
   1. "What the OpenAz project is attempting to do and why."
      Information in this category has been previously presented, but as
      the project progresses, additional perspective is gained, which enables cleaner representations of the concepts involved to be "explained".
   2. "What we have done so far."
      This is basically an explanation/tutorial of the OpenAz code and test programs, and how they address the objectives in the previous bullet.

Note: "Sketches" of some of the sections of the above-described "tutorial" are
included below in this email.

Comments and suggestions would be much appreciated.

Thanks,
Rich

****

First, here is a rough set of "instructions" of what "should work":

javadoc should be available by downloading the gnu tarball from:
http://openaz.svn.sourceforge.net/viewvc/openaz/azapi/doc/

the full project should be available by downloading the gnu tarball from:
http://openaz.svn.sourceforge.net/viewvc/openaz/

Assuming the project has been successfully downloaded and extracted, then to build and run one should only need to do the following (note: this assumes that java 6 and ant have been installed on your system):

After downloading and extracting the project to a directory that may be referred to as: <TOPOFPROJECT>

The following cmd lines, "in theory", should build the project without errors:

```
cd <TOPOFPROJECT>\openaz\test
ant clean
ant
```

Then to run the TestStyles.java program, again "in theory", the following
commands should work:

```
cd <TOPOFPROJECT>\openaz\test\bin
set baseline=<TOPOFPROJECT>\openaz
java -Djava.security.manager -cp
.;%baseline%\pep\bin;%baseline%\pdp\bin;%baseline%\azapi\bin;%baseline%\lib\jakarta-commons\commons-logging.jar;%baseline%\testsunxacml-
```

```
    test.TestStyles %baseline%\test\request\sensitive.xml
    %baseline%\test\policy\TestAzApi-GeneratedPolicy.xml
```

If the above command works, then what you should see is a long listing of log information that is put out as the TestStyles program runs. The total is probably about 1/2 meg of text if things run correctly. No attempt will be made in this email to explain the content, except that there should be no Exceptions or other errors encountered that terminate the program.

****

Following is a rough "sketch" of sections of a tutorial that is currently being scoped out:

Section 1: Motivational introduction

The following is preliminary sketch notes of some topics currently viewed as significant to present up front:

motivation for openaz:

will need common representation of attributes to be submitted for authorization; both for applications to define and for authorization systems to represent in policy.

current apis generally have platform-specific objects containing the data used for authorization decisions. As such, it is difficult to establish a common policy representation that will span platforms across an enterprise and between enterprises.

Attributes used for policy decisions are found in

- Subject data found in representations and entitlements associated w requesting user, intermediaries, systems,
- Action data in request messages that are issued to access resources,
- Resource metadata and data associated with the resources and within the resources
- Environment data describing the context of the decision, including time of day, system and application context.

It is assumed that XACML provides a foundation for the standard representation of Attributes to be submitted for an authorization decision, because each Attribute is represented by the following metadata:

- AttributeId: a URI that may be used as an identifier by both XACML Policies and external applications, repositories, and platforms to indicate that the associated attribute value is an instance that meets the criteria for this attribute as implied by the URI, itself. In particular, it is assumed that associated with the URI is an external description of the semantics associated with attributes

Category: grouping of attributes into logical entity; xacml has defined a core set, but it is extensible.

Finders: need for finder callbacks is implicit. The authorization problem is about evaluating a policy that uses attributes. The main feature is that the policy can change, and tomorrow may need attributes it doesn't use today, and others it may no longer need. Therefore, any attempt to pre-define all the required attrs is inherently futile. To address this requirement, the notion of a policy looking for an attribute in a request and upon not finding it, having the option to request it, is fundamental.

****

Section 2: Description of history of project as seen thru the test programs implemented at each phase:

Alert: this section contains some "soapboxing", which is subject to change as feedback will indicate. Also, it is somewhat lengthy and wordy, intended to head off anticipated questions, but, again, feedback will dictate whether the approach is desirable or not.

Preliminary text:

Testing and examples:

This section may be in the category of "more than anyone would ever want to know ...", however, rather than leaving interested people in the dark about the origin of the current state of affairs, a relatively brief history of the development of openaz in terms of what was delivered and what tests could be run with each delivery is presented.

In order to understand the current set of test programs, it is helpful to know the current structure of the test suite. Everything in this tutorial refers to the code and examples that may be found at:

http://openaz.svn.sourceforge.net/viewvc/openaz/

Note that the current test suite is the byproduct of the development process, with a primary focus on establishing end to end functionality with sufficient breadth to verify that all the key functionality is operational as well as that a few key data-types are also properly operational. As a result, less emphasis has been placed on automation of the testing or on testing all datatypes. It is expected that these improvements will be made as time permits as a matter of course on an ongoing basis.

There have been 3 phases of development so far, which will be listed here, which again is to provide context for understanding the current state:

1. org.openliberty.openaz.azapi (interfaces)
   org.openliberty.openaz.azapi.constants (constants, enums, etc)
   org.openliberty.openaz.pdp (impl of azapi interfaces)
   org.openliberty.openaz.pdp.provider (impl of dummy service)
   org.openliberty.openaz.test (impl of tests for azapi)

   AzApi: this was the initial phase of defining a Java-based API that captured the functionality inherent in XACML 2.0 core plus the XACML 2.0 multi-resource profile, plus some flexibility in preparation for eventual use with XACML 3.0. (this phase completed in Sep 2009, and comprised the initial release in the openliberty open source site).

   This phase included a dummy implementation of a pdp that returned canned responses designed to exercise the basic capabilities of AzApi. This was and is represented in:

   - TestAzAPI.java: a simple (but verbose) program that builds up a multiple request, submits it, and processes the responses.

2. org.openliberty.openaz.azapi.pep (interfaces)
   org.openliberty.openaz.pep (impl of pep interfaces)
   org.openliberty.openaz.test (impl of tests for pep)

   PepApi: this phase (Sep 2009 - Jan 2010) was designed to address the fact that AzApi is quite verbose, as a result of being a fairly direct presentation of XACML, and to show that PEP-oriented interfaces are easy to provide on top of the azapi framework.

   The basic theme here was to provide a "mapping layer" whereby a provider could provide a mapper for any kind of Java object that could be directly submitted for an authorization decision and all the code required to take information from the object and pass to azapi would be contained in the mapper, thus freeing appl developers from concerns in this area.

   Basically, all an application developer who wants an authorization decision made needs to know is three api calls to create the request, invoke the decision, and evaluate the response, which in many cases can be done in a single line of java code, for example:

   ```
   if ((pepReqFac.newPepRequest(obj1,obj2,obj3)).decide()).allowed())
   { ... do success processing }
   else
   { ... do failure processing }
   ```

   where obj1 is some kind of Subject object (ex. a JAAS Subject), obj2 is some kind of resource/action object (ex. a FilePermission), and obj3 is some kind of environment object (ex. a Date).

   The main test programs for this phase included
   - TestStyles.java: several tests showing different types of Java objects that could be submitted, including Strings, Dates, HashMaps, Permissions, and javax.security.auth.Subject including contained Principals.
   - BulkDecideTest.java: multiple decision tests

3. org.openliberty.openaz.pdp.provider (impl(wrap) sunxacml, attr finder)
   org.openliberty.openaz.pdp.provider.resources (impl of query support)

   SunXacml: This phase (Feb 2010 - present) has consisted primarily of proving in the use of the SunXacml open source implementation w AzApi and PepApi, and then filling in the missing major functions: query and attr finder callback.

   However, the main thing that changed in the testing arena was that we now started using actual XACML policies that are processed by SunXacml. This transition involved the following changes from phases 1 and 2:

   - instead of using the dummy service:
     openaz/pdp/provider/SimpleConcreteDummyService.java,
     the test programs now used the SunXacml implementation wrapper:
     openaz/pdp/provider/SimpleConcreteSunXacmlService.java

- first challenge was how to transition the test programs to use policies rather than the dummy service which had no policies, but only canned responses: the answer was, of course, to create XACML policies that would process the requests from the test programs. It turned out that all the test programs from phases 1 and 2 could be supported by a single XACML Policy, which is named:

  ```
  TestAzApi-GeneratedPolicy.xml
  ```

  A first attempt to explain the details of this policy is provided below, however, it turned out that when viewed from an attribute-oriented perspective, XACML is fairly straight-forward to understand and policies may be regarded as a hierarchy of defined attribute scopes, which enables it to adapt quite readily to the Java Permission model, hierarchical models, in general, as well as multi-tenant models.

- the second challenge was that it became almost immediately obvious that manually creating XACML policies is extremely labor intensive and that some kind of tool was required to avoid having this become a prohibitive task that would quickly interfere with progress.

  Fortunately, SunXacml came with the rudiments of a tool that could be quickly adapted to address these needs. The original tool is in the SunXacml sample/src directory, named:

  ```
  SamplePolicyBuilder.java
  ```

  This tool was adapted for use in openaz and is in the openaz/test/policies directory named:

  ```
  TestAzApiPolicySets.java
  ```

  Without going into detail, all the policy information for the phase 1 and 2 testing is contained in PolicyEnum.P1 in the above Policy builder.

- after securing the phase 1 and phase 2 functionality as "integrated w SunXacml" using the above Policy builder to build the Policy to handle those test cases, our attention turned to the main remaining functionality:

  - query (aka "whatIsAllowed")
  - attribute finder (aka "callback")

  Definition of the "query" capability revealed some interesting issues regarding the semantics of a query capability in the context of a policy engine. In particular, it raises an immediate issue of

  > "How does one determine what resources a given user has access to?"

  There are at least "two perspectives" on this, but before explaining the perspectives (which will be explained below), first consider the definition of the query method:

  ```
  Set<AzResourceActionAssociation> azRaa =
   AzService.query(scope, azReqCtx, permitOrDeny)
  ```

  where "scope" is a string, azReqCtx is the request context, and permitOrDeny is a boolean indicating whether only permit results are requested or only deny results. The return value, azRaa is a Set of AzResourceActionAssociation objects, each of which identifies a specific resource and a specific action on that resource for which the decision applies.

  The "scope" is intended to be resource-specific, and for example it may be thought of as a regular expression applied to a hierarchical file system. A concrete example would be what file resources within the scope "/A/B/*" does user "xxx" have access to, where user "xxx" is found within the request context. The answer to this would be a list of every file in the system that matched the path "/A/B/*" which would include every file in the directory /A/B, as well as every file in every subdirectory under that, to which user "xxx" was permitted access.

  We are now faced with a significant issue that emerges when policy is extracted from the resources themselves. On the one hand we have a file system, with unlimited numbers of files stored for any number of users. However, the file system, itself, is no longer the keeper of the information about who has access to which file. The actual file system may retain some legacy access control capabilities, but an organization that has decided to empower an external policy engine with the authority to make decisions about who has access to what, ultimately must allow these legacy capabilities to phased out of use, with the possible exception of some admin access capabilities to secure the physical resource.

  {Note: this paragraph is to recognize an issue that may be worthy of discussion, but is considered non-essential for the current analysis, except to move the issue aside: Leaving aside the issues that may arise from this evolutionary transition, for the sake of discussion we can assume that somehow the policy engine and the file system each maintain their own synchronized representation of the policy, and that all the front end policy engine is doing is saving cycles for the file system by giving users answers without actually contacting the file system. However, if a user attempts to access the file system directly, that user will be given exactly the same answer that the policy engine would have given since the undefined "synchronization" mechanism will guarantee that this is the case. (Note: the "synch" mechanism is for discussion only, as ultimately, all it is doing is duplicating a single function in two systems using two paradigms, which is a hard problem to solve, unless one makes the choice that one or the other of these two entities will be delegated w sole authority and the other will simply leave that capability in the hands of the other entity and not try to maintain a duplicate mechanism.)}

Returning to the main line of discussion, the issue that has emerged is that we now have two distinct entities: the policy engine that determines which resources the user has access to, and the resource repository which contains all the resources regardless of what users have access.

In fact, what has emerged is a situation where the policy engine has no knowledge of the existence or non-existence of specific resources, but only knowledge of whether specific users are permitted access to those resources if they exist. In a conjugate sense, we also have a resource repository that inherently has no knowledge of what users should be allowed access to any of the resources that actually exist in the repository.

Therefore, the only way to provide the answer to our queries is to "bind" the information in the policy engine to the information in the resource repository and use the combined information to produce the desired list.

Now we may meaningfully return to the question posed above:

> "How does one determine what resources a given user has access to?"

and explain the "two perspectives"

- one perspective is that in order to produce the list of allowed (or disallowed) resources, we should get the list of all the resources in the scope, which presumably can be obtained from repository by providing it with a representation of the scope and requesting a list of all resources within that scope, then submitting the items on the list one by one to the policy engine and getting a decision on each and only returning the items from the list for which the desired decision (permit or deny) has been returned.

  We have named this approach the

  resource-based query

- the second obvious perspective in this context is that in order to produce the list of allowed (or disallowed) resources in the scope, that we submit the scope and user directly to the policy engine and ask the engine what are the policies that are applicable to this user in this scope, and that the engine will return the policies themselves, which the caller can then presumably use to obtain the resource names from the repository.

  This probably requires a little more context to be clear. In particular, in the real world there are many kinds of resources that are named and organized in many different ways. However, once we choose a specific resource type then we may meaningfully consider the notion of "scope" and how it maps to subsets of the full resource collection.

  Taking a hierarchical file system as an example, access typically will be granted by saying user "xxx" is allowed access to all the files in directory /A/B and its subdirectories. This grant will take the form of the policy specification "/A/B/.*", which is a regular expression where ".*" matches any number of characters after "/A/B". So, if the user requests access to "/A/B/C", the answer will be yes. If the user requests access to "/A/D/C" then, absent any other information about the policy engine, the answer would presumably be no. (Obviously, more specific details would need to be added to tighten up this specification, but the point is simply to indicate the general way that this mechanism would work.)

  The final step is that presumably the policy engine would return the string "/A/B/.*" in response to our query, then presumably we could issue the request to the file system for a list of files meeting the definition "/A/B/.*" and we would then have the necessary information to return the same list to the caller as was done in the first perspective above.

  This approach has been named the

policy-based query

Both these perspectives are included in the current openaz project:

- resource-based query: The test program:
  /test/QueryTest.java
  implements this approach and is supported by the PolicySet, PS7, in TestAzApi-GeneratedPolicy.xml

- policy-based query: The test program:
  /test/TestStyles.java
  has test cases in the method testStyleQuery(), and is supported by the PolicySets PS3,PS4,PS5,PS6 in TestAzApi-GeneratedPolicy.xml

Details of these techniques from a policy perspective are preliminarily sketched below.

As should be clear from above, regardless of which perspective is chosen, it is necessary to "bind" the policy info with the resource info in order to produce a meaningful result.

Therefore, in order to test this capability it was decided to implement a dummy repository, which implements some of the features that would be required of a real repository in order to establish the policy to existing resource bindings required to return a list of allowed(disallowed) existing resources to the caller.

The repository capability is implemented in the following classes:

```
     org.openliberty.openaz.pdp.resources.
             OpenAzResourceDirectory.java
             OpenAzResourceQueryBuilder.java
             OpenAzTestResourceCollection.java
```

These classes together provide sufficient functionality to demonstrate the query capabilities in combination with the policy engine.

More background on the query can be found in the associated emails when the functionality was released, for example:
policy-based:
http://lists.openliberty.org/pipermail/openaz/2010-April/000050.html
resource-based:
http://lists.openliberty.org/pipermail/openaz/2010-May/000061.html

The "attribute finder" is the remaining functionality that has been implemented. The details of this functionality have been explained in the email:
http://lists.openliberty.org/pipermail/openaz/2010-May/000065.html

****

Section 3: policy analysis

It is fairly commonly mentioned that xacml policies are somewhat difficult to understand. Personally, I agree with that sentiment, however, after working with these policies off and on for the last 2 or 3 years, I have found that there are perspectives that make the policies easier to understand.

However, there are a few obvious "issues" with the policies from a usability perspective that should be fairly easy to clear up.

First, the policies are much too verbose, primarily because of the dictates of XML formatting. As a "first approximation" one should be able to separate the meaningful data items from the xml formatting overhead to simply reduce the size of the information that must be understood in order to understand the policy.

Second, the names in many cases are lengthy URIs, which remain even after the xml formatting has been removed. There should be simple abbreviations for most of the URI prefix information to allow users to represent policies in terms of the shorter names of items in their domain.

Both these steps have been taken in the OpenAz project to some degree. The Policy generating program, TestAzApiPolicySets.java, mentioned earlier has been developed based on the SunXacml SamplePolicyBuilder. However, while this program has enabled more efficient policy development, resulting in TestAzApi-GeneratedPolicy.xml, we are still left with some challenges.

First, as an aside, one key element of TestAzApiPolicySets is that it uses a structure AttributeMatchExpression.java, which is a container for a common pattern found in XACML Policies, which is attribute comparison. The structure has the members:

```
        int attributeDesignatorType;
        String attributeMatchId;
        String attributeDataType;
        String attributeValue;
        String attributeId;
        String attributeIssuer = null;
        String attributeFunctionId = null;
        boolean mustBePresent = false;
```

Basically, this structure looks like an Attribute, but it is a Policy-side attribute, in that the attributeValue is the value in the policy that will be compared to the value in the request. The attributeId in combination with the attributeDesignatorType is sufficient for an AttributeDesignator to be defined to get an Attribute from the Request. Finally, the attributeMatchId says what kind of comparison is to be done between the policy side attribute value and the request side attribute value. The attributeFunctionId is used to apply this structure to xacml conditions and mustBePresent can be used to invoke finders if the attribute is not found in the request. Also, as should be obvious, attributeDataType is a helper for the comparison function by representing the format of the data to be compared.

The way the policy builder works is that the Java programmer "simply" defines a sequence of these AttributeMatchExpressions, by providing data for the constructor (there are short constructors for those items that don't require all the members).

However, even with these optimizations/simplifications, one quickly finds that the program that is implemented to construct the policy grows to an unmanageable size and is not easy to maintain or modify. TestAzApiPolicySets.java is an example of this. Despite the fact that it easily generates TestAzApi-GeneratedPolicy.xml, it is large and the prospect of determining how to modify it to tweak policies even in a minor way is not trivial and is quite error-prone.

However, this is only a step along the way, and it appears to point in the right direction, which appears to be that one should not modify the Java code, but simply modify the information that is processed in the Java code. i.e. the objective would be to extract the data elements for the constructors and put that data in a simple framework that can be parsed to generate the code, or at least activate the code that would generate the Policy.

At the moment, all that has been done along those lines is to analyze TestAzApi-GeneratedPolicy.xml to try to extract the fundamental information and represent it in a form whereby the structure and intent of the Policy starts to become clear.

The following is a first cut at defining a parsable structure that could be used to generate these policies: the abbreviations should be self-explanatory: PS = PolicySet, PL = Policy, T = Target, R = Rule, O = Obligation. Similarly, the parameters should also be fairly obvious. The way to use this listing is to compare it line by line with TestAzApi-GeneratedPolicy.xml. There should be a pretty close match, although some of the rule naming has had some prefixes added as an experiment so there are discrepancies, but in terms of the order of elements and the semantics of the policies, the listing below should carry the same info as the xml file.

Given that, the next step is to firm up the structure below at a detailed level and to adapt TestAzApiPolicySets.java to parse the structure below to produce the desired policy.

```
      PS-01 (po)
        T (all)
        PS-02 (po)
          T (all)
          PL-01
            T ((sub-id,"Joe User"))
            T ((sub-id,"josh"))
            T ((sub-id,rfc822,"users.example.com"))
            T ((sub-id,x500name,"CN=Rich,OU=Identity Management,O=Oracle,C=US"))
            T ((res-id,anyURI,"http://www-example.com/toplevel"))
            T ((res-id,"file:C\toplevel"))
            T ((res-id,"file///C/toplevel/permissionTest"),(res-typ,"FilePermission))
            T ((res-id,regexp,"file///C/toplevel/permissionTest/.*"),(res-typ,"FilePermission))
            T ((res-id,"file:\\toplevel00"))
            T ((res-id,"file:\\toplevel01"))
            T ((res-id,regexp,"file:\\toplevel.*"))
            R (Permit)
              T ((act-id,"read"))
              T ((act-id,"Read"))
            R (Permit)
              T ((act-id,"write"))
              T ((act-id,"Write"))
            R (Permit)
              T ((act-id,"write"))
              T ((act-id,"Write"))
              C ((sub-role-id,orcl-weblogic,"developer"))
            R (Permit)
              T ((act-id,"commit"))
              C ((sub-group,admin at users.example.com,"dvelopers"))
            R (Deny,all)
            O (Permit, (sub-id,"user"),(res-id,"resource"))
            O (Deny, (sub-id,"user"),(res-id,"resource"),(act-id,"action")
        PS-03 (po)
          T (all)
          PS-04 (do)
```

```
T ((subject-id, "User1"))
T ((role-id, "admin")) ?? is this needed? (maybe not)
PS-05 (po)
  T ((res-id,".../A/B/.*",regexp), (res-typ,"TestResPerm"))
  T ((res-id,"/-"), (res-typ,"TestResPerm"))
  PL-02 (po)
    T ((res-id,".../A/B/.*"))
    R (Permit, all) (i.e. if target match, then it's Permit)
    O ((res-id,res-id-dsg), (res-typ,res-typ-dsg), (sub-id,sub-id-dsg))
  PL-03 (po)
    T ((res-id,"/-"),(res-typ,"TestResPerm"))
    R (Permit, all)
    O ((res-id,".../A/B/.*), (res-typ,res-typ-dsg), (sub-id,sub-id-dsg))

PS-06 (po)
  T ((res-id,".../A/D/.*"), (res-typ,"TestResPerm"))
  T ((res-id,"/-"), (res-typ,"TestResPerm"))
  PL-04 (po)
    T ((res-id,".../A/D/.*"))
    R (Permit, all) (i.e. if target match, then it's Permit)
    O ((res-id,res-id-dsg), (res-typ,res-typ-dsg), (sub-id,sub-id-dsg))
  PL-05 (po)
    T ((res-id,"/-"))
    R (Permit, all)
    O ((res-id,".../A/D/.*), (res-typ,res-typ-dsg), (sub-id,sub-id-dsg))

PS-07 (po)
  T ((res-typ,"EngineeringServer")
  T ((res-typ,"Menu")
  T ((res-typ,"FrisBee")
  PL-06 (po)
    T ((res-typ,"EngineeringServer")
    R (Permit)
      T ((sub-id,"fred"))
      T ((res-id,"resource-id-EngineeringServer-3"))
    R (Deny,all)
  PL-07 (po)
    T ((res-typ,"Menu")
    R (Deny,all)
  PL-08 (po)
    T ((res-typ,"FrisBee")
    R (Permit)
      T ((sub-id,"fred"),(priv-frisbee,mustbepres,"throw")
      T ((res-id,"resource-id-FrisBee-3")
    R (Deny,all)
```

```
-------------- next part --------------
An HTML attachment was scrubbed...
URL: http://lists.openliberty.org/pipermail/openaz/attachments/20100610/2351f5b8/attachment-0001.html
```

- Previous message: [OpenAz] OpenAz Conference Call - Thursday June 10, 1 PM ET
- Next message: [OpenAz] Slides on Multi-Decision Processing for Today's Meeting
- **Messages sorted by:** [date] [thread] [subject] [author]

More information about the OpenAz mailing list