Solving the Transitive Access Problem for the Services Oriented Architecture

Alan H. Karp, Jun Li

HP Laboratories
HPL-2008-204R1

Keyword(s):
SOA, web services, access control, RBAC, PBAC, ABAC, ZBAC

Abstract:
A key goal of the Services Oriented Architecture is the composition of independently written and managed services. However, managing access to these services has proven to be a problem. A particularly difficult case involves a service that invokes another service to satisfy an initial request. In a number of cases, implementations are able to achieve either the desired functionality or the required security, but not both at the same time. We say that this service composition suffers from the transitive access problem. We show that the problem arises from a poor choice of access control mechanism, one that uses authentication to make access decisions, and that the problem does not occur if we use delegatable authorizations.

External Posting Date: November 21, 2008 [Fulltext] Approved for External Publication
Internal Posting Date: November 21, 2008 [Fulltext]

Submitted to ACM Symposium on Access Control Models and Technologies, Stresa, Italy, June 3-5, 2009

Solving the Transitive Access Problem for the Services Oriented Architecture

Alan H. Karp
Hewlett-Packard Laboratories
1501 Page Mill Road
Palo Alto, CA 94304
650-857-3967
alan.karp@hp.com

Jun Li
Hewlett-Packard Laboratories
1501 Page Mill Road
Palo Alto, CA 94304
650-857-4087
jun.li@hp.com

ABSTRACT

A key goal of the Services Oriented Architecture is the composition of independently written and managed services. However, managing access to these services has proven to be a problem. A particularly difficult case involves a service that invokes another service to satisfy an initial request. In a number of cases, implementations are able to achieve either the desired functionality or the required security, but not both at the same time. We say that this service composition suffers from the *transitive access problem*. We show that the problem arises from a poor choice of access control mechanism, one that uses subject authentication to make access decisions, and that the problem does not occur if we use delegatable authorizations.

Categories and Subject Descriptors

K.4.4 **[Computers and Society]:** Electronic Commerce - Security
H.3.5 **[Information Storage and Retrieval]:** Online Information Services - Web-based services

General Terms

Security

Keywords

SOA, web services, access control, RBAC, PBAC, ABAC, ZBAC

1. INTRODUCTION

The Services Oriented Architecture (SOA) promises large gains in productivity by providing a means to assemble independently written and managed web services to satisfy a request. These gains have yet to be achieved, in part due to problems with controlling access to the services. A particularly difficult case occurs when an invoked service invokes another service to satisfy the initial request. This paper describes an approach to solving this problem that satisfies both the desired functionality and security requirements while conforming to the web services standards.

A real-world example of this problem comes from the Consolidated Afloat Network and Enterprise Service (CANES) program of the US Navy [27]. The goal of CANES is to standardize on a common set of hardware for all ships and unify software around a set of SOA based services.

Figure 1 shows a version of an important use case taken from the Net-Centric Enterprise Architecture specification [8] developed as part of CANES. The End User is in the battlefield with a machine that is not fully trusted because it is subject to loss, destruction, or capture. The component denoted "Mars Portal" runs in a more trusted environment and creates process denoted "WS Client" to act on behalf of the user.

In this scenario, the user, who we call Alice, is requesting a weather forecast for her ship. Her request goes to a Forecast service that is responsible for identifying the appropriate weather service to use. For example, Forecast may select a US service for predictions for the eastern Pacific or a UK service for a ship in the eastern Atlantic. The Forecast service then invokes the Weather service.

The transitive access problem arises because we have to decide which credentials get used when the Weather service is invoked. If we use the Forecast service's credentials, Alice might ask for something that the Forecast service is allowed to do but Alice is not, which makes the Forecast service a Confused Deputy [18]. We can address this problem by having the Weather service only accept requests signed by Alice, but that limits the value an intermediate service can add. If we use Alice's credentials, the Forecast service could ask for something Alice is allowed to do but doesn't want done. In other words, the Forecast service is able to impersonate Alice. We can address that problem by making the Forecast service fully trusted. With this approach, every service in a chain must fully trust every downstream service, even services it has never heard of. It's hard to argue that this large a violation of the Principle of Least Privilege [34] meets any reasonable definition of security.

These issues have been observed in practice. For example, in a Limited Technical Experiment (LTE) completed in February of 2008, civilian employees of the US Navy implemented a variation of the use case in Figure 1 and found that they could have either the desired security or the required functionality, but not both at the same time.

Although SOA is a framework that can be implemented using many different technologies, here we will consider only approaches based on the web services standards, SOAP for message transport, WSDL to specify the service interface, UDDI for service description and discovery, and SAML [30] for communicating security related information, all of which are expressed in

Submitted to *ACM Symposium on Access Control Models and Technologies*, June 3-5, 2009, Stresa, Italy.

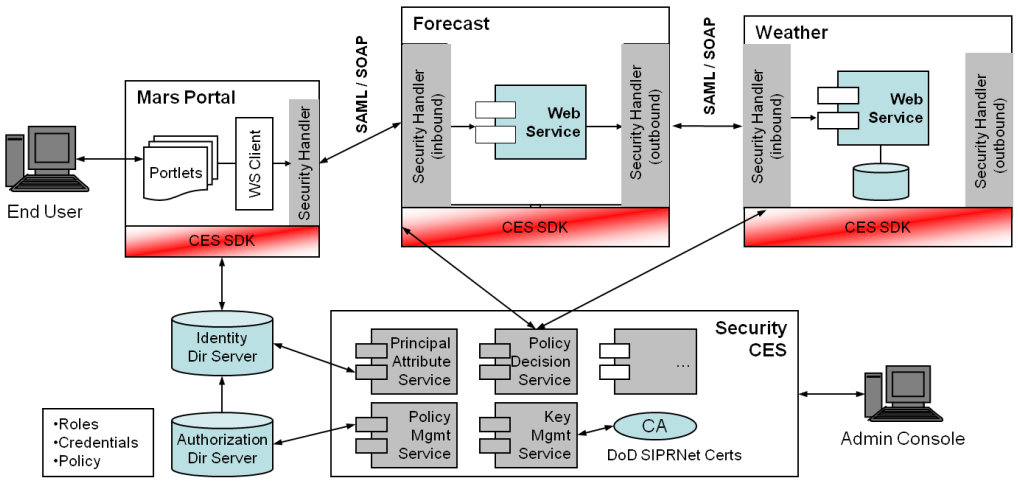

Figure 1. CANES use case showing service chaining.

XML. SAML is the most relevant of these here because access control is the focus of this paper.

A SAML assertion specifies a subject and the information being asserted, which is any combination of identity, attributes, and authorization. These assertions should only be accepted as valid if issued by a trusted source and submitted by the designated subject. In our examples, we verify the issuer's signature on the assertion and the submitter's proof of knowledge of the private key corresponding to the public key in the <Subject> field of the assertion.

The implementation for the CANES LTE was based on the Transited Provider pattern of the Liberty Alliance SAML Profile [19]. As illustrated in Figure 2, Alice presents her authentication to a trusted third party (TTP), which returns a SAML identification assertion. She constructs a SOAP message with this assertion in the header and a SOAP body specifying her request. She then signs the entire SOAP message and sends it to the Forecast service.

The Forecast service verifies the signatures on the SOAP message and the identity assertion and makes an access decision based on the specified identity. If access is allowed, the Forecast service sends its authentication information to the TTP asking for a Transited Provider assertion, such as

```
<TransitedProviderPath>
    <TransitedProvider>
        http://www.canes.gov/Forecast
   </TransitedProvider>
</TransitedProviderPath>
```

This assertion is not strictly required in this simple case because the Weather service is able to identify the Forecast service from the signature on the request. The Transited Provider assertion is needed for a longer chain, in which case an entry is added to the <TransitedProviderPath> for each additional service in the chain.

The Forecast service constructs a SOAP message, the body of which specifies its request and a header having both Alice's identity assertion and its Transited Provider assertion. The Forecast service then signs the SOAP message and sends it to the selected Weather service. The Weather service uses both the Alice's identity assertion and the Forecast service's Transited Provider assertion to make an access decision.

The Liberty Alliance documentation doesn't say how the Weather service should use those authentications to make that decision. An approach proposed for DCE [11] is one possibility. If the Weather service uses only the Forecast service's authentication, the Forecast service can become a confused deputy. If the decision is made based on Alice's authentication, then the Forecast service is able to use or abuse any of Alice's permissions. In the cited LTE, the implementation used the union of these sets of permissions, which has both of these vulnerabilities.

The scenario in Figure 1 is both too complicated and too simple for our purposes. It is too complicated because there are a large number of components to consider, most of which are not relevant to the transitive access problem. The use case is too simple because it involves only participants belong to the same security domain. Often, there will be two or three different organizations involved, a circumstance that only makes the trust assumptions more critical. More importantly, the use case doesn't capture the power of SOA because it doesn't involve passing references to services as parameters or returning a service reference.

This paper is organized as follows. In Section 2, we'll introduce an example that captures the issues faced by those implementing the CANES use case without extraneous components and that extends the example to include passing service references as parameters. Implementations of the CANES use case to date have based access decisions on the requester's authentication. Section 3 describes a number of security weaknesses of this approach. Section 4 describes these implementations and why they failed to achieve their goals of security or functionality. In Section 5, we introduce an access control approach based on explicit, delegatable authorizations. Section 6 describes implementation strategies and how we apply that strategy to the use case. Section 7 describes how we avoid inadvertent violation of access policies, and Section 8 describes related work.

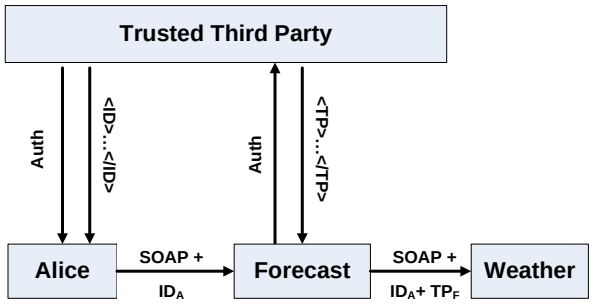

Figure 2. Use of the Transited Provider SAML profile.

2. SCENARIO

While the CANES example comes from a real deployment, it does not show the full range of access control issues that arise in service composition. The very simple set of services shown in Figure 3 captures the essential requirements.

Bob offers a Backup service with a backup method having a signature

```
ServiceRef backup(ServiceRef inRef)
```

where *ServiceRef* is a type used to denote any kind of invocable web service. For example, this type would be *EndPointReference* in WSRF [17]. A program running on Alice's behalf invokes this service

```
ServiceRef bRef = b.backup(fileRef)
```

Here *fileRef* is a reference to a service provided by Alice that returns the contents of a specific file, and *bRef* is a reference to a service that will hold the backup copy. Bob implements his backup method by invoking a copy service offered by Carol, which is implemented as

```
void copy(ServiceRef inRef,
          ServiceRef outRef)
{ outRef.write(inRef.read()); }
```

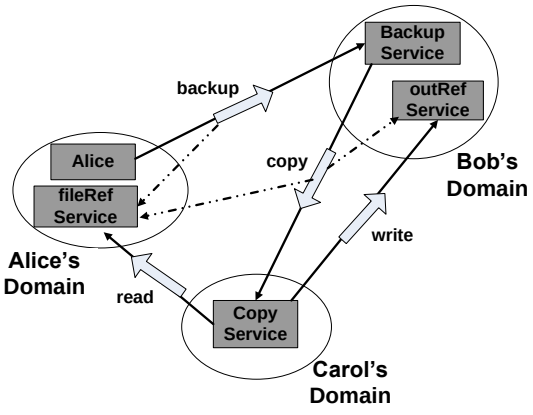

Figure 3: Illustration of the sample use case. Heavy arrows denote service invocation. Dotted lines, service delegations. For example, Alice invokes the backup service delegating to that service the right to use the fileRef service.

Bob's backup method's invocation of this service is

```
c.copy(fileRef,outRef)
```

Bob passes the reference he got as *fileRef* from Alice and a reference to a service that will hold the copy as *outRef*. Alice will get *outRef* as her return value.

This very simple example has many features of real web services composition. Services invoke other services, and services take references to other services as parameters. It is this last feature that is missing from the CANES example. Further, it may not be possible to change the service implementation. For example, we might want to change Carol's service to take the bits representing the files instead of references to services that provide them, but we must be prepared to deal with legacy applications.

In what follows, we will look at the consequences of various choices Alice makes for *fileRef* and Bob's service makes for *outRef* to see why such scenarios are so hard to deal with. Figure 4 shows a representation of the access policy as an access matrix [22], which shows the resources and principals. In this case, we see that Alice has permission to use *inRef*, while Bob and Carol have permission to use *outRef*. In what follows, we'll look at the implications of different choices for these permissions. First, though, we will examine the most widely used access control mechanisms.

	inRef	outRef
Alice	Allow	Deny
Bob	Deny	Allow
Carol	Deny	Allow

Figure 4. Representing permissions in an access matrix.

3. AUTHENTICATION-BASED CONTROL

The conventional approach bases the access decision on authentication of the subject presented with the request. The most common credential specifies the identity of the requester. We use the term Identification-Based Access Control (IBAC) to describe this approach. Each service keeps an access control list (ACL) specifying which operations each principal may use. One problem with this approach is that all those lists need to be updated each time a user's rights change. Coordinating those changes is difficult [32], especially across organizations.

Role-Based Access Control (RBAC) [13] was designed to address this problem. The ACLs list roles instead of, or in addition to, user identities. Users are assigned roles. When a user changes jobs, some other user is allowed to take on that role. No ACL changes are needed. Of course, sometimes only a few of the user's rights change. In that case, a new role needs to be introduced. Often the rights associated with a role depend on which user is acting in that role. In that case, too, a new role needs to be introduced. Also, all organizations need to agree on the rights associated with each role. Even small differences require the introduction of even more roles. The result is an explosion in the number of roles [16], which makes managing them difficult.

Policy-Based Access Control (PBAC) [4], which is called Attribute-Based Access Control (ABAC) in the US Defense Department jargon, extends RBAC to a more general set of proper-

ties. Subjects are assigned attributes. Each unique set of attribute values is effectively a role, which addresses the role explosion problem. Attributes can also be used to specify an identity or a role. When a request reaches a service, it tests the requester's attributes against a policy expressed in a special policy language to determine if access is allowed. PBAC has been shown to be capable of expressing a wide range of access policies. The flip side is the difficulty in understanding the rights being granted (or denied) when changing the set of attributes assigned to a user.

With PBAC, all participating organizations need to agree on the meanings of all the attributes, which is easier said than done. Recently, the National Security Agency spent a year reaching agreement on a set of attributes for the US Defense Department (DoD) Joint Enterprise Directory Service (JEDS) and came up with 13 attributes, most of them related to identity [1]. Additional work will be needed to standardize a more meaningful set of attributes. Reaching agreement with agencies from other countries or civilian first responders will be even more challenging.

We categorize IBAC, RBAC, and PBAC as autheNtication-Based Access Control (NBAC) because all of these approaches base the access decision on an authentication the subject presents along with the request. That authentication is used to look up, for IBAC and RBAC, or compute, for PBAC, an access decision. The service itself is not interested in the authentication, only the access decision.

There are a number of problems with using NBAC [21]. Here, we'll describe just two of them.

Delegation: It is hard for users to delegate subsets of their rights. In IBAC the ACLs of the relevant services need to be updated to reflect the changes. Since the ACL is a critical resource, such changes must be tightly controlled, putting a large burden on system administrators [32]. In RBAC, a new role needs to be introduced and the corresponding ACL entries created. PBAC is more problematic, since it is hard to know what attribute to assign to a user to grant a particular right. The result is that NBAC leads people to manage rights at rather coarse granularity.

Ambient authorities: An access decision depends on the authentication of the requester and the request being made. There is no means to specify which of the requester's rights apply to what arguments, which can lead to confusion. In the example of Section 2, Bob could inadvertently reverse the order of the arguments when invoking the copy service. If he has read permission on *outRef* and write permission on *inRef*, the request will succeed, because Bob has no way to express his true intent.

We'll see in the next Section two other problems, transitive access and confused deputy. All these problems, and others, are indicative of a failure to address the real problem, which is dealing with access policy. Identity, role, and attributes are only a means to an end, which for this discussion is making an access decision. The indirection they introduce in making that decision is the root cause of the transitive access problem. What is needed is a means of acting more directly on access policy [25].

4. TRANSITIVE ACCESS

We'll illustrate the problem with the scenario in Section 2. For each case, we'll show the access matrix as in Figure 4. There are a number of cases we need to consider.

Everyone has permission to read *inRef* and write *outRef*. Carol uses her permissions to read the input and to write the output. Alice's request succeeds. Alice can use her permission to recover the backup file.

	inRef	outRef
Alice	Allow	Allow
Bob	Allow	Allow
Carol	Allow	Allow

While everyone can read *inRef*, only Bob has permission to write *outRef*. Carol uses her permission to read the input, but she is unable to write the output. Bob could ask his system administrator to add Carol to the ACL for the service and later remove her, but the overhead of this operation is too high for most uses. Instead, Bob often grants Carol the ability to impersonate him for the duration of the request, as discussed in the Introduction. Carol gets far more privilege than she needs to complete the copy operation, but there is often no practical alternative. Without further action, Alice does not have permission to access the backup copy.

	inRef	outRef
Alice	Allow	Deny
Bob	Allow	Allow
Carol	Allow	Deny

Carol does not have permission to read the input file. Nor is it probable that Alice will have asked to have Carol added to the service's ACL, because Alice is unlikely to be aware that Carol's service gets invoked. Further, Alice necessarily has a trust relationship with Bob because she is using his service. No such relationship may exist with Carol. What often happens is that Alice allows Bob to impersonate her, and Bob allows Carol to impersonate Alice. Even worse than before, Carol has the rights of someone who may not know her, a particular problem if Alice and Carol are in different organizations.

	inRef	outRef
Alice	Allow	Deny
Bob	Deny	Allow
Carol	Deny	Allow

Carol does not have permission to use the *inRef* service or the *outRef* service. Further, impersonating Alice lets her read the input but not write the output. Impersonating Bob lets her write the output but not read the input. ***Carol's service must fail***. Changing the copy service implementation to take the contents of the files instead of references to them or to divide the application into read and write parts may not be possible, which is one of the constraints in the CANES scenario.

	inRef	outRef
Alice	Allow	Deny
Bob	Deny	Allow
Carol	Deny	Deny

Here, Alice specifies a file she is not allowed to read, and Bob specifies a file he does not have permission to write. Nevertheless, Carol can use her permissions to read the input and write the output. Even though neither Alice nor Bob has the required permissions,

	inRef	outRef
Alice	Deny	Deny
Bob	Deny	Deny
Carol	Allow	Allow

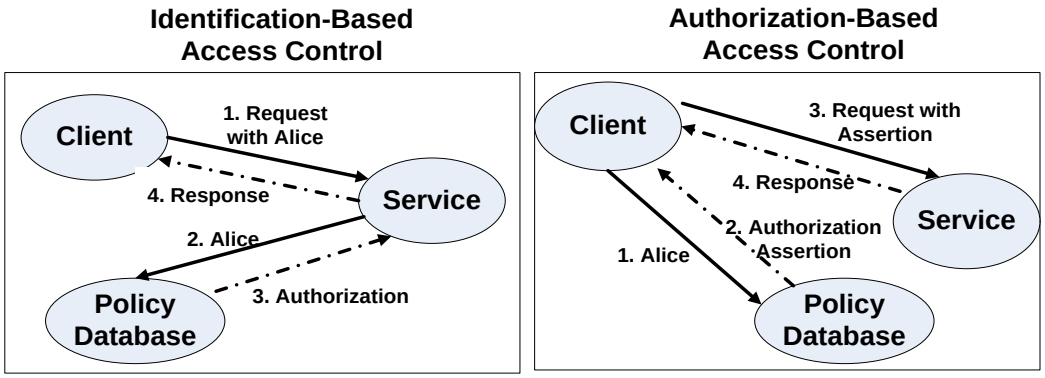

Figure 5. Comparison of NBAC and ZBAC.

Carol's service succeeds. Carol is a *confused deputy* [18] who has been induced to overwrite one of her files. Note that there may be no way for Carol to distinguish this case from those that should succeed. Indeed, it may be a security violation to give Carol the information needed to make the distinction.

This simple example captures the difficulty that has delayed progress on the CANES scenario and other SOA implementations. People have tried using roles or attributes instead of identities to no avail. That's not surprising since the problems arise from using subject authentication to make access decisions. The source of the difficulty is that the authentication is necessarily independent of the request.

5. AUTHORIZATION-BASED CONTROL

Every approach to access control begins with the user authenticating to the system and starting a program, a *user agent*, which acts on behalf of the user. In a system using subject authentication to make access decisions, the user agent must be able to transfer the right to authenticate as that user to every program it starts for the user.

Figure 5 compares using subject authentication (NBAC) with explicit authorizations (ZBAC) to make an access decision. For example, with IBAC a program Alice runs invokes Bob's service and includes proof of her identity. Bob's service looks up in a repository in his domain the ACL entry corresponding to the request and the authentication. If the entry matches, the service honors the request. Looking at it this way makes it clear that the authentication is only a way for the service to learn if the user is authorized to make this request. That being the case, let's turn things around.

The right half of Figure 5 shows an alternative approach. A user enters the system by authenticating, and the system starts a user agent with the ability to use that authentication. So far, this procedure is the same as an authentication-based system, but the next step is different. The user agent contacts a repository acting on behalf of the user's organization and receives explicit authorizations for each of the rights granted to the user. These rights can be individually delegated to programs running on the user's behalf, which makes users less vulnerable to erroneous or malicious programs they run [25]. The user's program submits the appropriate authorization along with each service request. The service only needs to verify the legitimacy of the authorization for the request. Subject authentication is not used to make the access decision, but it can be recorded for audit purposes. We call this approach *authoriZation-Based Access Control* (ZBAC).

Figure 6 shows how using delegatable authorizations also addresses the problem of federating access policy. Bob is probably not in a position to decide who is allowed to use his service, but an administrator in his organization is. Bob can delegate the right to use his service to that administrator, the domain controller in Figure 6. A corresponding domain controller in Alice's organization can arrange to use Bob's service and receives a delegatable authorization from Bob's domain controller. That authorization can be delegated to Alice's user agent when she authenticates to the system.

This approach has a number of advantages. Unlike authentication-based schemes, this approach does not require Bob's organization to know anything about Alice's rights. There is no need to federate identities, since authentication is only done within the users' organizations. Since there is only one authentication, there is no need to implement Single Sign-On. Roles and attributes can be helpful in managing the rights granted to individuals within a domain, but no global agreement on their meaning is needed since they are only used within an organization.

Figure 6 also illustrates how ZBAC makes the trust model explicit. With NBAC, Alice's credentials would be in a policy data-

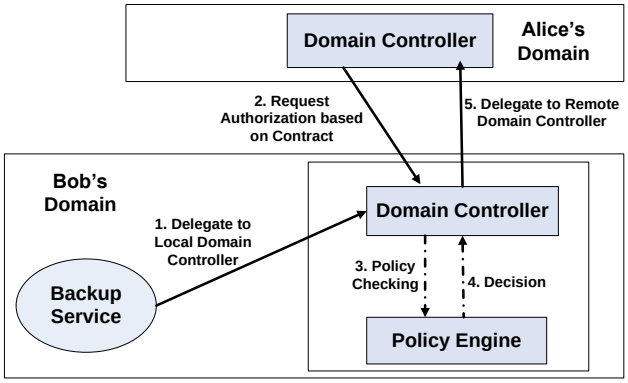

Figure 6. Federating access policy between domains.

base associated with the Backup service in Bob's domain even though Alice has no trust relationship with the Backup service. She has one with her domain, her domain has one with Bob's domain, and Bob's domain has one with the Backup service. That fact becomes obvious when Alice does something bad. The Backup service doesn't go after Alice. It tells its domain that Alice did something wrong, and that domain tells Alice's domain, which takes the appropriate action. This procedure requires information that is not defined in the NBAC model. The ZBAC delegation chain directly encodes these trust relationships.

6. DELEGATABLE AUTHORIZATIONS

SAML certificates were designed to communicate three kinds of security information, but most implementations use only the authentication and attribute fields. Typically, the authorization field is only used when a special service, such as the Policy Decision Service in Figure 1, makes the access decision. In those cases, the authorization field is used to convey to the invoked service a single bit of information, ALLOW or DENY. However, the standard allows using the SAML authorization field to carry more information [25]. In the following subsections, we'll represent the permission flow as signed digital certificates and show snippets of SAML assertions expressing those authorizations.

6.1 Installation Endowment

Before service invocations can be started, a subject, which can be an end-user such as Alice, or services, such as the backup and copy services, needs to receive SAML certificates that demonstrate permission to use certain services. This permission is embodied as a delegation chain rooted in the service itself [25]. We'll denote authorization certificates using a notation similar to that used for expressing speaks-for relations [23], *e.g.*,

```
Subject<-Service(Proof),
```

which can be read as "delegate to *Subject* the right to use *Service* using *Proof* to verify that the delegator has at least the rights being delegated." Another field, which we don't need for this example, can restrict which methods of the service the certificate allows. We'll see later how the certificate is validated. An example of a SAML authorization assertion is at

http://opra.hpl.hp.com/Fam/SamlAuthZCertExample.xml

which is explained in the Appendix.

In a *root certificate*, *Proof* is the public key corresponding to the private key used to sign the certificate, which we see from Figure 6 is the service owner's key. In a *delegation certificate*, *Proof* is a certificate granting the delegator the right to use at least the set of rights being delegated. For example,

```
SubjectB<-Service(SubjectA<-Service(Proof))
```

denotes that *SubjectB* has permission to use *Service* if *SubjectA* does. Each delegation results in an additional nesting of *Proof*, which allows full responsibility tracking. In this example, we know that *SubjectA* is responsible for *SubjectB*'s right to use *Service.*

The root certificates for the example in Section 2 are

c_0 = Copy<-Copy(Copy),

which denotes that the owner of copy service has delegated to itself the right to use the copy service, and

b_0 = Backup<-Backup(Backup).

Each service then delegates its right to whatever subject in its organization is responsible for managing access, denoted as B and, C respectively. For example,

c_1 = C<-Copy(c_0),

which delegates to C all rights to the copy service. Here *Proof* is the root certificate created by the service. We do the same for the backup service, *i.e.*,

b_1 = B<-Backup(b_0).

Starting with b_0 and c_0 instead of b_1 and c_1 saves a bit of key management because the service only needs to know its own private key to validate the root certificate.

Some entity in Alice's domain, call it A, negotiates with B for the right to use the backup service and receives from B a certificate authorizing this use as

a_B = A<-Backup(b_1).

B will receive a certificate permitting access to the Copy service from C represented as

b_c = B<-Copy(c_1).

Finally, A grants Alice two certificates for service invocations. One is for the Backup service, denoted

$Alice_{backup}$ = Alice<-Backup(a_B)

and one for the service providing the file to be backed up, denoted

$Alice_{fileRef}$ = Alice<-FileRef(FileSystemA),

where *FileSystemA* is the nested set of certificates rooted in the service providing Alice's file. Similarly, the Backup Service receives a certificate to invoke the Copy Service from B, namely

$backup_{copy}$ = Backup<-Copy(b_c)

and the right to use the service that will hold the output file,

$backup_{outRef}$ = Backup<-OutRef(FileSystemB).

Each of these certificates can be represented as a SAML authorization assertion [25]. *Proof* is encoded in the <Evidence> field in the SAML <AuthorizationDecisionStatement> as a nested set of authorization assertions.

This handling of files shown here is too simplistic, but it serves our purposes for this example. In a real implementation, the file system administrator will grant Alice read/write access to a particular sub-directory, for example, /users/alice. Such a constraint specification can be expressed in a SAML attribute statement as

```
<saml:AttributeStatement>
   …
<saml:Attribute
    AttributeName="AccessibleDirectory"
    AttributeNameSpace =
       http://www.domaina.com/CM.asmx
    <saml:AttributeValue>
       /users/alice
    </saml:AttributeValue>
</saml:Attribute>
       …
</saml:AttributeStatement>
```

Notice that the constraint has a scope represented by the XML *AttributeNameSpace* attribute, indicating that the constraint is applied to the entire service representing the file system. This <AttributeStatement> can be included in the SAML authorization assertion to restrict the rights being granted.

Also, real services have multiple methods. The restriction parameter that we didn't show can be used to limit which rights are being delegated. For example, if $\texttt{Alice}_{\texttt{fileRef}}$ includes both read and write permissions, Alice's delegation to the backup service can be represented as

$$\texttt{Backup<-Backup(Alice}_{\texttt{fileRef}}\texttt{,[read])}.$$

Which methods are being delegated is represented in the SAML certificate as a list of methods in the <AuthorizationDecisionStatement> [25].

6.2 Service Invocation

Alice starts a process to run the program that performs the backup. In an NBAC system, that process would be able to authenticate as Alice and would have all her rights. With ZBAC, that corresponds to Alice sharing her public key with the process, but she can do better. Say that Alice enters

```
backup(/users/alice/foo.pdf)
```

on the console. Based on Alice's installation endowment, the runtime system will map *backup* to the authorization to use the backup service and */users/alice/foo.pdf* to the authorization to use this file.

The runtime will then start a process to carry out the command. That process will create a new key pair and pass the public key to the runtime. The runtime will produce certificates delegating to that process the right to invoke the backup service and use the designated file,

$$\texttt{proc}_{\texttt{backup}} = \texttt{proc<-Backup(Alice}_{\texttt{backup}}\texttt{)}$$
$$\texttt{proc}_{\texttt{fileRef}} = \texttt{proc<-FileRef(Alice}_{\texttt{fileRef}}\texttt{)}.$$

At this point, the process has the least set of Alice's privileges it needs to fulfill her request but no more. Alice's risk is limited should the program the process runs be erroneous or malicious.

The program running Alice's request creates a SOAP message to invoke the backup service. The SOAP header designates the service invocation as the certificate $\texttt{proc}_{\texttt{backup}}$ and the argument to that service as a delegation in the SOAP body,

$$\texttt{ptob = Backup<-FileRef(proc}_{\texttt{fileRef}}\texttt{)}.$$

The backup service does the same when it invokes the copy service with certificate b_c, delegating the parameters, *i.e.*,

$$\texttt{inRef = Copy<-FileRef(ptob)}$$
$$\texttt{outRef = Copy<-OutRef(backup}_{\texttt{outRef}}\texttt{)}.$$

The copy service uses certificate *inRef* to read the contents of Alice's file and certificate *outRef* to write the backup. Finally, the backup service delegates to Alice the right to use the service holding the copy of her file,

$$\texttt{bref = Alice<-OutRef(backup}_{\texttt{outRef}}\texttt{)}.$$

Figure 7 shows the invocations and corresponding delegations.

We probably can't use the SAML authorization as an argument when invoking a legacy SOA service. In these cases, the SOAP body will contain the legacy representation of the arguments, most often as strings, and the authorizations will go into the SOAP header. Since we have separated designation from authorization, using this approach requires care to avoid confused deputy attacks.

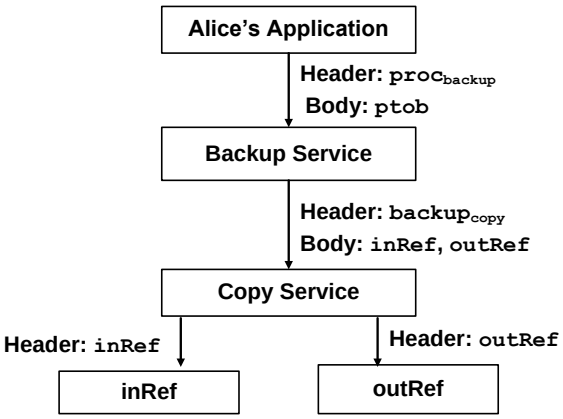

Figure 7. Permission flow in sample use case showing the certificates used for invocation and delegation and where they appear in the SOAP message.

6.3 Validation

SAML certificates are assumed to be public documents. Hence we need to verify the submitter's right to use them, which involves walking the delegation chain. We also need to verify that the submitter has the right to use certificates passed as delegations. Without that check, which was omitted in our earlier work [25], malicious subjects could delegate rights they don't have, leading to confused deputy attacks.

First, the invocation certificate in the SOAP header is checked to verify that it was issued to a public key corresponding to the private key used to sign the SOAP request. When a service is invoked with a parameter delegating a service reference, the invoked service verifies that the delegation was signed with the same private key used to sign the SOAP request.

The next step is to verify the proof in the invocation certificate. That step involves unpeeling the onion layers of the nested delegation certificates and verifying that each one was signed by the private key corresponding to the public key the delegation was issued to. The proof is complete when the verification reaches the service's root certificate.

While a certificate authority may be used when deciding to grant rights, there is no need for a certificate authority in the verification process, which only needs to check each signature against the corresponding public key until it reaches the root certificate signed with service's own private key. There is no need, except perhaps for audit, to attach an identity to any of the keys. Indeed, some of these keys are *ad hoc*, created and used for a single invocation, as we saw with the process running Alice's backup command.

6.4 Revocation

If we stop at this point, both the copy and backup services will accumulate rights each time they are invoked, which will eventually become a security exposure. The services can't just discard the SAML assertions, because we have assumed they are public documents. We use two mechanisms to address this problem.

First, we set an expiration time. Installation endowment certificates and return delegations can be set to expire when the corresponding contract ends. The situation is more complicated for delegation certificates passed as parameters. If timeouts were the only way to make a certificate invalid, this valid time interval would have to be short, but knowing how short is long enough is hard. Instead, we set an expiration time reasonably far into the future and rely on the invoker to revoke any delegations when the invocation completes.

Second, we explicitly revoke assertions. Each SAML assertion has a unique identifier, a UUID. We assume that each service supports a *revoke* method, which takes a UUID and a SAML assertion as arguments. The first argument specifies which authorization is being revoked. The second argument is the SAML assertion which appears as *Proof* in the assertion having the specified by the first argument. This way only the delegator of a right can revoke the delegation of that right.

Each service keeps a revocation list of unexpired, revoked authorizations. The first time an authorization with a UUID matching a revocation request is presented in an invocation, the service verifies that the revocation request is authorized. If so, the service adds this UUID to a local revocation list and refuses any requests containing that UUID anywhere in the delegation chain.

Unlike IBAC, we are only revoking a single authorization, not a user's identity, which has two advantages. First, timeouts can be short because service invocations have much shorter lifetimes than user identities. Second, there is no need to circulate certificate revocation lists (CRLs), because only the service itself needs to know which authorizations to use the service have been revoked. Both of these advantages lead to smaller, more manageable CRLs than when using IBAC.

6.5 Federating Access Policy

Figure 8 shows the evolution of the access matrix as services are invoked. We start with only Alice having permission to use *inRef* and only Bob having permission to use *outRef*, as shown at the top of Figure 8. Alice's invocation of Bob's backup service delegated the rights to use *inRef*, as shown in the second version of the access matrix. Bob then invoked Carol's copy service, delegating the least set of rights needed for his request to succeed, resulting in the third version of the access matrix. Bob delegates to Alice the right to use *outRef* when returning from her request, as shown in the last version of the access matrix.

This example shows that the effect of delegating the right to use services specified as arguments is to change the access policy to match the least privilege needed for the request to succeed. That's quite different from the situation when using subject authentication to make access decisions. In those cases, the access matrix was static because we needed some external mechanism to specify the required changes.

	inRef	outRef
Alice	Allow	Deny
Bob	Deny	Allow
Carol	Deny	Deny

	inRef	outRef
Alice	Allow	Deny
Bob	$Allow_A$	Allow
Carol	Deny	Deny

	inRef	outRef
Alice	Allow	Deny
Bob	$Allow_A$	Allow
Carol	$Allow_{A,B}$	$Allow_B$

	inRef	outRef
Alice	Allow	$Allow_B$
Bob	$Allow_A$	Allow
Carol	$Allow_{A,B}$	$Allow_B$

Figure 8. Permission flow ZBAC.

7. VOC

One objection commonly raised to the use of freely delegatable authorizations is loss of control. What if Alice's organization has a policy that says Bob should not be given access to Alice's file? It looks like there is no way to prevent Alice's program from delegating that right to Bob's service.

In an NBAC system, Alice would have to ask an administrator to add Bob to the ACL, let him take on some role, or assign him some attributes. Presumably, the administrator would refuse if honoring the request would violate policy. Unfortunately, this approach makes all delegations difficult, which leads people to bypass such mechanisms in order to get their work done. The result is that the policy ends up being violated in spite of the apparent control.

The first thing to recognize is that Alice can always send the file to Bob by some means if she can communicate with him. That may be as simple as reading the file herself and sending the contents to him or as complex as using a covert channel. Alice can even share her authentication credentials with Bob. So, whatever we do, we need Alice's voluntary cooperation. Nevertheless, access rules are complex, and they frequently change. Even though Alice may wish to comply with these rules, she may be oblivious of them. What we want is a system of that lets users voluntarily comply with the policy while being oblivious of it, something we call *Voluntary Oblivious Compliance (VOC)*.

There are many ways to support VOC. The simplest is to make a rule that people should ask an administrator before delegating any rights. That's clearly not a scalable solution, but it is the only one available with IBAC. Another approach is for Alice to send Bob a handle to the service and let the underlying system make the

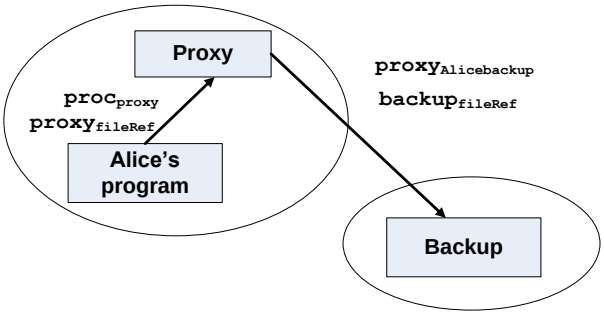

Figure 9. Using a proxy to enforce VOC.

access decision. A simple example of such a handle is a URL to a page inside Alice's firewall, which Bob can use only if the access policy lets him inside the perimeter. This approach is suitable for systems that use RBAC and PBAC. However, RBAC and PBAC, provide rather coarse control over access rights because they are tied to subjects. Administrators are faced with the dilemma of making the rules overly permissive, which leads to some violations of the desired policy, or overly restrictive, which leads people to circumvent the mechanisms.

Delegatable authorizations, which are the cause of concern, provide a solution, as well. Most delegations do not violate any policy, *e.g.*, when Alice's user agent delegates a right to a process it is starting on her behalf. We can proxy requests for those delegations that might violate policy, as is done when using an Enterprise Service Bus [2], illustrated in Figure 9. Instead of binding the invocation in Alice's program directly to the Backup service, the invocation is bound to a proxy for the service. So, Alice's user agent delegates

$$\mathtt{proc}_{\mathtt{backup}} = \mathtt{proc<\text{-}Proxy(Alice}_{\mathtt{backup}}\mathtt{)}$$
$$\mathtt{proc}_{\mathtt{fileRef}} = \mathtt{proc<\text{-}FileRef(Alice}_{\mathtt{fileRef}}\mathtt{)}$$

to the process running her program. This process invokes the proxy, delegating

$$\mathtt{proxy}_{\mathtt{fileRef}} = \mathtt{proxy<\text{-}FileRef(proc}_{\mathtt{fileRef}}\mathtt{).}$$

If the delegation to Bob's service would not violate policy, the proxy invokes the Backup service using an authorization given to the proxy for use when acting on Alice's behalf,

$$\mathtt{proxy}_{\mathtt{Alicebackup}} = \mathtt{Proxy<\text{-}Backup(asAlice),}$$

where *asAlice* is an authorization constructed by A, Alice's and the proxy's domain controller, specifying Alice's rights to the Backup service. The proxy delegates to the Backup service the *fileRef* authorization received from Alice's program as

$$\mathtt{backup}_{\mathtt{fileRef}} = \mathtt{Backup<\text{-}FileRef(proxy}_{\mathtt{fileRef}}\mathtt{).}$$

Neither Alice's program nor Bob's service need be aware of this indirection. We are just changing the name bindings of the authorizations.

8. RELATED WORK

The earliest form of explicit authorization is a *capability*, the defining characteristic of which is that it combines designation with authorization. Initially, capabilities were used to limit access to hardware resources, such as memory pages [7]. Later, they were used to protect other resources, such as files, and used across the network [9]. More recently, people have noted that object references can be used as capabilities controlling access at the object level [28]. In its purest form, what we demonstrate is that SAML authorization assertions can combine designation with authorization, thereby serving as capabilities.

The REST model of computation [14] uses GET and POST operations on URLs to implement web services. The original formulation had no access control, but the web calculus [6] makes URLs into capabilities. OAUTH [31] is not adequate for our needs. While it can be used for fine-grained authorization, it does not support chained delegation. Lampson's general speaks-for [23] can also be used as capabilities, but explanations of its use are rooted in an authentication-based ACL.

A number of systems similar to ours were developed before the web services standards were defined. Passport [35], Proxy [3], and Restricted Proxy [29] allow chained, restricted delegation, but the final access decision is based on the identity of the originator of the request. In a SOA environment, this requirement implies some sort of distributed identity management because the originator's identity must be in the service domain's policy database.

The Community Authorization Service [36] allows restricted, chained delegation using a central authorization service. Delegation between communities, which corresponds to domains in SOA, is not supported. At first glance the authorization profile for attribute certificates [12] seems to support the kind of delegation of authorization that we propose, but a close look shows that the authors use the phrase "authorization information" to mean what we call authentication information, such as role, group membership, and security clearance.

The closest to our approach is E-speak [20], which was based on the Simple Public Key Infrastructure [10] and used certificates as capabilities. However, with that approach all parameters had to be delegation certificates, which required changes to the application API. Our approach permits putting delegation certificates into the SOAP header. We extend the e-speak approach by including SAML attribute assertions in the SOAP header that allow the use of application specific constraints and to enforce Risk Adaptive Access Control (RAdAC) [26].

None of these approaches is suitable for SOA, which is based on XML and the corresponding web services standards. XACML, an XML specification for authorization decision making, has been adapted for delegation [5]. However, the thrust of this work is to alleviate the administrator's workload by defining which rights each user may delegate. This work does not support chained delegation.

Other distributed systems, particularly those that cross administrative boundaries can also benefit from switching from NBAC to ZBAC. The administrative burden of managing ACLs is widely recognized [32]. Administering a GRID [15] node involves creating and deleting accounts for users in many organizations. The Grid community has partially moved toward ZBAC with the Community Authorization Service [15]. People working on the DoD Global Information Grid (GIG) have reached the same conclusion [24]. One of the main complaints of Principal Investigators on PlanetLab [33] is the difficulty in delegating subsets of their authorities to their graduate students. Solutions to this problem that are currently being used for the 1,000 or so machines in PlanetLab today will not be practical as its size increases. ZBAC, by decoupling the policy decisions into manageable chunks, avoids the scalability issues inherent in NBAC.

9. CONCLUSIONS

SOA is different from more tightly coupled environments. SOA crosses administrative domains; it has far more users and separate components; it is far more dynamic in the rate and number of things that change; no one party is in charge. There is little reason to think that traditional designs are applicable to SOA. Yet that's what using authentication to make access decisions does. Authorization-based access control, which has advantages within an organization, and even on stand-alone computers, is a better match to the requirements of distributed systems that span administrative domains.

Service composition is an important use pattern for SOA. Implementations that use authentication to make access decisions have failed to achieve both the desired functionality and the required security. Using delegatable authorizations lets us federates access policy, which results in a fully functional system that enforces the Principle of Least Privilege.

10. ACKNOWLEDGMENTS

We'd like to thank Mike Davis and Harry Haury for helpful discussions and Tyler Close for pointing out a vulnerability.

11. REFERENCES

[1] Bachert, R., "Moving from One Theater to the Next Using Enterprise Directory Service (EDS)", LandWarNet Conf., November 2007, also http://www.afcea.org/events/pastevents/documents/Track4Session7-EDS.ppt

[2] BEA, AquaLogic Service Bus Concepts and Architecture, http://edocs.bea.com/alsb/docs30/pdf/concepts.pdf, Feb. 2008.

[3] Black, S. and Varadharajan, V., "An Analysis of the Proxy Problem," HP Labs Tech. Report HPL-90-163, 1990.

[4] Blaze, M., Feigenbaum, J., Ioannidis, J., and Keromytis, A., "The Role of Trust Management in Distributed Systems Security." Chapter in *Secure Internet Programming: Security Issues for Mobile and Distributed Objects*, (Vitek and Jensen, eds.) Springer-Verlag, 1999.

[5] Chadwick, D. W., Otenko, S., and Nguyen, T. A., "Adding Support to XACML for Dynamic Delegation of Authority in Multiple Domains," *Proc. of the IFIP Intl Conf. on Communications and Multimedia Security*, pp. 67-86, 2006.

[6] Close, T., "web-key: Mashing with Permission", IEEE W2SP 2008: Web 2.0 Security and Privacy. May 2008, http://www.waterken.com/dev/Web/REST/

[7] Dennis, J. B. and Van Horn, E. C., "Programming Semantics for Multiprogrammed Computations" *Comm. ACM*, 9(3):143-155, 1966.

[8] DISA, "A Security Architecture for Net-Centric Enterprise Services (NCES)", Defense Information Systems Agency, March 2004, also http://horizontalfusion.dtic.mil/docs/specs/20040310_NCES_Security_Arc.pdf.

[9] Donnelley, J. "A Distributed Capability Computing System (DCCS)", Third International Conf. on Computer Communication, Toronto, Canada, August 3-6, 1976.

[10] C. Ellison, http://www.ietf.org/rfc/rfc2692.txt 1999.

[11] Erdos, M. E. and Pato, J. N., "Extending the OSF DCE Authorization System to Support Practical Delegation", PSRG Workshop on Network and Distributed System Security. Feb. 11-12, 1993.

[12] Farrell, S. and Housley, R, "An Internet Attribute Certificate Profile for Authorization", IETF RFC 3281, April 2002.

[13] Ferraiolo, D.F. and Kuhn, D.R., "Role Based Access Control", *15th National Computer Security Conf.*: 554-563, 1992.

[14] R.Fielding, R. "Architectural Styles and the Design of Network-based Software Architectures"; Doctoral dissertation, University of California, Irvine; 2000.

[15] Foster, I., Kesselman, C., Pearlman, L., Tuecke, S., and Welch, V., "The Community Authorization Service: Status *In Proc. of Computing in High Energy Physics. 03* 2003.

[16] Freudenthal, E., Pesin, T., Keenan, Port, E. L. and Karamcheti, V., "dRBAC: Distributed Role-Based Access Control for Dynamic Coalition Environments", *Proc. of the Intl. Conf. on Distributed Computing Systems (ICDCS)*, 2002.

[17] Graham, S., Karmarkar, A., Mischinsky, J., Robinson, I., and Sedukhin, I., eds., "Web Services Resource 1.2", http://docs.oasis-open.org/wsrf/wsrf-ws_resource-1.2-spec-os.pdf 2006.

[18] Hardy, N., "The Confused Deputy", *Operating Systems Reviews*, **22**, #4, 1988.

[19] Hirsch, F., ed., "ID-WSF 2.0 SecMech SAML Profile, Version v2.0", http://www.projectliberty.org/liberty/content/download/894/6258/file/liberty-idwsf-security-mechanisms-saml-profile-v2.0.pdf

[20] Karp, A. H. "E-speak E-xplained", CACM, vol. 46. #7, pp. 113-118, July 2003.

[21] Karp, A. H., "Authorization Based Access Control for the Services Oriented Architecture", Proc. 4th Int. Conf. on Creating, Connecting and Collaborating through Computing (C5 2006), Berkeley, CA, IEEE Press, January 2006.

[22] Lampson, B., "Protection", Proc. 5th Princeton Conf. on Information Sciences and Systems, p437, Princeton, 1971.

[23] Lampson, B. "Practical Principles for Computer Security", In *Software System Reliability and Security*, Proceedings of the 2006 Marktoberdorf Summer school.

[24] Levin, R. E. "The Global Information Grid and Challenges Facing Its Implementation", GAO-04-858, 2004, also http://www.gao.gov/new.items/d04858.pdf

[25] Li, J. and Karp, A. H., "Access Control for the Services Oriented Architecture," ACM Workshop on Secure Web Services, ACM #459074, pp. 9-17, Fairfax, VA, November 2007.

[26] McGraw, R. W., "Securing Content in the Department of Defense's Global Information Grid", Secure Knowledge Management Workshop, Buffalo, NY, September, 2004.

[27] Miller, C., "Navy C4I Open Architecture Strategy", **SoftwareTech, 10**, #3, October 2007, also https://www.softwaretechnews.com/stn_view.php?stn_id=43&article_id=89

[28] M. S. Miller, "Robust Composition: Towards a Unified Approach to Access Control and Concurrency Control", Doctoral Dissertation, Johns Hopkins University, 2006, also http://erights.org/talks/thesis/index.html

[29] Neuman, B. C., "Proxy-Based Authorization and Accounting for Distributed Systems", *Proc. of the Intl. Conf. on Distributed Computing Systems (ICDCS)*, 1993.

[30] OASIS, "Security Assertion Markup Language (SAML) 2.0 Technical Overview", Working Draft 05**,** 10 May 2005.

[31] OAUTH, http://oauth.net

[32] Open Group, *CDSA Explained, An indispensable guide to Common Data Security Architecture*, The Open Group, 2001.

[33] PlanetLab, http://www.planet-lab.org/

[34] Saltzer, J. H. and Schroeder, M. D. "The protection of information in computer systems", Proceedings of the IEEE, 63(9):1278.1308, September 1975.

[35] Sollins, K. R., "Cascaded Authentication", Proc. of the IEEE Symposium on Research in Security and Privacy, pp. 156-163, 1988.

[36] Welch, V., Foster, I., Kesselman, C., Mulmo, O., Pearlman, L., Tuecke, Gawor, J., Meder, S., and Siebenlist, F., "X.509 Proxy Certificates for Dynamic Delegation", Proceedings of the 3rd Annual PKI R&D Workshop, 2004.

APPENDIX

This appendix annotates the authorization assertion Alice's proxy uses to delegate the right to read the input when it invokes the Backup service on her behalf. We've elided the cryptographic information in the <SubjectConfirmation> and <Signature> tags in the interest of space and rearranged some tags for clarity. The automatically generated assertion is at

http://opra.hpl.hp.com/Fam/SamlAuthZCertExample.xml.

The assertion is issued by the proxy acting on Alice's behalf and must be signed by the proxy's private key to be valid.

```
<saml:Assertion MajorVersion="1" MinorVersion="1" IssueInstant="2008-11-18T09:32:22Z"
      AssertionID="_98594eb1-c3d3-44cf-ab83-816bd95612a6"
      Issuer="CN="Proxy of Alice Jones O=Domain A""
       xmlns:saml="urn:oasis:names:tc:SAML:1.0:assertion">
  <Signature xmlns="http://www.w3.org/2000/09/xmldsig#">...</Signature>
```

We set the expiration time far enough in the future that we are sure the request will have completed because we expect Alice's program to revoke this delegation when it receives the return value.

```
  <saml:Conditions NotBefore="2008-11-18T09:12:22Z" NotOnOrAfter="2008-11-18T09:52:22Z"/>
  <saml:AuthorizationDecisionStatement Decision="Permit"
             Resource="http://www.DomainA.com/FileMgmt/FileMgmt.asmx">
```

The proxy is delegating to the Backup service.

```
    <saml:Subject>
      <saml:NameIdentifier NameQualifier=""
           Format="urn:oasis:names:tc:SAML:1.1:nameid-format:X509SubjectName">
        CN="Backup Service Authority O=Domain B"
      </saml:NameIdentifier>
      <saml:SubjectConfirmation>...</saml:SubjectConfirmation>
    </saml:Subject>
```

The right being delegated is the right to use the *ReadFile* method of the *FileMgmt* service.

```
    <saml:Action Namespace="http://www.DomainA.com/FileMgmt/FileMgmt.asmx">
       ReadFile
    </saml:Action>
```

The <AttributeStatement> limits this use to reading only the specified file, an example of how to specify an application specific restriction using an attribute.

```
    <saml:AttributeStatement>
      <saml:Attribute AttributeNamespace=http://www.DomainA.com/FileMgmt/FileMgmt.asmx
             AttributeName="AccessibleFile">
        <saml:AttributeValue>
             /users/content/alice/brochure.pdf
        </saml:AttributeValue>
      </saml:Attribute>
    </saml:AttributeStatement>
```

The *Proof* that this delegation is valid is the proxy's authorization assertion carried in the <Evidence> field, which has been signed by Alice.

```
<saml:Evidence>
  <saml:Assertion MajorVersion="1" MinorVersion="1" IssueInstant="2008-11-18T09:32:21Z"
    AssertionID="_71497a19-3193-4943-8e6e-b58f9f7c3aa0"
    Issuer="CN="Alice Jones O=Domain A"">
  <Signature xmlns="http://www.w3.org/2000/09/xmldsig#">...</Signature>
  <saml:Conditions NotBefore="2008-11-18T09:12:21Z"
                   NotOnOrAfter="2008-11-18T09:52:21Z"/>
  <saml:AuthorizationDecisionStatement Decision="Permit"
          Resource="http://www.DomainA.com/FileMgmt/FileMgmt.asmx" >
```

```
  <saml:Subject>
    <saml:NameIdentifier NameQualifier=""
              Format="urn:oasis:names:tc:SAML:1.1:nameid-format:X509SubjectName">
       CN="Proxy of Alice Jones O=Domain A"
    </saml:NameIdentifier>
    <saml:SubjectConfirmation>...</saml:SubjectConfirmation>
  </saml:Subject>
  <saml:Action Namespace="http://www.DomainA.com/FileMgmt/FileMgmt.asmx">
      ReadFile
  </saml:Action>
  <saml:AttributeStatement>
    <saml:Attribute AttributeNamespace=http://www.DomainA.com/FileMgmt/FileMgmt.asmx
           AttributeName="AccessibleFile">
      <saml:AttributeValue>
         /users/content/alice/brochure.pdf
      </saml:AttributeValue>
    </saml:Attribute>
  </saml:AttributeStatement>
```

The *Proof* that Alice has the right to delegate this authority comes from the delegation Alice received from the administrator.

```
<saml:Evidence>
  <saml:Assertion MajorVersion="1" MinorVersion="1" IssueInstant="2008-11-18T09:32:21Z"
       AssertionID="_ac2df433-3e73-4cb0-aa32-c57ad064d70b"
       Issuer="CN="Domain Access Right Controller O=Domain A"">
    <Signature xmlns="http://www.w3.org/2000/09/xmldsig#">...</Signature>
    <saml:Conditions NotBefore="2007-11-19T09:32:21Z"
                     NotOnOrAfter="2009-11-18T09:32:21Z" />
    <saml:AuthorizationDecisionStatement Decision="Permit"
              Resource="http://www.DomainA.com/FileMgmt/FileMgmt.asmx" >
      <saml:Subject>
        <saml:NameIdentifier NameQualifier=""
              Format="urn:oasis:names:tc:SAML:1.1:nameid-format:X509SubjectName">
           CN="Alice Jones O=Domain A"
        </saml:NameIdentifier>
        <saml:SubjectConfirmation>...</saml:SubjectConfirmation>
      </saml:Subject>
```

Notice that Alice has the right to read or write any file in the specified directory.

```
      <saml:Action Namespace="http://www.DomainA.com/FileMgmt/FileMgmt.asmx">
          ReadFile
      </saml:Action>
      <saml:Action Namespace="http://www.DomainA.com/FileMgmt/FileMgmt.asmx">
          WriteFile
      </saml:Action>
      <saml:AttributeStatement>
        <saml:Attribute AttributeName="AccessibleDirectory"
             AttributeNamespace="http://www.DomainA.com/FileMgmt/FileMgmt.asmx">
          <saml:AttributeValue>
            /users/content/alice
          </saml:AttributeValue>
        </saml:Attribute>
      </saml:AttributeStatement>
```

Alice's administrator has the right to read or write any file in the filesystem.

```
<saml:Evidence>
  <saml:Assertion MajorVersion="1" MinorVersion="1"
       AssertionID="_2cd7fea2-8ff1-446b-b5d3-723eccaa0014"
       Issuer="CN="File Management Service Authority O=Domain A""
       IssueInstant="2008-11-18T09:32:21Z">
    <Signature xmlns="http://www.w3.org/2000/09/xmldsig#">...</Signature>
    <saml:Conditions NotBefore="2007-11-19T09:32:21Z"
                     NotOnOrAfter="2009-11-18T09:32:21Z" />
```

```
    <saml:AuthorizationDecisionStatement Decision="Permit"
                    Resource="http://www.DomainA.com/FileMgmt/FileMgmt.asmx" >
      <saml:Subject>
        <saml:NameIdentifier NameQualifier=""
                 Format="urn:oasis:names:tc:SAML:1.1:nameid-format:X509SubjectName">
           CN="Domain Access Right Controller O=Domain A"
        </saml:NameIdentifier>
        <saml:SubjectConfirmation>...</saml:SubjectConfirmation>
      </saml:Subject>
      <saml:Action Namespace="http://www.DomainA.com/FileMgmt/FileMgmt.asmx">
           ReadFile
      </saml:Action>
      <saml:Action Namespace="http://www.DomainA.com/FileMgmt/FileMgmt.asmx">
            WriteFile
      </saml:Action>
```

The root of the authority is the filesystem service and is signed by the service itself.

```
<saml:Evidence>
  <saml:Assertion MajorVersion="1" MinorVersion="1"
                  AssertionID="_4ccb6557-d8f9-47e5-a239-2bc8ddd5c3c5"
                  Issuer="CN="File Management Service Authority O=Domain A""
                   IssueInstant="2008-11-18T09:32:21Z">
    <Signature xmlns="http://www.w3.org/2000/09/xmldsig#">...</Signature>
    <saml:Conditions NotBefore="2007-11-19T09:32:21Z"
                    NotOnOrAfter="2009-11-18T09:32:21Z" />
    <saml:AuthorizationDecisionStatement Decision="Permit"
                    Resource="http://www.DomainA.com/FileMgmt/FileMgmt.asmx" >
      <saml:Subject>
        <saml:NameIdentifier NameQualifier=""
                    Format="urn:oasis:names:tc:SAML:1.1:nameid-format:X509SubjectName">
          CN="File Management Service Authority O=Domain A"
        </saml:NameIdentifier>
        <saml:SubjectConfirmation>...</saml:SubjectConfirmation>
      </saml:Subject>
      <saml:Action Namespace="http://www.DomainA.com/FileMgmt/FileMgmt.asmx">
           ReadFile
      </saml:Action>
      <saml:Action Namespace="http://www.DomainA.com/FileMgmt/FileMgmt.asmx">
           WriteFile
      </saml:Action>
    </saml:AuthorizationDecisionStatement>
   </saml:Assertion>
</saml:Evidence>
```

Followed by the rest of the close tags.