Implementing Role Based Access Controls using X.509 Privilege Management - the PERMIS Authorisation Infrastructure

David W Chadwick, Alexander Otenko
IS Institute, University of Salford, M5 4WT, England
Email: d.w.chadwick@salford.ac.uk o.otenko@salford.ac.uk
Telephone: +44 161 295 5351
Fax: +44 161 745 8169

Abstract

This paper describes the PERMIS role based access control infrastructure that uses X.509 attribute certificates (ACs) to store the users' roles. Users' roles can be assigned by multiple widely distributed management authorities (called Attribute Authorities in X.509), thereby easing the burden of management. All the ACs can be stored in one or more LDAP directories, thus making them widely available. The PERMIS distribution includes a Privilege Allocator GUI tool, and a bulk loader tool, that allow administrators to construct and sign ACs and store them in an LDAP directory ready for use by the PERMIS decision engine. All access control decisions are driven by an authorization policy, which is itself stored in an X.509 AC, thus guaranteeing its integrity and trustworthiness. Authorization policies are written in XML according to a DTD that has been published at XML.org. A user friendly policy management tool is also being built that will allow non-technical managers to easily specify PERMIS authorisation policies. The access control decision engine is written in Java and has both a Java API and SAML-SOAP interface, allowing it to be called either locally or remotely. The Java API is simple to use, comprising of just 3 methods and a constructor. The SAML-SOAP interface conforms to the OASIS SAMLv1.1 specification, as profiled by a Global Grid Forum draft standard, thus making PERMIS suitable as an authorisation server for Grid applications.

Keywords

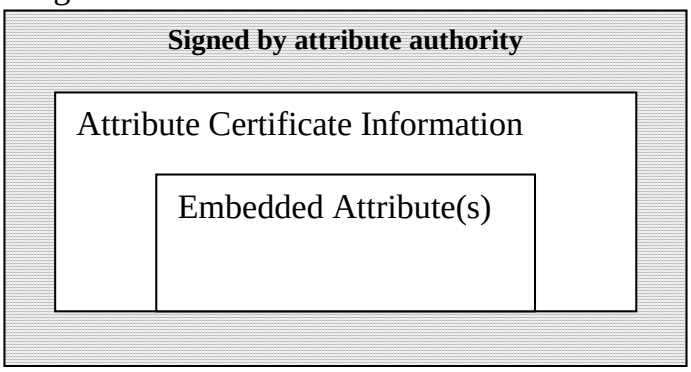
Trust Management, X.509, Attribute Certificates, Role Based Access Controls, XML, Privilege Management Infrastructure, SAML.

Introduction to X.509 PMIs

Most people familiar with PKIs will know that the standard format for a public key certificate is specified in the X.509 standard [14], published jointly by the ITU-T and ISO/IEC. The primary purpose of a PKI is to strongly authenticate the parties communicating with each other, though the use of digital signatures. But strong authentication on its own is insufficient for a process to determine who is allowed to do what. An authorisation mechanism is needed for this. Edition 4 of X.509 is the first edition of X.509 to fully standardise a strong authorisation mechanism, which it calls a Privilege Management Infrastructure (PMI). PMIs provide the authorisation function after the authentication has taken place, and have a number of similarities with PKIs. This paper assumes the reader is already familiar with the general concepts of PKIs, and these will not be repeated here. Readers wishing to learn more about PKIs may consult texts such as [1] or [12].

The primary data structure in a PMI is an X.509 Attribute Certificate (AC) (see Figure 1.) This strongly binds a set of attributes to its holder, and these attributes are used to describe the various privileges of the holder bestowed on it by the issuer. The issuer is termed an Attribute Authority (AA), since it is the authoritative provider of the attributes given to the holder. Examples of attributes and issuers might be: a degree awarded by a university, an ISO 9000 certificate issued by a QA compliance organisation, the role of supervisor issued by a manager, file access permissions issued by a file's owner. The whole data construct is digitally signed by the AA, thereby providing data integrity and authentication of the issuer.

Figure 1 Attribute Certificate



The attributes are embedded within the Attribute Certificate Information data construct (see Figure 2). This contains details of the holder, the issuer, the algorithms used in creating the signature on the AC, the AC validity time and various optional extensions. Anyone familiar with the contents of an X.509 public key certificate (PKC) will immediately see the similarities between a PKC and an AC. In essence the public key of a PKC has been replaced by a set of attributes. (In this respect a public key certificate can be seen to be a specialisation of a more general attribute certificate.) Because the AC is digitally signed by the issuer, then any process in possession of an AC can check its integrity by checking the digital signature on the AC. Thus a PMI builds upon and complements existing PKIs.

Figure 2 Attribute Certificate Info

```
Attribute Certificate Info ::= SEQUENCE {
        version                 AttCertVersion,
        holder                  Holder,
        issuer                  Issuer,
        signature               AlgorithmIdentifier,
        attrCertValidityPeriod  AttCertValidityPeriod,
        attributes              SEQUENCE of Attribute,
        issuerUniqueID          UniqueIdentifier  OPTIONAL,
        extensions              Extensions OPTIONAL }
```

Since a PMI is to authorisation what a PKI is to authentication, there are many other similar concepts between PKIs and PMIs. Whilst public key certificates are used to maintain a strong binding between a user's name and his public key, an attribute certificate (AC) maintains a strong binding between a user's name and one or more privilege attributes. The entity that digitally signs a public key certificate is called a Certification Authority (CA), whilst the entity that signs an attribute certificate is called an Attribute Authority (AA). Within a PKI, each relying party must have one or more roots of trust. These are CAs who the relying party implicitly trusts to

authenticate other entities. They are sometimes called root CAs[1] or trust anchors. Popular Web browsers come pre-configured with over 50 PKI roots of trust. The root of trust of a PMI is called the Source of Authority (SOA). This is an entity that a resource implicity trusts to allocate privileges and access rights to it. The SOA is ultimately responsible for issuing ACs to trusted holders, and these can be either end users or subordinate AAs. Just as CAs may have subordinate CAs to which they delegate the powers of authentication and certification, similarly, SOAs may have subordinate AAs to which they delegate their powers of authorisation and attribute assigning. For example, in an organisation the Finance Director might be the SOA for allocating the privilege of spending company money. But (s)he might also delegate this privilege to departmental managers (subordinate AAs) who can then allocate specific spending privileges (ACs) to project leaders.

When a problem occurs in a PKI, a user might need to have his signing key revoked, and so a CA might issue a certificate revocation list (CRL) containing the list of PKCs no longer to be trusted. Similarly if a PMI user needs to have his authorisation permissions revoked, an AA may issue an attribute certificate revocation list (ACRL) containing the list of ACs no longer to be trusted. The similarities between PKIs and PMIs are summarised in Table 1.

Concept	**PKI entity**	**PMI entity**
Certificate	Public Key Certificate (PKC)	Attribute Certificate (AC)
Certificate issuer	Certification Authority (CA)	Attribute Authority (AA)
Certificate user	Subject	Holder
Certificate binding	Subject's Name to Public Key	Holder's Name to Privilege Attribute(s)
Revocation	Certificate Revocation List (CRL)	Attribute Certificate Revocation List (ACRL)
Root of trust	Root Certification Authority or Trust Anchor	Source of Authority (SOA)
Subordinate authority	Subordinate Certification Authority	Attribute Authority (AA)

Table 1. A Comparison of PKIs and PMIs

Further details about X.509 PMIs can be found in [17].

Implementing RBAC with X.509

Recent research has focussed on Role Based Access Controls (RBAC) [22] [11], culminating in the publication of RBAC as an American National Standard (ANSI INCITS 359-2004) in February 2004. In the basic RBAC model, a number of roles are defined. These roles typically represent organisational roles such as secretary, manager, employee etc. In the authorisation policy, each role is given a set of permissions i.e. the ability to perform certain actions on certain targets. Each user is then assigned to one or more roles. When accessing a target, a user presents his role(s), and the target reads the policy to see if this role is allowed to perform this action. RBAC has the advantage of scalability over traditional discretionary access

[1] Unfortunately early versions of X.509 did not standardise the term root CA or any term for the root of trust. Because of this, disparate meanings for the term "root CA" have now evolved.

control (DAC) schemes such as access control lists (ACLs), and can easily handle large numbers of users as there are typically far fewer roles than users.

X.509 supports simple RBAC by defining role specification attribute certificates that hold the permissions granted to each role, and role assignment attribute certificates that assign various roles to the users. In the former case, the AC holder is the role, and the privilege attributes are permissions granted to the role. In the latter case the AC holder is the user, and the privilege attributes are the roles assigned to the user.

The hierarchical RBAC model is a more sophisticated version of the basic RBAC model. With this model, the roles are organised hierarchically, and the senior roles inherit the privileges of the more junior roles. So for example we might have the following hierarchy:

employee > programmer > manager > director.

If a privilege is given to an employee role e.g. can enter main building, then each of the superior roles can also enter the main building even though their role specification does not explicitly state this. If a programmer is given permission to enter the computer building, then managers and directors would also inherit this permission. Hierarchical roles mean that role specifications are more compact. X.509 supports hierarchical RBAC by allowing both roles and privileges to be inserted as attributes in a role specification attribute certificate, so that the latter role inherits the privileges of the encapsulated roles.

Another extension to basic RBAC is constrained RBAC. This allows various constraints to be applied to the role and permission assignments. One common constraint is that certain roles are declared to be mutually exclusive, meaning that the same person cannot simultaneously hold more than one role from the mutually exclusive set. For example, the roles of student and examiner, or the roles of tenderer (one who submits a tender) and tender officer (one who opens submitted tenders) would both be examples of mutually exclusive sets. Another constraint might be placed on the number of roles a person can hold, or the number of people who can hold a particular role. X.509 only has a limited number of ways of supporting constrained RBAC. Time constraints can be placed on the validity period of a role assignment attribute certificate. Constraints can be placed on the targets at which a role can be used, and on the policies under which an attribute certificate can confer privileges. Constraints can also be placed on the delegation of roles. However many of the constraints, such as the mutual exclusivity of roles, have to be enforced by mechanisms outside the attribute certificate construct e.g. within the policy enforcement function.

Of course, every target that relies on X.509 ACs to confer privileges, also needs to be configured with a policy, or a set of rules, that will tell it which access methods are to be granted by which privileges. Unfortunately X.509 does not standardise any type of policy and leaves this up to the applications using the X.509 PMI. This is one of the biggest challenges for anyone deciding to build an X.509 PMI.

The PERMIS PMI Architecture

The PERMIS PMI is, according to Blaze's definition in RFC 2704 [3], a trust management system. Consequently it must have the following five components:

i) A language for describing `actions', which are operations with security consequences that are to be controlled by the system.
ii) A mechanism for identifying `principals', which are entities that can be authorized to perform actions.
iii) A language for specifying application `policies', which govern the actions that principals are authorized to perform.
iv) A language for specifying `credentials', which allow principals to delegate authorization to other principals.
v) A `compliance checker', which provides a service to applications for determining how an action requested by principals should be handled, given a policy and a set of credentials.

X.509 attribute certificates specify mechanisms for ii) and iv). Principals are the holders and issuers of ACs and can be identified by their X.500 General Name (usually an X.500/LDAP distinguished name (DN) [16], IP address, URI or email address) or by reference to their public key certificate (issuer name and serial number) or if the principal is a software object by a hash of itself. Credentials are specified as X.500 attributes, which comprise an attribute type and value. Defining the policy (iii) and action (i) languages were significant tasks of the PERMIS project [5], as was building a privilege allocation subsystem that creates X.509 ACs, and a compliance checker (v) that validates them.

Defining the Policy Language

The authorisation policy needs to specify who is to be granted which roles, and what types of action on which targets are to be granted to the roles (optionally under which conditions). Domain wide policy authorisation is far more preferable than having separate access control lists configured into each target. The latter is hard to manage, duplicates the effort of the administrators (since the task has to be repeated for each target), and is less secure since it is very difficult to keep track of which access rights any particular user has across the whole domain. Policy based authorisation on the other hand allows the domain administrator (the SOA) to specify the authorisation policy for the whole domain, and all targets will then be controlled by the same set of rules.

Significant research has already taken place in defining authorisation policy languages. The Ponder language [7] is very compact and very powerful, but does not have a large set of supporting tools. The Keynote policy language [3] is also very comprehensive and covers many of our requirements, but is focussed on DAC rather than RBAC. Also, the policy assertions are very generic and are not related specifically to X.509. For example, the authoriser and licensees fields are opaque strings whereas we wanted them to have structure and meaning. Further, it does not seem to be possible to control the depth of delegation allowed from one authoriser to a subordinate. Finally, the syntax, comprising of ASCII strings and keywords, is specific to Keynote. The PERMIS project wanted to specify the authorisation policy in a well-known language that could be both easily parsed by computers, and read by the SOAs (with or without software tools). We decided that XML was a good candidate for a policy specification language, since there are lots of tools around that support XML, it is fast becoming an industry standard, and raw XML can be read and understood by many technical people. Shortly after we started our work, Bertino et al published a paper [2] that showed that XML was indeed suitable for specifying authorisation policies. Later, the OASIS consortium began work on the eXtensible

Access Control Markup Language (XACML) [18], and this is now an OASIS standard for specifying policies.

We needed a language tailored to X.509 and RBAC, so we specified a Data Type Definition (DTD)[2] for our X.500 PMI RBAC Policy. The DTD is a meta-language that holds the rules for creating the XML policies. Our DTD comprises the following components:

- RBAC Policy – this specifies the unique number for the policy, using a globally unique Object Identifiers (OID)
- SubjectPolicy – this specifies the subject domains i.e. only users from a subject domain may be authorised to access resources covered by the policy
- RoleHierarchyPolicy – this specifies the different roles and their hierarchical relationships to each other
- SOAPolicy – this specifies which SOAs are trusted to allocate roles. By including more than one SOA in this policy, the local SOA is effectively cross certifying remote authorisation domains
- RoleAssignmentPolicy – this specifies which roles may be allocated to which subjects by which SOAs, whether delegation of roles may take place or not, and how long the roles may be assigned for
- TargetPolicy – this specifies the target domains covered by this policy
- ActionPolicy – this specifies the actions (or methods) supported by the targets, along with the parameters that should be passed along with each action e.g. action Open with parameter Filename
- TargetAccessPolicy – this specifies which roles have permission to perform which actions on which targets, and under which conditions. Conditions are specified using Boolean logic and might contain constraints such as “IF time is GT 9am AND time is LT 5pm OR IF Calling IP address is a subset of 125.67.x.x”. All actions that are not specified in a Target Access Policy are denied.

A full description of the PERMIS X.500 PMI RBAC policy can be found in [4]. The policy DTD has been published by XML.ORG (http://www.xml.org) and is also available from our web site at http://sec.isi.salford.ac.uk/permis/Policy.dtd.

[2] We used a DTD rather than an XML schema, since XML schemas were still under development when we started our work, and few tools were available to support schemas. We are currently migrating towards the use of an XML schema.

```
<!ELEMENT RoleAssignmentPolicy (RoleAssignment)+ >
<!ELEMENT RoleAssignment  (SubjectDomain,Role,Delegate,SOA,Validity) >

<!ELEMENT SubjectDomain EMPTY>
<!ATTLIST SubjectDomain ID IDREF #REQUIRED>

<!ELEMENT Role EMPTY >
<!ATTLIST Role Type IDREF #IMPLIED
          Value IDREF #IMPLIED >

<!ELEMENT SOA EMPTY>
<!ATTLIST SOA ID IDREF #REQUIRED>

<!ELEMENT Validity (Absolute?, Maximum?, Minimum? ) >
<!ELEMENT Absolute EMPTY>
<!ATTLIST Absolute Start CDATA #IMPLIED
             End CDATA #IMPLIED >
<!ELEMENT Maximum EMPTY>
<!ATTLIST Maximum Time CDATA #IMPLIED >
<!ELEMENT Minimum EMPTY>
<!ATTLIST Minimum Time CDATA #IMPLIED >

<!ELEMENT Delegate EMPTY >
<!ATTLIST Delegate Depth CDATA #IMPLIED >
```

Table 2. The Role Assignment DTD

Table 2 shows a portion of the DTD that specifies the rules for the Role Assignment Policy, and Table 3 it is an example Role Assignment Policy for an electronic tendering application built by Salford City Council.

```
<RoleAssignmentPolicy>
   <RoleAssignment>
<!-- Role assignment for tender officers.
They must be employees of Salford City Council.  Valid only from close of tender.
Delegation not permitted -->
         <SubjectDomain ID="Employees"/>
     <Role Type="permisRole" Value="TenderOfficer"/>
     <Delegate Depth="0"/>
     <SOA ID="Salford"/>
     <Validity>
       <Absolute Start="2001-09-21T17:00:00"/>
     </Validity>
   </RoleAssignment>
   <RoleAssignment>
<!-- Role assignment for tenderers.
They must be dot com or co.uk companies.  Valid only until close of tender.
Delegation not permitted -->
         <SubjectDomain ID="Companies"/>
     <Role Type="permisRole" Value="Tenderer"/>
     <Delegate Depth="0"/>
     <SOA ID="Salford"/>
     <Validity>
       <Absolute End="2001-09-21T17:00:00"/>
     </Validity>
   </RoleAssignment>
   <RoleAssignment>
<!-- Role assignment for companies who are ISO9000 Certified.
They must be dot com or co.uk companies.  Valid only for a maximum of one year, as
companies have to be annually re-accredited. Certificates are issued by BSI.
Delegation not permitted -->
         <SubjectDomain ID="Companies"/>
     <Role Type="ISOCertified" Value="ISO9000"/>
     <Delegate Depth="0"/>
     <SOA ID="BSI"/>
     <Validity>
       <Maximum Time="+01"/>
     </Validity>
   </RoleAssignment>
</RoleAssignmentPolicy>
```

Table 3. An Example Role Assignment Policy

The SOA creates the XML authorisation policy for the target domain and stores this in a local file, say MyPolicy.XML. This will be used later by the Privilege Allocator tool to create the policy AC. However, creating XML files using generic XML editors such as IBM's Xeena, or text editors such as Notepad, is not intuitively easy, especially for non-technical managers. Consequently we are currently working on a joint project between the University of Salford and UCL to create a user-friendly graphical interface (GUI) to allow non-technical managers to easily create PERMIS policies. A screen shot from this is shown in Figure 3 below.

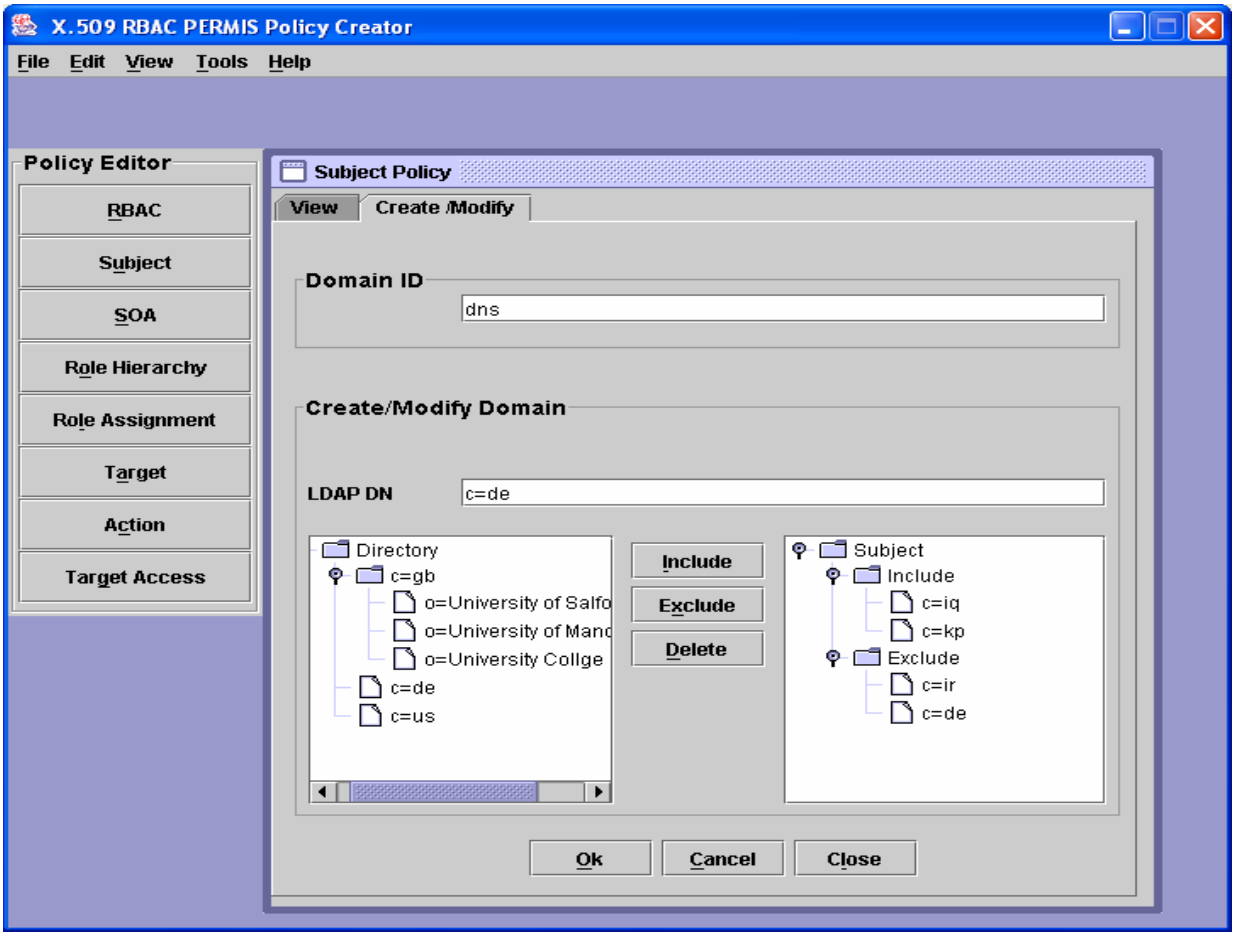


Figure 3. A screenshot from the PERMIS Policy Management GUI tool

Storing the policy

Policies are stored as digitally-signed authorisation policies, in a X.509 policy ACs. The policy AC is a standard X.509 AC with the following special characteristics: the holder and issuer names are the same (i.e. that of the policy issuer SOA), the attribute type is pmiXMLPolicy and the attribute value is the XML policy created as described above. The policy AC is signed by the root of trust of the PMI, and is similar to the self-signed public key certificate of the root CA of a PKI. Policy ACs are created using the Privilege Allocator (PA) tool (see later). The PA prompts the SOA for the name of the policy file (e.g. MyPolicy.XML) and then it copies the contents into the attribute value. After the SOA has signed the policy AC, the PA stores it in the SOA's entry in the LDAP directory. Each authorisation policy is given an Object Identifier [13], which is a globally unique number. This ensures that the PERMIS API (see later) always runs with the correct policy for the domain.

Role Assignment

A role in PERMIS is simply defined as an attribute type and value. We usually use the attribute type *permisRole* whose values are IA5 strings. Administrators (SOAs or AAs) can then put any value in the IA5 string to identify a specific role. Note however that there will have to be negotiation between the role allocating SOAs and the target

site SOA to ensure they use exactly the same attributes for the same roles. This is one reason that the Internet2 consortium has defined the EduPerson LDAP object class [23], so that all universities can use the same attribute types and values to mean the same things.

PERMIS provides two tools for assigning roles to individuals: the Privilege Allocation (PA) graphical user interface tool, and the bulk loader tool. The PA allows an administrator (SOA or AA) to assign one or more attributes (roles) to an individual, to digitally sign the encompassing AC, and then store the AC in the user's entry in an LDAP directory (see Figure 4). The bulk loader comprises a Java API and a sample program which searches an LDAP directory for entries that have a particular object class. For each retrieved LDAP entry, the sample program assigns a particular attribute to its Distinguished Name (DN), then calls the Java API which embeds the attribute in an X.509 AC, signs the AC using the private key of the SOA, and stores the AC in the entry of the given DN. In this way every person, e.g. student, manager, accountant etc. can be automatically assigned different role ACs.

In real life, role assignment can only be done by an authorised person. For example, allocating the role of professor to an academic may be decided by a university appointment's panel and the chair of the panel may inform the Registrar's department to assign the role to a particular individual. In the electronic world, the Registrar would need to issue and digitally sign an X.509 AC to the individual containing the professor role attribute. Within a university department, the head of school may assign the role Undergraduate Course Leader to a member of staff, and similarly issue an X.509 AC to this individual. The X.509 AC always provides the name of the assigning person (the AA), and this is bound to the AC through his/her digital signature, thus X.509 ACs cannot easily be forged.

In PERMIS, anyone is allowed to run the PA or bulk loader tools, i.e. to issue role ACs, but only authorised individuals will be trusted by the PERMIS decision engine, according to the PERMIS trust model (see later). Since all the ACs are digitally signed by their assigning authorities, the PERMIS decision engine can easily determine trusted ACs from non-trusted ACs. Since ACs are digitally signed by the AA which issued them, they are tamper-resistant, and therefore there is no modification risk from allowing them to be stored in a publicly accessible LDAP directory. This also means that authorities who issue digital ACs can store them locally, but give global access to them (just as they might for X.509 public key certificates).

The attribute certificate revocation lists (ACRLs) of revoked certificates (if any), also need to be stored in a publicly accessible place, such as the local LDAP directory, so that relying parties can have easy access to them. Thus there is little advantage in general of distributing the ACs to their holders, rather than keeping them centrally stored in an LDAP directory, since a relying party will still need to access the issuing authorities LDAP directory in order to retrieve the latest ACRL. On the other hand, if the ACs are not distributed to their holders, but are retained in the local LDAP directory, then there is no need for ACRLs to be issued, since the administrator simply needs to delete a user's ACs from their entry in order to "revoke" the AC. We thus believe that the so called "pull" model for AC retrieval [10], where ACs are fetched by PERMIS from various LDAP directories, rather than the user pushing them

to PERMIS from his PC, is the simplest model for managing ACs, and the most efficient for decision making, since ACRLs are not needed.

As can be seen, the PERMIS infrastructure supports distributed authorisation management, in which different sites can run the PA and/or bulk loader tools and allocate ACs to their users and store them in their local LDAP directories. This will significantly ease the management of privileges in large distributed environments, such as GRID networks, Internet marketplaces etc, as the policy governing access to target resources only needs to tell the PERMIS API which LDAP directories to contact and which remote SOAs to trust.

We see that this mechanism can be extended to any type of attribute or role certification e.g. Microsoft Certified Engineer, BSc (Hons) University of Salford etc. and that these "roles" can easily be built into PERMIS via the authorisation policy. We have already used this mechanism in an electronic prescribing system that we built [9], to allow the Royal College of Pharmacy to allocate pharmacist roles to qualified pharmacists, and the General Medical Council to allocate doctor roles to qualified doctors.

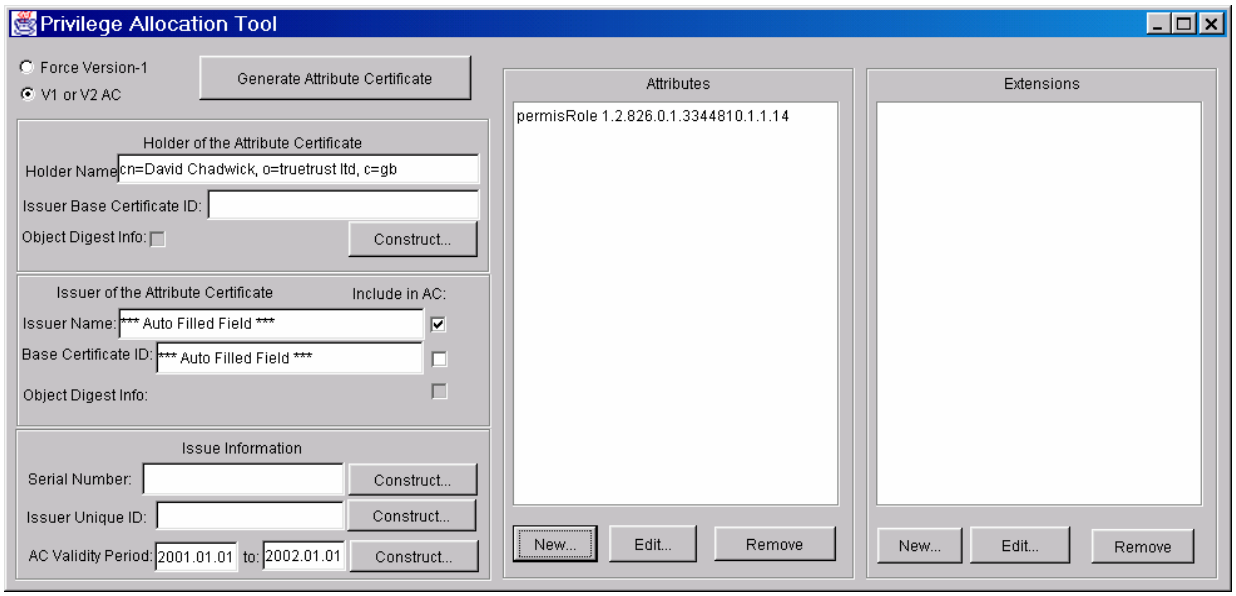


Figure 4. The Privilege Allocation GUI Tool

The PERMIS Decision Engine

An application gateway (see Figure 5) is responsible for authenticating and authorising remote users and providing access to targets. The ISO 10181-3 Access Control Framework [15], splits the functionality of the application gateway into two components: an application-specific component termed the Access Control Enforcement Function (AEF), and an application-independent component termed the Access Control Decision Function (ADF). In this way, all access controls decisions in a domain can be consistently enforced by the ADF independent of the application. The ADF makes its decisions based on the authorisation policy for the domain, on who is initiating a request, what action is being requested on which target, and

environmental factors such as the time of day. The PERMIS decision engine is an implementation of an ADF.

The AEF and ADF can be interconnected via either a local application programmable interface (API) for local invocation, or via a communications protocol for remote invocation.

An API between the AEF and ADF has already been defined by the Open Group. It is called the AZN API [19], and is specified in the C language. A similar API is also being developed by the IETF, called the Generic Authorization and Access control (GAA) API [21]. PERMIS has drawn on the work of the Open Group in order to specify the PERMIS API in Java. This is described in the following section.

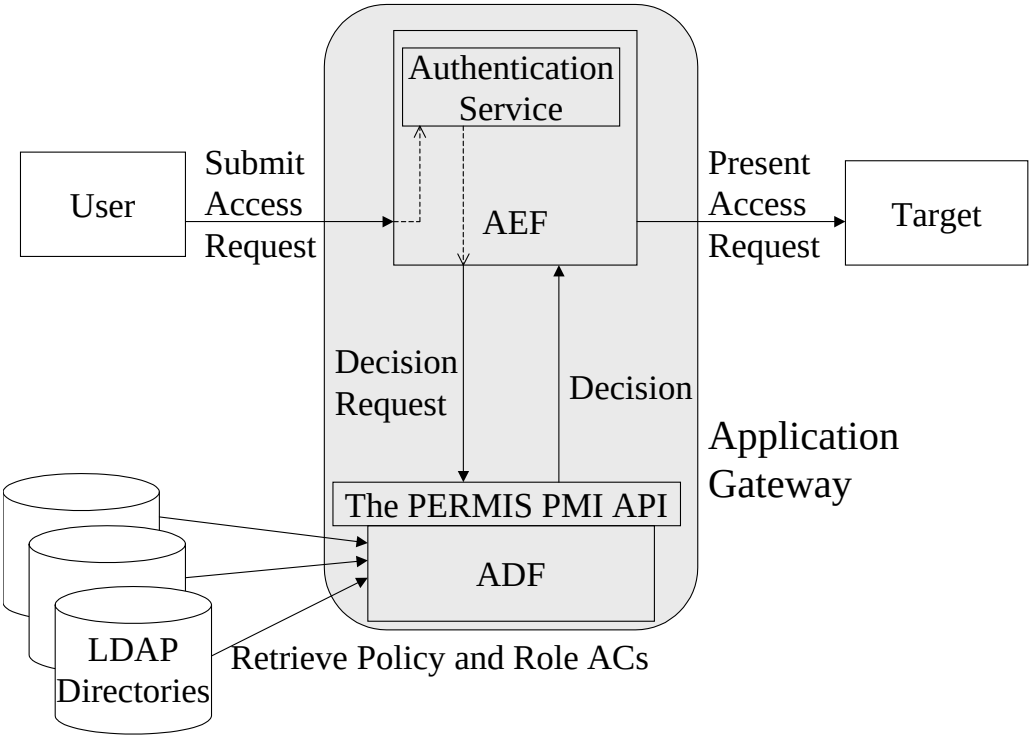


Figure 5. The PERMIS Application Gateway

The OASIS group has defined the SAML protocol [6] as a standard protocol for invoking authorisation decision requests from a remote ADF service. In 2003 we started to work with the Global Grid Forum to produce a profile of SAML for use by Grid applications. This profile [8] has in fact identified a number of deficiencies in the standard SAMLv1.1 protocol, and has defined a few extensions to it. The resulting protocol has now been implemented by PERMIS, and this will allow the PERMIS ADF to run as a stand-alone service. The protocol is also shortly to be incorporated into Globus Toolkit V3, thus allowing grid applications to invoke PERMIS to make authorisation decisions on its behalf.

User Authentication

In order to get through the application gateway, each user needs to be issued with an application specific authentication token acceptable to the gateway. If a PKI is being used, this will be a digitally signed public key certificate. If a conventional authentication system is being used it will be a username/password pair. Alternatively, the application gateway may support Kerberos authentication, one-time password authentication, biometric authentication or any other scheme. The PERMIS API has been designed to be authentication agnostic. PERMIS simply trusts the calling application gateway to have performed authentication of the user to its own satisfaction. All that PERMIS requires is that the application gateway pass to PERMIS the authenticated LDAP DN of the user. PERMIS will then use this to retrieve the user's X.509 ACs, on which to make its authorisation decisions.

To summarise, in the PERMIS model, a user accesses resources via an application gateway. The AEF authenticates the user in an application specific way, then asks the PERMIS ADF if the user is allowed to perform the requested action on the particular target resource. The PERMIS ADF accesses one or more LDAP directories to retrieve the policy AC and the role ACs for the user, and bases its decision on these. If the decision is *grant*, the AEF will access the target on behalf of the user. If the decision is *deny*, the AEF will refuse access to the user. The AEF talks to the PERMIS ADF via either the PERMIS Java API or the GGF SAML protocol.

The PERMIS Java API

The PERMIS Java API comprises 2 simple methods: GetCreds, and Decision, and a Constructor. The Constructor builds the PERMIS API Java object. For construction, the AEF passes the name of the SOA (the root of trust for target authorisation), the Object Identifier of the policy to be used, and a list of LDAP URIs from where the ADF can retrieve the policy AC and subsequently the role ACs. The policy AC is always retrieved from the SOA's entry in the LDAP directory pointed to by the first URI in the list. The Constructor is usually called just once, whilst the AEF starts up. After construction of the API has completed, the PERMIS ADF will have read in and validated the XML of the authorisation policy that will control all future decisions that it makes.

When a user initiates a call to the target, the AEF authenticates the user, then passes the LDAP DN [16] of the user to the PERMIS ADF through a call to GetCreds. In pilot services users have been authenticated in different ways. In an e-tendering application, the user sent an S/MIME email message to the AEF; in an e-planning application, the user opened an SSL connection using an X.509 client certificate.

In all cases the authentication yielded the user's LDAP DN. The PERMIS ADF uses this DN to retrieve all the role ACs of the user from the list of LDAP URIs passed at initialisation time (this is termed the "pull" model [10]). The role ACs are validated against the policy e.g. to check that the DN is within a valid subject domain, and to check that the ACs are within the validity time of the policy etc. Invalid role ACs are discarded, whilst the roles from the valid ACs are extracted and kept for the user, and returned to the AEF as a subject object. (The GetCreds interface also supports the "push" model [10] (called user-pull by Park et al [20]), whereby the AEF can push a set of ACs to the ADF, instead of the ADF pulling them from the LDAP directories,

but since our ADF currently does not retrieve ACRLs, this mechanism is unused at present).

Once the user has been successfully authenticated he will attempt to perform certain actions on the target. At each attempt, the AEF passes the subject object, the target name, and the attempted action along with its parameters, to the PERMIS ADF via a call to Decision. Decision checks if the action is allowed for the roles that the user has, taking into account all the conditions specified in the TargetAccessPolicy. If the action is allowed, Decision returns Granted, if it is not allowed it returns Denied. The user may attempt an arbitrary number of actions on different targets, and Decision is called for each one. In order to stop the user keeping the connection open for an infinite amount of time (for example until after his ACs have expired), the PERMIS API supports the concept of a session time out. On the call to GetCreds the AEF can say how long the session may stay open before the credentials should be refreshed. If the session times out, then Decision will throw an exception, telling the AEF to either close the user's connection or call GetCreds again.

The AEF can stop using a particular ADF at any time, for example if circumstances dictate that the authorisation policy should be changed. If the AEF invokes a new PERMIS ADF constructor, standard Java garbage collection will delete the old ADF. This can happen, say, if the SOA wants to dynamically impose a new authorisation policy on the domain. The AEF can invoke a new Constructor call, and this will cause the ADF to read in the latest authorisation policy and be ready to make access control decisions again based on this.

The PERMIS Trust Model

The PERMIS trust model has a single trusted root of authority, from which all other trust is inherited. When the PERMIS ADF is initialised, it is passed the name of the SOA for the target (the root of trust), the unique OID of the policy to use, and a list of LDAP URIs to access. This allows the PERMIS ADF to fetch the policy AC from the SOA's entry stored in the LDAP directory pointed to by the first LDAP URI in the list. PERMIS checks the digital signature on this policy AC, and if it does not correspond to that of the SOA, then the policy is not to be trusted and it is discarded. If the policy AC has been signed by the SOA, then it is trusted, and its embedded unique OID is compared to that passed at initialisation time. If they are the same, this confirms that the correct policy is being used.

Inside the policy is the SOAPolicy. This gives a list of remote SOAs and AAs to be trusted to allocate role ACs to users. Further, the RoleAssignmentPolicy states which attributes/roles these SOAs and AAs are trusted to issue. Attributes not in this policy will be discarded by PERMIS at decision time. Also the RoleAssignmentPolicy says which subjects these attributes can be assigned to. This ensures that remote SOAs do not allocate roles to wrong groups of users. However, the remote SOAs are trusted to, and expected to, authenticate the subjects prior to issuing them with role ACs. In this way the PERMIS ADF can be assured that all the roles it uses in its decision making have been assigned according to the policy, and are trusted.

Finally, the policy AC says what access rights are given to each role. The PERMIS ADF makes decisions according to this, and the AEF trusts PERMIS to make the correct decisions.

Conclusion

We have shown how the standard X.509 PMI can be adapted to build an efficient role based trust management system, in which the role assignments can be widely distributed between organisations, and the local authorisation policy determines which roles are trusted and what privileges are to be given to them. The PERMIS authorisation policy governs all aspects of access to the targets in the local domain. The PERMIS authorisation policy is written in XML and the DTD has been published by XML.ORG. A user friendly policy management GUI is being built that will allow non-technical managers to easily create PERMIS policies.

PERMIS provides an authorisation decision engine which determines which users are allowed to perform which actions, based on the authorisation policy and the X.509 role ACs allocated to the users. Two tools are provided for allocating role ACs and storing them in LDAP directories: a graphical Privilege Allocator tool and a bulk loading tool.

A simple Java API and SAML interface are provided which allow applications to easily incorporate the PERMIS decision engine as either their local or remote authorisation decision making machine. A public release of PERMIS is available for educational and research use from our web site (http://sec.isi.salford.ac.uk/permis.

Acknowledgments

This initial PERMIS project was 50% funded by the EC ISIS programme, and partially funded by the EPSRC under grant number GR/M83483. The SAML interface and the GUI policy management tool are being funded by JISC. The authors would also like to thank Entrust Inc. for making their PKI security software available to the University on preferential terms.

References

[1] Adams, C., Lloyd, S. (1999). “Understanding Public-Key Infrastructure: Concepts, Standards, and Deployment Considerations”. Macmillan Technical Publishing, 1999

[2] Bertino, E., Castano, S., Farrari, E. “On specifying security policies for web documents with an XML-based language”. Proceedings of the Sixth ACM Symposium on Access control models and technologies 2001, available from ACM digital library.

[3] Blaze, M., Feigenbaum, J., Ioannidis, J. “The KeyNote Trust-Management System Version 2”, RFC 2704, September 1999.

[4] D.W.Chadwick, A. Otenko. “RBAC Policies in XML for X.509 Based Privilege Management” in Security in the Information Society: Visions and Perspectives: IFIP TC11 17th Int. Conf. On Information Security (SEC2002), May 7-9, 2002, Cairo, Egypt. Ed. by M. A. Ghonaimy, M. T. El-Hadidi, H.K.Aslan, Kluwer Academic Publishers, pp 39-53.

[5] D.W.Chadwick, A. Otenko. “The PERMIS X.509 Role Based Privilege Management Infrastructure”, Proc 7th ACM Symposium On Access Control Models And Technologies (SACMAT 2002), Monterey, USA, June 2002. pp135-140.

[6] “OASIS eXtensible Access Control Markup Language (XACML)” v1.0, 12 Dec 2002, available from http://www.oasis-open.org/committees/xacml/

OASIS. “Assertions and Protocol for the OASIS Security Assertion Markup Language (SAML)”. 19 April 2002. See http://www.oasis-open.org/committees/security/
[7] Damianou, N., Dulay, N., Lupu, E., Sloman, M. “The Ponder Policy Specification Language”, Proc Policy 2001, Workshop on Policies for Distributed Systems and Networks, Bristol, UK 29-31 Jan 2001, Springer-Verlag LNCS 1995, pp 18-39
[8] Von Welch, Frank Siebenlist, David Chadwick, Sam Meder, Laura Pearlman. “Use of SAML for OGSA Authorization”, Jan 2004, Available from https://forge.gridforum.org/projects/ogsa-authz
[9] D.W.Chadwick, D.Mundy. "Policy Based Electronic Transmission of Prescriptions". Proc of Fourth IEEE Int Workshop on Policies for Distributed Systems and Networks, Lake Como, Italy, 4-6 June 2003, p197-206
[10] Farrell, S., Housley, R. “An Internet Attribute Certificate Profile for Authorization”, RFC 3281, April 2002
[11] Sandhu, R., Ferraiolo D., Kuhn, R. “The NIST Model for Role Based Access Control: Towards a Unified Standard”. In *proceedings of 5th ACM Workshop on Role-Based Access Control*, pages 47-63. (Berlin, Germany, July 2000).
[12] Housley, R., Polk, T. “Planning for PKI: Best Practices Guide for Deploying Public Key Infrastructure”. John Wiley and Son, ISBN: 0-471-39702-4, 2001
[13] ITU-T Recommendation X.680 (1997) | ISO/IEC 8824-1:1998, Information Technology - Abstract Syntax Notation One (ASN.1): Specification of Basic Notation
[14] ISO 9594-8/ITU-T Rec. X.509 (2001) The Directory: Public-key and attribute certificate frameworks
[15] ITU-T Rec X.812 (1995) | ISO/IEC 10181-3:1996 “Security Frameworks for open systems: Access control framework”
[16] Wahl, M., Kille, S., Howes, T. "Lightweight Directory Access Protocol (v3): UTF-8 String Representation of Distinguished Names", RFC2253, December 1997
[17] D.W.Chadwick, “The X.509 Privilege Management Infrastructure” in *Proceedings of the NATO Advanced Networking Workshop on Advanced Security Technologies in Networking, Bled, Slovenia, June 2003*
[18] “OASIS eXtensible Access Control Markup Language (XACML)” v1.0, February 2003, available from http://www.oasis-open.org/committees/tc_home.php?wg_abbrev=xacml
[19] The Open Group. “Authorization (AZN) API”, January 2000, ISBN 1-85912-266-3
[20] Park, J.S., Sandhu, R., Ahn,G. “Role-Based Access Control on the Web”, ACM Transactions on Information and Systems Security, Vol 4. No1, Feb 2001, pp 37-71.
[21] Ryutov, T., Neuman, C., Pearlman, L. “Generic Authorization and Access control Application Program Interface C-bindings” <draft-ietf-cat-gaa-cbind-05.txt>, November 2000. See http://www.isi.edu/gost/info/gaaapi/
[22] Sandhu, R.S., Coyne, E.J., Feinstein, H.L., Youman, C.E. “Role Based Access Control Models”. IEEE Computer 29, 2 (Feb 1996), p38-43.
[23] See http://www.educause.edu/eduperson/