Chapter V
Dynamic Delegation of Authority in Web Services

David W. Chadwick
Computing Laboratory, UK

ABSTRACT

Delegation of authority (DOA) is an essential procedure in every modern business. This chapter enumerates the requirements for a delegation of authority Web service that allows users and services to delegate to other users and services authority to access computer-based resources. The various models and architecture that can support a DOA Web service are described. A key component of the DOA service is the organisation's delegation policy, which provides the rules for who is allowed to delegate what to whom, and which needs to be enforced by the DOA service. The essential elements of such a delegation policy are outlined. The chapter then describes a practical DOA Web service that has been built and piloted in various grid applications. It concludes by reviewing some related research and highlighting where future research is still required.

INTRODUCTION

Delegation of authority is an essential procedure in every modern business. A delegate is defined as "A person authorized to act as representative for another; a deputy or an agent" (www.dictionary.com). Without delegation of authority (DOA), managers would soon become overloaded. DOA allows tasks to be disseminated between employees in a controlled manner. A delegate may be appointed for months, day, or minutes, for one task, a series of tasks, or all tasks associated with a role. DOA needs to be fast and efficient with a minimum of disruption to others. Delegators should not need permission from their superiors for each act of delegation they undertake, or otherwise their superiors would soon become overburdened with delegation requests from subordinates. Instead, a delegation policy should be in place so that delegators know when they are empowered to delegate (i.e., what and to whom) and when they are not.

The recipient (or service provider) who is asked to perform a service for a delegate should be able to independently verify that the delegate has been properly authorized to act as a representative for the delegator, before granting the request. If the delegate has not been properly authorised, the delegate's request should be declined. The recipient will therefore enforce the delegation policy of its organization and deny service requests from unauthorized delegates.

In a computing environment there is also a need for DOA. One computer process may need to delegate to another computer process. One person may need to delegate his privileges to another person in order to allow the later to undertake the computer-based tasks of the former. Similarly, in a service-oriented world, computer services also need the ability to delegate tasks to other services, so that the latter can perform subtasks on the former's behalf. Service providers need to be able to verify that each service requestor is properly authorized. If the service requestor has been dynamically delegated authority by another authorized entity, service providers need to be able to verify that this was done in accordance with their delegation policy.

The objective of this chapter is to present a model for dynamic delegation of authority in a Web services world, in which users can delegate to other users, services to other services, and users to services. This chapter also describes a current implementation of this model and compares and contrasts it with other delegation systems that only partially implement the model.

BACKGROUND

In Grid computing today, which is based on Web services, delegation from a user to his Grid job is enacted via the process of proxy certificates (Tuecke, Welch, Engert, Pearlman, & Thompson, 2004). The purpose of these is two fold. Firstly, it allows a user to start a Grid job, and then leave it to run in his absence for as long as is required, without him needing to be there to continually log in and authorize the use of new Web services by the job. Secondly, it allows the job to migrate around the Grid, and to spawn new subtasks to run on other machines as necessary. These subtasks can themselves authenticate as proxies of the user and consume Web services (or resources) that the user is entitled to have. The process works as follows. The Grid user, who must have an asymmetric key pair and X.509 certificate, initializes his Grid job, and during this process the job creates its own asymmetric key pair. The user then issues an X.509 proxy certificate for the Grid job, which certifies the public key of the job. The proxy certificate also contains the name of the job (which must be subordinate to the user's own distinguished name), the name of the user as the issuer, and the signature of the user. The Grid job can now authenticate to any Web service Grid resource by digitally signing requests using its own newly created private key, and the Web service can authenticate the job using the job's newly created proxy certificate. Because the name of the Grid job is subordinate to that of the user, then the Web service knows that it has to check if the user is authorized to access this service, and if so, then the service is to be consumed on behalf of the user. When a new subtask needs to be spawned, to run elsewhere on the Grid, the spawned subtask can generate its own new asymmetric key pair, and the original Grid job can issue a second proxy certificate for the spawned subtask, with a name that is subordinate to its own. In this way the job can delegate as necessary in order to achieve its aims. In each case, the Web service checks if the user, and not the job itself, is authorized to consume its resources. This is easily achieved because the name of the job is linked to the name of the user by being subordinate to it.

Note that in the basic proxy certificate scheme, determining the user's authorization rights is left to the Web service. In Globus Toolkit, a grid mapfile is used to map the user's authenticated (proxy certificate) name onto a local user account name, and the normal operating system mechanisms are then used to control the access rights of each user account. Proxy certificates do have a field (the Proxy Policy field) to contain authorization information, but no standard mechanisms are currently defined for what this field should contain, other than "inherit all" or "independent." In the basic proxy scheme, the Proxy Policy field is set simply to "inherit all," meaning that the proxy certificate inherits all the user's access rights, whatever they are. The Proxy Policy field may alternatively be set to "independent," meaning that the proxy should be treated as an independent entity that has its own authorization rights issued to it, and it inherits no rights of the issuing user, but we do not believe this latter value is currently being used much, if at all.

A slightly more sophisticated mechanism has recently been engineered in the Virtual Organization Management Service (VOMS) (Alfieri, Cecchini, Ciaschini, Dell'agnello, Frohner, Lorentey, & Spataro, 2005). This allows a user to delegate an explicit subset of his roles to his grid job. The user asks his local VOMS server to issue him with one or more short lived X.509 attribute certificates (ACs) which contain (possibly a subset of) his roles in the virtual organization (VO). These attribute certificates are then placed inside the job's proxy certificate (in a new certificate extension field—called AC Sequence—defined specifically for this) and in so doing, the ACs can be transported around the grid by the job and its spawned subtasks. The purpose of these ACs is to delegate to the job a specific (sub)set of roles held by the user, so that the job only inherits the (sub)set of permissions assigned to these roles. Note that VOMS does not use the Proxy Policy field for this, even though it was designed for this purpose. This is so that service providers which do not understand VOMS ACs and the new AC Sequence certificate extension field, can still utilize the proxy certificate in the basic way, by using the Proxy Policy field and assigning all the user's permissions to the grid job. Whether this is good practice from a security perspective or not is open to debate.

As good as the proxy certificate scheme is, nevertheless there are a number of problems with its approach. Firstly, the delegator must have an asymmetric key pair in order to sign the proxy certificate. Most users today do not have X.509 certificates and signing keys. Thus, we need a delegation process that does not mandate that a delegator has an asymmetric key pair. Secondly, proxy certificates cannot be revoked. Instead, they are designed to be relatively short lived. This means that once the Grid job has started, and its proxy certificate has been issued, it cannot be stopped automatically, and neither can any of the spawned sub tasks. Instead, some form of manual intervention by the Web service administrators will be needed to kill the job. To try to limit the damage, proxy certificates (and VOMS attribute certificates) are given short lifetimes, typically of the order of 24 hours, although the actual duration is application dependent. Ideally, we need a more proactive way of revoking proxy certificates after they have been issued and before they have expired, for example, by re-evaluating their permissions at specific intervals or every time new subtasks are spawned. Finally, the proxy certificate approach only works for DOA from a user to a Grid job and from a job to a spawned subtask, and does not work from user to user. This is a significant limitation in its applicability. In order to overcome all these limitations we need a better approach to DOA, one that is general purpose and can cater for delegation from person to person, person to task, task to task, service to service, and so forth, in which the delegators are not mandated to have PKI key pairs, and in which the act of delegation can be revoked prematurely.

REQUIREMENTS FOR WEB SERVICES DELEGATION OF AUTHORITY

As stated above, the first requirement is for a general purpose delegation of authority service that can delegate from any type of entity to any other type of entity (Requirement 1).

Secondly, we need to be able to independently name the delegator and the delegate. It might be acceptable in person to job delegation that the job takes a name subordinate to that of the person, but in person to person delegation and Web service to Web service delegation we should not have to make the delegate assume a principal name which is subordinate to that of the delegator. For the reason of prudent accountability, if nothing more, every principal should authenticate with its own identity, and not with that of another. So delegation should be from one named entity to another, where their names do not need to bear any relationship to each other (Requirement 2).

In order to build a scalable authorization infrastructure, we need to move toward attribute or role-based access controls, where a principal is assigned one or more attributes, and the holder of a given set of attributes is given certain access rights to certain resources. In this way we can give access rights to a whole group of principals, for example, to anyone with an IEEE membership attribute, or to any member of project X, or any Web service of a specific type, without needing to list all the members individually, as there might be many thousands of them (Requirement 3).

The delegation scheme will benefit from a hierarchical model for roles and attributes so that delegators can delegate a subset of their roles/attributes. With hierarchical roles and attributes, a principal with a superior role (or attribute) inherits all the permissions of the subordinate roles (or attributes), and may delegate a subordinate role rather than the most superior role he holds. For example, a project manager may be superior to a team leader who is superior to a team member who is superior to an employee. Principals should to be able to delegate any of their roles and attributes to other principals, so that the delegate may perform on their behalf only those tasks that are enabled by the delegated attributes. For example, a project manager should be able to delegate the subordinate role of team member to an employee (Requirement 4).

All organizations need to be able to control the amount of delegation that is possible, in order to stop "wrong" delegations from being performed. For example, a project manager should not be able to delegate his age or name attributes to anyone else, nor be able to delegate the team member role to one of his children. So we need to have a Delegation Policy, and an effective enforcement mechanism that will control both the delegation process itself (is this delegator allowed to delegate these attributes to this delegate?) and the verification process by the consuming Web service (is this delegate properly authorized to access this service?) (Requirement 5).

We may want very fine grained delegation, in order to delegate a specific task rather than attributes or roles, because the latter usually confer permissions to perform a set of tasks (Requirement 6).

Users must not be constrained to having a PKI key pair before they can delegate to another entity. Users should be able to authenticate and prove their identity without having to possess a public key certificate (Requirement 7).

A delegator should be able to prematurely revoke an act of delegation, without the delegation lasting for its originally intended period of time. When delegation takes place, its effect should be instantaneous. There are many reasons why premature revocation may be needed, for example, the delegator returns early from vacation or sick leave and wishes to continue in his role himself, or the delegate proves to be untrustworthy or incompetent in the delegated role, or the delegate moves position in the organization and the delegation is no longer appropriate, and so forth (Requirement 8).

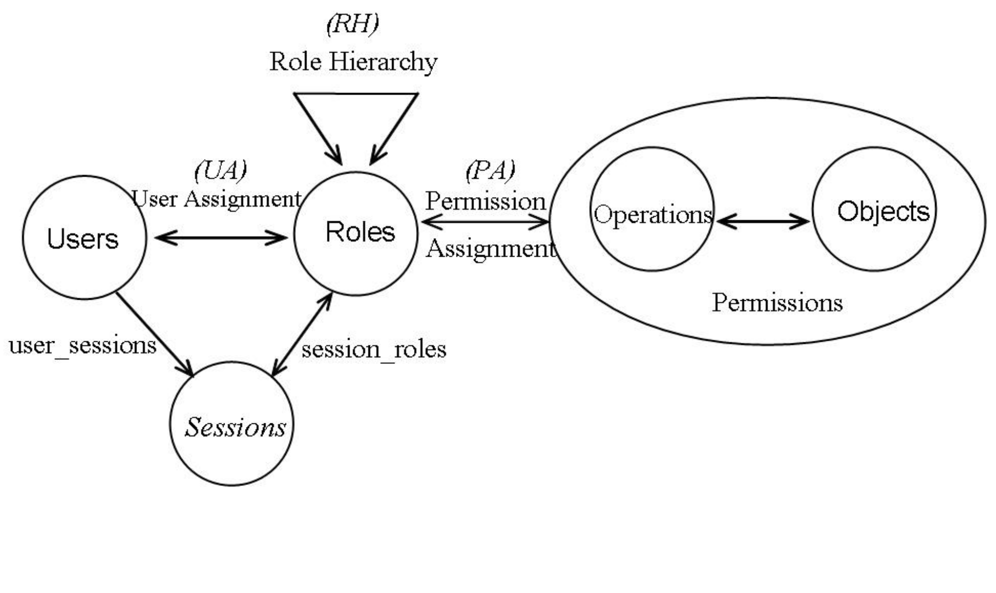
Finally, we may wish to make the whole DOA system Web services compliant, so that it will integrate nicely with the service-oriented architectures (SOA) Web services world that is the subject of this book. (Note, however, that this last requirement is not a functional requirement of DOA, because we can map the concepts and designs onto any underlying infrastructure, such as IPv6 protocols, CORBA, and so forth. Rather, it is an implementation requirement to facilitate integration with the Web services world.) (Requirement 9).

The rest of this chapter is structured as follows. The next section describes the hierarchical role/attribute-based access control model, where principals are given any attributes rather than simply roles, and these attributes are used to gain access to resources which are identified by their attributes. The section after that describes a Web services-based architecture of a DOA infrastructure that will allow any principal to delegate any attribute to any other principal, providing it is in accordance with the organisation's Delegation Policy. The following section describes the features that are needed in an organisation's Delegation Policy in order to allow principals to delegate their attributes to other principals. The penultimate section describes one practical implementation of this DOA Web services infrastructure, and shows screen shots of a browser interface that allows humans to delegate attributes to and revoke attributes from other humans. The final section concludes with a comparison of several other schemes and indicates where further research is still needed.

THE HIERARCHICAL RBAC/ABAC MODEL

Role-based access control (RBAC), our Requirement 3, was standardised by NIST and is now published as an American National Standard (ANSI, 2004). Figure 1 shows the ANSI RBAC model. Users are assigned roles via user assign-

Figure 1. The NIST/ANSI RBAC model



ments (UAs). A user may be assigned zero, one, or more roles. A user with zero roles currently assigned will not be able to access any protected objects.

A role may have zero, one, or more users assigned to it at any time, in order to cater for the natural migration of users between roles in an organisation. Roles are assigned permissions via Permission Assignments (PAs). A permission is the ability to perform an operation on a protected object or resource, for example, print to a laser-jet printer, or invoke a Web service. A role may have zero, one, or more permissions assigned to it. A permission may also be assigned to a set of roles, for example, in order to read certain files of project X, the roles of employee and member of project X are needed. A user obtains the permission to perform an operation on an object by being assigned the role or roles that has (have) the required permission(s) assigned to it (them). Users are statically constrained from having certain permissions by not being assigned the required roles. For example, in a bank the same person cannot usually audit transactions and be a teller, so static constraints will forbid the same user from being assigned both the teller and auditor roles.

In the ANSI RBAC model, roles may be organised in a hierarchy to suit the particular needs of an organisation. The reason for having a hierarchy is that senior or superior roles inherit the permissions assigned to their junior or subordinate roles, so that the permissions do not need to be explicitly assigned to the senior role. This simplifies permission assignment and provides a solution for Requirement 4. For example, say a project manager role is superior to a team leader role, and the team leader role has the permission *sign off project task* assigned to it. The project manager role automatically inherits this permission from the team leader role. Role hierarchies may be general hierarchies in which there is an arbitrary partial order between all the roles, or may be limited hierarchies in which some restrictions apply, for example, the hierarchy forms a tree structure or inverted tree structure. An organisation's general role hierarchy can be a disjoint set of several hierarchies, in which there is no single most superior role or most subordinate role. This allows limited permission inheritance to propagate between the roles.

By extending the ANSI RBAC model to include attributes of a user, such as age, name, and qualifications, in the role hierarchy, we can assign permissions to attributes as well as to roles, and migrate toward an attribute based access control (ABAC) model. Furthermore, by extending RBAC so that permissions can refer to operations on classes of objects identified by their attributes instead of operations on specifically named objects, we extend the migration to ABAC. Also by supporting resource class hierarchies, in which subordinate resource object classes inherit the attributes of their superior more generic object classes, we allow permission assignments to be inherited by subordinate object classes. For example, a permission assignment that says that users with the employee role can print on printers, through the process of role and resource object class inheritance we simultaneously allow managers (who are superior to employees) to print on laser jet printers (which are subordinate to printers).

In order to cater for dynamic constraints, in which a conflict of interest might arise if a user acts in multiple roles simultaneously, but not if a user acts in the roles independently, the concept of sessions is introduced. For example, say the *traveller* role is allowed to complete travel expenses claim forms and the *manager* role is allowed to authorise completed forms. People who are managers are usually travellers as well, but obviously managers are not allowed to authorise their own completed travel claim forms. Thus, a user cannot simultaneously act as a traveller and a manager in a session in order to complete a travel claim form and then authorise the completed form. When a user wishes to use the system he must activate one or more of his roles in a session, and the dynamic constraint will not allow him to activate conflicting roles in the same session. In the case of the travel claim request scenario, the user session starts when the travel claim form is opened and finishes when the user has finished accessing it.

Applying ABAC to Distributed Web Services

The ANSI RBAC standard says nothing about how roles are assigned to users or permissions are assigned to roles. In this chapter, we assume that each role and attribute has an administrative authority that controls the assignment of roles and attributes to users. We further assume that in a distributed environment there will be many such attribute authorities (AAs) that reside in different domains from each other and from the Web service that is being accessed. Consequently, it must be the Web service provider itself that decides who are the attribute authorities that it trusts to assign which roles and attributes to which users, and furthermore, what permissions to confer on each attribute and role. In this way, each Web service remains autonomous and in direct control of who is authorised to access its resources.

Roles (or attributes) are assigned to users in the form of attribute assertions, or attribute certificates (ACs), in which an issuer (an AA) asserts that a holder has a particular attribute. An AC is a digitally signed or "certified" attribute assertion. Each attribute assertion should contain: the name (or identity) of the holder, the attributes that have been assigned to the holder by the issuer, the name (or identity) of the issuer that is, the AA, and the period of time the assertion is valid. Attribute assertions may also optionally contain the policy rules of the issuer, for example, limiting the resources that the assertion may be presented to. The assertion may be digitally signed by the issuer to prove or certify that the contents are authentic. In a distributed environment the digital signature will always be necessary, unless a trusted path exists between the issuer and the consuming Web service. Examples of ACs are: a degree certificate issued to a graduate by a university, a state registered nurse certificate issued to a nurse by the Royal College of Nursing, an employee certificate issued to a member of staff by the employing organisation, and a project manager certificate issued to a person by a VO manager.

There are two standard formats for attribute assertions, or ACs. The first is the ISO/ITU-T X.509 Attribute Certificate format (ITU-T, 2005), and the second is the OASIS SAML attribute assertion format (OASIS, 2005). The primary difference between the two formats is that the former is a binary encoding of the assertion, while the latter is an XML text encoding; furthermore, the digital signature is mandatory in the X.509 AC format, and optional in the SAML attribute assertion format. Both formats are infinitely extensible to allow for bespoke tailoring by applications and issuers, for example, to add application specific policy rules. Both of these token formats can be used as authorisation credentials by users, and a Web services infrastructure should be able to cater for both of these formats as a minimum. If a user presents an X.509 AC or SAML attribute assertion to a Web service, the service should be able to determine from the attribute authorities that it trusts if the user has sufficient attributes (or roles) to be granted access to its resources.

A WEB SERVICES-BASED DELEGATION OF AUTHORITY ARCHITECTURE

The ISO Standard 10181-3 (ITU-T, 1995) provides a general architectural model for controlling access to networked resources (see Figure 2). In this model, the access control enforcement function (AEF) or policy enforcement point (PEP)—the terms are synonymous—intercepts an initiator's access request and asks the application independent access control decision function (ADF) or policy decision point (PDP)—again the terms are synonymous—if the initiator is allowed to perform the requested action on the target resource. The PDP examines the authorisation credentials of the initiator and consults its policy—which can be an ABAC or RBAC policy—to determine if the initiator has sufficient attributes (or roles) to be granted access to the target resource. From this evaluation it returns a granted or denied response to the PEP. The initiator's credentials may be provided by either the initiator in its access request, or the PDP can retrieve them itself from the issuer or a credential repository.

The architectural model (Figure 2) is ideal for controlling access to Web services. The Web service endpoint reference is the PEP that traps the user's service request. It then forwards the user's request to the PDP asking for an authorisation decision. The PEP and PDP can be collocated, or distributed, and communicate via an open protocol such as in Welch, Ananthakrishnan, Siebenlist, Chadwick, Meder, and Pearlman (2006). If the PDP returns granted, the user is allowed to consume the resources of the Web service, but if the PDP returns denied, the user's request will be rejected by the PEP. The complex task of deciding if the user has the correct set of attributes for the requested service is handed over to the application independent PDP to determine. It is the PDP that will decide if the user has been properly assigned or delegated the attributes that are asserted in the user's credentials according to the authorisation policy that is written by the Web service administrator.

Interestingly, we can also utilise the above model when creating a delegation of authority (DOA) Web service (see Figure 3). The DOA Web service will receive a delegation request from a delegator to delegate an attribute or attributes to a delegate. The delegator can be any Web ser-

Figure 2. X.812/ISO 10181-3 access control framework

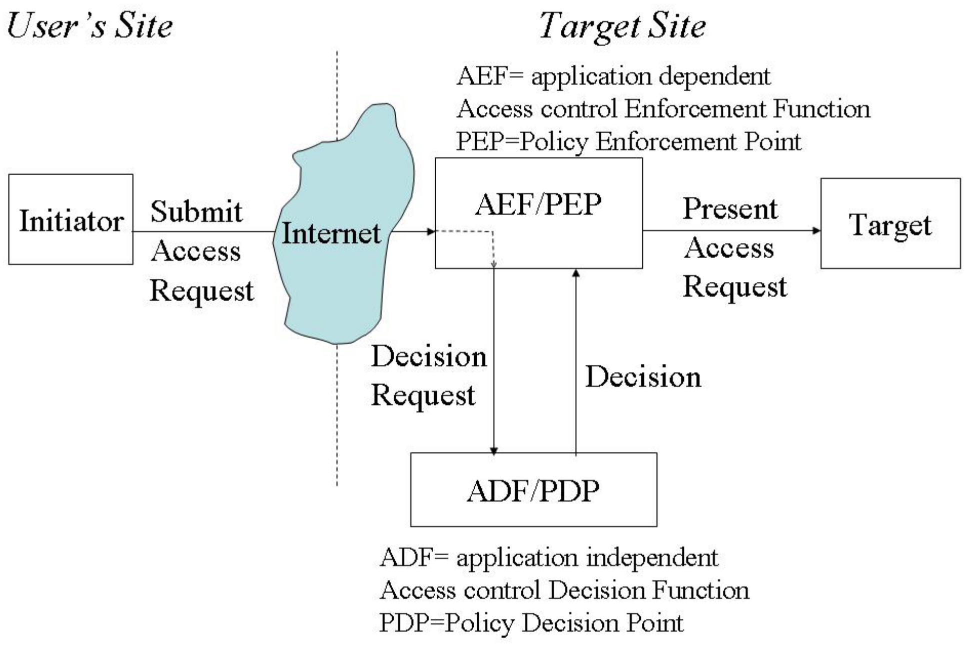
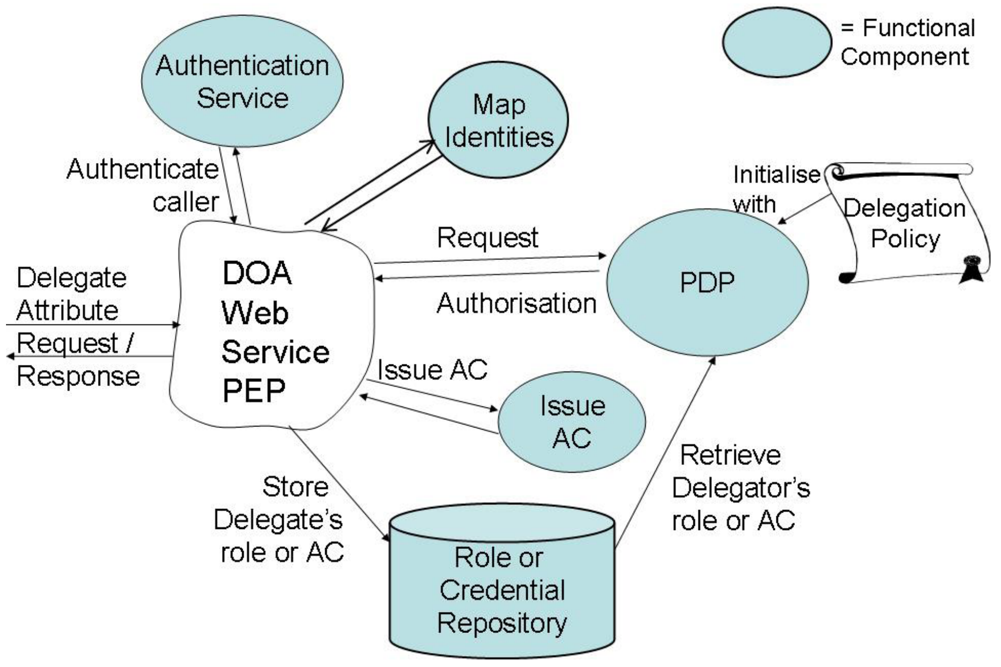

User's Site
Target Site
AEF= application dependent
Access control Enforcement Function
PEP=Policy Enforcement Point
Initiator
Submit Access Request
Internet
AEF/PEP
Present Access Request
Target
Decision Request
Decision
ADF/PDP
ADF= application independent
Access control Decision Function
PDP=Policy Decision Point

Figure 3. The delegation of authority Web service architecture



vice, or a human being acting via a Web services user interface. The delegate can be another Web service or another human being. In this way, we achieve the desired objective of person to person, service to service, person to service, and service to person delegation of authority (Requirement 1). The target resource is the Web service software that is able to issue an authorisation credential, in the form of an AC, for the delegate, on behalf of the delegator. This *issue AC* software should be capable of creating the attribute certificate in either X.509 AC or signed SAML attribute assertion format. This *issue AC* software should have its own digital signing key pair for this task, so that future credential recipients can verify that the issued credential is authentic. Because most users do not have their own PKI key pairs, they cannot issue their own ACs. This is why we require the DOA Web service to sign the credential on the delegator's behalf. This solves Requirement 7.

The delegator's request will be intercepted by the PEP, and passed to the PDP to ask if this user is allowed to delegate this/these particular attribute(s) to this delegate, according to the organisation's delegation policy (Requirement 5). The PDP retrieves the delegator's current set of authorisation credentials or roles/attributes from the local repository, and consults the delegation policy to see if the requested delegation is allowed or not. If the policy allows the delegator and delegate to be independently named, then this solves Requirement 2. As a result of evaluating the policy, the PDP replies granted or denied to the PEP. If granted, the PEP will ask the *Issue AC* software to issue a delegated authorisation credential to the delegate on behalf of the delegator, and will then either publish this in the local credential repository or return it to the requestor, or both. The delegate will now be able to use the issued credential to gain access to the service that has been delegated to him, and may also be able to further delegate the embedded attribute to other delegates, if allowed by the delegation policy. If the local repository stores delegated attributes instead of credentials, the *Issue AC* software will still create the delegated attribute(s) for the delegate, but not sign them, and the delegated attribute(s) will be stored in the repository. Subsequently, the delegate will be able to ask the DOA Web service to issue a new credential for him, based on the attributes that are stored for him in the local repository.

When a delegator makes a Delegate Attribute request to the DOA Web service, the delegator is first authenticated to determine who he or she is. Delegator authentication can be by any suitable means, and can be via an internal authentication service or external Web service. This model does not dictate any particular authentication scheme (Requirement 7). It is up to an implementation to determine the most appropriate authentication mechanism to use. That being said, digital signatures would be the most appropriate and secure mechanism for Web service to Web service authentication, but for authenticating a human user that is accessing the DOA Web service via a Web services user interface, a username and password stored in the local LDAP directory might be appropriate.

The next step is to optionally map the requestor's authenticated name into the authorisation name that is held in the authorisation credentials. This step is only needed if the two names are different, for example, when proxying is used (this will be described in more detail in the Implementation Section) or when the authentication mechanism uses a different name form to that stored in the issued credentials. Ideally this step should not be needed in the latter case, because the authenticated name should be held in the authorisation credential. If the mapping is needed, how this is performed is not part of the model, but care will be needed because a security vulnerability will be introduced if the mapping is not made in a secure manner.

Once the PEP has the delegator's authorisation name, it asks the PDP if this user is allowed to delegate this/these particular attribute(s) to the delegate. If granted is returned, the PEP then asks the target resource (*Issue AC*) to issue the new authorisation credential to the delegate, on behalf of the delegator. It then publishes the new credential in the repository or returns it to the requestor. If the delegate wishes to further delegate this credential to someone else, then the delegate will now take on the role of delegator and access the DOA Web service to request delegation of this/these attribute(s) to someone else. In this way, delegation can continue automatically from one user to another, providing, of course, that each delegation is in accordance with the organisation's delegation policy.

The model supports two different modes of operation, depending upon whether the repository stores credentials or attributes/roles. In both cases, delegation only takes places once, but credential issuing may take place zero, one, or more times. When the repository stores credentials, they are only issued once by the DOA Web service, they will typically have a relatively long lifetime (the period of the delegation), and they can be retrieved at will from the repository by users or by Web services that wish to validate the authority of a user to access its service. When the repository stores attributes/roles, the DOA Web service can be called repeatedly to issue typically short lived credentials based on the attributes/roles that have been delegated and stored in the repository. When the DOA Web service is only issuing already delegated attributes, the delegator's name is not required, only the name of the delegate. In both modes of operation the repository will need to record the validity period of the delegation and any policy conditions that are attached to it. If credentials are stored, this information is embedded in the issued credentials, if attributes are stored, separate fields will be needed in the repository to record it. When the repository stores attributes, it has to be strongly secured to prevent tampering with its contents and an attacker inserting false attributes. When the repository stores credentials, because the latter are digitally signed, it is not possible for an attacker to insert false credentials into the repository without first gaining access to the private signing key of the *Issue AC* service. Even if the repository is only weakly protected, the worst an attacker could do would be to remove a user's credentials, a denial of service attack.

The Advantages of a DOA Web Service

Here, we summarise the benefits of using a DOA Web service instead of each delegator issuing their own delegated credentials. Firstly, the DOA Web service can support a fully secure audit trail and a repository, so that there is an easily accessible record of every authorization credential/attribute that has been issued and revoked throughout the organization. If each delegator were allowed to independently issue their own credentials, then this information would be distributed throughout the organization, making it difficult or impossible to collect, being possibly badly or never recorded or even lost.

Secondly, the DOA Web service can be provided with the organization's delegation policy, and apply control procedures to ensure that a delegator does not overstep her authority by issuing greater permissions to delegates, or even to herself, than the organization's policy allows. For example, a delegator may have an attribute that they are allowed to delegate to others, but not allowed to assert themselves. Without proper controls a delegator may delegate the attribute to himself so that he is then allowed to assert the attribute. A well constructed delegation policy and PDP enforcement mechanism can ensure that this does not happen.

Thirdly, we don't get cascading revocations. In a traditional certificate chain, such as a PKI certificate chain, if any superior certificate in the chain is revoked, then all the subordinate certificates are also automatically revoked. Thus, if a delegator issued her own ACs, and her delegates then issued their own ACs, then if her AC was subsequently revoked, then all the delegates' ACs would also become immediately invalid. We typically don't want this to happen in an organization. For example, if a manager delegates various roles to members of staff in her department, and is then replaced and her role is revoked, we don't want all the delegated roles to be immediately revoked as well, or the department might grind to a halt. This does not happen with a DOA Web service. Because all the ACs are issued and signed by the DOA Web service, then a delegator's AC can be revoked without causing any of the delegate's ACs to be automatically revoked. Note however, if we record the name of the delegator in each issued AC, we are still able to implement cascading revocations if we require them.

Fourthly, the complexity of AC chain validation is significantly simplified. When delegators issue the ACs themselves, the AC chains can become arbitrarily long. When the DOA Web service issues the ACs to delegates, the AC chain length will always be a maximum of two, depending upon who the relying party trusts. If the relying party trusts the administrative authority that operates the DOA Web service and the former has delegated the issuing of ACs to the latter, then the chain length will always be two (trusted authority) $\leftarrow$ (AC of DOA Web service) $\leftarrow$ (AC of delegate). If the relying party trusts the DOA Web service as a root of trust, then AC chain lengths are reduced to just one, the AC of the delegate issued by the trusted DOA Web service.

Finally, a delegator does not need to hold and maintain her own private signing key, which would be needed if the delegator were to issue and sign her own ACs. Only the DOA Web service (the *Issue AC* component) needs to have an AC signing key.

The only disadvantage of using a DOA Web service is that the AC signing key must be permanently online and ready to be used to sign ACs when requested. In some highly secure systems and applications, this will be unacceptable.

Revocation of Authority

There are several different approaches that have been taken to the complex issue of revocation of authority, and of informing remote relying parties when revocation has taken place. Relying parties in our context refers to Web service providers who consume the issued credentials. The primary objective of revocation is to remove a credential (and all its copies, if any) from circulation as quickly as possible, so that relying parties are no longer able to use it. If this is not possible, a secondary objective is to inform the relying parties that an existing credential in circulation has been revoked and should not be used or trusted. The latter can be achieved by requiring either the relying parties to periodically check with the credential issuer, or the credential issuer to periodically notify the relying parties. Of these, requiring the relying parties to periodically check with the credential issuer is preferred, because it places the onus on the relying parties rather than on the issuer, because in general an issuer may not know who all the relying parties are, but the latter will always know who the issuer is.

The simplest approach, that used by X.509 proxy certificates (Tuecke et al., 2004), VOMS ACs (Alfieri et al., 2005), and SAML attribute assertions (OASIS, 2005), is to never revoke a credential, and instead to issue short lived delegation/ authorisation credentials that will expire after a short period of time and thus be effectively and automatically removed from circulation within a fixed period. The assumption in this case is that it is unlikely that authorisations will need to be revoked immediately after they have been issued and before they have expired. Because they are only valid for a short period of time, the opportunity to inflict damage through the illegitimate use of the authorisation credentials is short lived. Of course, the amount of damage that can be done in a short period of time can be huge, so short lived credentials are not always the best solution. Consequently, SAML attribute assertions also have the optional feature of containing a "one time use" element, which means that the consuming Web service can only use the attribute assertion once to grant access, and then it should never be used again. Instead, a new attribute assertion should be obtained from the attribute authority each time the user requests access to the Web service. This feature could be used in our DOA architecture, either at delegation time, in which case it would allow a delegator to delegate an attribute for one time use only by the delegate, or at issuing time (if attributes are stored in the repository) in which case the short lived ACs would be flagged for one time use.

An advantage of short lived credentials is that they effectively remove a credential from circulation after a short period of time, and consequently they mandate that users or service providers must frequently contact the credential issuer in order to obtain new freshly minted credentials.

The main disadvantage of short lived credentials is knowing how long to issue them for. They should be valid for the maximum time that anyone is likely to need them for, or otherwise one of the later steps of a user's task may fail to be authorised before the task has been completed, which could lead to the task being aborted and all the processing lost. This is a current well-known problem with proxy certificates. On the other hand, the longer they are valid, the greater their period of vulnerability to misuse without any direct way of withdrawing them from circulation. This has caused some researchers to suggest that proxy certificates should be revocable!

A second disadvantage of short lived credentials is that the bulk of the effort is placed on the issuer, who has to keep reissuing the short lived credentials. This could become a bottleneck to performance. A better solution should put the bulk of the processing effort onto the relying parties, because these are the ones who want to use the issued credentials.

A different approach to achieving the secondary objective of revocation is to notify the relying parties when revocation has taken place by issuing revocation lists. A revocation list is a digitally signed list of revoked credentials, usually signed by the same authority that issued the original credentials. Revocation lists have an expiry time and are updated and issued periodically. Relying parties are urged to obtain the next issue of the revocation list before the current one has expired, in order to keep as up to date as possible. The latest revocation list can be sent by the user along with his credentials, to prove that his credentials have not been revoked, or the relying party can independently download them from the issuer's repository. The use of certificate revocation lists (CRLs) is the approach standardised in X.509 (ITU-T, 2005) and is most frequently used by X.509 public key infrastructures. Revocation lists ensure that relying parties are eventually informed when a credential has been revoked, no matter how many copies of the credential there are in circulation, but revocation lists have several big disadvantages. Firstly, there is always some delay between a user's credential being revoked and the next issue of the revocation list appearing. This could be 24 hours or even longer, depending upon the frequency of issue of the CRLs. Thus, in order to reduce risk to a minimum, a relying party would always need to delay authorising a user's request until it had obtained the latest CRL that was published *after* the user issued his service request, which of course is impractical for most scenarios. If the relying party relies on the current revocation list, then the risk from using a revoked credential equates, on average, to half that of using a short lived credential, assuming the validity period of a short lived credential is equal to the period between successively issued CRLs. This reduced risk comes at an increased processing cost.

CRLs can put a significant processing load on both the issuer and the relying party. CRLs have to be issued at least once every time period, regardless of whether any credentials have been revoked or not during that period. In a large system the lists can get inordinately long containing many thousands of revoked credentials. These have to be reissued every time period, distributed over the network, and read in and processed by the relying parties. Delta revocation lists (ITU-T, 2005) have alleviated this problem, but again by increased processing complexity. Consequently, few people, if any, today are using revocation lists with authorisation credentials.

An alternative approach to notifying relying parties is to use the online certificate status protocol (OCSP) (Myers, Ankney, Malpani, Galperin, & Adams, 1999). Rather than a relying party periodically retrieving the latest revocation list from the issuer's repository, the OCSP allows a relying party to ask an OCSP responder in real time if a certificate (i.e., credential) is still valid or not. The response indicates if the certificate is good, or has been revoked, or its status is unknown. Because most OCSP responders base their service on the latest published revocation lists, the revocation status information is no more current than if the relying party had consulted the latest revocation list itself; thus the risk is not lessened. But what an OCSP responder does do is reduce the amount of processing that a relying party has to undertake in order to validate a user's credential/certificate. This reduced cost to the relying parties is offset by the cost of setting up and running the OCSP service.

We can see that none of the above approaches to revocation is ideal. Delegation of authority might last for a long period of time, especially when humans delegate roles that are meant to last for months or even years. We could issue long lived credentials, but the use of CRLs for revocation has many disadvantages. We could issue short lived credentials, but there is an inherent conflict between long lived delegation and short lived credentials that needs to be resolved. In the proposed architectural model this can be resolved by storing delegated attributes in the repository, along with the validity period of the delegation, and then repeatedly issuing short lived credentials as and when they are required until the delegation period has expired. Early revocation of the delegation is then achieved by removing the user's attributes from the repository. This approach is viable, but we are still left with the problem of determining the validity period of the short lived credentials.

Consequently, we propose an alternative scheme that we believe is superior to short lived credentials, CRLs, and OCSP servers. We believe that the optimum approach to credential issuing should have the following features. A user's credential should be issued just once and stored in the issuer's repository with its own unique URL. The credential should be valid for as long as the delegation is required, which can be a relatively long or short period of time. This minimises the effort of the credential issuer (and the delegator). A credential should be able to be used many times by many different service providers, according to the user's wishes, without having to be reissued. This mirrors the situation today with our plastic credit cards and other similar types of credential. A credential should be capable of being revoked at any time, and the revocation should be instant. This can be achieved by the issuer simply deleting the credential from its repository and requiring relying parties to contact the issuer's repository periodically, using the URL of the credential, to check if the credential is still present or has been revoked. This period can be determined by the relying party according to its risk mitigation strategy. This period can vary per application or per user request, and is set by the relying party as appropriate, and not by the issuer, which is putting the responsibility where it belongs. Ideally, a relying party should contact the repository when the credential is first used, and then periodically during the life of the authorisation session according to its own risk assessment. In order to strongly bind the repository to the credential, the credential's URL is embedded in the credential, so that the relying party knows where to go to check for the revocation status of the credential. This design minimises the processing effort of the issuers and the relying parties, because issuers do not need to continually mint new credentials, and relying parties do not need to process potentially large revocation lists. A secure network lookup, for example using TLS (Dierks & Allen, 1999) to bind to the repository URL, is all that is needed to ensure that a credential is still valid and has not been revoked. A simple bitwise comparison of the initial validated credential with subsequently retrieved copies is all that is needed to ensure that the credential is still the same one. Finally, there is little possibility of the credential expiring before the user's task has been completed, because it is likely to be long lived, which is not the case with short lived nonrevocable credentials.

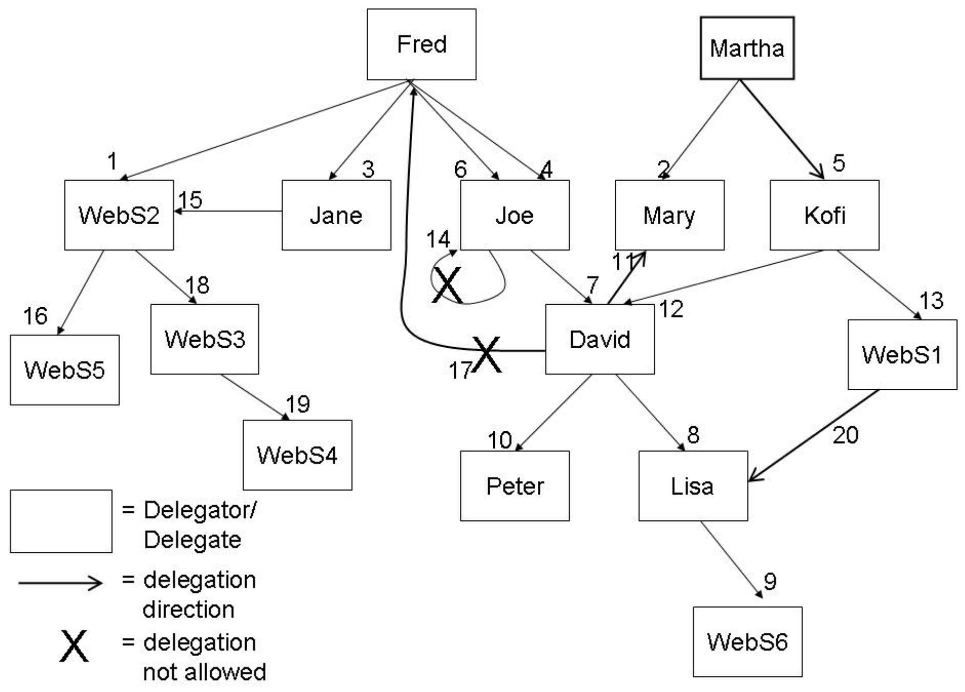
THE DELEGATION POLICY

In essence, the delegation policy needs to say who (i.e., the delegator) is entitled to delegate what (i.e., which roles and attributes and if fine grained delegation is also required, which tasks or permissions as well) to whom (i.e., the delegate), and under what constraints. The process of delegation forms a directed acyclic graph (DAG), with the initial attribute holders that is, initial delegators, as the sources of the graph (see Figure 4). Intermediate nodes in the graph represent delegates who subsequently act as delegators and further delegate their attributes (or permissions) to others. Sink nodes represent delegates who have not further delegated their attributes (or permissions) to others. Edges in the graph represent the attributes or permissions that have been delegated from the delegator to the delegate. Successor edges must always represent the same or less attributes and permissions than the union of their predecessor edges; otherwise a delegator will have delegated more privileges than he himself possessed. The graph is acyclic because a delegator should not be able to delegate to herself or to a predecessor (e.g., edges 14 and 17 in Figure 4). Rationally, there is a reason for this; a delegate should never *need* to delegate to an entity that previously delegated directly or indirectly to it. But there is also a security reason for this. There is a potential security loophole if a delegator, who is allowed to delegate a privilege but not to assert it, does subsequently delegate it to herself, as then she would be able to assert the delegated privilege (see later).

The delegation policy specifies the schema for this directed acyclic graph, thereby controlling which entities can be sources, sinks, and intermediate nodes, and what the attribute relationships between the nodes are.

A simplified form of the directed graph is a delegation tree, in which there is only one source or root node which holds all the attributes that can be delegated, and each act of delegation creates a separate delegate subordinate node. If a delegate receives attributes from two or more delegators in separate acts of delegation, such as edges 7 and 12 in Figure 4, then these are represented as separate edges and nodes in the tree, without merging the delegate nodes together. The purpose of this is

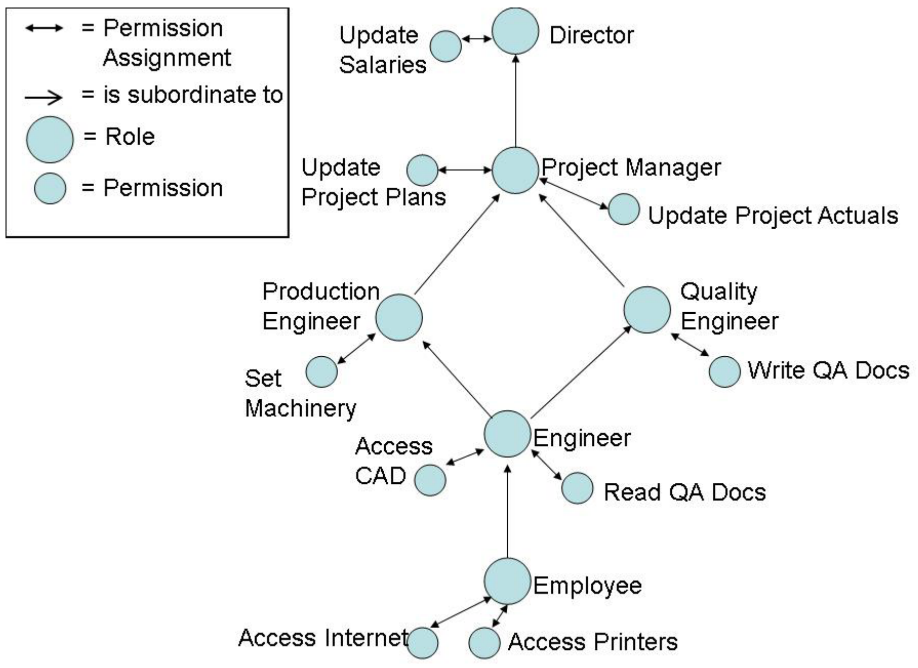
Figure 4. An example delegation directed acyclic graph



to forbid such a delegate from combining their various attributes together and delegating them to another delegate in a single act of delegation, such as in edges 8, 10, or 11 of Figure 4. Instead, multiple separate acts of delegation must take place, thereby maintaining the tree structure. The reason for this is that subsequent delegation and revocation become cleaner and easier to determine. In the case of delegation it is easier to prevent cycles from occurring. For example, in Figure 4, should the delegation from David to Fred take place in edge 17? The answer is no if it contains attributes from edges 6 or 4, but yes if it only contains attributes from edge 12. Consequently, determining which delegation is allowed and which is not can be quite complex in a DAG, but it is much easier in a tree. The process of revocation is to remove a delegation edge from the DAG and any consequential edges dependent upon the revoked edge. When delegation forms a tree, revoking an edge simply removes the whole subtree in a single act of revocation. With a DAG, there may be multiple incoming edges to a delegate node (from the same or different delegator nodes, as in edges 4 and 6 to Joe or 7 and 12 to David, respectively), and multiple outgoing edges to further delegates. If one of the incoming edges is revoked, the process of determining which outgoing edges and further delegate nodes should be deleted and which should remain becomes much more complicated. Thus, delegation trees significantly simplify delegation DAGs.

Concerning what can be delegated from a delegator to a delegate, this can be determined by reference to the role hierarchy. A delegator should be allowed to delegate any of the roles or attributes that he possesses or any of their junior roles from the role hierarchy. We have already described the role hierarchy in the fourth section, which specifies the partial order relationship

Figure 5. Combining permissions with the role hierarchy to determine what can be delegated



between the attributes and roles, but we can also add the permissions that each role or attribute has been granted into this hierarchy as well, making them the leaves of the delegation role hierarchy (see Figure 5). In this way the holder of an attribute (represented by a large circle in Figure 5) can delegate this particular attribute or any of its subordinate attributes from the role hierarchy or any of their associated permissions (represented by small circles in Figure 5), to a delegate in the DAG or delegation tree. For example, referring to Figure 5, a person holding the project manager role should be able to delegate this role, or any of its subordinate roles, for example, Quality Engineer, or any of the associated permissions, for example, Update Project Plans, to a delegate. This gives the delegator fine grained control over what he is able to delegate (Requirement 6). In the delegation DAG, a successor edge in the delegation graph must contain the same or less attributes/permissions than the union of its predecessor edges, with reference to the role and permissions hierarchy.

Note, however, that there is one significant difference between the roles and the permissions in Figure 5. The roles are assigned by the attribute authorities in one domain, while the permissions are assigned to the roles by the service providers in possibly different domains. Furthermore, different service providers may assign different permissions to the same role/attribute. For example, you might posses an American Express credit card, and find that it is not valid in one shop, is valid for any purchases in another shop, and is only valid for purchases over £5 in a third shop. The attribute has not changed, but the permission assigned to it has, according to the policy of the service provider. Thus, in order to achieve fine grained authorisation at the permission level, the attribute authority will need to closely liaise with the various service providers in order to add these permissions to its delegation role hierarchy. Note that we can achieve the same fine grained control over delegation if we create new uni-permission roles as the leaves of the role hierarchy, where each new uni-permission role is assigned just one of the permissions of the superior "real" role, for example, we can create an AccessPrinter role subordinate to the Employee role in Figure 5 to replace the Access Printer permission. This uni-permission role will not be assigned to a person initially, but it may be delegated to another entity dynamically. However, for this dynamic fine grained delegation of authority to work in a Web services world, the service provider that has assigned the permission to the role, for example, Access Printer to the Employee role, will now need to update its access control policy and add the new uni-permission role, for example, AccessPrinter to its role hierarchy. Service providers may be reluctant to make these changes to their RBAC policies, in which case permissions instead of uni-permission roles will need to be delegated.

There are additional policy rules that may need to be included in the delegation policy, such as: is a delegate allowed to delegate the credential again, that is, is the delegation process recursive or not, and if it is recursive, how many times can the delegation recurse, an infinite number of times or a limited number of times? We also need to consider if a delegator is empowered to assert the attributes and tasks that he is delegating, or is only allowed to delegate them, and if an attribute can be asserted, is there a control on where it can be asserted, that is, with only a subset of service providers? Consider, for example, an airline manager who is assigning a duty roster to pilots. The manager is delegating permission to fly an aircraft (say the "on flight duty" attribute) during certain periods of the day to pilots (the delegates). Clearly, the manager should not be able to invoke this permission himself, and empower himself to fly one of the aircraft, and thus the role (or task) may be delegated but not asserted by the airline manager. (Note that there is an alternative way of modeling this, by requiring a person who is authorised to fly an aircraft to have two attributes, say "on flight duty" and "qualified pilot," and to only give the airline manager permission to delegate the "on flight duty" attribute. Then, the airline manager would only be able to fly the aircraft if he was a qualified pilot. But in order to make our model flexible enough, we see that it is an advantage to have an assertion flag in our delegation policy.) A delegation policy may also contain conditions that a candidate delegate must fulfil before delegation can take place. For example, before a person can be delegated the fire officer role they must first have obtained a first aid certificate. Many of these conditions can be expressed in terms of attributes or roles a candidate delegate must possess before the new attribute or role can be delegated to them.

A flexible delegation policy language will allow the policy writer to specify the delegators and delegates by their attributes or roles, as well by specifically naming them. For example, we should be able to say "heads of department may delegate the fire officer role to members of staff within their department" as well as "Joe can delegate the fire officer role to David." The former allows whole groups of users to be delegators and delegates, the latter only allows specifically named individuals. In order for a person to delegate the fire officer role under the first policy rule, this person must have been assigned the assertable head of department role and the assertable or nonassertable fire officer role (depending upon whether he can act as a fire officer himself or only delegate this role) and the delegate must have been assigned the member of staff role and have the same department attribute as the delegator. In order for the latter policy rule to take effect, user Joe only needs to have been assigned the fire officer role and user David needs to exist. Note that the first policy rule on its own does not constitute a complete delegation policy. In order to be complete, a delegation policy must always specify which users are the delegation sources of authority in the DAG, that is, named individuals or services, and what attributes they are allowed to assign to whom. Otherwise, the PDP will not be able to determine if a particular delegation is allowed or not. For example, if we only have the former policy rule, and Joe attempts to delegate the fire officer role to Fred, then the PDP will not know if Joe is a head of department or not, or who is allowed to say that Joe is a head of department and who is allowed to say that Fred is a member of staff. If we only have the latter policy rule, the PDP will not know who is allowed to say that Joe is a fire officer. Without additional policy rules the PDP will not be able to determine if the delegator or possibly the delegate are bona-fide. Thus, delegation source of authority (SoA) policy rules are needed. These SoA rules may be completely general, and say, for example, that Person X is the trusted source of authority who may issue any credentials to anyone containing any attributes or permissions, or they may be much more specific and say, for example, that Person X is trusted to assign the head of department attribute to anyone in the organisation, but may not assert this attribute himself. With these SoA rules in place the PDP is then able to make authorisation decisions.

To summarise, a delegation policy needs to be able to:

1. Specify the delegation process in terms of a delegation directed acyclic graph (or a simplified delegation tree). This is done by specifying the rules for the delegation relationships that can exist between pairs of nodes in the DAG.
2. Identify the delegator and delegate nodes in the DAG by their attributes or roles or unique names/identifiers.
3. Specify trusted sources of authority of the DAG by their unique names/identifiers.
4. Specify what can be delegated in terms of an attribute/role hierarchy.
5. For very fine grained delegation optionally include the various attribute permissions as the leaf nodes in the attribute/role hierarchy.

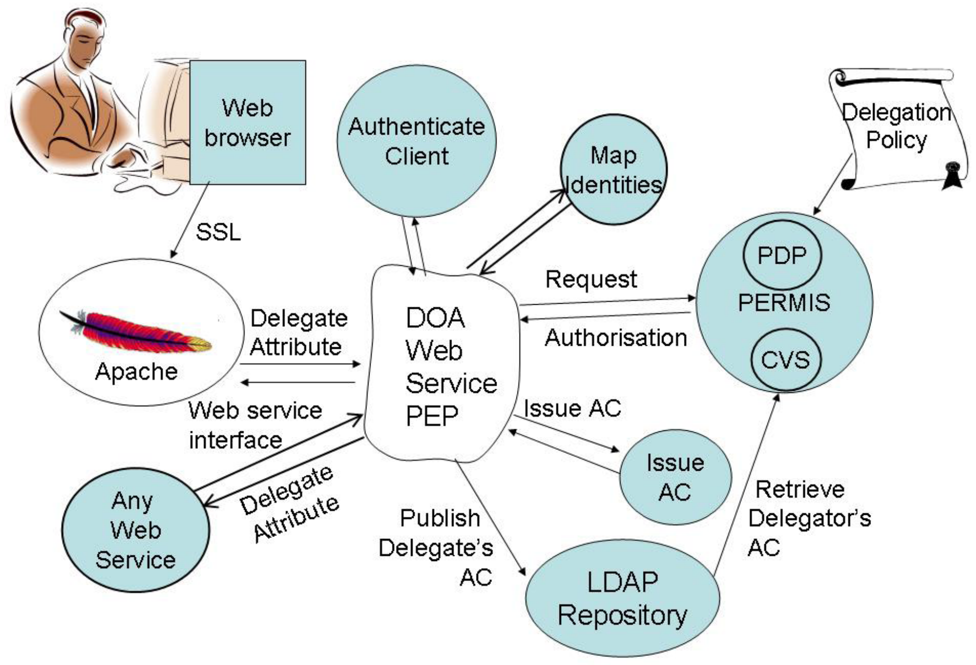
6. Specify whether delegator nodes in the DAG can or cannot assert the attributes that they are allowed to delegate.
7. Control the depth of the delegation graph (length of delegation chains).
8. Optionally specify other policy rules that can control when, where, or how delegates may assert the privileges that have been delegated to them.
9. Optionally place conditions on candidate delegates that must be fulfilled before delegation can take place.

IMPLEMENTING A PRACTICAL DOA WEB SERVICE

One can see that building a dynamic DOA Web service is reasonably complex and many competing choices have to be made. Primary choices are: should the DOA Web service repository store attributes or credentials? Should the issued credentials be short lived or long lived? If long lived, then how should revocation be performed? As stated in the fifth section, there are two main modes of operation that can be envisaged for the DOA Web service.

In the first mode of operation, the repository is an internally trusted component of the system and stores attributes rather than credentials. It is assumed that the repository cannot be tampered with by attackers, and therefore the attributes within it are safe. The DOA Web service issues short lived credentials to clients on demand. Consequently, no revocation is necessary. Users can delegate their attributes to other users according to the delegation policy. Clients can make repeated requests to the service for short lived credentials to be issued to users based on the attributes held by the users.

Figure 6. A practical delegation of authority Web service



In the second mode of operation, the repository is accessible to the outside world in read mode via secure links and stores relatively long lived credentials which are tamperproof. Users delegate these credentials to other users, and the DOA Web service has write access to the repository. Clients retrieve the credentials by contacting the repository directly using the URLs of the credentials. Credentials are revoked by removing them from the repository. Relying parties (service providers) must periodically check the repository to see if a credential is still there or not, according to their own risk assessments.

We have chosen to implement the second mode of operation because of its various advantages given in the fifth section. In our first version, we have used an LDAP server as the credential repository, and in the second version we are adding an Apache WEBDAV server (Goland, Whitehead, Faizi, Carter, & Jensen, 1999).

Delegation Policy Enforcement

The most complex and crucial component in a DOA Web service is the PDP that can support the organisation's delegation policy. The PDP essentially has two complementary functions to perform.

- Firstly, it must validate a delegator's claim to have the necessary set of attributes that it wishes to delegate and then validate if the chosen delegate has the necessary set of attributes to qualify as a delegate (this is the process of attribute or credential validation).
- Secondly, it must determine if the delegator is allowed to delegate these attributes to the chosen delegate (i.e., determine if the delegation request conforms to one of the delegation policy rules).

XACML (OASIS-2, 2005) is an OASIS standard for an access control policy language in XML, and an open source implementation of an XACML PDP exists, written by Sun, and available from http://sunxacml.sourceforge.net/. XACML provides a rich language for specifying who is allowed to do what. Access control subjects, resources, and actions are specified in terms of their attributes. If we make *the delegator* the access control subject and *the delegate* the access control resource, while *to delegate* is the access control action of an XACML access control rule, then an XACML PDP can decide if a delegator with a given set of attributes is allowed to delegate some of these attributes to a potential delegate who possesses another set of attributes. This is the second of the functions described above. Consequently, an XACML PDP should work very well in the first mode of operation where the repository stores user attributes, and the attributes do not need to be validated (because their presence in the trusted repository is sufficient to say they are valid).

However, XACML does not support credential validation, and therefore on its own cannot be used by either service providers that receive delegated credentials, or a DOA Web service that stores credentials instead of attributes. An XACML PDP works on the assumption that it is given a valid set of subject, resource, and action attributes upon which to make its access control decision. This can only work at a service provider site which directly trusts the issuers of all received credentials, so that there are no delegation chains to follow. An XACML PDP cannot determine if the credentials possessed by a delegated subject are valid or not. Consequently, an XACML implementation on its own is unable to enforce our delegation policy at the service provider site or in our DOA Web service that stores credentials, without significant enhancements, specifically the addition of a credential validation service (CVS). For this reason, we chose not to use an XACML PDP in our first implementation.

PERMIS (Chadwick & Otenko, 2003) is another open source PDP implementation that supports RBAC policies in XML. A PERMIS PDP comprises two components, a credential validation service (CVS) that validates users credentials, and a PDP that makes access control decisions. The PERMIS policy says who is entitled to assign which attributes to whom and whether delegation is allowed or not, as well as which attributes are needed to access which resources. Furthermore, PERMIS policies have an integer to control the depth of delegation. Thus, a PERMIS policy can be used to create an organisation's delegation policy, as well as enforce it as a service provider's site. PERMIS can be configured to either pull a user's credentials from an external repository, or to have them presented by the PEP, and so is ideal for our delegation scenario where the user does not have to present his existing credentials in order to request the delegation of attributes to a delegate. The credential format primarily supported by PERMIS is the X.509 attribute certificate, and so this is the format we adopted for our delegated credentials. LDAP repositories support the storage and retrieval of X.509 attribute certificates, and so we chose to use LDAP as our credential repository. PERMIS also supports the no assertion flag and does not allow delegators to delegate attributes to themselves.

One of the limitations of PERMIS is that its delegation policy does not support the specification of delegators and delegates by any of their attributes, but rather only by the naming domains of which they are members. Naming domains are specified using LDAP/X.500 distinguished names. This means that we cannot specify a delegation policy such as "heads of department can delegate the fire officer role to senior members of staff in their department." Instead, we have to name the individual heads of department, and specify the naming domain that potential delegates reside in, for example, cn=John Smith,o=myorg,c=gb can delegate the fire officer role to principals who are from the naming domain "ou=deptA,o=myorg, c=gb." This means our PERMIS delegation policies will be more restrictive or less efficient than ones we could write in the XACML language, but we are able to fully enforce them, while with XACML we can write richer delegation policies but we are not able to fully enforce them because XACML cannot validate (delegated) credentials. Consequently, in our first DOA Web services implementation we chose to use PERMIS on it own, but in the next implementation we plan to investigate the combination of the PERMIS CVS functionality with the XACML policy decision functionality.

Client Access

Our implementation of the DOA Web service is written in Java, and runs inside a Tomcat application server and Apache AXIS SOAP server. Consequently, it can be invoked through SOAP calls. The DOA Web service (actually the containing Tomcat server) has its own X.509 public key certificate, and requires the requesting Web service to have one as well. These certificates are used to open a secure SSL (https) connection with the DOA Web service using mutual authentication. All other types of authentication method or connection are rejected. We chose to use SSL certificate-based mutual authentication rather than XML signed SOAP messages due to SSL's superior performance and ubiquity. The SSL client must either be the entity directly making the request (i.e., the requestor), or a trusted proxy acting on its behalf. In the latter case the name of the requestor is taken from the first parameter of the Web services operation (except the storeACforMe and revokeACforMe operations, which cannot come from a trusted proxy). The names of the trusted proxies are read in at initialisation time from a configuration file. We have implemented an Apache server as a trusted proxy, using LDAP username-password authentication of the users, and this will be described later.

The DOA Web service publishes a standard WSDL file that allows other Web services to determine how to access its services. It supports five operations:

- *delegateForMe*, whose arguments are: the distinguished name (DN) of the requestor (the delegator), the DN of the delegate, the attributes to be delegated, the validity time of the delegation (from and to), whether the delegated attributes can be asserted or not (yes/no), and how many more times the attributes can be delegated (the delegation depth, an integer). If the delegator is allowed to delegate this attribute to this delegate, an X.509 AC is created, with the DOA Web Service set as the credential issuer and the delegator's name placed in the IssuedOnBehalfOf field. The latter is a standard X.509 AC extension defined in the 2005 edition of X.509.
- *revokeForMe*, whose arguments are: the distinguished name (DN) of the requestor and the set of credentials that should be revoked. Each credential is identified by the DN of the holder, the DN of the issuer, and the serial number of the credential. This method allows the requestor to revoke many credentials at the same time (in one request). The DOA Web Service has built in rules for who is allowed to request the revocation of a credential. The allowed revokers are: the holder of the credential (i.e., the delegate himself), the issuer of the credential (which is usually the DOA Web Service but could be the delegator), who the credential was issued on behalf of (usually the delegator but could be blank), the source of authority of the delegation graph, or anyone who could have issued this credential. The rationale for allowing the latter category of revocation requestor was purely one of expediency. It was reasoned that if revocation was deemed to be necessary, then it should be able to be done fast by anyone in authority, in order to minimise the risk of damage from use of an unauthorised credential. If a user has the authority to issue a credential, and could have issued it, even though she did not actually issue it, then she should still be allowed to revoke it. While this does provide a minimal chance for a denial of service attack by a person in authority, the risk from this was deemed to be less than allowing a credential that should be revoked to remain in circulation longer than it should have been, say because the actual delegator was not available to revoke it.
- *storeACforMe*, whose argument is a fully formed digitally signed X.509 AC, sent as a base64 encoded string. This is a repository service for an external user who has the ability to sign and issue credentials herself, but does not have write access to the credential repository. The requestor must be authenticated via SSL client authentication and have the same name as the issuer of the credential. The DOA Web Service checks if the issuer is allowed to delegate this credential according to the delegation policy, and if so, stores the credential in the delegate's LDAP entry in the repository and reports success to the requestor. Otherwise, it reports unauthorised to the requestor and discards the credential.
- *revokeACforMe*, whose argument is the X.509 AC that is to be revoked, encoded as a base64 string. The DN of the requestor is taken from the client certificate of the established SSL connection and must be one of the allowed revokers, according to the rules presented in *revokeForMe* above; otherwise the revocation request is rejected. If the requestor is allowed to revoke the credential, then the AC is removed from the delegate's LDAP entry.
- *searchRepository*, whose arguments are the DN of the requestor and the DN of the delegate. The authenticated SSL client must be the trusted proxy or the requestor. This

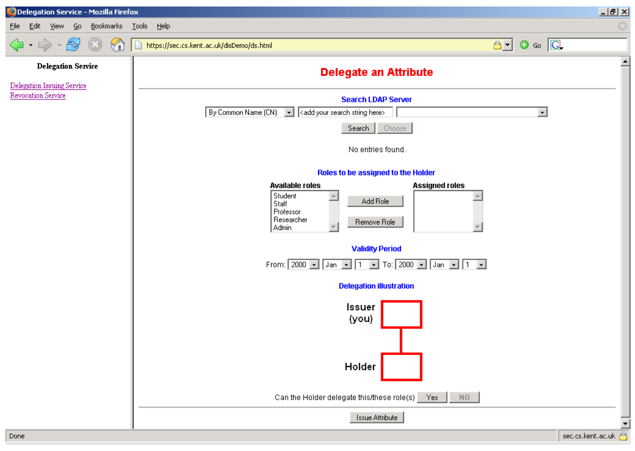
service searches through the repository for credentials issued to the delegate. The service then checks if the requestor is authorised to view the retrieved credentials. This is determined from a configuration parameter, which can be set to either *anyone* or *revokers*. If *anyone*, all the retrieved credentials are returned, and there is no privacy protection on viewing a user's credentials. If *revokers*, each credential is checked to see if the requestor is allowed to revoke it, using the same rules as in *revokeForMe* above. The procedure removes from the result all those credentials that the requestor is not authorised to revoke. In this way, the privacy of the delegate's credentials is protected, because only those requestors who are authorised to revoke the credentials are allowed to search for them and retrieve them.

When Apache is acting as a trusted proxy on behalf of a human delegator, the human is presented with the Web page shown in Figure 7. In order to access this page, the user must first be authenticated by Apache. Any type of authentication supported by or plugged into Apache can be used. We have chosen to use standard Apache LDAP authentication, using usernames and passwords stored in our organisation's LDAP server, because this is the authentication mechanism used by all our users to access the university's network and services. The displayed delegation page invites the user to search through the organisation's LDAP service to find the user he wishes to delegate to, for example, be entering the surname. A picking list of users who match the entered criteria is displayed, and the user chooses the correct person. The user then selects the attributes that he wishes to delegate to this person, fills in the validity time of the delegation (from and to), and can then choose if the person should be allowed to further delegate these attributes or not. If further delegation is selected, the user can set the depth of further delegation and choose between allowing or forbidding the user to assert the roles that have just been delegated to him. Finally, the user presses the Issue Attribute button and if everything is in accordance with the delegation policy, the delegation is allowed. If the user has tried to do something counter to the delegation policy, one of two things might happen, according to the *downgradeable* configuration parameter of the delegation service. If the infringement is minor, for example, setting the validity period too long, and *downgradeable* is true, the delegation is still allowed to go ahead but the user's parameters are overridden by ones that conform to the policy. If *downgradeable* is false, or the infringement cannot be downgraded, for example, the delegator is trying to delegate to someone not allowed by the policy, then the delegation is rejected.

The delegator (and all other allowed revokers) can revoke the issued credential at any time by entering the Revocation Service Web page. Again, the requestor must be authenticated by Apache before the revocation page is displayed. Upon entering the revocation page, the requestor is again invited to search through the organisation's LDAP service to find the delegate he wishes to revoke an attribute from. After entering the search criteria, a list of users is displayed. Upon choosing one of them, the system invokes the *searchRepository* Web service and one of three responses will be displayed, either: a list of the user's credentials that are visible to the requestor, or a message saying that this user does not have any attributes, or an error message saying that the requestor is not allowed to search and view this user's attributes.

This DOA Web service has been piloted in various grid applications by the National e-Science centre at the University of Glasgow, and details of these trials can be found in Sinnott, Stell, Chadwick, and Otenko (2005), Sinnott, Watt, Jiang, Stell, and Ajayi (2006), and Watt, Sinnott, Jiang, Ajayi, and Koetsier (2006).

Figure 7. Web-based front end to our delegation service



Delegation Service - Mozilla Firefox

File Edit View Go Bookmarks Tools Help

https://sec.cs.kent.ac.uk/disDemo/ds.html

Delegation Service

Delegation Issuing Service
Revocation Service

Delegate an Attribute

Search LDAP Server

By Common Name (CN) <add your search string here>

Search Choose

No entries found.

Roles to be assigned to the Holder

Available roles: Student, Staff, Professor, Researcher, Admin

Add Role Remove Role

Assigned roles

Validity Period

From: 2000 Jan 1 To: 2000 Jan 1

Delegation illustration

Issuer (you)

Holder

Can the Holder delegate this/these role(s) Yes NO

Issue Attribute

Done

sec.cs.kent.ac.uk

CONCLUSION AND FUTURE TRENDS

Comparison with Other Work

VOMS (Alfieri et al., 2005) is a Web services-based credential issuing service, but it is not a delegation service. It only implements part of the model specified in the fifth section, specifically the *Issue AC* and repository services. A user can make repeated requests to a VOMS service, for it to issue short lived X.509 attribute certificates derived from a subset of the attributes held in the user's repository entry. The user must be in possession of an X.509 public key certificate (PKC) in order to utilise the VOMS service, because the holder field of the credential points to the public key certificate of the user (PKC issuer and serial number) rather than the distinguished name of the user. The repository holds the various attributes of the users, but these can only be inserted into the repository by the VO manager. Users are not able to delegate their attributes to other users. They must ask the VO manager to insert attributes into other user's entries for them. The VOMS service therefore places a high administrative and maintenance load on the VO manager, because he is responsible for all delegations and revocations, and this task cannot be dynamically delegated to the VO users.

Signet (McRae, Nguyen, Cohen, & Vine, 2004) and Grouper (see *http://middleware.internet2.edu/dir/groups/grouper/)* from the Internet2 consortium are developing software that will allow users to assign permissions and delegate privileges between each other. The system is architecturally much simpler than the one depicted in Figure 3. It is designed for human users and is Web server rather than Web services based. The user interface is any standard Web browser, and the server functionality is written as Java servlets and jsp which can run in any container such as Tomcat. The repository is a RDBMS with SQL interface which stores a user's group memberships (or roles) and individual permissions (for fine grained control) along with their validity times and other policy related information such as prerequisites before a privilege can be granted and conditions on its use after it has been granted. Consequently, there is no PDP holding the delegation policy as a separate entity. Rather, the policy is distributed throughout the repository in the various tables. Signet does not issue credentials, and this functionality has to be provided by an external plugin that retrieves and packages the data from the repository in an appropriate way, for example, as a SAML assertion.

Work has been on going since 2004 in OASIS to add support for delegation of authority to XACMLv2 (OASIS, 2007). This work is designed to allow the setting of access control policies to be delegated between administrators, and is complementary to the DOA work described here. Unfortunately, the DOA work in XACML has progressed rather more slowly than originally anticipated, and at the time of writing it is not clear what the outcome will be. In parallel with the OASIS work, we devised a mechanism whereby the PERMIS CVS could be incorporated with a XACMLv2 PDP at a service provider site, in order to provide valid attributes to the PDP from delegated credentials. The valid attributes can then be fed into the XACML PDP for it to make access control decisions. This work is described fully in Chadwick, Otenko, and Nguyen (2006).

Future Work

We have developed a secure audit Web service (SAWS) which allows events to be securely audited in a tamperproof log (Xu, Chadwick, & Otenko, 2005). We propose to incorporate this into a future version of the DOA Web service so that every delegation decision can be securely logged for future reference. This might be important, for example, when trying to retrospectively trace how a person became authorised, or when he was revoked and by whom.

We are currently building a WEBDAV repository to replace the existing LDAP repository, so that individual credentials can be uniquely identified by their URLs. A disadvantage of using LDAP repositories is that a URL can only usually refer to all the credentials of a particular user, rather than to individual credentials. This is because a set of ACs are usually all held together as a set of values within a single LDAP attributeCertificate attribute. This makes it impossible to retrieve a single credential of a user.

We are currently adding the ability to perform fine grained delegation based on individual permissions rather than attributes. As pointed out above, a number of complexities are introduced when this occurs in a multiple domain environment, due to the fact that a permission that is understood and valid in one domain may not be recognised in another domain. We are addressing this problem at the attribute level by adding role/attribute mappings to our service provider PDP policies. This will allow an attribute that is issued in one domain to be recognised in a service provider domain. Extending this mapping to permissions would allow fine grained authorisations to be understood between domains.

While oursystem currently only supports credentials in X.509 attribute certificate format, it will be relatively easy to add signed SAML attribute assertions as well due to the modular construction of PERMIS. Once the performance of signed SAML assertions improves, this addition will be made.

Finally, we are investigating how best to combine the PERMIS CVS functionality with the XACML policy decision functionality to allow richer delegation policies to be specified through the identification of delegators and delegates by their attributes rather than by their membership of a specific domain.

As a general trend, we expect to see more Web-based interfaces being gradually introduced to allow users to delegate authority to other users, and more willingness on the side of administrations to empower users to delegate among themselves, providing they can specify adequate delegation policies to control this. We also expect to see users and Web services dynamically delegating authority to subordinate Web services to do work on their behalf, so that work flows can be automated and distributed throughout and between organisations. We also expect to see much richer functionality to be gradually introduced into the Web front ends and the back end delegation policies. We have taken the first tentative steps along this path, by allowing dynamic delegation of authority between users and Web services, and we fully expect more sophisticated and richer mechanisms to follow.

ACKNOWLEDGMENT

We would like to thank the UK JISC for funding this work under the DYVOSE project.

REFERENCES

Alfieri, R., Cecchini, R., Ciaschini, V., Dell'Agnello, L., Frohner, A., Lorentey, K., & Spataro, F. (2005). From gridmap-file to VOMS: Managing authorization in a Grid environment. *Future Generation Computer Systems, 21*(4), 549-558.

ANSI (2004). Information technology: Role-based access control. ANSI INCITS 359-2004.

Chadwick, D.W., & Otenko, A. (2003). The PERMIS X.509 Role-based privilege management infrastructure. *Future Generation Computer Systems, 19*(2), 277-289.

Chadwick, D.W., Otenko, S., & Nguyen, T.A. (2006, October 19-21). Adding support to XACML for dynamic delegation of authority in multiple domains. In *Proceedings of the 10th IFIP TC-6 TC-11 International Conference, CMS 2006,* Heraklion, Crete, Greece (pp. 67-86). Springer-Verlag.

Dierks, T., & Allen, C. (1999). The TLS Protocol Version 1.0, RFC 2246.

Goland, Y., Whitehead, E., Faizi, A., Carter, S., & Jensen, D. (1999). HTTP extensions for distributed authoring – WEBDAV. RFC 2518.

ITU-T. (1995). Security frameworks for open systems: Access control framework. ITU-T Rec X.812 | ISO/IEC 10181-3:1996.

ITU-T. (2005). The directory: Public-key and attribute certificate frameworks. ISO 9594-8 (2005) /ITU-T Rec. X.509.

McRae, L., Nguyen, M., Cohen, A., & Vine, J. (2004). Signet functional requirements. Retrieved June 3, 2007, from http://middleware.internet2.edu/signet/docs/signet_func_specs.html

Myers, M., Ankney, R., Malpani, A., Galperin, S., & Adams, C. (1999). X.509 Internet public key infrastructure: Online certificate status protocol – OCSP, RFC 2560.

OASIS. (2005). Assertions and protocol for the OASIS Security Assertion Markup Language (SAML) V2.0, OASIS Standard.

OASIS. (2007). XACML v3.0 Administrative Policy Version 1.0, working draft 15. Retrieved June 3, 2007, from http://www.oasis-open.org/committees/tc_home.php?wg_abbrev=xacml

OASIS-2. (2005). eXtensible Access Control Markup Language (XACML), Version 2.0. OASIS Standard.

Sinnott, R.O., Stell, A.J., Chadwick, D.W., & Otenko, O. (2005). Experiences of applying advanced grid authorisation infrastructures. In *Proceedings of the European Grid Conference (EGC),* Amsterdam, Holland.

Sinnott, R.O., Watt, J., Jiang, J., Stell, A.J., & Ajayi, O. (2006). Single sign-on and authorization for dynamic virtual organizations. In *Proceedings of the 7th IFIP Conference on Virtual Enterprises, PRO-VE 2006*, Helsinki, Finland.

Tuecke, S., Welch, V., Engert, D., Pearlman, L., & Thompson, M. (2004). Internet X.509 Public

Key Infrastructure (PKI) proxy certificate profile. RFC3820.

Watt, J., Sinnott, R.O., Jiang, J., Ajayi, O., & Koetsier, J. (2006). A Shibboleth-protected privilege management infrastructure for e-science education. In *Proceedings of the 6th International Symposium on Cluster Computing and the Grid, CCGrid2006*, Singapore.

Welch, V., Ananthakrishnan, R., Siebenlist, F., Chadwick, D., Meder, S., & Pearlman, L. (2006). Use of SAML for OGSI Authorization, GFD.66. Retrieved June 3, 2007, from http://www.ggf.org/documents/GFD.66.pdf

Xu, W., Chadwick, D., & Otenko, S.(2005). A PKI-based secure audit Web service. IASTED Communications, Network and Information Security CNIS, November 14 - November 16, Phoenix, AZ.