Implementing Role Based Access Controls Using X.509 Attribute Certificates – the PERMIS Privilege Management Infrastructure

David W Chadwick, Alexander Otenko, Edward Ball
IS Institute, University of Salford, M5 4WT, England

Email: D.W.Chadwick@salford.ac.uk
Telephone: +44 161 295 5351
Fax: +44 161 745 8169

Implementing Role Based Access Controls Using X.509 Attribute Certificates – the PERMIS Privilege Management Infrastructure

Abstract

This paper describes a policy driven role based access control system. The user's roles, and the policy are stored in X509 Attribute Certificates. The policy, written in XML, describes who is trusted to allocate roles to users, and what permissions each role has. The DTD has been published at XML.org. Access control decisions are made by an Access Control Decision Function consisting of just three Java methods and a constructor. The decision is made according to the requested mode of access, the user's trusted roles and the policy. A tool making and storing ACs is also described.

Keywords

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

Introduction

The primary purpose of a Public Key Infrastructure (PKI) is to strongly authenticate the parties communicating with each other. The standard format for a public key certificate is specified in the X.509 standard [X509]. But authentication on its own is not enough. As well as knowing who a remote party is, one also needs to know what action(s) the remote party is authorised to undertake. Thus we also need an authorisation mechanism. Edition 4 of X.509 is the first edition to fully standardise a strong authorisation mechanism, which it calls a Privilege Management Infrastructure (PMI). A PMI provides the authorisation function after the authentication function has taken place, and has a number of similarities with a PKI (see side box). The X.509 PMI standard does not favour any particular authorisation scheme. Discretionary, Mandatory and Role Based Access Control (DAC, MAC and RBAC) schemes can all be supported. RBAC (see side box) has the advantage that it can significantly simplify the management of access controls for large numbers of users, since the permissions are allocated to roles rather than to individual users.

The EC funded PERMIS project was given the challenge of building an X.509 role based PMI that can be used by very different applications in 3 cities of Europe. The project had members from the cities of Barcelona (Spain), Bologna (Italy) and Salford (UK). All three centres already had experience of running pilot PKIs, and so it was natural for them to want to add a PMI capability. The chosen applications of the 3 cities are very different in character, so they were a good test of the generality of the developed PMI.

Bologna wanted to allow architects to download street maps of the city, to update the maps with their proposed plans, and to upload the new building plans and requests for building licenses to the city planning office's server.

Barcelona is a major tourist and commercial centre but parking is very restricted. Many parking tickets are frequently issued to hired cars but by the time the car hire companies receive the parking tickets, the hirers have left the country. The plan was to give the car hire companies on line access to the city's parking ticket database, so that

the companies can instantly check to see if any parking tickets have been issued for returned cars. Each company will be able to send the details of the driver to the city, thereby transferring the fine to the individual. Legislation requires that a car hire company can only access the tickets issued to its own cars, and so authorisation will need to be at the record level.

Salford is implementing an electronic tendering application. The tendering process will start when the city places the Request for Proposal (RFP) documents on its web site, allowing anyone to download them. However, in some restricted tendering instances, only companies previously authorised by Salford will be able to submit tenders. In other cases it may be a requirement that a company has ISO 9000 or other certification in order to submit a tender. Once the tenders have been submitted, they must remain anonymous until the winner has been chosen. The city tender officers must not be given access to the electronic tender store before the closing date of the RFP, and tenderers must not be allowed to submit tenders after the closing date.

The challenge for the PERMIS project was to build a generalised role based X.509 PMI that can cater for these very different applications.

Trust Management with X.509 PMIs

According to the definition in RFC 2704 [Blaze], a trust management system comprises 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.

The objective of the PERMIS project was to build a trust management system from the supplier's perspective, rather than simply a role based X.509 PMI, since the latter is missing some vital system components. Whilst X.509 specifies mechanisms for ii) and iv) above, it does not specify i), ii) or v). In X.509 principals are the holders and issuers of ACs and are usually identified[1] by their X.500 General Name (usually an X.500 distinguished names or IP address, URI or email address) or by reference to their public key certificate (issuer name and serial number). Credentials are specified as X.500 attributes within ACs and they comprise an attribute type and value. These attributes will hold the credentials of the particular access control scheme (MAC, DAC or RBAC) being implemented. X.509 describes how these attributes can be delegated from holder to holder. However X.509 does not standardise the languages for the policy (iii) and actions (i). These were significant tasks of the PERMIS project, as was building the compliance checker (v) (which is part of our privilege verification subsystem). A final essential component of a functional system, which is not explicitly

[1] If the principal is a software object, it can be identified by a hash of itself.

mentioned in [Blaze] or [X509], is a privilege allocation subsystem that allows an issuer to create and digitally sign X.509 ACs.

Implementing RBAC with X.509

The X.509 PMI standard supports $RBAC_0$ [Sandhu] by defining two types of attribute certificate. *Role specification ACs* hold the permission assignments granted to each role, e.g. Role Employee can Print files to Laser 1. *Role assignment ACs* hold the roles assigned to each user e.g. D. Chadwick has Role Employee (user assignments in $RBAC_0$ terminology). In role specification ACs, the AC holder is the role, and the privilege attributes are permissions granted to the role. In role assignment ACs the AC holder is the user, and the privilege attributes are the roles assigned to the user. If one were to implement the X.509 standard as written, then in order to determine the permissions granted to a user, the compliance checker would need to read in all the role assignment ACs granted to the user, all the role specification ACs for each granted role, plus validate the signatures of each AC and check the various CRLs to ensure that no AC has been revoked since creation. Further, if delegation of authority is supported, the chain of role assignment ACs from the user up to the trusted SOA would need to be validated. This is clearly a time consuming and onerous task, as confirmed by Knight [Knight]. In order to make our system more efficient we made two performance enhancing design decisions. Firstly we decided to place all the role specifications/permission assignments in just one policy AC, thereby significantly simplifying the permission validation process. Secondly we decided to store all the role assignment ACs in distributed LDAP directories and implement the "pull" model [Farrell] (called server-pull by [Park]), so that CRLs become unnecessary (revoked ACs are simply deleted from their LDAP directories). Also, in the initial release, we do not support delegation of authority.

X.509 fully supports hierarchical RBAC by allowing both roles and privileges to be inserted as attributes in a role specification AC, so that the latter role inherits the privileges of the encapsulated roles. For example, Role Manager can Delete files from printer queue of Laser 1 and has role of Employee. Since in PERMIS our policy AC subsumes the functionality of role specification ACs, it is our policy that describes the role hierarchy (see later.)

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 permission 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 AC construct e.g. within the policy and compliance checker. Consequently the PERMIS project has only partially implemented $RBAC_2$ and this will be the subject of future research.

Defining the Policy Language

The policy needs to specify who is to be granted what type of action on which targets, and under what conditions. Having a domain wide policy for authorisation is preferable to 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 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 policy languages. The Ponder language [Ponder] is compact and powerful, but does not have a large set of supporting tools. The Keynote policy language [Blaze] is also 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 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 is extensive tool support for XML, it is fast becoming an industry standard, and raw XML can be easily read and understood. Shortly after we started our work, Bertino et al published a paper [X-Sec] that showed that XML was indeed suitable for specifying authorisation policies and the Lawrence Berkeley National Laboratory implemented XML policies in their Akenti system [Thompson]. Later, the OASIS consortium began work on the eXtensible Access Control Markup Language [XACML].

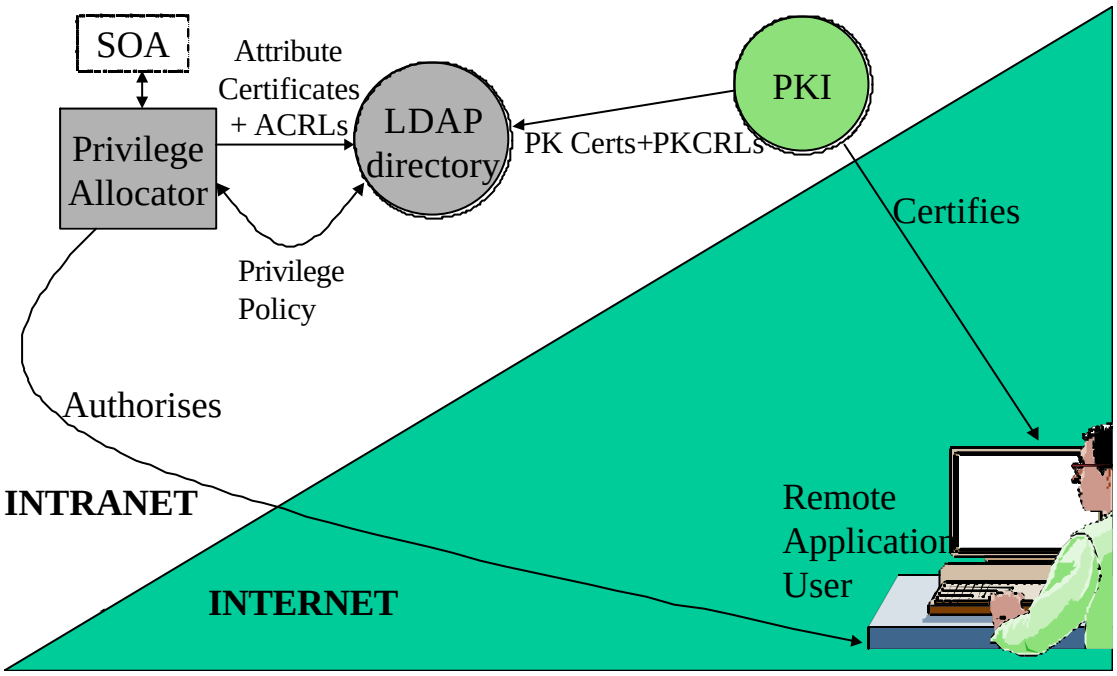
We drew on the previous research, particularly Ponder, and took many of its concepts and tailored them to support X.509 and RBAC. There are many similarities between our resulting Data Type Definition (a DTD is a meta-language that holds the rules for creating XML documents) and the later XACML work. Our X.500 PMI RBAC Policy comprises the following components:

?? SubjectPolicy – this specifies the subject domains i.e. only users from a subject domain may be authorised to access resources covered by the policy. Each domain is specified as an LDAP subtree, using Include DN and Exclude DN statements
?? 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 trusting remote SOAs, and this enables the distributed management of roles to take place
?? 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. This sub-policy effectively states who is trusted to allocate which roles to whom, and is central to the distributed management of trust
?? TargetPolicy – this specifies the target domains covered by this policy. Each domain is specified as an LDAP subtree, using Include DN and Exclude DN statements
?? 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 policy can be found in [Chadwick]. An SOA creates the policy for his domain using his preferred XML editing tool, and stores this in a local file, say MyPolicy.XML, to be used later by the Privilege Allocator to create the policy AC.

Figure 1. The Privilege Allocation Subsystem



The Privilege Allocation Subsystem

The privilege allocation subsystem (see Figure 1) comprises a Privilege Allocator (PA) (see below) that issues X.509 role assignment ACs for users, and also signs the policy AC that will control the RBAC API (see later). These are stored in an LDAP directory for subsequent use. In addition to privileges, each user will also need to be issued with an application specific authentication token. 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. The PERMIS API is authentication agnostic.

As the distributed management of roles is supported so different sites can allocate ACs to their users and store them in their LDAP directories. This significantly eases the management of privileges in large distributed environments, such as GRID networks, Internet marketplaces etc, as the local SOAPolicy tells the PERMIS API which remote SOAs to trust. Thus we can have role assignment ACs issued by SOA_1 that are trusted by one domain and not trusted by another domain.

The Privilege Allocator (PA)

This tool is used by the SOA or an AA to allocate privileges to users. Since PERMIS is using RBAC, the SOA uses the PA to allocate roles to users in the form of role assignment ACs. A role in PERMIS is simply defined as an attribute type and value. We are using two attribute types *permisRole* and *ISOCertified*, whose values are IA5 strings. In the case of Bologna, there are two permisRole values: Map-Readers and Architects. Map-Readers can download any maps produced by the municipality, whereas Architects are allowed to download maps and upload digitally modified maps. In the case of Barcelona, there are also two permisRoles defined: Generalised and Authorised. Any citizen or business can be allocated the Generalised role. Anyone with the Generalised role has permission to read their own pending car parking fines. Businesses that have signed an agreement with the Barcelona city council are given the Authorised role. Authorised roles can read their own pending fines and also may modify the details of them (e.g. update the driver's name and address). Salford is different to the other sites, in that whilst it will allocate two permisRoles, that of Tenderer and Tender-Officer, it will also rely on an external SOA (in this case the British Standards Institute) to allocate the ISOCertified roles to users. (In fact in the project we set up a proxy BSI SOA to allocate these roles, as the BSI is not a project partner.)

The ACs are stored in an LDAP directory. This can be made publicly accessible since there is no modification risk as the ACs are digitally signed. This also means that authorities who issue ACs can store them locally but allow global access. We think this might be particularly useful for example in the case of ISOCertified roles. Anyone wishing to know if a person has an ISOCertified role may access the BSI LDAP directory and retrieve the person's X.509 AC. The ACRLs of revoked certificates (if any) will also be stored here. Thus there is little advantage in general of distributing the ACs to their holders, since a relying party will still need to access the issuing authorities LDAP directory to retrieve the latest ACRL (or call an OCSP responder once these become available for attribute certificates). This mechanism can be extended to any type of privilege certification e.g. Microsoft Certified Engineer, etc. and these "roles" can easily be built into our API via the policy. We are already using this in an electronic prescribing system that we are building that could allow the Royal College of Pharmacy to allocate roles to qualified pharmacists, and the General Medical Council to allocate roles to qualified doctors.

The PA can also create digitally-signed policy ACs. These are standard X.509 ACs with the following special characteristics: the holder and issuer names are the same (i.e. that of the SOA), the attribute type is pmiXMLPolicy and the attribute value is the XML policy created as described above. The policy AC indicates the root of trust of the PMI, and is similar to the self-signed public key certificate of the root CA of a PKI. 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, it is stored in the SOA's entry in the LDAP directory. Each policy is given an Object Identifier, which is a globally unique number. This is passed to the PERMIS API during construction (see later) to ensure it always runs with the correct policy, and allows policies to be dynamically changed.

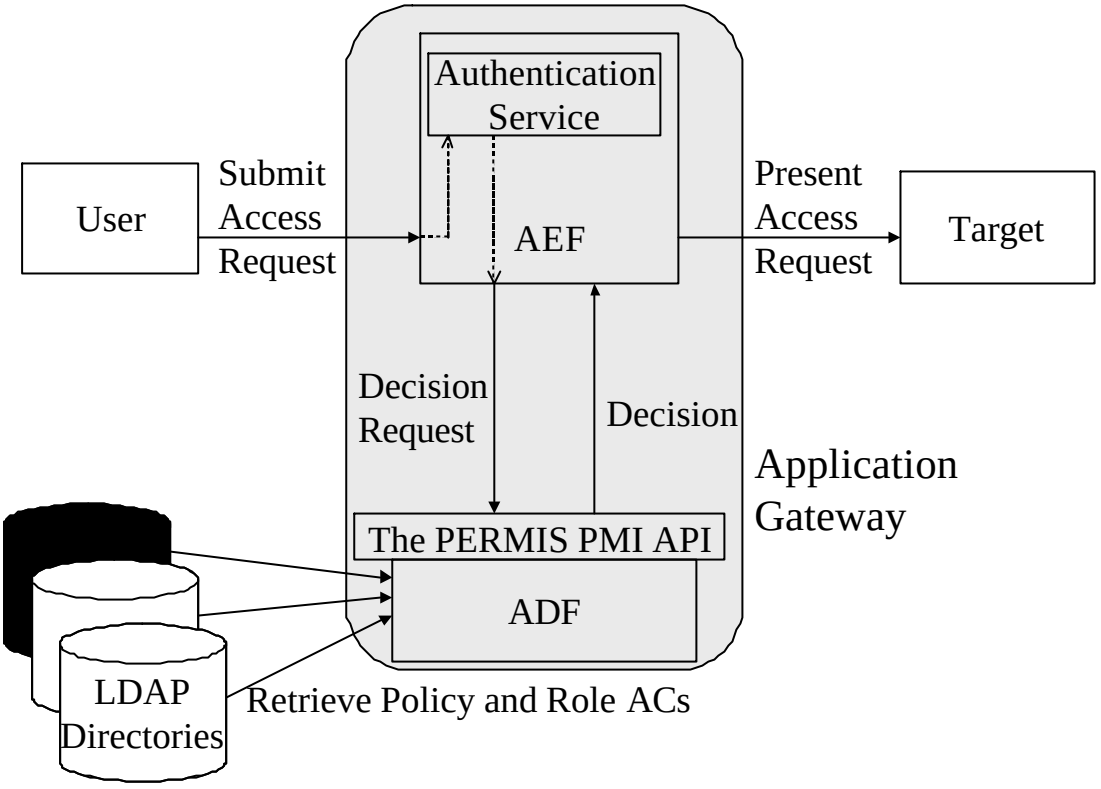
The Privilege Verification Subsystem

The privilege verification subsystem (see figure 2) is responsible for authenticating and authorising the remote user and providing access to the target. The primary component is the application gateway. As we wanted a policy to control all access within a domain we followed the guidance in ISO 10181-3 Access Control Framework [X812] which 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). The ADF is equivalent to Blaze's compliance checker [Blaze] and ensures that 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 policy for the domain.

Two application programmable interfaces (APIs) between the AEF and ADF have been defined; the AZN API [AZN] and the Generic Authorization and Access control (GAA) API [GAA] with its C binding [Ryutov]. Both have somewhat complex C interfaces, but we wanted a simple to use Java API. Basing our work on the AZN API firstly we specified the PERMIS API in Java rather than in C, and secondly we simplified the API by assuming that the Target and the AEF are either co-located or can communicate across a trusted LAN. Without this latter simplification the authorisation token carried from the AEF to the Target would need to be protected, for example as an X.509 attribute certificate, and so we would have gained little from implementing an application gateway.

In summary, a PERMIS user accesses resources via an application gateway. The AEF authenticates the user in an application specific way, then asks the ADF if the user is allowed to perform the requested action on the particular target resource. The ADF accesses one or more LDAP directories to retrieve the PERMIS RBAC policy 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 ADF via the PERMIS Java API.

Figure 2. The Privilege Verification Subsystem



The PERMIS PMI API

The PERMIS API comprises 3 simple methods: GetCreds, Decision, and Shutdown, 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 authorisation), the Object Identifier of the policy, 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 first URI in the list. The Constructor is called when the AEF starts up. After the API has been constructed, the ADF reads in and validates the XML policy that will control all future decisions.

When a user initiates a call to the target, the AEF authenticates the user, then passes the LDAP distinguished name (DN) of the user to the ADF through a call to GetCreds. In the 3 cities the users will be authenticating in different ways. In Salford the user will be sending an S/MIME email message to the AEF, in Barcelona and Bologna he will be opening an SSL connection. In all cases the user will be digitally signing the opening message, and verification of the signature will yield the user's LDAP DN. The ADF uses this DN to retrieve all the role ACs of the user from the list of LDAP URIs passed at initialisation time (the "pull" model). 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 [Farrell] (called user-pull by [Park]), 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 CRLs, this mechanism is unused at present).

Once the user has been successfully authenticated he will attempt to perform 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 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. Because GetCreds has performed the onerous task of role validation, Decision is very quick to execute. Had we simplified the API by merging GetCreds and Decision into one call, then performance would have been adversely affected for those users performing more than a single action.

Shutdown can be called by the AEF at any time. Its purpose is to terminate the ADF and cause the current policy to be discarded. This could happen when the application is gracefully shutdown, or if the SOA wants to dynamically impose a new policy on the domain. The AEF can follow the call to Shutdown with a new Constructor call, and this will cause the ADF to read in the latest policy and be ready to make access control decisions again.

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 policy determines which roles are to be trusted and what privileges are to be given to them. The local policy is written in XML and provides the rules governing all aspects of access to the targets in the local domain. A simple Java API is provided which allows all target applications to easily incorporate this system. The generality of the PERMIS API has already proven its worth, as it is being used in 4 very different applications throughout Europe.

Acknowledgments

This work has been 50% funded by the EC ISIS programme PERMIS project, and partially funded by the EPSRC under grant number GR/M83483. The authors would also like to thank Entrust Inc. for making their PKI security software available to the University on preferential terms.

An earlier version of this paper was presented at the SACMAT 2002 conference.

References

[AZN] The Open Group. “Authorization (AZN) API”, January 2000, ISBN 1-85912-266-3

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

[Chadwick] Chadwick, D.W., Otenko, A. “RBAC Policies in XML for X.509 Based Privilege Management” to be presented at IFIP SEC 2002, Egypt, May 2002

[Farrell] Farrell, S., Housley, R. “An Internet Attribute Certificate for Authorization”, <draft-ietf-pkix-ac509prof-05.txt>, August 2000

[GAA] T. Ryutov, C. Neuman, L. Pearlman. “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/
[Knight] Knight, S., Grandy, C. “Scalability Issues in PMI Delegation”. Pre-Proceedings of the First Annual PKI Workshop, Gaithersburg, USA, April 2002, pp67-77
[Park] J.S.Park, R. Sandhu, G. Ahn. “Role-Based Access Control on the Web”, ACM Transactions on Information and Systems Security, Vol 4. No1, Feb 2001, pp 37-71.
[Ponder] 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
[Ryutov] Ryutov, T., Neuman, C. “Generic Authorization and Access control Application Program Interface: C-bindings”, <draft-ietf-cat-gaa-cbind-05.txt>, November 2000.
[Sandhu] Sandhu, R.S., Coyne, E.J., Feinstein, H.L., Youman, C.E. “Role Based Access Control Models”. IEEE Computer 29, 2 (Feb 1996), p38-47.
[Thompson] M. R. Thompson, S. Mudumbai, A. Essiari, W. Chin. “Authorization Policy in a PKI Environment”, Proceedings of the First Annual PKI Workshop, Gaithersburg, USA, April 2002, pages 137-149.
[X509] ITU-T Rec. X.509 (2000) | ISO/IEC 9594-8 The Directory: Authentication Framework
[X812] ITU-T Rec X.812 (1995) | ISO/IEC 10181-3:1996 “Security Frameworks for open systems: Access control framework”
[XACML] “OASIS eXtensible Access Control Markup Language (XACML)” v0.11, March 2002, available from http://www.oasis-open.org/committees/xacml/docs/
[X-Sec] 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.

SIDE BOX

Introduction to X.509(2000) PMIs

PMIs provide the authorisation function after the authentication has taken place, and have a number of similarities with PKIs.

The primary data structure in a PMI is an X.509 Attribute Certificate (AC). This strongly binds a set of attributes to its holder, and these attributes are used to describe the various privileges that the issuer has bestowed upon the holder. 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.

Each AC 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, so as to provide authorisation instead of authentication. (In this respect a public key certificate can be seen as 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.

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 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 implicitly 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. 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 will 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 will issue an attribute certificate revocation list (ACRL) containing the list of ACs no longer to be trusted.

More information about attribute certificates can be found in [Farrell].

SIDE BOX

Introduction to RBAC

Much research has focussed on Role Based Access Controls (RBAC) e.g. [Sandhu][Park]. In the basic RBAC model, $RBAC_0$, a number of roles are defined. These roles typically represent organisational roles such as secretary, manager, employee etc. Each role is assigned a set of permissions i.e. the ability to perform certain actions on certain targets (termed permission assignment). Each user is then assigned to one or more roles (termed user assignment). When accessing a target, a user presents his role(s), and the target reads the permission assignments to see if this role is allowed to perform this action.

The hierarchical RBAC model, $RBAC_1$, 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 permission assignment does not explicitly state this. If a programmer role is assigned permission to enter the computer building, then managers and directors would also inherit this permission. Hierarchical roles mean that permission assignments are more compact.

Another extension to basic RBAC is constrained RBAC, $RBAC_2$. This allows various constraints to be applied to the user 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.

Finally the consolidated RBAC model, $RBAC_3$, includes the features of both $RBAC_2$ and $RBAC_1$.

RBAC is seen to be so important that NIST is now facilitating the voluntary development of an RBAC standard. For more details see http://csrc.nist.gov/rbac/ .
