Version	Date	Author	Comments
1.0	6 February 2012	ISSRG, Univ Kent	First public release

Decision Making in PERMIS

This document discusses how the core PERMIS API should make its decisions and how the returned responses and exceptions should be translated into one of the four possible XACML decisions (Grant, Deny, Indeterminate, NotApplicable). If the current PERMIS implementation (Feb 2010) does not work this way it is flagged below.

The Access Control Part of the PERMIS Policy

Target Specification

The PERMIS policy provides a means to specify the different domains that the policy will protect. This is done using the `TargetPolicy` element. Domains (or targets) can be specified as either LDAP resources or resources specified by HTTP-like URLs. The following gives an example `TargetPolicy` element which specifies three different target domains, one specified as an LDAP resource and two specified as a URL.

Table 1 Specifying the Available Targets

```
<TargetPolicy>
  <TargetDomainSpec ID="printers">
    <Include LDAPDN="dc=kent,dc=ac,dc=uk">
      <Exclude LDAPDN="dc=library,dc=kent,dc=ac,dc=uk"/>
    </Include>
    <ObjectClass Name="Printer"/>
  </TargetDomainSpec>
  <TargetDomainSpec ID="unsecured">
     <Include URL="http://kent.ac.uk/"/>
  </TargetDomainSpec>
  <TargetDomainSpec ID="secured">
     <Include URL="https://kent.ac.uk/"/>
  </TargetDomainSpec>
</TargetPolicy>
```

The union of all the domains specified in the `TargetPolicy` element is called the **coverage domain** of the policy.

A requested target that is not a member of the coverage domain of the policy can never be mentioned in any of the access rules (see later) of that policy because this is checked at policy parsing time. Note

that the opposite is not true: it is perfectly possible for a requested target to belong to the coverage domain of the policy but not have any access rule mention this particular target.

Action Specification

The actions recognised by the PERMIS policy are specified in the `ActionPolicy` element. Each action must have a name and one may optionally specify arguments for each action. It is also possible to restrict the domains on which the action makes sense by including references to `TargetDomainSpec` elements inside the `Action` element. The following gives an example of an `ActionPolicy` which specifies three actions each of which are (supposed to be) valid on the coverage domain of the policy.

Table 2 Specifying Actions Valid on the Coverage Domain

```
<ActionPolicy>
  <Action Name=”Print” ID=”Print”>
    <Argument Name=”nrOfPages” Type=”Integer”/>
  </Action>
  <Action Name="GET" ID="GET"/>
  <Action Name="POST" ID="POST"/>
</ActionPolicy>
```

However, it should be clear that in a case like this it doesn't make much sense to 'Print' on the target http://kent.ac.uk/, nor does it make much sense to 'GET' on a printer. So, we can restrict the applicability of the different actions by specifying a `TargetDomain` inside the relevant `Action` element. This is the recommended way of constructing policies and it is done as follows:

Table 3 Restricting Action Validity

```
<ActionPolicy>
  <Action Name=”Print” ID=”Print”>
    <Argument Name=”nrOfPages” Type=”Integer”/>
    <TargetDomain ID=”printers”/>
  </Action>
  <Action Name="GET" ID="GET">
    <TargetDomain ID=”secured”/>
    <TargetDomain ID=”unsecured”/>
  </Action>
  <Action Name="POST" ID="POST">
    <TargetDomain ID=”secured”/>
    <TargetDomain ID=”unsecured”/>
  </Action>
</ActionPolicy>
```

When specifying explicitly which domains the action makes sense on, the policy parser will prevent you from granting permissions which don't make sense. In the example above for instance you will not be able to grant permission to Print on http://kent.ac.uk . This error will be picked up at policy parsing time and as a consequence the decision engine will not be created.

Target Access Specification

The most important part of the PERMIS policy is the `TargetAccessPolicy` element which effectively grants permissions to holders of certain roles, subject to optional conditions and optionally specifying obligations as well. This part of the policy consists of a list of Target Access Rules (TAR) which is represented as a list of `TargetAccess` elements in the PERMIS policy.

For instance, the following `TargetAccess` element grants public access (no roles required) to all unsecured web pages in the kent.ac.uk domain for the GET and POST actions:

Table 4 Permission with Explicit Actions

```
<TargetAccess>
   <RoleList/>
   <TargetList>
     <Target>
       <TargetDomain ID="unsecured"/>
       <AllowedAction ID="GET"/>
       <AllowedAction ID="POST"/>
     </Target>
    </TargetList>
</TargetAccess>
```

In this case, we have explicitly specified all allowed actions, which also happen to be all actions that make sense on the specified target. The PERMIS policy syntax allows one to use a shortcut in this case: not specifying any `AllowedAction` explicitly is equivalent to specifying all actions *that are defined for the target at hand*. Thus the above can be rewritten as (assuming the `ActionPolicy` as defined in Table 5):

Table 5 Permission with Implicit Actions

```
<TargetAccess>
   <RoleList/>
   <TargetList>
     <Target>
       <TargetDomain ID="unsecured"/>
     </Target>
    </TargetList>
</TargetAccess>
```

The `RoleList` element of a TAR limits the applicability of the TAR by specifying the roles a subject should have in order to be granted the specified permission. It is possible to limit the applicability of the TAR even further by attaching a condition to the TAR. For instance, suppose we want to limit printing jobs of students to be less than a hundred pages in size. Also, when the student doesn't have sufficient printing credit available the job should not be accepted. When the job has been done, the student's

available credit should be decreased by the number of pages printed which is expressed as on `Obligation`[1]. This could be done as follows:

Table 6 Limiting the Applicability of a TAR with a Condition

```
<TargetAccess>
   <RoleList>
     <Role Type=”permisRole” Value=”student”/>
   </RoleList>
   <TargetList>
     <Target>
       <TargetDomain ID="printers"/>
     </Target>
    </TargetList>
    <IF>
     <AND>
        <LT>
          <Arg Name=”nrOfPages” Type=”Integer”/>
          <Const Type=”Integer” Value=”100”/>
        </LT>
        <GE>
          <Environment Parameter=”printCredit” Type=”Integer”/>
          <Arg Name=”nrOfPages” Type=”Integer”/>
        </GE>
      </AND>
    </IF>
    <Obligations>
      <Obligation>
         Decrease printer credit by number of printed pages[2]
      </Oblgation>
    </Obligations>
</TargetAccess>
```

How does PERMIS Make a Decision?

The general (and original) idea behind the PERMIS policy is:

Everything Not Explicitly Allowed Is Denied

The original assumption was that the PERMIS PDP would be the only PDP ever called by the PEP, so if PERMIS did not know the answer to a query it would return Deny. However this assumption no longer holds true, so the PERMIS PDP should reply Not Applicable and Indeterminate as well as Deny.

Every TAR represents a permission (or a set of permissions) that is available to subjects possessing certain roles, so by adding TARs to a policy, it may happen that ‘Non Grant’ results are turned into ‘Grant’ results but everything that was Granted should still be Granted.

[1] The Obligation is not specified fully but this is not the point of this document anyway.
[2] The Obligation is not specified fully but this is not the point of this document anyway.

However, in order to make PERMIS usable in a multi-policy environment we should take care not to return a Deny when in fact the policy has nothing sensible to say about the request. To this end, when asked to make a decision the PERMIS engine works in two stages.

- In the first stage it tries to determine whether the request can ever lead to a Grant result by inspecting the requested Target and Action (i.e. the permission), as well as the Subject passed in. If it is clear that the request is outside the scope of the policy then Not Applicable is returned. Only when it is clear that the request *might* be granted (because it falls within the scope of the policy) does the second stage start.
- In the second stage, PERMIS inspects the TARs and if it finds a TAR that grants access, the returned result will be Grant. If it doesn't find such a TAR the returned answer may either be Deny or an exception may be thrown.

These two stages are further described below.

Determining the Applicability of a Request

The PERMIS API receives a request through four different parameters:

- `Subject`: contains the valid roles of the subject as well a reference to the PERMIS engine that created it
- `Action`: contains the requested action
- `Target`: contains the request target
- `Environment`: contains the environmental variables that are available during decision making

PERMIS's first stage of decision making checks whether

- None of the given `Subject`, `Action` and `Target` is missing. If one of them is, an exception is thrown "[Indeterminate. Subject|Action|target] is missing" and the request stops here. Note that the current implementation only returns [general `PbaException`] so this needs enhancing.
- The `Subject` has been created by the decision engine that is being interrogated. If it isn't an exception is thrown "Indeterminate. Unrecognised Subject Object" and the request stops here. Note that the current code only returns [general `PbaException`] so this needs enhancing.
- [3]The `Target` belongs to the coverage domain of the policy. It is doesn't, an exception is thrown and the request stops here. [Not Applicable, `TargetOutOfDomainException`]
- The `Action` is specified in the policy (by name and parameters) and whether the requested target belongs to the target domain associated with the action. If any of these is not satisfied, an

[3] The current code calls `RiskAssessment` (PEP callback mechanism) at this point but this is wrong. It should only be called after a Grant response has been determined since there is no risk in returning a deny or not applicable. E.g. if the callback mechanism signals that the Subject object can no longer be used, an exception is thrown telling the PEP to call the CVS again.

exception is thrown and the request stops here. [NotApplicable-`ActionNotInPolicyException`]

Hence, when a request passes the first stage of decision making, it has been established that there may be a rule granting the requested permission to the subject.

Evaluating the Target Access Rules

Evaluating a Single Target Access Rule

This section describes the evaluation of a single Target Access Rule. The combination of multiple TARs into a global decision is discussed in the next section.

The evaluation of a TAR results in one of three possible outcomes:

- The Boolean value TRUE is returned. This means that this TAR grants access to the requested resource.
- The Boolean value FALSE is returned. This means that the TAR cannot grant access to the requested resource.
- An exception is thrown meaning that something went wrong during the evaluation of the TAR's condition.

The evaluation of a TAR works as follows:

- First, it is determined whether the requested action matches one of the allowed actions (which were either specified explicitly or implicitly) for the TAR. If it isn't FALSE is returned.[4]
- Next, it is determined whether the requested target belongs to the domain of the TAR. If it isn't then FALSE is returned.
- Next, it is determined whether the TAR's roles (the required roles) are subordinate to the subject's roles. If they aren't FALSE is returned (because the subject's roles aren't sufficient).
- If the TAR does not have a condition, then TRUE is returned.
- If the TAR does have a condition, the condition is evaluated.
    - If the TAR's condition evaluates to TRUE, then TRUE is returned.
    - If the TAR's condition evaluates to FALSE, then FALSE is returned.
    - It there is a problem evaluating the TAR's condition, then an Indeterminate exception is thrown.

Note that the condition of a TAR is only evaluated once it has been established that the action and target match and that the subject's roles are sufficient. The exception thrown will make a distinction between a missing environmental variable [Indeterminate-`NotInEnvironmentException`] and any other error [Indeterminate-`EvaluationException`]. Note that the exception is not returned to the PEP at this point, but is held in memory and the next TAR is evaluated, since it might still be possible to get a TRUE response from a subsequent TAR.

[4] Note: in the actual implementation this step is skipped because the TARs are indexed by action name, and hence only the TARs for which the action name matches are actually considered.

Evaluating the List of Target Access Rules

The final question is how to combine the individual TAR evaluation results into an overall decision. In the absence of conditions (and obligations) in the policy PERMIS' rule combining algorithm can be stated as 'permit overrides': the moment one TAR returns TRUE, then an overall Grant decision is made, if none of them do (meaning in this case that all TARs returned FALSE) then an overall Deny decision is made.

The next question is how to treat condition evaluation errors. An evaluation error can either be caused by an (environmental) attribute necessary for the evaluation of the condition not being present in the environment or by a syntax/runtime error of some kind (e.g. bad syntax of attribute or division by zero).

First, we describe how PERMIS works now (Feb 2010)[5]:

- Each TAR is evaluated in turn.
- If a TAR evaluates to TRUE, then Grant is returned (along with any obligations attached to the TAR).
- If a TAR evaluates to FALSE, then it is simply ignored.
- If a TAR's evaluation results in an exception, then an exception is thrown.

Thus, currently evaluation of the TARs stops immediately once an evaluation exception has occurred, and this exception is returned to the caller.

A downside of this approach is that there may actually be a further TAR which would grant the subject access but this TAR is never reached because the evaluation of TARs has already been suspended. There are no security issues with this approach but there is a potential denial of service.

The preferred approach only differs slightly from the current implementation and is as follows:

- Each TAR is evaluated in turn.
- If a TAR evaluates to TRUE, then Grant is returned (along with any obligations attached to the TAR).
- If a TAR evaluates to FALSE, then it is simply ignored.
- If a TAR's evaluation results in an exception, then this is noted (by using local variables) and the next TAR is tried.

If after all the TARs have been evaluated none of them returned TRUE, then either all TARs returned FALSE (in which case Deny is returned as the overall decision) or at least one TAR's evaluation resulted in an exception (in which case an exception is returned as the overall decision). The issue now becomes which exception should be returned if several exceptions were encountered? Here we could arbitrarily choose to return the exception associated with the first evaluation error (i.e. only remember the first exception and ignore subsequent ones, which would give the same result as the current implementation), or we could try to find the first evaluation error that was a result of a missing attribute (if we think that the PEP is likely to be able to supply the missing attribute). The second option seems to result in more successful requests and so we choose this one. This means that the first exception will be

[5] The use of MSoD has been ignored in this description.

remembered, and it will be replaced by a [Indeterminate-`NotInEnvironmentException`] if that occurs subsequently.

This approach ensures that

- Access will be granted if there is at least one rule that says so.
- Access will be denied if all rules explicitly deny access.
- An exception will be thrown if at least one TAR has an error in it
- The [Indeterminate-`NotInEnvironmentException`] will take precedence and will be returned if it has occurred, allowing the PEP to supply more environmental parameters which might result in a subsequent Grant result.

Because of the benefits, this is the approach we will take.

With respect to which obligations are returned, the proposed implementation (based upon the existing one) will ensure that the obligations associated with the first applicable TAR will be returned.

Alternatives for Evaluation of the Target Access Rules

Another approach is as follows. When it is discovered that an attribute is missing, the PDP returns the [Indeterminate-`NotInEnvironmentException`] and the AIPEP then calls some configured in 'attribute finders[6]' to supply a value for the missing attribute. The AIPEP than calls the PDP again. This would certainly be a helpful addition to the current PERMIS standalone authorization server.

Yet another alternative is that the PEP supplies values for all attributes that may be needed by the policy. With PERMIS, it is possible to specify all environmental attributes mentioned in the policy by calling the `getEnvAttributes()` on the `XMLPolicyParser` class. However, computing the actual value of all these attributes may actually be an expensive operation for the PEP and hence one could supply 'stub' values for some of these expensive attributes. When it has been determined that the actual value of the attribute (rather than the stub value) is needed, the PEP will be informed of this (somehow) and can submit a second request when it was able to compute the missing values. Again, this requires more changes to the existing code base than the proposed TAR combining mechanism.

---

[6] Attribute Finder as such is not in the XACML standard. Some people see this as Sun's implementation of part of the functionality of the Context Handler, which is among other things, responsible for contacting the PIPs. (see message http://markmail.org/message/2jwfyrijxqpe5qel from Anne Anderson)
Basically, you register the Attribute Finders with the PDP, and when the PDP sees that it is 'missing' an attribute, it will query the Attribute Finders (which is in fact step 5 of the XACML2 Data-Flow diagram. The answers returned are equivalent to step 10. Such a facility would make supporting BTG much easier. Thus the big difference is that the PDP does not return the Indeterminate answer, but goes looking for the missing attribute through the Attribute Finders. However, if we want to make this facility available to all PDPs, then we can place the functionality into the AIPEP, which is our enhanced version of the contect handler.

PERMIS Responses and Exceptions and XACML Response Codes

In this section we will describe how the different PERMIS Responses may be translated into XACML response codes by an XACML wrapper around the core PERMIS API.

The obvious ones are:

- When the core PERMIS API returns TRUE, then this translates into a Grant XACML response code.
- When the core PERMIS API returns FALSE, then this translates into a Deny XACML response code.
- A NotApplicable-`TargetOutOfDomainException` is translated into a NotApplicable XACML response code.
- An Indeterminate-`NotInEnvironmentException` is translated into an Indeterminate XACML response code with a missing-attribute status. The `MisssingAttributeDetail` element may be used to signal which attribute was missing.
- An `EvaluationException` is translated into an Indeterminate XACML response code with a syntax-error status.

When an `ActionNotInPolicyException` is thrown, then most of the time it is fine to translate this into a NotApplicable response code. Only in the case when the action was not found in the policy because of a missing action parameter could an Indeterminate with missing-attribute status be more appropriate. The current code, however, does not distinguish between these occurrences, and hence the following translation takes place:

- An `ActionNotInPolicyException` is translated into a NotApplicable XACML response code.

Since, the XACML wrapper code has to create the Subject, Action and Target objects from the XACML request context, it can check whether any of them is null. If they are then an Indeterminate response code will be returned before the PERMIS PDP is actually called. The status of the Indeterminate response may be syntax-error (if e.g. an LDAP DN could not be parsed) or missing-attribute (if e.g. resource-id was missing). Also, the code takes care of the fact that the Subject object is most definitely created with the correct PermisRBAC reference as its creator.