Programming with the PERMIS API

Version	Date	Comments
0.1	28th August 2007	First draft by George Inman
0.2	2nd September 2007	QA by Linying Su
0.3	30th April 2010	Md. Sadek Ferdous. Updated the document with the current state of the API.
0.4	24th of May 2010	Stijn does QA and makes a number of changes in the process.
0.5	25 May 2010	David. Introduction
0.6	28 May 2010	Stijn. Minor changes.

1. Introduction

The PERMIS API provides a programmable interface to the PERMIS RBAC engine. Role Based Access Control (or Attribute Based Access Control more generally) comprises two parts:

1. Assigning Roles and/or Attributes to Users
2. Assigning Permissions to sets of Roles/Attributes.

A PERMIS authorization policy provides the rules for both of the above i.e. 1) who is trusted to assign which roles/attributes to which users (the credential validation policy) and 2) which roles/attributes are needed in order to be granted access to which resources under which conditions (the access control policy).

Consequently the PERMIS RBAC engine offers two primary services/APIs,

1) the credential validation service (CVS), called via the getCreds method, which validates user credentials according to the credential validation policy, and

2) the policy decision point (PDP), called via the authzDecision method, which makes an

access control decision based on the access control policy.

An application will typically call getCreds first, and be returned the user's set of valid attributes. It will then call authzDecision, and be returned the access control decision. The call to authzDecision may be repeated multiple times using the same set of valid attributes, as and when the user makes requests to access different resources. The call to authzDecision is typically a couple of orders of magnitude faster than the call the getCreds since the latter typically involves cryptographic operations as well as network accesses (in pull mode).

2. Constructing the Decision Engine

Three elements should be considered when constructing the Decision Engine i.e. PERMIS PDP and CVS, which is implemented as a PERMIS RBAC object. The first is a Policy Finder object, which is used to load and parse the PERMIS policy. We provide several Policy Finder objects, discussed below. The second element that should be considered is the CustomisePERMIS class used to customize the initial variables and components used by the PERMIS RBAC. The final step in constructing the Decision Engine is initializing the Decision Engine object itself.

2.1 Policy Finder Objects

Each PERMIS RBAC object requires a PERMIS policy that specifies the access control and credential validation rules that the decision engine will follow. The policy finder object takes this policy and parses it for use by the decision engine. In order to initialize the CVS and PDP, a policy finder object must be provided.

We provide various implementations of policy finder objects and the one that you use depends on where the policy is stored and the format that it is stored in.

2.1.1 Simple Permis Policy Finder (issrg.simplePERMIS.SimplePERMISPolicyFinder)

The Simple Permis Policy finder (issrg.simplePERMIS.SimplePERMISPolicyFinder) is the easiest policy finder object to use and is located inside the issrg.simplePERMIS package. It allows a plain text XML policy file stored on the hard drive to be loaded and parsed.

The constructor takes two inputs, a File object initialized with the XML policy file on the hard drive and a Principal object representing the Source of Authority for this policy.

Usage:

```
PolicyFinder pf = null;
Principal prin = null;
try {
  prin = new  LDAPDNPrincipal("cn=A Permis Test User,o=permis,c=gb");
} catch (RFC2253ParsingException e1) {
  e1.printStackTrace();
}
try{
  pf = new SimplePERMISPolicyFinder(new File("path_to_file.xml"),prin);
} catch(java.io.IOException e){
  e.printStackTrace();
}
```

There are four other variations of the constructor that allow different ways to load the XML policy file on the hard drive with different options. Look at the SimplePERMISPolicyFinder class to know more about them.

2.1.2 Simple Permis AC Policy Finder (issrg.pba.rbac.x509.SimplePERMISACPolicyFinder)

The Simple Permis AC Policy Finder (issrg.pba.rbac.x509.SimplePERMISACPolicyFinder) can be used to parse a policy stored inside an X.509 attribute certificate. This class checks that an attribute certificate stored in a byte array is signed by the principal and that the policy's unique identifier is correct.
Constructor: The most general constructor takes five inputs, a byte[] containing an attribute certificate, a String object containing the unique policy identifier, a java security principal object which represents the SOA who signed the policy and a signature verifier object to check the AC's digital signature and a satLevel which is the logging level for the SAWS logging. The last argument is optional and can be left out if not required. The signature verifier object can be null if no signature checking is required. If the signature verifier is null, no signature verification will be done when the credentials are validated (i.e. during the call to getCreds()).Usage:

```
byte [] byteArrayAC = /* read byte array from file */
String policyId = ”thePolicyIdentifier”
Principal = new LDAPDNPrincipal(”cn=the policy issuer”);
SignatureVerifier = null; /* no signature verification will be done */
int satLevel = 0; /* no SAWS logging required */
PolicyFinder pf = null;
try{
  pf = new SimplePERMISACPolicyFinder(byteArrayAC, policyId, principal,
            sv, satLevel);
} catch(issrg.pba.PbaException e) {
  e.printStackTrace();
}
```

2.1.3 Repository AC Policy Finder (issrg.pba.rbac.x509.RepositoryACPolicyFinder)

The Repository AC Policy Finder, which extends the Simple Permis AC Policy Finder, is the most commonly used policy finder class and can be used to load and parse a policy stored in any repository object that implements the issrg.utils.repository.AttributeRepository interface. This class retrieves the attribute certificates stored in the SOA's user entry and searches for a policy with the correct unique identifier, checks the X.509 AC's signature and then parses the XML policy stored in the X.509 AC.

Constructor: The constructor takes four inputs, an instance of the Repository interface representing a collection of attribute certificates, a String object containing the unique policy identifier, a java security principal object which represents the SOA who signed the policy and a signature verifier object to check the AC's digital signature. The signature verifier object may be null if no signature checking is required.

Usage:

```
AttributeRepository attRep = /* create attribute repository here */
String policyId = ”thePolicyIdentifier”
Principal principal = new LDAPDNPrincipal(”cn=The Policy Issuer”);
SignatureVerifier sv = null; /* no signature verification will be done */
PolicyFinder pf = null;
try{
  pf = new RepositoryACPolicyFinder(attRep, policyId, principal, sv);
} catch(issrg.pba.PbaException e) {
  e.printStackTrace();
}
```

2.2 Customising the PERMIS RBAC object

Before initialising the PERMIS RBAC object its components can be customized according to the needs of the application. The CustomisePERMIS class (issrg.pba.rbac.CustomisePERMIS) can be used to set all these system variables and the most commonly used methods are explained below. These calls should be invoked in a static synchronized block with the creation of a PermisRBAC object to prevent other threads from modifying your settings. Please note that these methods are optional and that not all must be invoked before the decision engine is created. If this class is not utilized then sensible default values are used.

2.2.1 Setting the System Clock (CustomisePERMIS.setSystemClock(String className))

This is an optional method that allows you to set a secure system clock to be used by the decision engine. The CustomisePERMIS class contains a default value of "issrg.pba.rbac.SystemClock" which is sufficient for most applications but any class that implements the abstract issrg.pba.rbac.Clock class can be used. In order to set a new class as the system clock the setSystemClock method should be called with a String object containing the new class name as an argument.

Usage:

```
CustomisePERMIS.setSystemClock("foo.MyClock");
```

where foo.MyClock extends the abstract issrg.pba.rbac.Clock class.

2.2.2 Setting the AuthzTokenParser (CustomisePERMIS.setAuthzTokenParser(String className))

The AuthzTokenParser interface defines methods for extracting credentials from Authorisation Tokens and as such the AuthTokenParser can be seen to define the format in which PERMIS will expect these credentials. By default PERMIS uses the MultiAuthzTokenParser as its default AuthzTokenParser.

The CustomisePERMIS.setAuthzTokenParser method allows one to set the format of these authorisation tokens by passing the name of a new AuthzTokenParser class as a String argument. PERMIS supports a number of of AuthzTokenParser implementations:

1. the SimplePERMISTokenParser class that parses SimplePERMISToken objects. (issrg.simplePERMIS.SimplePERMISTokenParser) described above,
2. RoleBasedACParser (issrg.pba.rbac.x509.RoleBasedACParser) that parses attribute certificates for roles, and
3. ShibbolethAuthzTokenParser (issrg.shibboleth.ShibbolethAuthzTokenParser) that parses plaintext shibboleth attributes for roles.
4. The default MultiAuthzTokenParser which can handle multiple token types and basically acts as a dispatcher.

Usage:

```
CustomisePERMIS.setAuthzTokenParser("issrg.pba.rbac.x509.RoleBasedACParser");
```

2.2.3 Setting Attribute Certificate and PKC LDAP attribute names

Different LDAP servers have different schemas that name LDAP attributes differently, by default PERMIS expects attribute certificates to be stored in "attributeCertificateAttribute;binary" LDAP attributes and for public key certificates to be stored in "userCertificateAttribute;binary"

LDAP attributes.

If you wish to use a different attribute name for attribute certificates such as “attributeCertificateAttribute” the CustomisePERMIS.setAttributeCertificateAttribute(String) method is available to pass the new attribute name to the decision engine.

Usage:

```
CustomisePERMIS.setAttributeCertificateAttribute(“attributeCertificateAttribute”);
```

If you wish to use a different attribute name for user public key certificates such as “userCertificateAttribute” the CustomisePERMIS.setUserCertificateAttribute(String) method is available to pass the new attribute name to the decision engine.

Usage:

```
CustomisePERMIS.setUserCertificateAttribute(“userCertificateAttribute”);
```

2.2.4 MultiAuthzTokenParser (issrg.pba.MultiAuthzTokenParser)

This class represents an AuthzTokenParser capable of parsing different token types (e.g. X.509 ACs, SimplePERMISToken, etc.). This class holds a mapping from the TokenType to the actual AuthzTokenParser instances. Its main purpose is to dispatch the call to parse a certain Attribute to the relevant parser.
The easiest way to make sure that that the correct parser can be found is to use the TokenType methods, e.g.TokenType.X509AC.asAttribute() for the X.509 ACs or TokenType.SIMPLE_PERMIS_TOKEN.asAttribute() for simple PERMIS tokens.

Usage:

```
byte[] bytes = null;
try {
  bytes = FileUtility.getBytesFromFile(new File("Path_to_file.ace"));
} catch (IOException e) {
  e.printStackTrace();
}

String holder = "cn=the holder";
String issuer = "cn=simple token issuer";
String roleType = "permisRole";
String roleValue = "student";
SimplePERMISToken simpleToken =
    new SimplePERMISToken(holder, issuer, roleType, roleValue);

Object[] obj = {TokenType.X509AC.asAttribute(bytes),
                TokenType.SIMPLE_PERMIS_TOKEN.asAttribute(simpleToken)};

try {
  /* Push in the two credentials. This assumes that the (default)
   * MultiAuthzTokenParser is used.
   */
  Subject s = pbaApi.getCreds(java.security.Principal user, obj);
} catch (issrg.pba.PbaException pe) {
  pe.printStackTrace();
}
```

2.2.5 Configuring PERMIS for use with X509 Certificates (CustomisePERMIS.configureX509Flavour())

Whilst the PERMIS RBAC is set up to use the MultiAuthzTokenParser by default many users

prefer to use X509 certificates to store policies and roles and a method is available to set all the necessary variables and register the necessary extensions to facilitate this. The CustomisePERMIS.configureX509Flavour() method can be used to register the correct AuthzTokenParser and AC extensions for the user. After this method has been called, PERMIS will accept X509 certificates by default.
Usage:

```
CustomisePERMIS. configureX509Flavour();
```

3. Initialising the Decision Engine

Once the above configurations have taken place you are ready to create the PERMIS decision engine object. The decision engine itself is an instance of the class issrg.pba.rbac.PermisRBAC. Because it has already been statically set up in the steps above the actual process of creating the engine is relatively simple, requiring only that one of the class's constructors are called properly.

The PermisRBAC class contains four callable constructor methods which we will discuss here. The first constructor is the most basic of all and only takes the PolicyFinder object as an input, this has the advantage of being simple to use but the disadvantage of preventing the decision engine from being able to work in pull mode unless additional repository locations are defined in the policy. In the pull mode, PermisRBAC is able to get authorization tokens from some specified repositories. However, in push mode, users have the tokens in advance and PermisRBAC is used to validate them.

Usage:

```
PermisRBAC engine;
PolicyFinder pf;

try{
  pf = new SimplePERMISPolicyFinder(new File("path_to_file.xml"),principal);
} catch(java.io.IOException e) {
  e.printStackTrace();
}

try{
  /* Create the PERMIS decision engine. */
  engine= new PermisRBAC(pf);
} catch(issrg.pba.PbaException pba) {
  pba.printStackTrace();
}
```

The second constructor takes three inputs; a PolicyFinder object, an AttributeRepository object and an AuthzTokenParser object and can be used to create a decision engine object which uses a single predefined repository object to pull user attributes. An AttributeRepository object can be a single type of repository such as LDAPRepository, FileRepository or WebDAVRepository, or be a hybrid repository, which consists of multiple (types of) repositories. A hybrid repository is represented as an issrg.utils.repository.MultiRepository object. This is the most commonly used constructor as it allows for the most flexibility as well as allows for the use of various repositories in the same decision engine. The AttributeRepository object itself can be used to pull user attributes when decisions are being made, if this input is null push mode will be used for credential validation unless additional repository locations are defined in the policy. For example, PermisRBAC is able to get credentials from both local files and LDAP servers in a single call. If the AuthzTokenParser object is null then the default AuthzTokenParser set in CustomisePERMIS will be used.

Usage:

```
PermisRBAC engine = null;
PolicyFinder pf = null;
AttributeRepository ar = null;

try {
  ar = issrg.pba.rbac.URLHandler.getRepositoryByURL(
          "ldap://sec.cs.kent.ac.uk/c=gb");
} catch(BadURLException e ) {
  e.printStackTrace();
}

try {
  pf = new SimplePERMISPolicyFinder(new File("path_to_file.xml"),principal);
} catch(java.io.IOException e){
  e.printStackTrace();
}

try {
  engine = new PermisRBAC(pf, ar, null);
} catch(issrg.pba.PbaException pba){
  pba.printStackTrace();
}
```

The third constructor takes three inputs; a PolicyFinder object, a String object containing the URL of a repository and an AuthzTokenParser object. This constructor is very similar to the second constructor except for the fact that the repository has not been initialized previously leaving the decision engine to determine the repository type and create the repository object itself. The Attribute Repositories are constructed from URLs using URLHandler.getRepositoryByURL (much like has been done in the above example), so the appropriate URLHandlers must be registered. Permis supports a number of URLHandlers, including handlers to deal with the protocols http, https and file type. A new URLHandler can be registered by calling URLHandler.addProtocol(u), where u is an instance of the URLHandler class. An instance of URLHandler can be created as

URLHandler u = new URLHandler(<string name of the protocol (without trailing ":")>, <the number of the default port the protocol listens on>)

Usage:

```
PermisRBAC engine = null;
PolicyFinder pf = null;

try{
  pf = new SimplePERMISPolicyFinder(new File("path_to_file.xml"),principal);
} catch(java.io.IOException e){
  e.printStackTrace();
}

try{
  pbaapi= new PermisRBAC(pf, "ldap://sec.cs.kent.ac.uk/c=gb", null);
} catch(issrg.pba.PbaException pba ){
  pba.printStackTrace();
}
```

The fourth constructor allows us to initialise a decision engine that retrieves user credentials from multiple repository sources. This constructor takes three inputs; a PolicyFinder object, a String array containing the URLs of multiple repositories and an AuthzTokenParser object. This constructor is identical to the previous constructor except in the fact that instead of taking one String object containing a URL it takes a String Array containing multiple URLs. The Attribute Repositories are constructed from URLs using URLHandler.getRepositoryByURL, so the appropriate URLHandlers must be registered.

Usage:

```
PermisRBAC engine = null;
PolicyFinder pf = null;
String[] urls = new String[] {"ldap://sec.cs.kent.ac.uk/c=gb",
                              "ldap://issrg-beta.cs.kent.ac.uk/c=gb"};

try{
  pf = new SimplePERMISPolicyFinder(new File("path_to_file.xml"),principal);
} catch(java.io.IOException e){
  e.printStackTrace();
}

try{
  engine = new PermisRBAC(pf, urls, null);
} catch(issrg.pba.PbaException pba ) {
  pba.printStackTrace();
}
```

3.1 Using the PermisRBAC object

In order to make a decision the caller must first obtain the user's credentials. This can be done by calling any of the range of getCreds methods, providing both Push and Pull models. If Pull model is used, the user authorisation tokens are retrieved from the repository, specified in PERMIS RBAC constructor or policy. After that a delegation chain is established, and a set of valid authorisation tokens is calculated by parsing the tokens and (optionally) verifying signatures on them. The PermisRBAC object uses Role attributes from the authorisation tokens, for which allowed role type OID and range of values are specified in the Policy. Thus the valid credentials for the user are calculated and stored in the Subject object.

Now the Subject can be used many times by the PermisRBAC to make an authorisation decision. This depends on AEF or PEP and is appropriate before the credentials become outdated and need to be refreshed. Note also that a Subject object can only be used by the PermisRBAC object that created it. It cannot be used by other PermisRBAC objects, even if they are initialised with the same Policy OID: to check this the object ascertains if the owner of the Subject object is itself and throws an exception at decision time if it is not. This is done to ensure that the credentials of the Subject are up to date and are not forged or confused with credentials obtained from another PermisRBAC instance.

3.2 Credential Validation Stage: getCreds()

It is possible to use the PermisRBAC object solely as a credential validation service. If one wants to use the PermisRBAC object to make authorisation decisions then the credential validation step is still a prerequisitie. The CVS functionality is provided through a range of (overloaded) methods all called getCreds.

There are six different implementations of the getCreds() method and the one to use depends on how user credentials are presented to the decision engine. These six methods can be further divided into push and pull methods.

If credentials are always pulled by the decision engine from a predefined Attribute repository, e.g. an LDAP server then the two pull methods getCreds(`java.security.Principal subjectDN`) and getCreds(`java.security.Principal subjectDN, CredentialsService service`) should be used. The first of these takes a java security principal object that represents the user requesting a service and retrieves his attributes from a repository, parses them for valid roles and returns a Subject object representing them. The second does the same as `getCreds(Principal)`, but in addition sets a service specified by the AEF; for example session time, after which the credentials will be automatically expired.

If user authorisation tokens are always collected by the PEP before getCreds() is called then these tokens should be pushed to getCreds which will then parse and remove roles that are not recognized by the policy. There are two push getCreds() methods getCreds(java.security.Principal subjectDN, java.lang.Object[] creds) and getCreds(java.security.Principal subjectDN, java.lang.Object[] creds, CredentialsService service). The first of these returns the credentials of the subject whose Principal and Authorisation Tokens are supplied and the second returns the credentials of the subject whose Principal and Authorisation Tokens are supplied and sets a service specified by the AEF; for example session time, after which the credentials will be automatically expired.
`The simplest way to push in the credentials is to use the MultiAuthzTokenParser. In Section 2.2.4, there are examples on how to use the MultiAuthzTokenParser to push credentials.` Credentials can also be pulled from a set of repositories referenced by a String Array of repository urls using the getCreds(java.security.Principal subject, java.lang.String[] urls, AuthzTokenParser authParser) method. This method doesn't check whether all of the repositories are connectable. If they are not, a partial or even empty set of credentials may be returned.

3.3 Making an Authorisation Decision: authzDecision()

Getting an authorisation decision is done through the authzDecision method which together with the authorisation decision itself also returns any obligations associated with this decision. More specifically, it returns a Response object that represents both the authorisation decision and the obligations that should be observed before the decision is valid. This method takes four parameters; a Subject object which describes the subject wishing to carry out the action on the target (provided by getCreds), an Action object that represents the action that the subject wishes to perform, a Target object that represents the object on which the subject wishes to act and a java.util.Map object containing a list of application contextual parameters. The java.util.Map object can be null, which indicates no contextual parameters are applicable.

3.4 Example: Making a Decision after Pulling Credentials

The example below shows how to make a decision after pulling credentials from predefined repositories. In the example below, the environment is never used.

```
//Return a string describing the result of the authorisation decision
public String execute(Principal user, Action action, Target target) {
 Response resp = null;
 try {
   Subject s = pbaApi.getCreds(user); /* pbaApi is the PermisRBAC object */
   resp = pbaApi.authzDecision(s, action, target, null);
   if (!resp.isAuthorised()){
    return "1: the action is not allowed";
   }
 } catch (issrg.pba.PbaException pe) {
     return "2: invalid input: " + pe.getMessage();
 } catch (Throwable th) {
     return "3: run-time error: " + th.getMessage();
 }
 String res = "0: action succeeded";
 if (resp.getObligations() != null && resp.getObligations().size() > 0) {
   res += "\nThere are obligations associated with the response";
 }
 return res;
}
```

3.5 Example: Making a Decision after Pushing Credentials

```
public String execute(Principal user, Object []creds,
                      Action action, Target target) {
 Subject s = null;
 try {
  s = pbaApi.getCreds(user,creds);
  Response resp = pbaApi.authzDecision(s, action, target, null);
  if (!resp.isAuthorised()){
   return "1: the action is not allowed";
  }
 } catch (issrg.pba.PbaException pe){
   return "2: invalid input: " + pe.getMessage();
 } catch (Throwable th){
   return "3: run-time error: " + th.getMessage();
 }
 return "0: action succeeded";
}
```

4. Example Classes

We provide some complete examples which are provided as a separate package (zip file). This zip file contains different example classes (starting from simplest possible implementation towards more complex one) along with all required libraries and resources. The examples can be downloaded from this URL: http://sec.cs.kent.ac.uk/permis/documents/relatedDoc.shtml . You will need to register and agree to the terms of use in order to obtain the sample classes, as the package includes the iaik_jce library.

4.1 Setting up and running the example classes

1. Unzip the examples.zip into a directory of your choice.
2. Make sure the directory structure is as follows:

```
Your JAVA Project
    examples
       lib
       resources
       src
       build.xml
```

   Now from the command prompt, change directory to examples.

3. Run the following command from the command prompt to compile all example sources:

```
ant compile
```

4. Execute the following command to run any example class (e. g. XMLPolicyHD)

```
ant XMLPolicyHD
```

Here, XMLPolicyHD is the name of an example class. Change it to other example classes to run them.

If the output is as follows, then everything is working fine.

```
0: action succeeded
```

The above output indicates that the user is authorised to access the resource using the policy and the AC.

For more complex examples (e.g. RepositoryAcWithSV.java) in which both the policy and the AC are retrieved from an LDAP server, the following steps should be performed before running the example class.

1. Use the PolicyEditor (or another LDAP client) to add an X.509AC containing the xml policy in to AA's LDAP entry. For the example programs the policy issuer is hardcoded to be “cn=A PERMIS Test User,o=permisv5,c=gb”, so the policy should be put in this LDAP entry. The PolicyEditor can be downloaded from: http://sec.cs.kent.ac.uk/permis/downloads/Level1/policy.shtml.

2. Use the Attribute Certificate Manager (ACM) to add the X.509 AC to the subject's LDAP entry. For the example program, the subject requesting the action is hardcoded as “cn=User4,o=permisv5,c=gb”, so the X.509 AC should be put in this entry. The ACM can be downloaded from: http://sec.cs.kent.ac.uk/permis/downloads/Level1/acm.shtml.

XMLPolicyHD.java represents a very simple application of the PERMIS API and loads the policy from an XML file located on the local hard drive. The credential (X.509 AC) is also loaded from the hard drive.

```java
package issrg.examples;

import issrg.pba.Action;
import issrg.pba.PbaException;
import issrg.pba.Response;
import issrg.pba.Subject;
import issrg.pba.Target;
import issrg.pba.rbac.BadURLException;
import issrg.pba.rbac.CustomisePERMIS;
import issrg.pba.rbac.LDAPDNPrincipal;
import issrg.pba.rbac.PermisAction;
import issrg.pba.rbac.PermisRBAC;
import issrg.pba.rbac.PermisTarget;
import issrg.pba.rbac.PolicyFinder;
import issrg.simplePERMIS.SimplePERMISPolicyFinder;
import issrg.utils.RFC2253ParsingException;
import issrg.utils.repository.AttributeRepository;
import issrg.utils.repository.FileRepository;
import issrg.utils.repository.RepositoryException;

import java.io.File;
import java.io.IOException;
import java.security.Principal;

/**
 * The simplest possible example class for the PERMIS API. This class loads the
 * policy from an XML File on hard drive. The credential (X.509 AC) is also
 * loaded from the hard drive.
 *
 * @author sadek
 */
public final class XMLPolicyHD {
  private static PermisRBAC pbaApi = null;

  private XMLPolicyHD() {}

  public static void main(String[] args) {
    PolicyFinder finder = null;
    Principal prin = null;
    AttributeRepository fileRepository = null;

    /**
     *creates a LDAP DN principal object for the user who signed the policy
     * This should be set by the application not hard-coded or the Same user
will
     * always be used.
     */
    try {
      prin = new LDAPDNPrincipal("cn=A PERMIS Test User,o=permisv5,c=gb");
    } catch (RFC2253ParsingException e1) {
      e1.printStackTrace();
    }

    /**
     * Loads the required AC (.ace) from the hard drive
     */
    try {
      fileRepository = new FileRepository(
"file://examples/resources/issrg/examples/User.ace?attributeCertificateAttribute
;binary=^.*.ace$&userCertificate;binary=^.*.cer$");
    } catch (RepositoryException e1) {
      e1.printStackTrace();
    }
    // initialises and creates the decision engine
    try {
      // Sets the LDAP attribute names

CustomisePERMIS.setAttributeCertificateAttribute("attributeCertificateAttribute"
);
      CustomisePERMIS.setUserCertificateAttribute("userCertificateAttribute");
      // Sets the AuthTokenParser to allow PERMIS to use X509 //certificates
      try {

CustomisePERMIS.setAuthzTokenParser("issrg.pba.rbac.x509.RoleBasedACParser");
      } catch (ClassNotFoundException e2) {
        e2.printStackTrace();
      }

      // loads the policy from the hard drive
      try {
        finder = new SimplePERMISPolicyFinder(new
File("examples/resources/issrg/examples/policy.xml"),
            prin);
      } catch (IOException ie) {
        ie.printStackTrace();
      }

      /**
       * creates the decision engine object in pull mode using the Policy finder
       * and any decisions will use the repository object fileRepository
       */
      pbaApi = new PermisRBAC(finder, fileRepository, null);
    } catch (PbaException e) {
      // Catches any creation errors
      e.printStackTrace();
    }

    // A new principal Object for the user making the decision, again
    // should not be hard-coded in production builds
    Principal user = null;
    try {
      user = new LDAPDNPrincipal("cn=User4,o=permisv5,c=gb");
    } catch (RFC2253ParsingException ex) {
      ex.printStackTrace();
    }
    // creates an action object that specifies what the user wants to do
    Action action = new PermisAction("GET");

    // The target object specifies where the user wants to do the action
    // defined above
    Target target = null;
    try {
      target = new PermisTarget("http://localhost/secure2/index.html");
    } catch (BadURLException be) {
      be.printStackTrace();
    }

    // make a decision using the execute method
    // and print out the result to the console
    System.out.println(execute(user, action, target));

  }

  /*
   * this method calls the decision engine for a decision using the parameters
   * passed to it
```

```
    */
  private static String execute(java.security.Principal user, Action action,
Target target) {
    try {
      // calls getCreds for a subject object containing
      // the principal's valid roles pulled from local AC.
      Subject s = pbaApi.getCreds(user);

      // call permis for the decision
      Response resp = pbaApi.authzDecision(s, action, target, null);
      // see if the decision failed
      if (!resp.isAuthorised()) {
        return "1: the action is not allowed";
      }
      // catch potential errors
    } catch (issrg.pba.PbaException pe) {
      // if one of the parameters did not match the policy return
      // an error message that states which one and why
      return "2: invalid input: " + pe.getMessage();
    } catch (Throwable th) {
      // any other errors unrelated to the policy
      // throw a different error
      return "3: run-time error: " + th.getMessage();
    }
    // The user is authorised to use the resource; return OK
    return "0: action succeeded";
  }
}
```

5. Other PERMIS Packages and Classes

5.1 Repositories (issrg.utils.repository)

The Permis API currently has number of different types of repositories implemented, e.g. LDAP server, file hash, MultiRepository, WebDav server and Virtual repositories. Each of these implements the AttributeRepository interface, this means that they can all be used interchangeably within your code as long as they are defined as AttributeRepository objects rather than individual repository types. Each repository type can hold policies, roles and public key certificates in a variety of formats. As PERMIS offers the same interface to each of its repositories a basic guide to using the repository classes is provided here. The issrg.utils.repository.AttributeRepository interface defines five methods for accessing repositories:

1. The getAllAttributes(java.security.Principal) method returns all of a specified users attributes as an instance of the javax.naming.directory.Attributes interface.
2. The getAttributes(java.security.Principal dn,java.lang.String attributeName) method returns only the entries for the attribute type specified in the request and is used when the format of credentials is known before the request.
3. If multiple types of credentials are required and known at the time of the request then the method getAttributes(java.security.Principal dn, java.lang.String[] attributeNames) should be used. The previous method is equivalent to calling this one with an array of length one.
4. The getStatus() method returns a status response and can be used to ensure that a repository object is available for use.
5. The final method getDiagnosis() can be used to return the last exception thrown.

5.1.1 issrg.utils.repository.FileRepository

This repository uses File URLs and the hash of the certificates DN to find attributes in an entry. The File URLs should be of the form:

```
file://<path>[?<attribute type>=<file name mask>[&...]...]
```

The path should normally point to the directory where the files are. However, it may point to a file, if a single file has to be used. (This may be useful when loading a Policy ACIn this case it is noit required that the filename contains the hash of the distinguished name of the holder. The path may be absolute or relative; in the latter case the path is computed from the current directory of the application.

The query part of the URL tells the FileRepository the mapping between the attribute types and file name mask (a regular expression). You should specify at least one attribute type mapping (a URL is meaningless otherwise - no attributes can be retrieved). If the attribute type doesn't have the ";binary" suffix, the file is treated as a text file with one value in it, and it is returned as a string. Otherwise, a byte[] is the attribute value.

Example:

```
file:///c:/certificates/?attributeCertificateAttribute;binary=^.*\.ace$&userCert
ificate;binary=^.*\.cer$&eduPerson=^.*\.edu$
```

This URL will tell the File Repository to look for attributeCertificateAttribute;binary in *.ace files, and look for userCertificate;binary attributes in *.cer files in the c:/certificates/ subdirectory.

In order to use this repository with PERMIS you should first create the folder containing the hashed DN certificates. The DN hashing of certificates can be performed using the openssl command.

```
openssl x509 -noout -hash -in cert.pem (e.g. cert.pem is an AC or a PKC)
```

which will output the hash value. The DN hash value has to be a part of the file name. An alternative to the openssl command is to use the utility class issrg.utils.mains.Hash. Run this class without arguments on the command line to see how it should be called. Once the physical repository has been set up an issrg.utils.repository.FileRepository object should be created by passing the repository url as defined above to the constructor which will set up the repository for use with PERMIS.

5.1.2 issrg.utils.repository.LDAPRepository

This class is the implementation of the Attribute Repository for LDAP. It can be built out of an array of DirContext. Each DirContext object constitutes a root for LDAP searches and is obtained by establishing an LDAP connection with the root directory concerned.

The repository object can be used for retrieving similar information from multiple directories simultaneously. For example, this can be seen to be useful when retrieving X.509 Attribute Certificates for PMI entities that possess ACs issued by different issuers.

In fact, the repository object uses the MultiRepository object to create multiple threads, and acts as a proxy object for backwards compatibility (earlier versions of this object had a constructor with an array of DirContext). *It is better to use a MultiRepository object for multi-root clusters of LDAP repositories* rather than a LDAPRepository object but it is possible.

This repository has two publicly accessible constructors. The first LDAPRepository(javax.naming.directory.DirContext Context) allows us to create a repository using a DirContext object and the second LDAPRepository(javax.naming.directory.DirContext[] Contexts) allows us to crteate a

MultiRepository object using an array of DirContext objects. In actual practice we generally wish to create our repository using only the repositories URL for which there is not a easily defined constructor, however by statically calling the issrg.pba.rbac.URLHandler.getRepositoryByURL(String url) method the ldap protocols url handler object will create the repository object and return it as a AttributeRepository object. This is the easiest way of constructing an LDAP repository. Note that a LDAP URL handler has to be registered before calling this method.

5.1.3 issrg.utils.repository.MultiRepository

This class is the implementation of the Attribute Repository for multithreaded access to a cluster of repositories. It can be built out of an array of Repositories. Each of these repositories constitutes a root for searches.

The object can be used for retrieving similar information from multiple directories simultaneously. For example, it is useful when retrieving X.509 Attribute Certificates for PMI entities that possess ACs issued by different issuers (therefore, stored in different repositories available to them).

The object creates multiple threads (one per repository) when attributes are requested and performs the searches simultaneously. This improves efficiency, since most of the time the repositories are waiting for a reply. The object waits till all of the repositories return anything or report an error, so the result is always complete.

This class is used when multiple repositories are required. Whilst the LDAP repository object supports the usage of multiple LDAP repositories, this class allows us to use multiple repositories of different types. In order to construct this repository each individual Attribute Repository to be included must have been initialised previously and then passed to one of the classe’s two (static) factory methods. The first factory method takes a single AttributeRepository object and is used when only one repository is required; otherwise the second factory method which takes an array of AttributeRepository objects is used. Once the repository has been initialised it allows us to search and query multiple repository objects at the same time for a single user’s attributes.

The following example is to create a MultiRepository object, named multi, which includes two repositories: a virtual repository and a LDAP repository.

```
AttributeRepository[] arr = new AttributeRepository [2];
arr [0] = new VirtualRepository();
arr [1]  =
issrg.pba.rbac.URLHandler.getRepositoryByURL(“ldap://sec.cs.kent.ac.uk/c=gb”);
AttributeRepository multi = MultiRepository.getMultiRepository(arr);
```

5.1.4 issrg.utils.repository.VirtualRepository

This class is the implementation of the AttributeRepository for use with local files loaded at run time by PERMIS. It represents a repository stored in memory that can be produced at runtime and then discarded once a session has ended.

The virtual repository class offers a single constructor to takes no arguments that is used to create an empty shell that must then be populated with attributes

The virtual repository class differs from all the other repository classes in that it must be populated with attributes after construction and as such contains an extra method called populate that takes three arguments: a String containing the LDAP DN of the user entry, a String containing the attribute type of the credential and a java object containing the credential.

Example:.

```
byte [] attributeCertificateAttribute = /* load byte array here */
```

```
VirtualRepository vr = new VirtualRepository();
vr.populate("cn=user0,o=permis,c=gb", "attributeCertificateAttribute", 
attributeCertificate);
AttributeRepository r = vr;
```

Once the repository has been populated it can be passed to PERMIS to be used to pull credentials. Please note that once the repository has been passed to PERMIS any changes in contents will not be reflected in decision responses.

5.1.5 Issrg.utils.repository.WebDavRepository

This class is an implementation of the AttributeRepository interface for use with a WebDav server.

In order to initialize this repository type there are two constructors. The first should be used if the WebDav server is publicly accessible over HTTP and takes two inputs: a String containing the host url and an integer containing the server's port number. If the WebDav server is contacted via a SSL connection then the second method should be used it takes 4 inputs; a String containing the host url, an integer containing the server's port number, a String containing a path to a local p12 file to be used for authentication and the password to this p12 file. The p12 file holds a key pair for authenticating the WebDAV server to PERMIS CVS.

e.g.

Foe non-SSL connection

```
AttributeRepository r = new WebDavRepository("http://issrg-beta.cs.kent.ac.uk",80);
```

For SSL connection

```
AttributeRepository r = new WebDavRepository("https://issrg-beta.cs.kent.ac.uk", 
433, "C:\p12example.p12", "pass");
```

5.2 Subjects, Actions, Targets and Environment

Permis makes its authorisation decisions based on the subject , action, target and environment objects passed to it. These four objects define the authorisation request. The Subject contains the credentials of the requesting subject; the action states what the user wants to do on the target; the target tells the location of it; and optionally the environment supplies some variables (e.g. the current time and date) to support his request. The decision engine then checks against the policy whether the target is mentioned in the policy and if so whether the user has the credentials to perform the requested action/s on the target and whether the variables satisfies the policy condition (e.g. the current time is before 12:00 pm).

5.2.1 Subjects

Subject objects should implement the issrg.pba.Subject interface and in most cases the issrg.pba.rbac.PermisSubject class will be used. We would recommend that this class should not be constructed directly. It will be created by the CVS when one of the PermisRBAC classes getCreds() methods is called.

5.2.2 Actions

Action objects should implement the issrg.pba.Action interface and in most cases the issrg.pba.rbac.PermisAction class will be sufficient. This class takes either a String value representing an action or a String value representing an action and an array of Arguments as its constructor and once it has been constructed it is ready to be passed to the decision method.

Usage:

```
Action action = new PermisAction("GET");
Action action = new PermisAction("PUT", params);
```

where params is an array of Argument objects. Each Argument object contains a string type name (e.g. "Integer") and a string value (e.g. "19").

5.2.3 Targets

Target objects should implement the issrg.pba.Target interface and in most cases the issrg.pba.rbac.PermisTarget class will be sufficient. This class provides two constructors: one for URL targets and the other for LDAP DNs. Be sure not to confuse these two!

For URL targets the PermisTarget(java.lang.String url) constructor should be used. The String value should contain the url of the target. For example,

```
try{
  Target target = new PermisTarget("http://sec.cs.kent.ac.uk/permis");
} catch (BadURLException b) {
  b.printStackTrace();
}
```

For LDAP DNs the PermisTarget(java.lang.String dn, java.lang.String[] objectClasses) class should be used. The String object should contain the LDAP DN of the target being requested and the String[] object should contain an array of strings that are the various object classes that the Target has. These can be represented as user friendly strings e.g. "printer" or as an OID string e.g. 1.2.3.4.5.6. If the array is null, then the Target is of any class.

```
try{
      Target target = new PermisTarget("cn=target,o=permis,c=gb", null);
}catch (RFC2253ParsingException r){
      r.printStackTrace();
}
```

5.2.4 Environment

Environment object should implement the java.util.Map interface. A Map object is needed if policy conditions involve some environmental elements. A Map object consists of a set of entries. Each entry is identified by an environmental name and has a Java object value.

The current PERMIS can take Integer, Double and String objects as environmental values. The environmental name can be any string, which depends on what environmental names are used in policy conditions.

The following example illustrates how to create an environment object.

```
Map<Object, Object> env = new HashMap<Object, Object>();
env.put("time", new String("14:38:00"));
env.put("temperature", new Double("18.5"));
env.put("number_of_doors", new Integer("10"));
```

6. Descriptions of Important Classes

- AuthzTokenRepository objects are responsible for returning valid authorisation tokens. It calls AuthzTokenParser to check if a token retrieved from an AttributeRepository is an authentic parsed token, to see if it is issuer valid i.e. has been signed by the issuer.

- AuthzTokenParser objects are responsible for parsing the raw tokens and validating their signatures (technical validation, or authenticity check) using a SignatureVerifier module. AuthzTokenParser objects are passed raw binary tokens and output authentic parsed tokens. Different AuthzTokenParsers and signatureCheckers are needed for ACs, PKC, SAML assertions, Shibboleth attributes etc.

- PermisRBAC objects are the implementation of the PERMIS API. It mainly consists of the two methods: getCreds and authzDecision. The main responsibility of getCreds is to get valid credentials for the relying party according to the policy which the relying party has defined. It also makes an authorization decision for a holder based on the received credentials and the policy with the authzDecision method.

- AttributeRepository objects are repositories for retrieving authorization tokens from a selected store. The different storage methods can be hard drive, LDAP and WebDAV.

- Policy Finder Objects are responsible for parsing the raw XML policy files into a form that PERMIS can read. It separates the policy in a variety of different segment for use throughout Permis. There are several different policy finder objects that can be used to parse policies in a variety of formats and storage locations.

- Action Objects tell the PDP what action the user wishes to perform on a selected target.

- Target objects tell the PDP what entity the user wishes to perform an action upon.

- Subject Objects represent the user’s credentials as determined by the CVS and can tell the PDP who the user is, his set of credentials, and defines a Credential service which can be used to provide an additional service on the credentials. In PERMIS these are also linked to the specific CVS/PDP pair that produced them.

7. Descriptions of Permis Packages

This section gives a brief explanation of each of the PERMIS Java packages and what their contents do:

issrg.ac – this package contains the classes that are used to represent X509 Attribute Certificates and their attributes within PERMIS
issrg.acm- this package contains the classes that are used to build the Attribute Certificate Manager software. It contains classes, which represents a GUI for creating, saving and modifying X509 Attribute certificates for use with PERMIS.
issrg.aef – this package contains the classes for a simple testing application for PERMIS, called the sample aef. These classes are a good starting point for those learning to use the PERMIS API
issrg.akentipermis – this package contains classes that implement the time interfaces to PERMIS
issrg.dis – this package contains classes that are used to build and use the Delegation Issuing Service which is a web based service that implements delegation of authority

issrg.editor2 – this package contains the classes that are used to build and use the new graphical

policy editor application
issrg.globus – this package contains the classes that are used to build and use the GT3 globus Permis Authorisation services
issrg.gt4 - this package contains the classes that are used to build and use the GT4 globus Permis Authorisation services
issrg.ldapadmin – this package contains the source code for the LDAP bulk loader application
issrg.log – this package contains the custom log4j levels and appenders that can be used by PERMIS

issrg.pba - this package contains the PERMIS decision engine itself. issrg.pba itself mostly contains interface classes that are implemented by the classes in issrg.pba.rbac which contains the core PERMIS CVS and PDP.

issrg.policytester – this package contains the classes for the graphical policy tester application
issrg.policywizard – this package contains the classes for the graphical policy creation wizard, which is used by the policy editor application

issrg.saws- contains the classes for the secure audit web service
issrg.security – contains classes that are used to verify credentials such as PKC signature checkers
issrg.shibboleth – contains the classes used to construct the Java portion of an authorisation module for apache.
issrg.simplePERMIS – contains a set of classes that can be used in conjunction with the classes in issrg.pba to create a simplified version of Permis that uses plain text attributes and XML policies
issrg.test – contains a set of regression tests for Permis designed to be used by Permis developer to ensure the integrity of Permis after some new functionality is added into it. The regression test is build and run by taking advantage of ant script.
issrg.utils – contains a set of useful classes that are used throughout each permis application such as name parsers and repository objects.

issrg.xacml – This package contains an XACML interface to PERMIS that allows users to use PERMIS with Sun's XACML package.

7. Further Information

For additional information on Permis Java classes please refer to the Permis java documentation available on our website (http://sec.cs.kent.ac.uk/permis/documents/permis/javadoc/index.html). These can be used to find method and constructor information on all the permis interfaces, classes and their packages.

For additional information on the creation of Permis policies and Attribute Certificates please refer to the Installing and Configuring Permis document (http://sec.cs.kent.ac.uk/permis/downloads/Level2/decisionEngine.shtml) that provides a walkthrough for the creation of a simple PERMIS policy and role certificates.

If you wish to download the complete set of Java classes for PERMIS please refer to the OpenPERMIS website www.openpermis.com that provides several download options of the latest PERMIS distribution.