Zebra Copy: A Reference Implementation of Federated Access Management

Jun Li, Alan H. Karp
Advanced Architecture Laboratory
HP Laboratories Palo Alto
HPL-2007-105
June 28, 2007*

Services Oriented Architecture, SOA, web services, access control, Federated Identity Management, FIdM

Federated Identity Management (FIdM) is being applied to Services Oriented Architecture (SOA) deployments that cross enterprise boundaries. These systems have been found to be inflexible, unscalable, and difficult to use, manage, and upgrade. We contend that a major reason for these difficulties is that FIdM solves the wrong problem. Specifically, FIdM says nothing about federating access policies. What is needed instead of FIdM is a system for Federated Access Management (FAccM). This report demonstrates the benefits of FAccM over FIdM for SOA deployments and includes a step-by-step explanation of code needed to deploy, manage, and use a sample service.

Zebra Copy: A Reference Implementation of Federated Access Management[1]

Jun Li and Alan H. Karp
Hewlett-Packard Laboratories
Palo Alto, California

Abstract

Federated Identity Management (FIdM) is being applied to Services Oriented Architecture (SOA) deployments that cross enterprise boundaries. These systems have been found to be inflexible, unscalable, and difficult to use, manage, and upgrade. We contend that a major reason for these difficulties is that FIdM solves the wrong problem. Specifically, FIdM says nothing about federating access policies. What is needed instead of FIdM is a system for Federated Access Management (FAccM). This report demonstrates the benefits of FAccM over FIdM for SOA deployments and includes a step-by-step explanation of code needed to deploy, manage, and use a sample service.

1. Introduction

The Services Oriented Architecture (SOA) [22,27] may yet deliver on the promise of loosely coupled application development that didn't materialize from earlier attempts, such as CORBA [29,11]. The SOA is based on Web Services standards - SOAP for invocation [26], WSDL for interface definition [31], and UDDI for service discovery [28], all of which use XML [7] as the communications format. These standards remove many dependencies on machine architecture and operating system, making composition of independently developed components far easier.

One of the things holding back the widespread use of the SOA is the delay in reaching consensus on how to secure the services. There are a number of aspects of securing web services, such as encryption, message integrity, authentication, authorization, *etc*., and there appears to be at least one standard for each of them, XML DSIG [32], XACML [6], *etc*. The relevant standard for a discussion of access control is the Security Assertion Markup Language (SAML) [21]. The goal of SAML is to provide a means for exchanging security information across organizational boundaries, a requirement if the SOA is to reach its full potential.

The SAML specification is quite general in the kind of assertions that can be made, but most of the examples include a specification of the user's identity. For example, the SAML Technical Overview [21] includes the statement, "At the heart of most SAML assertions is a ***subject*** (a principal – an entity that can be authenticated – within the context of a particular security domain) about which something is being asserted." The Liberty Alliance, which is developing a framework for distributed identity management, has adopted SAML 2.0, another indication of the importance of identity assertions in SAML.

[1] This document expands upon [14] and freely copies some of the text.

It is no surprise, then, that most implementations based on the SOA tie access control decisions to the identity of the requester. This approach is spelled out in the introduction to the SAML specification [21], which states,

> "For example, a typical assertion from an identity provider might convey that 'This user is John Doe, he has an email address of john.doe@company.com, and he was authenticated into this system using a password mechanism.' A service provider could choose to use this information, depending on its access policies, to grant access to local resources."

To judge by the preponderance of talks at security conferences, such as RSA 2007 [25], most implementers assume that a Federated Identity Management (FIdM) framework is needed to associate an access policy with a given identity when crossing organizational boundaries. Based on the problems they report encountering, people implementing these systems are learning that federating access policies is much harder than federating identities.

Left unspecified in the SAML specification is how the service provider uses the identity of the requester to make access control decisions. Typically, the service uses the identity to look up the appropriate policy in some local database and bases the access decision on that information. So, it appears that the identity of the requester isn't the critical information; it is the authorization information in the database that matters. If that is indeed the case, why not just have the request convey the authorization information instead of or in addition to the requester's identity? This report shows that managing access policies using explicit authorizations with Federated Access Management (FAccM) is simpler than managing access policies using FIdM.

In the remainder of this report, we'll describe the Zebra Copy scenario, explain why FIdM is not a good solution, show the advantages of FAccM using Authorization-Based Access Control (ABAC), and show how to express FAccM with SAML assertions. In the Appendix, we'll walk through sample code that implements the Zebra Copy scenario.

2. The Scenario

Zebra Copy has just introduced a service for printing high quality brochures of up to 20 pages in press runs ranging from 10 to 10,000. HP has a contract with Zebra Copy for other services that are used by about 2,000 HP employees. That contract has just been extended to include the new service.

The Hewlett-Packard marketing department provides brochures customized for each trade show where HP has a booth.[2] Employees of that department are eager to use the new service from Zebra Copy. Some employees in this department work the large trade shows, but Bob is responsible for the smaller ones. He never needs to print more than 500 copies for any show.

[2] We have no idea if this statement is true, but we'll assume it is for the scenario.

The files to be printed are quite large, and Zebra Copy doesn't want to hold them until the job is processed. Instead, Zebra Copy requires HP provide a service that it can invoke to get the files for printing. The service invocation specifies the file to be used for that particular print job.

Bob works with a number of contractors to produce the brochures. When he gets busy or is to be away from the office, he often needs one of them to handle the routine work of ordering the brochures. These people come and go quite frequently, so Bob needs to keep close tabs on exactly what they're allowed to do.

Zebra Copy bills HP for all jobs ordered by people with the proper rights. HP wants to track those orders so it can bill back to the department the order came from. That's not too hard for regular employees, but it is an issue for contractors who work for more than one department. Bob is expected to monitor how much is spent on brochures for the trade shows he deals with. That way he can immediately report any errors or misuse.

In the next two sections, we'll contrast how Bob's access to the service is managed with Federated Identity Management (FIdM) based on Identification-Based Access Control (IBAC) and Federated Access Management (FAccM) based on Authorization-Based Access Control (ABAC). We'll assume that all accesses are via web services running as part of a Services Oriented Architecture (SOA) environment.

3. The Federated Identity Approach

HP needs to specify which of its employees may use the Zebra Copy Brochure service. If the service was run by HP, that policy could be expressed by an entry in an access control list. The problem is that HP doesn't control the Zebra Copy ACL. Instead, HP needs a policy engine to express such rules.

Let's say that HP maintains such a policy engine describing which employees may use the Zebra Copy Brochure service and how large a press run each is allowed to order. HP can set up an automated system that will forward policy changes to the relevant parties. So, when Bob is granted the right to order press runs of up to 500 copies[3], the Brochure service gets a message telling it to set up an account for Bob with that limit. The web service returns a credential Bob can use to identify himself to Zebra Copy. That's needed because Zebra Copy and HP may use different authentication mechanisms. Bob gets an email with his credential and a WSDL description of the Zebra Copy web service.

Since Zebra Copy needs to be able to upload the file, Bob has to do some setup. Let's assume HP runs a web service something like Microsoft SharePoint. If Bob doesn't already have a directory on the HP File Content service, he needs to get one[3]. Next, he sets up an account for the Brochure service to use. He also creates a folder to contain the files to be printed and adds read permission for the account he just created.

[3] How that decision is made is left as an exercise for the reader.

Bob needs to set up his account at Zebra Copy before he can order any jobs, so he invokes the Zebra Copy setup service. Among other things, this service allows Bob to provide Zebra Copy the credential it will need to read the files to be printed.

When Bob wants to place an order, he runs a program that invokes the Zebra Copy Brochure web service. The parameters to that service include the name of the file and the number of copies to be printed. When Bob's order is submitted to Zebra Copy, the identity associated with the order is checked against the list of authorized users. If Bob's credentials match an entry in the Zebra Copy database, the system verifies that the order is consistent with the policy associated with Bob. In this case that means the number of copies must be less than 500. Only then is the order processed. When Bob needs Alice, a contractor to order brochures for him, he must tell HP to tell Zebra Copy to set up an account for her.

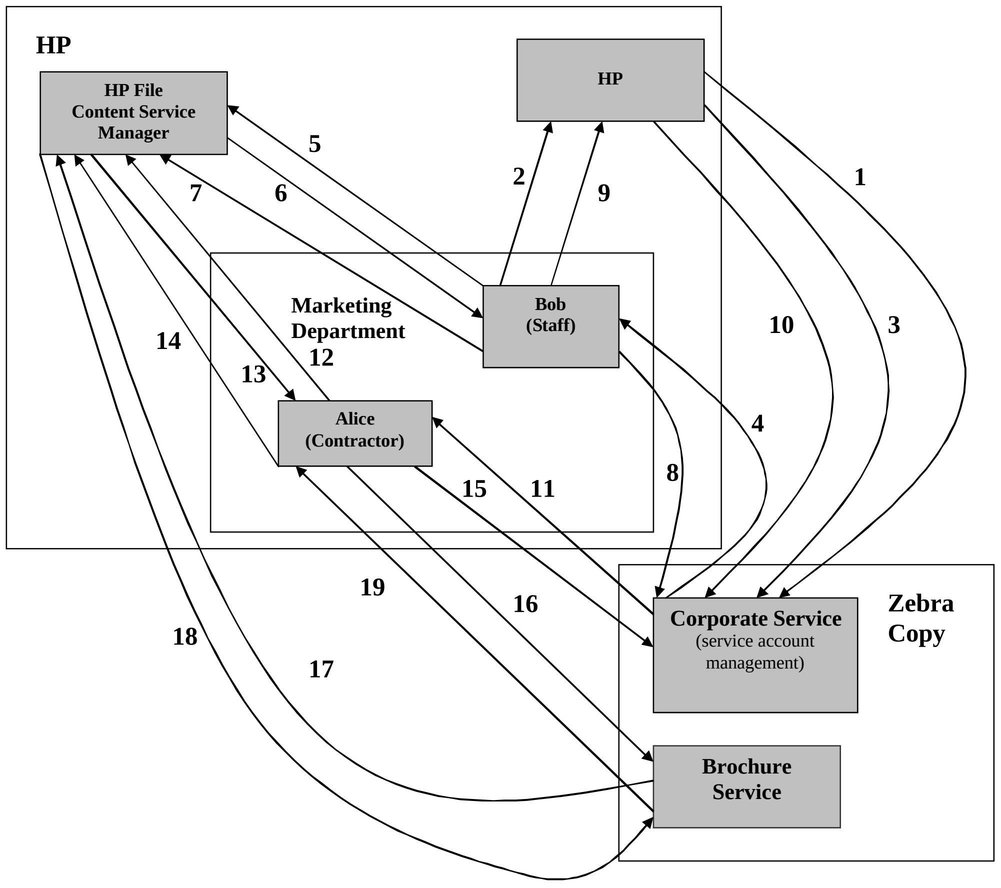

Figure 1: Workflow with Identification Based Access Control.

Figure 1 summarizes the scenario using Identification Based Access Control. Notice the absence of a line in the figure connecting Alice and Bob. That indicates the lack of an easy way to know that Bob is responsible for Alice's access to the Brochure service.

1. HP signs a contract with Zebra Copy.

2. Bob requests permission to use the Brochure service.
3. If approved, the request is forwarded to Zebra Copy.
4. Zebra Copy creates an account for Bob, puts an entry in its ACL, and returns his credentials to him.
5. Bob gets an account at the HP File Content service and creates an account for Zebra Copy.
6. The HP File Content service returns credentials for Zebra Copy to use.
7. Bob creates a directory in the file service, setting the ACL appropriately.
8. Bob creates his account at Zebra Copy, providing to Zebra Copy the credentials for the HP File service.
9. Bob asks HP to set up an account for Alice at Zebra Copy.
10. If approved, the request is forwarded to Zebra Copy.
11. Zebra Copy creates an account for Alice, puts an entry in its ACL, and returns his credentials to her.
12. Alice gets an account at the HP File Content service and creates an account for Zebra Copy. (She may not know that one already exists.)
13. The HP File Content service returns credentials for Zebra Copy to use.
14. Alice creates a directory in the file service, setting the ACL appropriately.
15. Alice creates her account at Zebra Copy, providing to Zebra Copy the credentials for the HP File service.
16. Alice submits her order to the Brochure service.
17. The Brochure service uses its credentials to request the file.
18. The contents of the file are returned to the Brochure service.
19. The Brochure service returns the status report.

The way Zebra Copy manages its services was a good example of FIdM based on Identification-Based Access Control (IBAC). It's quite familiar, but there are a number of problems. Note that Role-Based Access Control (RBAC) [8] and Policy-Based Access Control (PBAC) [2] don't help much since they address only the manageability problems.

3.1. Manageability

There is substantial management overhead. When Bob changes jobs, HP must make sure its policy is updated, which should result in Bob losing some of his rights and gaining others. In our example, Zebra Copy will have to update its ACLs to reflect these changes. Companies like HP can have10,000 or more business partners. Even small companies like Zebra Copy often have hundreds of customers. The cost of updating the access lists can be substantial.

The approach is unmanageable for users, too. Bob may work with dozens of business partners. He may well end up with a different credential for each of them. Worse, each might be based on a different technology, perhaps as similar as SAML 1.1 and SAML 2.0 or as different as X.509 and Kerberos. Bob will have to learn how to use each of them. The number of tools being developed to simplify things for users is proof that the problem is real. These tools include a variety of Single Sign-On (SSO) products [1, 4, 24], and identity management tools, such as Card Space [19]. The problem is getting all the business partners Bob deals with to agree on a single approach.

3.2. Ambient Authorities

Every request Bob makes to Zebra Copy is accompanied by proof of his identity. Zebra Copy searches its policy database entries for him looking for a match with the request. If a match is found, the request is honored. It is very hard for Bob to give a process running on his behalf a subset of his rights. That means a virus running in Bob's browser can do anything Bob is allowed to do, even if he doesn't want it done. For example, Bob might have logged in to check his account balance, but the virus orders 500 copies of a 20 page brochure containing Bob's medical records. Single Sign-On exacerbates the problem by giving each process Bob runs even more authority.

3.3. Delegation

Bob is busy as the deadline for a trade show approaches. He'd like his new contractor, Alice, to handle ordering the needed brochures. He tells HP to authorize Alice to make up to 100 copies at Zebra Copy. HP, in turn, tells Zebra Copy. Zebra Copy sets up an account for Alice. Bob also needs Alice to have access to the directory where she'll put the file to be printed. If Alice doesn't already have an account at the HP File Content service, Bob will have to arrange to have one set up. Once that's done, Bob can add Alice to the list of authorized users. In many cases Bob won't have permission to change the ACL on his directory will have to ask a sysadmin to do it for him. This process is so onerous that people often share credentials, such as passwords and private keys [23].

3.4. Revocation

Bob returns and takes over ordering brochures again. He needs to remove Alice's right to do that job and her access to the directory holding the files to be printed. He changes the ACL at the HP File Content service if he can or asks a sysadmin to do it for him. He also tells HP to tell Zebra Copy to remove Alice's entries from their databases. But what if Alice had access for a reason independent of Bob? For example, Alice may also be preparing brochures for Edward at HP. By revoking Alice's access, Bob is inadvertently preventing Edward from getting his job done.

3.5. Responsibility tracking

An order was placed telling Zebra Copy the name of the file to be printed. The HP computers can audit the fact that Zebra Copy accessed the file and that the file was in a directory assigned to Bob. The HP computers have no way of verifying that Alice placed the order. That's not much of an issue for a sales brochure, but it is for medical records accessed at a hospital. Furthermore, there is no easy way to know that Bob authorized Alice to make such requests.

3.6. Confused Deputy

Bob invokes the brochure service, providing the names of the file to be uploaded and an image file at Zebra Copy. The brochure service uploads the input file, writes the image file, and updates the accounting records. Bob needs to know the name of the accounting file so he can monitor his expenditures. If Bob specifies the name of the accounting file where he should have put the name of the image file, Zebra Copy will end up using its privileges to overwrite the accounting information with the image. That happens because Zebra Copy has no way to assign Bob's rights to arguments of its invocations.

3.7. Transitivity

Let's say the brochure service uses a file read service to get Bob's file. The question is whose authentication gets used. It can't be Bob's because the file read service never heard of him. However, care is needed because the Brochure service is likely to have rights it doesn't want to exercise on Bob's behalf, such as the ability to update accounting records.

Another issue comes up when the file read service reads the file from the HP file service. It can't use its own identity because HP never heard of it, so it must impersonate the Brochure service. That might be a serious issue if the file read service is run by one of Zebra Copy's business partners. There are also many cases where such impersonization isn't enough. Something as simple as outFile.write(infile.read()) won't be allowed unless one identity has both permission to read the input file and permission to write the output file.

3.8. Policy Compliance

Unbeknownst to Bob, only HP employees are allowed to use Zebra Copy's Brochure service. When Bob tries to delegate to Alice, HP denies the request, and the policy is enforced. Unfortunately, all delegations are difficult, and most of them don't violate policy. People get around this problem by sharing credentials. In our scenario, Bob tells Alice his Zebra Copy password, and the policy is violated.

4. Using Federated Access Management

A simpler approach is to implement FAccM with Authorization-Based Access Control (ABAC). Most of this report consists of a detailed description of the code that implements the Zebra Copy scenario with FAccM. This section summarizes the workflow of that implementation within the SOA framework. Details come with the discussion of the code.

The owner of the Brochure service at Zebra Copy creates a SAML certificate containing the location and name of the service, the owner's public key, and an authorization field specifying that press runs of up to 10,000 will be approved. That certificate is delegated to whoever in Zebra Copy is responsible for handling contracts to use the Brochure service. Delegation involves creating a new certificate containing the public key of the delegatee and a copy of the assertion being delegated. It is only valid for someone who knows the private key corresponding to the public key in the assertion. This new certificate is signed by the delegator.

When Zebra Copy signs its contract with HP, it delegates to HP a SAML certificate representing the right to use the Zebra Copy service. HP delegates to Bob the right to order press runs of up to 500 copies based on information in the HP policy. If Bob wants to print 37 brochures, he generates a new key pair, creates a SAML certificate specifying the new public key, and sets the job limit to 37. Bob then passes that certificate and the new private key to the program that will place the order. Bob delegates to Alice the right to make press runs of up to 100 copies the same way.

Bob gets an authorization to create files maintained by the HP File Content service. Bob uses his authorization to create a file that the Brochure service will upload and a SAML certificate granting read access to that file. When Bob submits his job, he delegates this certificate to the service he is invoking and uses the assertion in it to designate the file to be read. The brochure service uploads the file by presenting this certificate to the HP file service.

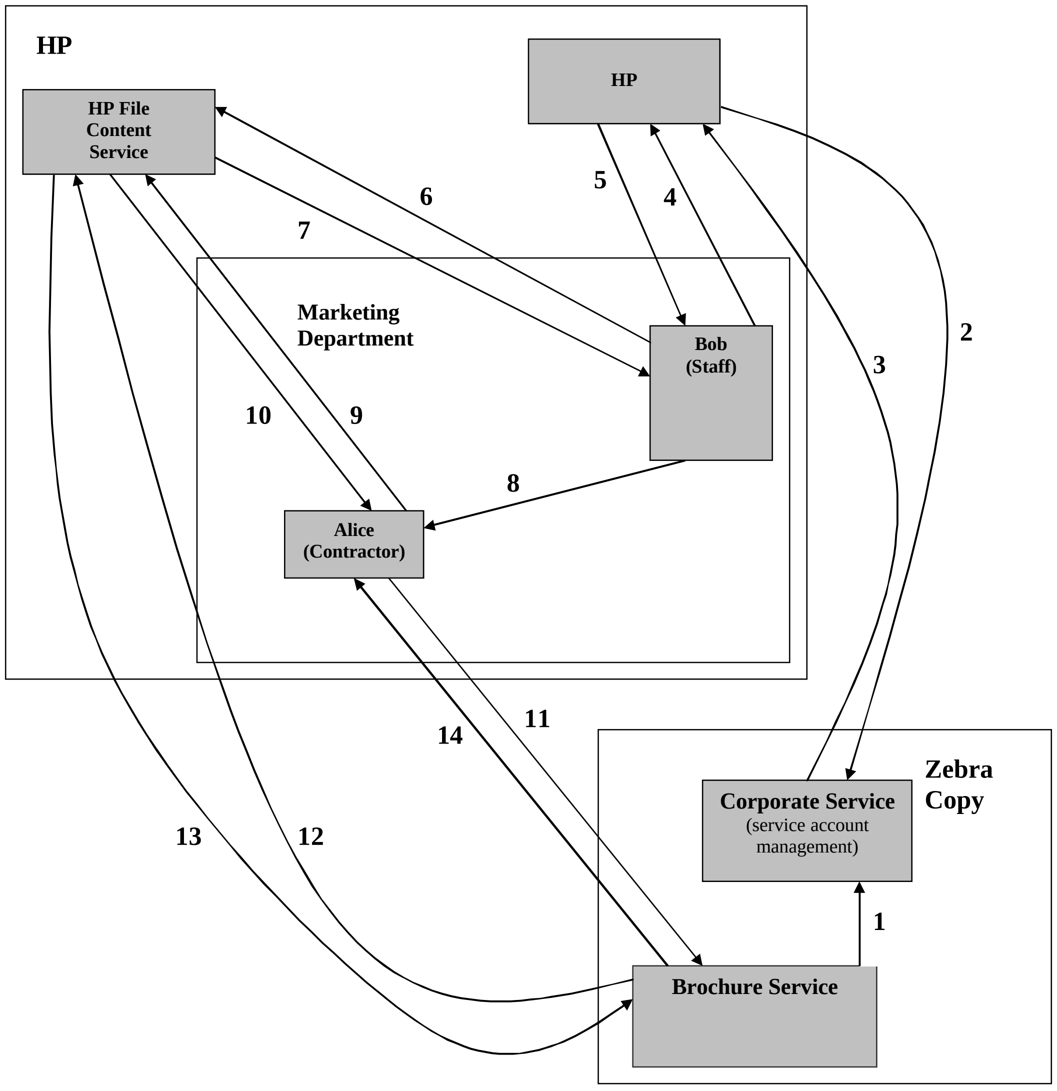

Figure 2: Workflow with Federated Access Management.

When Alice moves on to another job, Bob sends a message directly to the brochure service telling it to stop honoring the certificate Bob delegated to Alice's. Once that's done Alice won't have the right to order copy jobs, so any delegations she made will be invalid.

Figure 2 shows the workflow when using Federated Access Management.

1. The owner of the Brochure service creates a self-signed certificate granting all rights to the service to Zebra Copy Corporate.
2. HP and Zebra Copy sign a contract granting HP the right to use the service.
3. Zebra Copy delegates to HP a certificate granting the part of its rights.
4. Bob asks for access to the Brochure service.
5. HP delegates some of his rights to Bob.
6. Bob asks the File Content service for authorization to create a file.
7. The File Content service sends an authorization for that file to Bob.
8. Bob delegates to Alice a subset of his rights to use the Brochure service.
9. Alice asks the File Content service for authorization to create a file.
10. The File Content service sends an authorization for that file to Alice.
11. Alice invokes the Brochure service, delegating to it the right to read the file.
12. The Brochure service uses that authorization to read the file.
13. The File Content service returns the contents of the file.
14. The Brochure service sends Alice the status report.

Using FAccM with ABAC has a number of advantages.

1. **Manageability:** The flow of authority is exactly the way we have been managing companies and the military for hundreds of years. People with responsibilities delegate to their subordinates without needing approval from some third party who may lack adequate knowledge of the situation to make an informed decision.
2. **Ambient authorities:** There are none. Bob gave the process acting on his behalf exactly the subset of his rights that he wanted it to have.
3. **Delegation:** Delegation is the key to simplifying distributed policy management. Note the way Bob manages his rights without needing to bother people who know nothing about his organization.
4. **Revocation:** Revocation is simple and only involves the two end-points of the request. That reduces the delay in revoking access and simplifies the mechanism. Further, there is no danger that Bob revoking the rights that he delegated to Alice will affect the authorization Alice received from Edward.
5. **Confused deputy:** Deputies cannot become confused because the authorization and the designation are combined. Each resource is tied to exactly the intended set of rights.
6. **Transitive access:** Forwarding rights is simple. Should the Brochure service need to invoke a web service provided by a business partner, it simply delegates the needed rights. Some of those may come from Bob; some may come from the

Brochure service. Hence, outFile.write(inFile.read()) works when the necessary authorizations have been delegated.

7. **Policy Compliance:** Delegation is easy, even if it violates policy. However, identity and attribute information can be required with the authorization. That information can be used to enforce policy. In our scenario in which only HP employees may use the brochure service, the service can deny Alice's request because she can't prove she is an HP employee.

5. Using SAML Certificates as Authorizations

So far, the discussion hasn't provided any details of how to use SAML assertions as authorizations. In this section, we'll walk through some of the examples generated from the sample code described in Appendix D. The sample code is available for download from HP [16].

The basic idea is to have a service create a certificate issued to its organization. In the following example, the Brochure service issues a certificate to Zebra Copy Corporate granting full rights to the service. When a contract is signed with HP, Zebra Copy Corporate can delegate a subset of its rights to HP. The evidence that the Brochure service should honor a request signed by HP is the assertion granting Zebra Copy Corporate the rights being delegated. Subsequent delegations allow further limits to the rights being granted, right down to the process running on behalf of the user. The approach presented here is similar to the way HP's e-speak product [12] used Simple Public Key Infrastructure certificates [5].

a. Initial Authorization Certificate

The assertion named ZebraCopyCorporateBrochureSevice.xml in ApplicationDir under the ZebraCopySample directory represents the right to use all aspects of the service and specifies any limits that the service will not exceed. Here, Zebra Copy Corporate is granted the right to use this service. This certificate is created when the brochure service is ready to be brought on line. It starts with some XML boiler plate specifying the version and character encoding.

```
<?xml version="1.0" encoding="utf-8"?>
```

Next we see the first tag that describes what the XML file contains. The tag "certificate-log" is used for logging and debugging. It is not part of the SAML authorization certificate. It explains what the enclosed assertion is about and when it was produced. Each time you run the sample code, the time stamp will change.

```
<certificate-log
   timestamp="5/7/2007 10:18:07 AM"
   label="Zebra Copy Corporate Authorization to Brochure Service">
```

The assertion contains statements the issuer makes about the rightful user of the certificate. The rightful user is whoever knows the private key corresponding to the public key contained in the SubjectConfirmationData in the assertion. In a system based on Identity- or Role-Based Access Control (IBAC, RBAC), certificates are used to make assertions about identity or role of the rightful user. In a Policy-Based Access Control (PBAC) system, certificates are used to make assertions about properties the rightful user has, such as employment or citizenship. When using the certificate as an authorization in an Authorization-Based Access Control (ABAC) system, we only need one assertion that specifies the rights being granted.

In the following Assertion, we see that the issuer is the Brochure Service Authority. Each assertion also has a unique ID and a time stamp set to the time the certificate was created. Version information that we won't be using is included.

```
<saml:Assertion MajorVersion="1" MinorVersion="1"
  AssertionID="_cc0a2088-905f-41f8-a47c-dbd720fc2fc0"
  Issuer="Brochure Service Authority"
  IssueInstant="2007-04-03T16:57:51Z"
  xmlns:saml="urn:oasis:names:tc:SAML:1.0:assertion">
```

We can set a time interval during which this certificate is valid. In an IBAC, RBAC, or PBAC system, the interval is usually made short because it is so difficult to revoke a privilege. That's not the case with ABAC, so we'll set the maximum possible range.

```
<saml:Conditions NotBefore="0001-01-01T00:00:00Z"
                 NotOnOrAfter="9999-12-31T23:59:59Z" />
```

The next part of the certificate specifies the service that will be invoked. In this case, the certificate permits access to the Zebra Copy Brochure service.

```
<saml:AuthorizationDecisionStatement
   Resource="http://www.zebracopy.com/services/BrochureService.asmx"
   Decision="Permit">
```

Next we provide information of who is the rightful user of the certificate, Zebra Copy Corporate in our example. Note that the name is specified as an X.509 distinguished name. We're only using two of the possible fields, CN for common name and O for organization. The Brochure service gets the subject information it will need from the Zebra Copy Corporate X.509 certificate. The SubjectConfirmationData is a Base64 encoding of the subject's X.509 certificate. The ellipsis denotes that there are many characters in this field that have been elided.

```
<saml:Subject>
    <saml:NameIdentifier
        NameQualifier=""
        Format=
        "urn:oasis:names:tc:SAML:1.1:nameidformat:X509SubjectName">
            CN="Corporate O=Zebra Copy"
    </saml:NameIdentifier>
    <saml:SubjectConfirmation>
      <saml:ConfirmationMethod>
        urn:oasis:names:tc:SAML:1.0:cm:bearer
      </saml:ConfirmationMethod>
      <saml:SubjectConfirmationData>
          MIIBkTCB+wIERWP+/jANBgkqhkiG9w0BAQQFADAQ...
      </saml:SubjectConfirmationData>
    </saml:SubjectConfirmation>
</saml:Subject>
```

The next field specifies which actions on the brochure service are being authorized. Notice that revocation is done by invoking a method on the service. This method allows delegators to revoke certificates they delegate to others.

```
<saml:Action
   Namespace="http://www.zebracopy.com/services/BrochureService.asmx">
     Print
</saml:Action>
<saml:Action
   Namespace="http://www.zebracopy.com/services/BrochureService.asmx">
     Revoke
</saml:Action>
```

That ends the first authorization statement.

```
</saml:AuthorizationDecisionStatement>
```

Next, we specify any restrictions on the use of the service. Note that the subject field inside the AttributeStatement block is identical to the previous AuthorizationDecisionStatement, so it's been elided. It seems redundant to have the same subject appear in both places, but most people use AttributeStatements to say things about subjects. Since it would be an error not to provide a Subject in that case, the XML serializer provided by the SAML SDK that we used requires each AttributeStatement include a Subject.

```
<saml:AttributeStatement>
   <saml:Subject>
      ...
   </saml:Subject>
```

The brochure service allows press runs of up to 10,000 copies. Note that the SOA standards don't specify how to tie an Attribute specifying any limits on the use of the service to the Action of the corresponding method. That's not a problem for this example, but it could be for one with multiple methods that take arguments of the same type. The trick we use is to define a distinct Namespace URI for each Action/Attribute pair. Notice in the snippet below the end of the value of the AttributeNamespace is the Action this AttributeValue applies to. There is no name space explosion problem because the scope of the name space is limited to the SAML Assertion it appears in.

```
<saml:Attribute
   AttributeName="PrintLimit"
   AttributeNamespace="urn:zebra:copy:brochure_service:Print">
   <saml:AttributeValue>
      10000
   </saml:AttributeValue>
</saml:Attribute>
```

That's the end of what this certificate authorizes.

```
</saml:AttributeStatement>
```

Next is the signature needed to protect the certificate from changes and forgery and use by those who don't know the corresponding private key. First, there is the information on the algorithm used to sign the certificate. See the SAML documentation [21] for details

```
<Signature xmlns="http://www.w3.org/2000/09/xmldsig#">
   <SignedInfo>
     <ds:CanonicalizationMethod
         Algorithm="http://www.w3.org/2001/10/xml-exc-c14n#" \
         xmlns:ds="http://www.w3.org/2000/09/xmldsig#" />
     <SignatureMethod
         Algorithm="http://www.w3.org/2000/09/xmldsig#rsa-sha1" />
     <Reference URI="#_97fcf75e-fcd2-4781-bff9-cc3762643652">
        <Transforms>
           <Transform
               Algorithm=
             "http://www.w3.org/2000/09/xmldsig#enveloped-signature" />
            <Transform Algorithm=
              "http://www.w3.org/2001/10/xml-exc-c14n#">
              <ec:InclusiveNamespaces
                PrefixList="#default code ds kind rw saml samlp typens"
                xmlns:ec="http://www.w3.org/2001/10/xml-exc-c14n#" />
            </Transform>
          </Transforms>
          <DigestMethod
              Algorithm="http://www.w3.org/2000/09/xmldsig#sha1" />
          <DigestValue>1vcLB7+KYAdflO/J58S1bR1obPs=</DigestValue>
        </Reference>
      </SignedInfo>
```

The next piece is the Base64 representation of the signed hash of the document. The ellipsis denotes that much of the content has been elided.

```
<SignatureValue>
  Ya3leEcWzCC2TmRAdC0lwUvWU3kJi...
</SignatureValue>
```

The last piece of the signature is information on the key used to compute the SignatureValue. In this case it is the Base64 representation of the issuer's X.509 certificate. As before, much of the content has been elided.

```
<KeyInfo>
    <X509Data>
      <X509Certificate>
         MIIBnDCCAQUCBEVj/...
      </X509Certificate>
    </X509Data>
  </KeyInfo>
</Signature>
```

Finally, we close the remaining open tags, and we're done.

```
   </saml:Assertion>
</certificate-log>
```

Zebra Copy Corporate is now authorized to use its own Brochure service.

b. Delegation to HP

When the contract between the two companies is signed, Zebra Copy delegates to HP a portion of its rights to use the brochure service. This certificate is called ZebraCopyToHP.xml in the ApplicationDir. After the boilerplate and logging tag, the certificate has a single assertion. This time the issuer is Zebra Copy corporate.

```
<saml:Assertion MajorVersion="1" MinorVersion="1"
   AssertionID="_7c629054-fdc3-4911-95ba-3467b522b71a"
   Issuer="Zebra Copy Corporate Centralized Authority"
   IssueInstant="2007-04-04T23:22:18Z"
   xmlns:saml="urn:oasis:names:tc:SAML:1.0:assertion">
```

The conditions are set as before, except now the valid interval of the certificate is the contract term.

```
<saml:Conditions
   NotBefore="2007-04-01T00:00:00Z"
   NotOnOrAfter="2008-04-01T00:00:00Z" />
```

Next is the actual authorization, which is the right to use the brochure service.

```
<saml:AuthorizationDecisionStatement
   Resource="http://www.zebracopy.com/services/BrochureService.asmx"
   Decision="Permit">
```

As before, the authorization statement specifies who may use the authorization, HP in this case.

```
<saml:Subject>
   <saml:NameIdentifier NameQualifier="" Format=
      "urn:oasis:names:tc:SAML:1.1:nameid-format:X509SubjectName">
         CN="HP O=Hewlett-Packard Company"
   </saml:NameIdentifier>
```

Except for the actual data, the SubjectConfirmation is the same as before, so its content has been elided.

```
   <saml:SubjectConfirmation>
      ...
   </saml:SubjectConfirmation>
</saml:Subject>
```

Zebra Copy gets to decide which of the methods that it is allowed to use that it wishes to grant to HP. In this case, there is only one method (ignoring revocation), so it is granted.

```
<saml:Action
```

```
   Namespace="http://www.zebracopy.com/services/BrochureService.asmx">
      Print
</saml:Action>
<saml:Action
   Namespace="http://www.zebracopy.com/services/BrochureService.asmx">
      Revoke
</saml:Action>
```

The Brochure service will need to know that the delegation comes from someone authorized to do so. We use the assertion from Zebra Copy Corporate's authorization certificate as that proof. The elided text is the rest of the Assertion granting rights to Zebra Copy Corporate, which is exactly what was described in Section 5a.

```
<saml:Evidence>
   <saml:Assertion MajorVersion="1" MinorVersion="1"
     AssertionID="_cc0a2088-905f-41f8-a47c-dbd720fc2fc0"
     Issuer="Brochure Service Authority"
     IssueInstant="2007-04-03T16:57:51Z"
     xmlns:saml="urn:oasis:names:tc:SAML:1.0:assertion">
      ...
</saml:Evidence>
```

That ends the

```
</saml:AuthorizationDecisionStatement>
```

Now we specify the attributes associated with each allowed action.

```
<saml:AttributeStatement>
```

Zebra Copy Corporate has the authority to print press runs of up to 10,000 copies. However, the contract with HP is limited to press runs of no more than 5,000 copies.

```
<saml:Attribute
   AttributeName="PrintLimit"
   AttributeNamespace="urn:zebra:copy:brochure_service:Print">
   <saml:AttributeValue>
      5000
   </saml:AttributeValue>
</saml:Attribute>
```

That's the end of what this certificate authorizes.

```
</saml:AttributeStatement>
```

Next, we see the signing information. It's similar to that in the previous certificate except that the issuer is Zebra Copy Corporate instead of the Brochure service.

```
<Signature xmlns="http://www.w3.org/2000/09/xmldsig#">
   ...
</Signature>
```

Finally, we close the remaining open tags, and we're done.

```
   </saml:Assertion>
</certificate-log>
```

We now see the pattern. When HP delegates to Zelda, it includes its authorization certificate as evidence. That certificate includes as evidence the certificate authorizing Zebra Copy Corporate to use the service. That process continues through all subsequent delegations. At each delegation the rights that are authorized can be constrained, and typically get reduced when the certificate is delegated. Each certificate shows the complete delegation chain, which is exactly what is needed for responsibility tracking. If any of the certificates is revoked, all downstream delegations will be invalid because the Evidence won't pass the verification test.

6. Conclusions

Large scale distributed systems are inherently different from stand-alone computers. There is little reason to think that designs for the latter are applicable to the former. Yet that's what Identification-Based Access Control does. It takes an access model designed to deal with mainframe users and tries to make it work in a distributed environment. Authorization-Based Access Control, which has advantages even on stand-alone computers, is a better match to the requirements of distributed systems that span administrative domains.

The SOA is quite different from the familiar systems that use Identity-Based Access Control. The 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; and no one party is in charge of managing updates. If the SOA is to achieve its goals, it is critical to reduce the coupling between domains to the greatest possible extent. Identity-Based Access Control results in a tightly coupled system, one that requires federated identity management, results in information leakage between domains, and makes delegation and upgrade difficult. Authorization-Based Access Control can simplify federated access management while increasing system flexibility and responsiveness to the people trying to get work done inside the system.

We have discussed one possible format of the authorizations, but there are many options. Which one is chosen by a domain depends on many factors. The approach described here is to use SAML to assert the authorization. Although the SAML documentation focuses on providing identity information, the specification allows other assertions. All we need do is assert that an access is allowed, and we've converted the SAML assertion from one about identity to one about authorization.

Zebra Copy was a real company, a Mom and Pop shop, near HP Corporate headquarters with a contract with HP. It controlled access with Identification-Based Access Control, employing a person to manually check the name on print orders. Managing the access list must have been a substantial burden. Zebra Copy is no longer in business. The cost of managing that list isn't the only reason the company failed, but it may have been a factor.

Acknowledgements

We would like to thank Joe Pato for getting us to implement the Zebra Copy scenario and Raj Rajagopalan for figuring out how to show ABAC in action. We'd also like to thank Suhayl Masud and Cat Okita for suggesting improvements to the text.

References

1. ActiveIdentity, Single Sign-On, http://www.actividentity.com/solutions/technology/esso__overview.php
2. Blaze, M.; Feigenbaum, J.; Lacy, J., "Decentralized trust management," Proceedings of IEEE Symposium on Security and Privacy, pp. 164-173, 1996.
3. ComponentSpace, SAML .NET Toolkit, http://www.componentspace.com/saml.net.aspx
4. Computer Associates, Single Sign-On, http://www.ca.com/us/products/product.aspx?id=166
5. Ellison, C., Frantz, B., Lampson, B., Rivest, R., Thomas, B., and Ylonen, T., "SPKI Certificate Theory", IETF RFC 2693. http://www.ietf.org/rfc/rfc2693.txt
6. Extensible Access Control Markup Language (XACML) V1.1, http://www.oasis-open.org/committees/xacml/repository/cs-xacml-specification-1.1.pdf
7. Extensible Markup Language (XML), http://www.w3.org/XML/
8. Ferraiolo, D. F. and Kuhn, D. R, "Role Based Access Control" *15th National Computer Security Conference*, 1992.
9. Ferrara, A. and MacDonald, M., *Programming .NET Web Services*, O'Reilly Media, Inc., 2002.
10. Gamma, E., Helm, R., Johnson, R., and Vlissides, J, *Design Patterns: Elements of Reusable Object-Oriented Software*, Addison-Wesley, Addison, Mass., 1995
11. Henning, M. and Vinoski, S., *Advanced CORBA Programming with C++*, Addison-Wesley, 1999.
12. Hewlett-Packard, *e-speak Architectural Specification*, Release A.03.14.00, 2001.
13. Housley, R, Ford, W., Polk, W., Solo, D. Internet X.509 Public Key Infrastructure Certificate and CRL Profile. IETF RFC 2459, January 1999. http://www.ietf.org/rfc/rfc2459.txt.
14. 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), http://www.hpl.hp.com/techreports/2006/HPL-2006-3.html
15. Knuth, D. E., *Literate Programming*, CSLI Lecture Notes #27, Center for the Study of Language and Information, Stanford, California, 1992.
16. Li, J. and Karp, A., "Zebra Copy sample code", http://www.hpl.hp.com/personal/Alan_Karp/ZebraCopy.zip
17. Li, J. and Karp, A., "Zebra Copy sample code with SOAP interception", http://www.hpl.hp.com/personal/Alan_Karp/ZebraCopyExtension.zip
18. Microsoft, MSDN, http://www.msdn.microsoftcom.
19. Microsoft, "Introducing Windows CardSpace", http://msdn2.microsoft.com/en-us/library/aa480189.aspx

20. Microsoft, “Using SOAP Extensions in ASP.NET”, http://msdn.microsoft.com/msdnmag/issues/04/03/ASPColumn/
21. OASIS, “Security Assertion Markup Language (SAML) 2.0 Technical Overview, Working Draft 05”**,** 10 May 2005, http://www.oasis-open.org/committees/download.php/12549/sstc-saml-tech-overview-2%5B1%5D.0-draft-05.pdf
22. Papazoglou, M.P and Georgakopoulos, D., “Service-Oriented Computing,” Communications of the ACM, Vol. 46, No. 10, pp. 25-8, Oct. 2003.
23. Ping Identity, “Reducing Account Sharing with Federated Single Sign-On”, Webinar, http://www.pingidentity.com/p/03yVcBqM?elq=F993B4D596D54D5B91838E8F7ECD6DE6
24. Ping Identity, Single Sign-On, http://www.pingidentity.com/resources/88
25. RSA Conference 2007, http://www.rsaconference.com/2007/US/.
26. Simple Object Access Protocol (SOAP) 1.1, W3C Note, http://www.w3.org/TR/2000/NOTE-SOAP-20000508/
27. Stojanovic, Z. and Dahanayake, A. (eds), *Service-Oriented Software System Engineering: Challenges and Practices*, Idea Group Publishing, 2005.
28. Universal Description, Discovery, and Integration (UDDI), http://www.uddi.org/.
29. Vinoski, S., “CORBA: Integrating Diverse Applications within Distributed Heterogeneous Environments,” IEEE Communications Magazine, vol.35, no.2, pp. 46-55, Feb. 1997.
30. Web Services Addressing (WS-Addressing), http://www.w3.org/Submission/ws-addressing/.
31. Web Services Description Language (WSDL) 1.1, W3C Note, http://www.w3.org/TR/wsdl.html.
32. XML-Signature Syntax and Processing, W3C Recommendation, http://www.w3.org/TR/xmldsig-core/

Appendix A. Introduction to the Sample Code

We have implemented all the web services in the Zebra Copy scenario in C# with the Microsoft .Net Framework 2.0 using Microsoft Visual Studio 2005 Professional Edition. A file containing the code can be downloaded from HP [16]. The distribution includes the ComponentSpace library [3] for .Net 1.1. This commercial package has a number of useful functions for manipulating SAML certificates. You may only use this package for the purposes of running the sample code. If you want to develop your own applications, there is a trial version you may use free for 30 days.

There are seven directories in the distribution.

1. **ApplicationDir**: The various key stores and certificates needed to run the sample applications. All SAML certificates generated during a run are also stored here.
2. **Authorization**: The code used to validate the generic part of the authorizations. The application specific checks, such as verifying the number of copies, are done in the actual service code.
3. **ExternalLibraries**: Contains the external DLLs that you will need to manipulate the SAML certificates.
4. **HPWebSite**: Code for the HP web service that handles the authorization from Zebra Copy and the HP web service Zebra Copy uses to upload the file to be printed.
5. **ServiceClassLibrary**: Client side code that is specific to the brochure service.
6. **UserApplication**: The code used to run the scenario.
7. **ZebraCopyWebSite**: Code for Zebra Copy corporate web service and the Brochure service.

We assume the reader is familiar with the tools being used and can configure the various web services. See the Microsoft documentation at MSDN online [18] or one of the many reference books [9, for example] for assistance.

Appendix B. Real Life versus the Sample

In real life, HP and Zebra Copy run independent web sites, and they start their web services independently. Each company has its own way of identifying its employees and services. A timeline of the scenario might look like the representation in Figure A1. The vertical dashed lines separate the two web sites and the user application. The solid horizontal lines represent requests; the horizontal dotted lines, responses.

Although other orderings are possible, Figure A1 shows the HP web site starting the HP Corporate web service (H), the Zebra Copy web site starting its Corporate web service (Z). The HP web site starts it File web service (F) while the Zebra Copy web site is starting the Brochure web service (B). B creates the SAML certificate granting Z the right to use B. Next we see H signing the contract with Z and getting back a delegation of Z’s right to use B. User Bob (U) asks to use B and HP delegates some of its rights to U. U asks F for the right to read and write the file to be printed and delegates the read authority to B. U then invokes B. B asks F for the file and prints the contents.

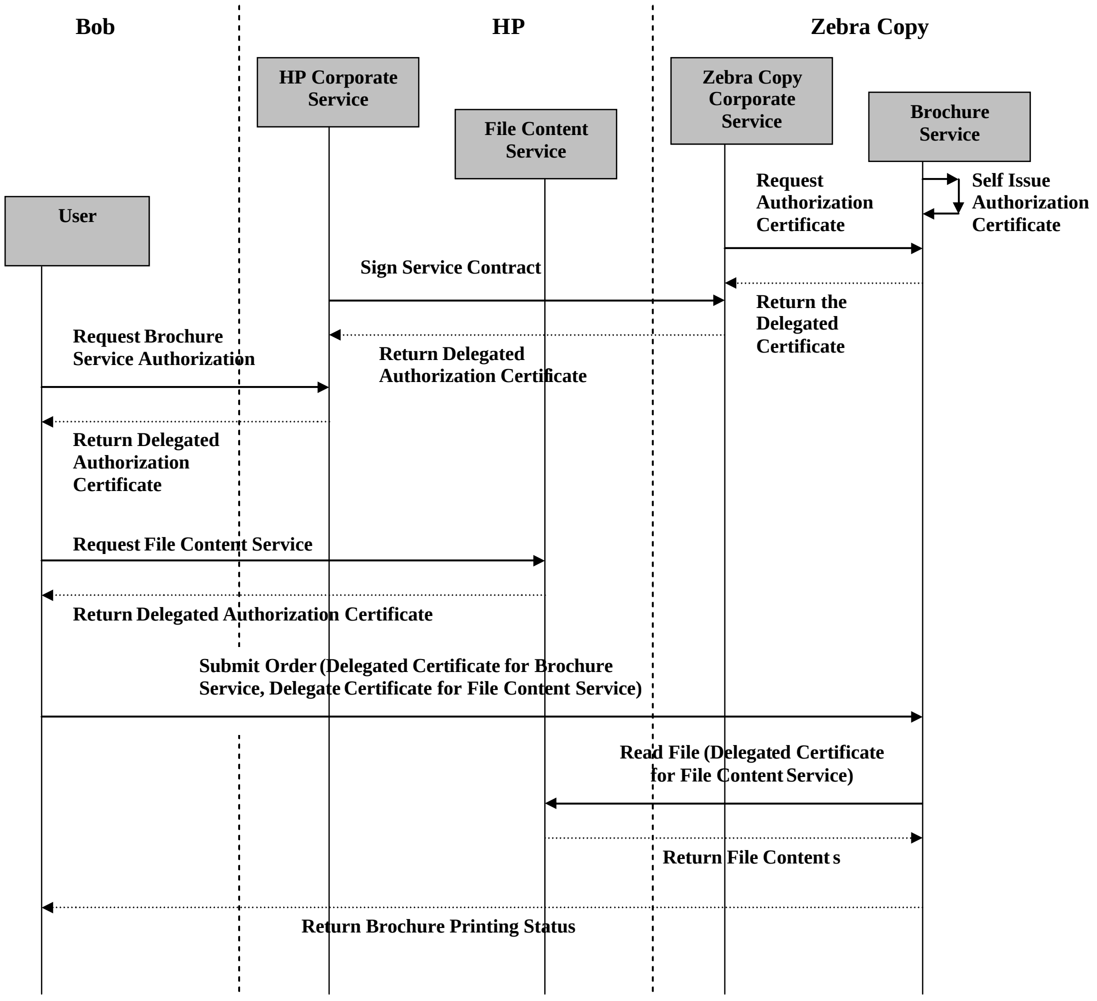
Bob
HP
Zebra Copy
HP Corporate Service
File Content Service
Zebra Copy Corporate Service
Brochure Service
User
Request Authorization Certificate
Self Issue Authorization Certificate
Sign Service Contract
Return the Delegated Certificate
Request Brochure Service Authorization
Return Delegated Authorization Certificate
Return Delegated Authorization Certificate
Request File Content Service
Return Delegated Authorization Certificate
Submit Order (Delegated Certificate for Brochure Service, Delegate Certificate for File Content Service)
Read File (Delegated Certificate for File Content Service)
Return File Contents
Return Brochure Printing Status

Figure A1: A possible real-world timeline for the scenario

Our sample was designed so that single stepping through the code starting with the user's application will eventually reach all the components, including the startup of the relevant web services. We do that by deferring the instantiation of each web service until the first time it is invoked. Figure A2 shows the timeline. User U instantiates the HP Corporate web service H, which instantiates the Zebra Copy Corporate web service Z, which in turn instantiates the Brochure service B. B creates the authorization certificate for Z, which Z delegates to H and H delegates to U. U then instantiates the HP File service F and gets back the authorization to a file. U delegates this certificate to B and invokes the service. B reads the file and prints the job.

Appendix C. ApplicationDir

This directory contains the certificates used to represent authorizations in the sample system and the files containing the signing keys of all parties. In an actual deployment, the files containing the keys would be stored with the corresponding entities.

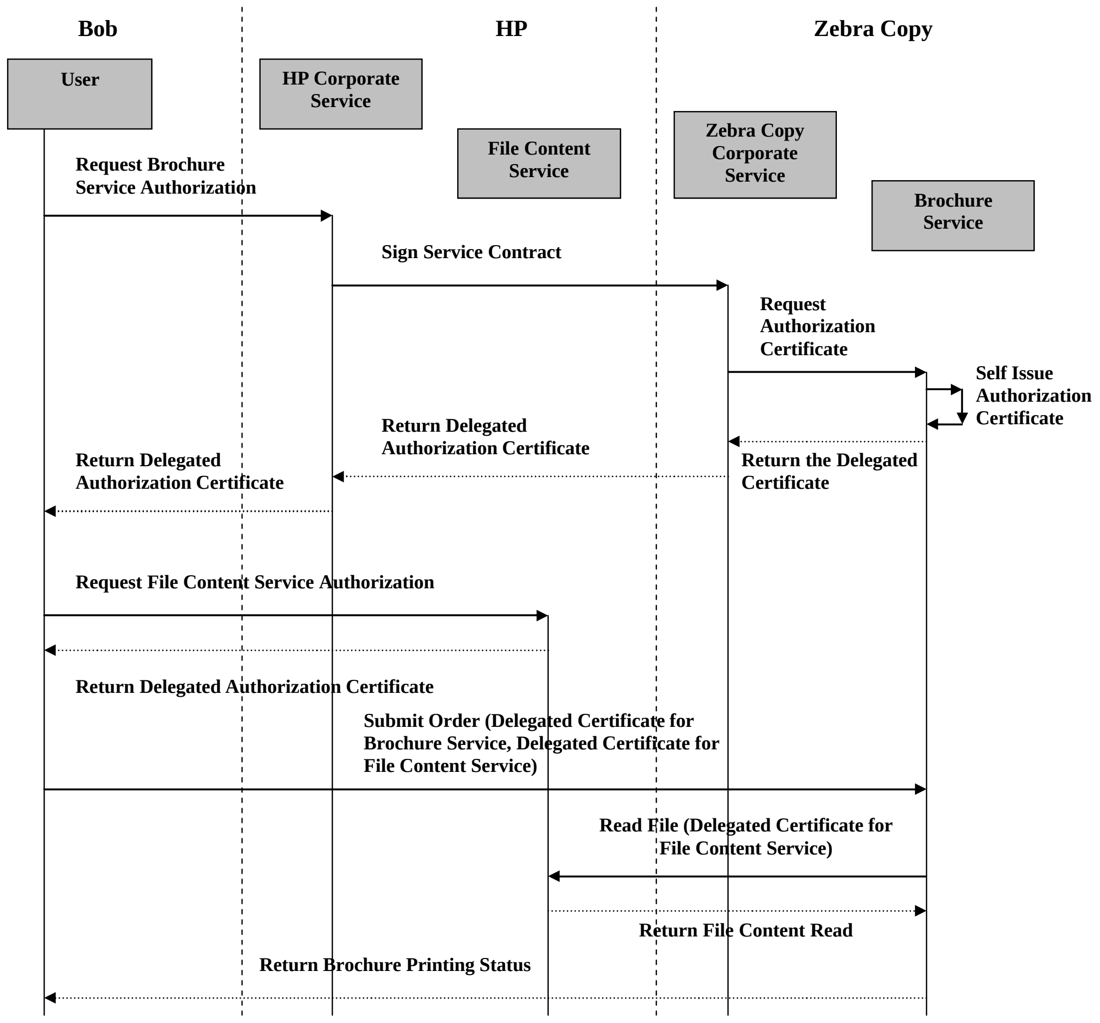
Bob
HP
Zebra Copy
User
HP Corporate Service
File Content Service
Zebra Copy Corporate Service
Brochure Service
Request Brochure Service Authorization
Sign Service Contract
Request Authorization Certificate
Self Issue Authorization Certificate
Return Delegated Authorization Certificate
Return the Delegated Certificate
Return Delegated Authorization Certificate
Request File Content Service Authorization
Return Delegated Authorization Certificate
Submit Order (Delegated Certificate for Brochure Service, Delegated Certificate for File Content Service)
Read File (Delegated Certificate for File Content Service)
Return File Content Read
Return Brochure Printing Status

Figure A2: The timeline in the sample implementation for the scenario.

The sample code uses X.509 [13] certificates to hold the keys used for authorization and delegation. In a real deployment the Brochure service would already have Zebra Copy's certificate, HP would have the certificates for Bob and Alice and would provide them to the file service; Bob would get Alice's certificate from HP; and HP would provide its certificate to Zebra Copy when the contract for the service is signed. At no point is a Certificate Authority needed.

All needed certificates are provided in ApplicationDir. If you encounter a problem with them, you can use the keytool that comes with the Java 1.5 distribution to generate the needed key pairs and certificates. On a Windows system keytool is often found in the bin directory of the JRE, *e.g.*,

```
C:\Program Files\Java\JRE1.5.0_10\bin
```

Generate a public/private key pair for each participant. The examples that follow assume that keytool is in your execution path. Note that each command must be entered on a single line.

```
keytool -genkey -keystore brochure.pfx  -alias brochure  -keypass
      password -dname "CN=Brochure Service O=Zebra Copy"
keytool -genkey -keystore zebracopy.pfx -alias zebracopy -keypass
      password -dname "CN=Corporate O=Zebra Copy"
keytool -genkey -keystore hp.pfx        -alias hp        -keypass
      password -dname "CN=HP O=Hewlett-Packard Company"
keytool -genkey -keystore hpfiles.pfx   -alias hpfiles   -keypass
      password -dname "CN=File Server O=Hewlett-Packard Company"
keytool -genkey -keystore bob.pfx       -alias bob       -keypass
      password -dname "CN=Bob Doe O=Hewlett-Packard Company"
keytool -genkey -keystore alice.pfx     -alias alice     -keypass
      password -dname "CN=Alice Jones O=Consultants R Us"
```

You will need to set the environment variable RSAHome to point to the directory holding these key stores. Say the sample is installed in C:\temp\ZebraCopySample. Then the response to

```
      C:\temp\ZebraCopySample>echo %RSAHome
```

should be

```
      C:\temp\ZebraCopySample\ApplicationDir
```

Appendix D. Code Walk-through

The sample code consists of two web sites, each running two web services, and a user application. Normally, each service in these two web sites is instantiated independently. For example, the Zebra Copy web site would start and initialize the Brochure Service and the service representing Zebra Copy Corporate. That's the right way to run your web services but not best if you want to explain them and see the service initializations. We have designed the sample code so that all of the code is touched if you start in the user application by initializing each service when the first request for it arrives. In this Appendix we'll single step the program to show all the steps in applying Federated Access Management to the Zebra Copy scenario.

We have left out all error checking code in the reference implementation. If something goes wrong during a run, you'll get an unhandled exception, and your web service will crash. That's not good coding practice, but it simplifies the tutorial.

Open the solution by double clicking on ZebraCopySample.sln in the ZebraCopySample directory. After Visual Studio opens, right click on the UserApplication project and set it as the StartUp Project. Open UserApplication.cs in the UserApplication project, and set a breakpoint at the line containing

```
User bob = new User("bob.pfx", "password");
```

If this is the first time you've opened the project, do a Rebuild. If this is not the first time you're running the application, make sure to stop the two web servers used for the HP and Zebra Copy web services. They don't show up in the list of Applications in the Windows Task Manager, but you can see them in the process list as WebDev.WebServer.EXE.

You're now ready to run the program. Hit F5. A command shell window should appear, and Visual Studio will stop at your breakpoint. Wait for the startup of the two web servers that host the HP and Zebra Copy web sites. You should see them in your System Tray. If you simply hit F5, the program will run to completion. After a few seconds' delay, you'll see some text in the command shell window. Then, a browser will open showing you the certificate granting Bob the right to use the Brochure service. Press Enter in the command shell window. After a few seconds, some more text will appear in the command shell window, and another browser will open showing Bob's authorization to read and write the file to be printed. Press Pressing Enter takes you to the next step. Once more and you'll see Bob's delegation to Alice. Eventually, the sample code will run to completion. The command shell window will close when you press Enter one last time. You can examine the displayed certificates to see how the delegations are represented.

In the remainder of this Appendix, we'll single step through the code, which involves separate processes for the user application and the two web sites. We describe all the code, even code that seems obvious. It has been our experience that "obvious" is in the eye of the beholder. Think of this approach as Literate Programming [15].

Since you're unlikely to finish the program in a single setting, you can set a break point to get back to where you left off more quickly. Be careful to set the break point only in code running in the user application process. You can set break points in the various web services, but you have to start your program differently to see them. You may also find that the web services time out as you single step. Your only recourse is to set a break point in the user process, stop the debugging session, and restart. You don't have to stop the web services, though, because you've already seen the initialization up to that point.

Stop the two web servers. You can right click on the icon in the system tray and select Stop, or kill them from the Process list in the Task Manager. When they have been stopped, hit F5 in Visual Studio. Execution stops at your break point.

```
User bob = new User("bob.pfx", "password");
```

We see that the constructor for the User class takes two strings, the name of the user's key store and the password needed to access its private key. In a real application, this code would be in a process running in the user's account.

Step into the User constructor. Note that we've removed some indentation from the included code in order to help readability.

```
public class User
```

```
{
    private KeyPair keyPair;
    private X509Certificate x509Cert;
    public User(string filename, string password)
    {
        keyPair = new KeyPair(filename, password);
```

There are two members of this class, one for the user's public/private key pair and one for the user's X.509 certificate. Single step three more times to enter the KeyPair constructor. The KeyPair class has two members, the user's X.509 certificate and the private key. We've also included a static variable that gets the location of ApplicationDir from the environment variable RSAHome. Note that you cannot use a relative path here because this variable is accessed by code running in a variety of directories.

```
public class KeyPair
{
    private X509Certificate x509Certificate = null;
    private System.Security.Cryptography.RSA privateKey = null;
    public static string baseDir =
            Environment.GetEnvironmentVariable(
                "RSAHome").TrimEnd(@"\".ToCharArray()) + "\\";
```

As you continue to hit single step, you'll move through the code that extracts these elements from the key store file. We now construct the file name.

```
pfxfileName = directory + pfxfileName;
```

Next, we declare variables to hold the contents of the file and use static methods from the ComponentSpace library to read in the key store and extract the certificates it holds.

```
CertificateContext[] certificateContexts = null;
CertificateStore certificateStore = null;
```

We've set up the key stores to hold only one certificate, so we can easily extract the user's X.509 certificate and private key. An exception means that you forgot to set the RSAHome environment variable, the key store you created is in the wrong format, or that the key store is protected by a different password.

```
certificateStore = CertificateStore.ImportPfxFile(
       pfxfileName, password);
certificateContexts =
        CertificateContext.FindAllCertificates(certificateStore);
x509Certificate = certificateContexts[0].X509Certificate;
privateKey = certificateContexts[0].PrivateKey;
```

Finally, we close the key store file.

```
certificateStore.Close();
CertificateContext.CloseCertificateContexts(certificateContexts);
```

Back in the User constructor in Users.cs, we set the key and certificate members and return to UserApplication.cs.

```
    x509Cert = keyPair.Certificate;
}
```

Although it looks like a simple access of a member of KeyPair, the code actually invokes a getter in KeyPair.cs.

```
public X509Certificate Certificate
{
    get{return this.x509Certificate;}
}
```

A few more single steps and you're back in UserApplication.cs. If you don't get past this point, either the RSAHome environment variable is not set correctly, or you forgot to supply the key stores.

The next step is for Bob to get permission to use the Brochure service. This process will take us to the HP web service, the Zebra Copy Corporate web service, the Brochure service, and back again. To see how that works step into the GetBrochureServiceToken method of the User class, where we see our first interaction with a web service. In order to use a web service, a client needs a local object to handle all the communication with the remote service.

```
ServiceClassLibrary.HPWs.HPService hpService =
        new ServiceClassLibrary.HPWs.HPService();
```

The HPService class is a proxy class that defines such things as the communication protocol, SOAP, and namespace for the interface schema. Stepping into this constructor takes us to some system generated code in Settings.Designer.cs in the ServiceClassLibrary.

```
public static Settings Default
{ get{return defaultInstance;} }
```

Stepping into through this method takes us to more system generated code that contains configuration information for the service. In particular, it specifies the URL of the service.

```
[global::System.Configuration.ApplicationScopedSettingAttribute()]
[global::System.Diagnostics.DebuggerNonUserCodeAttribute()]
[global::System.Configuration.SpecialSettingAttribute(
        global::System.Configuration.SpecialSetting.WebServiceUrl)]
[global::System.Configuration.DefaultSettingValueAttribute(
        "http://localhost:1499/HPWebSite/HPService.asmx")]
public string ServiceClassLibrary_HPWs_HPService
{
  get {return ((string)(this["ServiceClassLibrary_HPWs_HPService"]));}
}
```

Single stepping takes us back to Users.cs where we set a long timeout on this service, so we can single step without getting a time-out exception.

```
hpService.Timeout = 3600000;
```

Bob will provide his X.509 certificate to HP when he requests access to the Brochure service. HP will use his public key to delegate an authorization to him. The defined interface requires the certificate be in the form of a byte array, as the native X509Certificate object is not XML serializable in .NET web services. Then we invoke the service at the HP web site to get the authorization certificate for the Brochure service.

```
byte[] userCertificate = _x509Cert.Export(X509ContentType.Cert);
XmlElement hpServiceToken =
      hpService.ObtainServiceCapability(userCertificate);
```

This invocation takes us to the HP web site, where we enter the ObtainServiceCapability method of HPService.cs. We are now looking at code running in the process running the HP web server. Since this process is different from the process running UserApplication, break points and console output won't be seen.

Note that we didn't need any authorization to use this service. We would expect that for a service that we want lots of people to use, such buying from our online catalog. We might also expect the service to be open if it were only accessible inside the corporate firewall. That's not the case here. The HP File Content service is accessible to anyone who knows the URL. In a real implementation, we would distribute the authorization to use this service to all employees. We didn't want to illustrate this bootstrap step for the sample code. Instead, we implemented a very simple access policy that you'll see soon.

We first instantiate the class that does the actual work for the web service. While we could include this code in the class implementing the web service, it's easier to test code if it is in a local class and later expose its functionality as web service methods. Normally, this class would be defined as a Singleton [10] and started by the web service, but then we wouldn't be able to see the initialization. Note that the space between the constructor and the declaration [WebMethod] is significant

```
[WebService(Namespace = "http://www.hp.com/services")]
[WebServiceBinding(ConformsTo = WsiProfiles.BasicProfile1_1)]
public class HPService : System.Web.Services.WebService
{
    public HPService () {}

    [WebMethod]
    public XmlElement ObtainServiceCapability(byte[] certificate)
    {
       HP localHP = new HP();
```

Class HP has three members, the filename of HP's key store, HP's key pair, and Bob's public key in the form of a string. Bob's public key wouldn't normally be here, but we want to implement a very simple access policy that allows only Bob to get authorization to use the Brochure service. In a real deployment there would be some policy engine used to make such decisions. However it is done, this policy shows how identity can enter into Federated Access Management. Note that some characters in Bob's public key have been elided.

```
private string bobKey = "30818902818100856E3...";
public const string testCertificateFileName_HP = "hp.pfx";
private static KeyPair hpKeyPair;
```

As you single step, you should reach the statement assigning hpKeyPair. If you don't, that's because you forgot to stop your web servers before you started debugging UserApplication.cs. HP's KeyPair is defined as a static variable, which means it stays set as long as the HP web server is running. At any rate, if you did reach this statement, you can step over (F10) the call because you've already seen what it does.

```
public HP()
{
   if (null == hpKeyPair)
   {
      hpKeyPair = new KeyPair(testCertificateFileName_HP,"password");
   }
}
```

On returning to HPService.cs, we invoke GetTokenForHP, which will ask Zebra Copy Corporate for authorization to use the Brochure service. Next we'll step into the method that gets the authorization.

```
Assertion assertion = localHP.GetTokenForHP();
```

This call takes us back to HP.cs, where a local proxy for the Zebra Copy Corporate web service is instantiated, initialized with data needed to sign the SOAP message, and set with a long time out. This service represents the entity at Zebra Copy that negotiates the contract with HP. Note that single stepping didn't take us into the system generated code for the ZebraCopyCorporateService because the compiler doesn't generate debugging symbols in the DLLs of proxy classes.

```
ZebraCopyCorporateWs.ZebraCopyCorporateService srv =
              new ZebraCopyCorporateWs.ZebraCopyCorporateService();
srv.Timeout = 3600000;
```

Next, we invoke the service, asking for the right for press runs of up to 5,000 copies. First, we need to convert HP's X.509 certificate into a byte array that can be transferred to the Zebra Copy web site. Stepping into its SignCopyServiceContract takes us to the Zebra Copy web site.

```
byte[] certificate =
       hpKeyPair.Certificate.Export(X509ContentType.Cert);
XmlElement assertion = srv.SignServiceContract(5000, certificate);
```

That takes us to the SignCopyServiceContract in ZebraCopyCorporateService.cs on the Zebra Copy Web site. This class configures the web services parameters and provides a static variable to hold the authorization to use the Brochure service. In a real implementation, this authorization would be handled differently.

```
[WebService(Namespace = "http://www.zebracopy.com/")]
[WebServiceBinding(ConformsTo = WsiProfiles.BasicProfile1_1)]
public class ZebraCopyCorporateService : System.Web.Services.WebService
{

    private static Assertion serviceCertificate = null;

    public ZebraCopyCorporateService () {}
```

The SignCopyServiceContract method's arguments are the size of the largest allowed press run and a byte array containing the requester's X.509 certificate. The method also needs the SOAP signing information.

```
[WebMethod]
[SoapHeader("msgSigningHeader", Direction = SoapHeaderDirection.In)]
public XmlElement SignCopyServiceContract(int number,
    byte[] certificate) {
```

The first thing we do is instantiate an instance of the Zebra Copy Corporate class. As before, in a real deployment, the Zebra Copy web site would take care of this step. The constructor sets up the key pair for this entity.

```
ZebraCopyCorporate corporate = new ZebraCopyCorporate();
```

The setup in ZebraCopyCorporate.cs is the same as for HP Corporate, with one difference. Zebra Copy Corporate provides a static method so that its services can read its X.509 certificate.

```
private const string pfxFileName = "zebracopy.pfx";
private static KeyPair keyPair;
public static X509Certificate getCert(){ return keyPair.Certificate; }
public ZebraCopyCorporate()
{
  if (null == keyPair)
  {
    keyPair = new KeyPair(pfxFileName,"password");
  }
}
```

Back in ZebraCopyCorporateService.cs, we instantiate a local Brochure object, called the *service surrogate*, for the corresponding Bochure web service and create the authorization certificate. In real life, this certificate would be created by the Brochure service when it is started by the web site.

```
    if (null == serviceCertificate)
    {
        Brochure brochure = new Brochure();
```

Initializing the Brochure service surrogate consists of getting its key pair and the X.509 certificate for Zebra Copy Corporate, which is stored in a static variable for later use. In a real implementation, the Zebra Copy web site would provide this certificate to the Brochure service.

```
public class Brochure
{
  private const string certificateFileName = "brochure.pfx";
  private static KeyPair brochureServiceKeyPair;
  private static X509Certificate zebracopyX509;
  public Brochure()
  {
    if (null == brochureServiceKeyPair)
    {
      brochureServiceKeyPair = new
           KeyPair(certificateFileName,"password");
      zebracopyX509 = ZebraCopyCorporate.getCert();
```

The getter in ZebraCopyCorporate.cs returns the requested X.509 certificate.

```
public static X509Certificate getCert(){ return keyPair.Certificate; }
```

The return takes us through the Brochure constructor back to ZebraCopyCorporateService.cs. Here we invoke the BrochureStartUp method which returns the authorization token for Zebra Copy Corporate to use the Brochure service.

```
serviceCertificate = brochure.BrochureStartUp();
```

First, we’ll instantiate an assertion authorizing use of the Brochure service.

```
public Assertion BrochureStartUp()
{
    Assertion brochure_assertion = BrochureAssertion();
```

We then step into the BrochureAssertion method in this file. This method builds the actual SAML certificate.

It instantiates a new Assertion from the ComponentSpace library and defines a human readable string to denote the certificate issuer. Next, it sets a condition specifying the valid lifetime of the certificate. In an IBAC, RBAC, or PBAC system, revocation is hard. (See Section 3.4 if you forgot why.) People simplify their lives by making certificates with short lifetimes. Later we’ll see how easy revocation is with ABAC, so we’re free to set a long lifetime on the certificate. In this case, the Brochure service sets the maximum possible interval.

```
private Assertion BrochureAssertion()
{
  Assertion assertion = new Assertion();
  assertion.Issuer = "Brochure Service Authority";
  assertion.Conditions = new Conditions(
                              DateTime.MinValue,DateTime.MaxValue);
```

Next, we extract the subject information from Zebra Copy Corporate's X.509 certificate.

```
NameIdentifier nameIdentifier =
     new NameIdentifier("", NameIdentifier.Formats.X509SubjectName,
                         zebracopyX509.Subject);
     SubjectConfirmation subjectConfirmation =
          new SubjectConfirmation(ConfirmationMethod.Methods.Bearer);
```

We then convert that X.509 certificate into a string for inclusion in the SAML assertion as the subject confirmation data. We also construct a Subject object instantiated with the name identifier in the X.509 certificate and this confirmation data.

```
byte[] exportResult = zebracopyX509.Export(X509ContentType.Cert);
string result = Convert.ToBase64String(exportResult);
subjectConfirmation.SubjectConfirmationData =
         new SubjectConfirmationData(result);
Subject subject = new Subject(nameIdentifier, subjectConfirmation);
```

Next, we state what is being authorized in an AuthorizationDecisionStatement, which specifies the Brochure web service, the action (web service method) being allowed on it, and the decision, which in this case is "Permit". It also specifies a Revoke method. Normally, Revoke would be a generic service provided automatically for every web service. Here, however, we leave the implementation to the web services. Every authorization should include the ability to revoke delegated certificates.

```
AuthorizationDecisionStatement authorizationDecisionStatement =
      new AuthorizationDecisionStatement();
authorizationDecisionStatement.Resource =
      "http://www.zebracopy.com/services/BrochureService.asmx";
Action print = new Action(
      "http://www.zebracopy.com/services/BrochureService.asmx",
      "Print");
Action revoke = new Action(
      "http://www.zebracopy.com/services/BrochureService.asmx",
      "Revoke");
```

Next we add the authorized actions and the Subject to the AuthorizationDecisionStatement and add it to the Assertion being constructed.

```
authorizationDecisionStatement.Actions.Add(print);
authorizationDecisionStatement.Actions.Add(revoke);
authorizationDecisionStatement.Decision = Decision.Permit;
authorizationDecisionStatement.Subject = subject;
assertion.Statements.Add(authorizationDecisionStatement);
```

Next, we use an AttributeStatement to specify any application dependent constraints, a press run limit of 10,000 in our example. Note that the second parameter to the Attribute method is a URN. This URN has the same data as the URL in the resource specification in the decision statement plus the name of the method that this constraint applies to. Note that we must include a subject. That seems odd, since we just included the same Subject in the AuthorizationDecisionStatement. However, if we leave it out, the serializer of the toolkit we're using fails. We believe that happens because the normal use of AttributeStatements is to say something about a Subject, in which case not including a Subject would be an error.

```
AttributeStatement attributeStatement = new AttributeStatement();
attributeStatement.Subject = subject;
attributeStatement.Attributes.Add(
     new ComponentSpace.SAML.Assertions.Attribute("PrintLimit",
         "urn:zebra:copy:brochure_service:Print", "10000"));
```

Finally, we add this statement to the assertion and return to BrochureStartUp in this file.

```
assertion.Statements.Add(attributeStatement);
return assertion;
```

Next, we use the static SignAssertion method of AuthZUtilities to sign the assertion to prevent tampering and forgery. The arguments are the assertion we just created, the Brochure service X.509 certificate, and its private key.

```
Assertion signedBrochureAssertion =
            AuthZUtilities.SignAssertion(brochure_assertion,
            brochureServiceKeyPair.Certificate,
            brochureServiceKeyPair.PrivateKey);
```

In AuthZUtilities.cs in the Authorization project, we start by converting the assertion to XML because that is the form it will be delivered in.

```
public static Assertion SignAssertion(Assertion assertion,
            X509Certificate certificate,
            System.Security.Cryptography.RSA privateKey)
{
   XmlElement xmlElement = assertion.ToXml();
```

The method in the ComponentSpace library that generates the signature takes the XML representation of the assertion, the private signing key, and X.509 certificate of the issuer and adds a signature field in XML to the assertion.

```
AssertionSignature.Generate(xmlElement, privateKey, certificate);
```

We then define a name space manager for parsing the SAML assertion, extract the signature as an XML node, and add it to the assertion and return the signed assertion. We use Xpath instead of the signature retrieval method provided by the ComponentSpace SAML toolkit because it doesn't support nested certificate chaining.

```
XmlNamespaceManager nsm = new XmlNamespaceManager(new NameTable());
nsm.AddNamespace("saml", "urn:oasis:names:tc:SAML:1.0:assertion");
nsm.AddNamespace("dsig", SignedXml.XmlDsigNamespaceUrl);
XmlNode sigNode = xmlElement.SelectSingleNode(
               "/saml:Assertion/dsig:Signature", nsm);
```

```
assertion.Signature = (XmlElement)sigNode;
return assertion;
```

Back in BrochureStartUp, we return the resulting signed assertion.

```
return signedBrochureAssertion;
```

We are now back in the SignServiceContract method of ZebraCopyCorporate.cs, where we write the SAML certificate to disk so you can see what's in it.

```
AuthZUtilities.OutputAssertionToFile(
          serviceCertificate,
          "Zebra Copy Corporate Authorization to Brochure Service",
          "ZebraCopyCorporateBrochureService.xml");
```

Stepping into OutputAssertionToFile takes us to AuthZUtilities of the Authorization project. It shows that we get the directory from the static variable set in KeyPair.cs and construct the fully qualified name of the output file.

```
public static void OutputAssertionToFile(XmlElement xmlElement,
         string label, string fn)
{
   string baseDir = KeyPair.baseDir;
   string filename = baseDir + fn;
```

We next convert the assertion to XML and instantiate a new XML document. If the file exists, we delete it and instantiate a file stream to create the output file.

```
XmlElement xmlElement = assertion.ToXml();
XmlDocument dom = new XmlDocument();
FileStream fs = null;
if (File.Exists(filename)) { File.Delete(filename); }
fs = new FileStream(filename, FileMode.Create);
```

We start the output document with the XML boiler plate described in Section 5. Next, we get a time stamp to include in the certificate-log tag. Then, we add the label passed in as an input parameter to the heading. Finally, we add the assertion to the output file and write it to disk.

```
XmlDeclaration xmlDecl = dom.CreateXmlDeclaration("1.0","utf-8", null);
dom.InsertBefore(xmlDecl, dom.DocumentElement);
XmlNode rootNode = dom.CreateNode(XmlNodeType.Element,
      "certificate-log", String.Empty);
dom.AppendChild(rootNode);
XmlNode timeAttributeNode =
      dom.CreateNode(XmlNodeType.Attribute, "timestamp", String.Empty);
timeAttributeNode.InnerText = DateTime.Now.ToString();
rootNode.Attributes.Append((XmlAttribute)timeAttributeNode);
XmlNode labelAttributeNode =
      dom.CreateNode(XmlNodeType.Attribute, "label", String.Empty);
labelAttributeNode.InnerText = label;
rootNode.Attributes.Append((XmlAttribute)labelAttributeNode);
```

```
XmlNode newNode = dom.ImportNode(xmlElement, true);
rootNode.AppendChild(newNode);XmlNode newNode =
      dom.ImportNode(xmlElement, true);
rootNode.AppendChild(newNode);
fs.Position = 0;
dom.Save(fs);
fs.Close();
```

Back in ZebraCopyCorporate.cs, we define the certificate that will grant HP the right to use the Brochure service. The IssueCertificate method of ZebraCopyCorporate takes its authorization, HP's X.509 certificate, the largest authorized press run, and the dates the project is valid. Here we set the contract to start now and run for a year. That means a new certificate will have to be issued if the contract is renewed, but it also means that no special action is needed if the contract is allowed to expire.

```
Assertion resultCertificate = null;
DateTime begin = DateTime.Now;
DateTime end = DateTime.Now.AddDays(365.0);
if (serviceCertificate != null) {
    resultCertificate = corporate.IssueCertificate(
           serviceCertificate, certificate, number, begin, end);
```

The first step done by the IssueCertificate method in ZebraCopyCorporate.cs is to define the assertion specifying the rights being granted to HP. Its arguments are Zebra Copy Corporate's authorization certificate, HP's X.509 certificate, the largest press run being allowed, and the begin and end dates of the contract.

```
Assertion delegatedBrochureAssertion = DelegateBrochureAssertion(
      serviceAssertion, cert, number, begin, end);
```

We are now in the DelegteBrochureAssertion method in ZebraCopyCorporate.cs. After instantiating a new assertion, we set the valid time interval of the certificate to be the specified begin and end times.

```
private Assertion DelegateBrochureAssertion(
      Assertion incoming_assertion, byte[] requesterX509,
      int number, DateTime begin, DateTime end)
 {
    Assertion assertion = new Assertion();
    assertion.Issuer = "Zebra Copy Corporate";
    assertion.Conditions = new Conditions(begin, end);
```

We next take the byte array form of HP's X.509 certificate and turn it into an object. As before, we get the subject information from that certificate.

```
X509Certificate x509Certificate = new X509Certificate(requesterX509);
NameIdentifier nameIdentifier = new NameIdentifier("",
      NameIdentifier.Formats.X509SubjectName, x509Certificate.Subject);
SubjectConfirmation subjectConfirmation = new
      SubjectConfirmation(ConfirmationMethod.Methods.Bearer);
string result = Convert.ToBase64String(requesterX509);
subjectConfirmation.SubjectConfirmationData = new
```

```
      SubjectConfirmationData(result);
Subject subject = new Subject(nameIdentifier, subjectConfirmation);
```

Next, we add the AuthorizationDecisionStatement. There is one addition to what was done before. The AuthorizationDecisionStatement now includes an Evidence member, which consists of the assertion supplied as the first argument. This assertion is the one that grants Zebra Copy Corporate the right to use the service. We'll see how this evidence is used when we walk through an invocation of the Brochure service.

```
AuthorizationDecisionStatement authorizationDecisionStatement =
       new AuthorizationDecisionStatement();
authorizationDecisionStatement.Resource =
       "http://www.zebracopy.com/services/BrochureService.asmx";
Action action = new Action(
       "http://www.zebracopy.com/services/BrochureService.asmx",
       "Print");
Action revoke = new Action(
       "http://www.zebracopy.com/services/BrochureService.asmx",
       "Revoke");
authorizationDecisionStatement.Actions.Add(action);
authorizationDecisionStatement.Actions.Add(revoke);
authorizationDecisionStatement.Decision = Decision.Permit;
authorizationDecisionStatement.Subject = subject;
authorizationDecisionStatement.Evidence =
       new Evidence(incoming_assertion);
```

The AttributeStatement, specifies that HP is allowed press runs of up to the specified amount. Finally, we return.

```
assertion.Statements.Add(authorizationDecisionStatement);
AttributeStatement attributeStatement = new AttributeStatement();
attributeStatement.Subject = subject;
attributeStatement.Attributes.Add(
      new ComponentSpace.SAML.Assertions.Attribute(
          "PrintLimit", "urn:zebra:copy:brochure_service:Print",
          number.ToString()));
assertion.Statements.Add(attributeStatement);
return assertion;
```

Back in IssueCertificate of ZebraCopyCorporate.cs we sign the certificate with Zebra Copy Corporate's private key using a method we've previously stepped through.

```
Assertion signedHPAssertion = AuthZUtilities.SignAssertion(
          hpcopyAssertion, keyPair.Certificate, keyPair.PrivateKey);
```

We write the certificate to a file so you can examine it, and return the result to the SignCopyServiceContract method in ZebraCopyCorporateService.cs, where we return the XML form of the certificate.

```
AuthZUtilities.OutputAssertionToFile(signedAssertion,
          "Zebra Copy Corporate Service","ZebraCopyToHP.xml");
return signedAssertion;
```

That takes us back to the SignServiceContract method in ZebraCopyCorporateService.cs where we return the XML form of the authorization.

We are now back at the HP Corporate web service on the HP web site in method GetTokenForHP in file HP.cs. There we convert the assertion to an object and return.

```
return (new Assertion(assertion));
```

That takes us back to the ObtainServiceCapability method of HPService.cs where we write out the authorization granted to HP and invoke the GetTokenFromHP method in HP.cs that will delegate a portion of HP's rights to Bob.

```
string verification = "";
AuthZUtilities.OutputAssertionToFile(
                   assertion,
                   verification,
                   "HPBrochureService.xml");
Assertion resultAssertion = localHP.GetTokenFromHP(
        assertion, certificate);
```

The arguments are the SAML assertion representing HP's right to use Zebra Copy's Brochure service and a byte array containing Bob's X.509 certificate.

```
public Assertion GetTokenFromHP(Assertion incomingAssertion,
       byte[] certificate)
{
  if (hpKeyPair.Loaded)
  {
    Assertion userAssertion = CreateUserAssertion(
                                 incomingAssertion, certificate);
```

The CreateUserAssertion method of HP.cs implements a trivial access policy. It checks the public key in the input X.509 certificate to see if it matches the value stored in member bobKey. That means that only someone who can use Bob's private key can get an authorization to use the Zebra Copy Brochure service. The previous sentence is carefully worded to make it clear that people can and do share private keys. In reality, there would be some policy engine within HP that would be invoked to make the access decision. This code does show that identification can play a role in Federated Access Management, just a different role than in Federated Identity management. In FAccM, identity can be used to distribute rights, but, as we'll see, not to make an access decision at request time. With FIdM, identity is used for both. The implications of this seemingly small difference are significant.

Our sample does not show how Bob proves the right to use the authorizations. That's because it's done the same way Bob proves the right to use an authentication assertion submitted in an IBAC system. In both, Bob would sign the entire SOAP body of the request message and embed the signature in the SOAP header. The service can then verify that the signature matches the SOAP body and uses the same public key as the one appearing in the assertions. We show how that's done in Appendix E.

```
private Assertion CreateUserAssertion(Assertion incoming_assertion,
      byte[] certificate)
{
  X509Certificate x509Certificate = new X509Certificate(certificate);
  string pubkey = x509Certificate.GetPublicKeyString();
  if (pubkey != bobKey) return null;
```

We declare the issuer to be the HP Brochure Service Authority, Zelda in the scenario described in Section 2. Note that we can safely set the valid time span to be the maximum possible. That's because this certificate will be valid only so long as HP's is valid, but that interval has been set to be the duration of the contract.

```
Assertion assertion = new Assertion();
assertion.Issuer = "HP Brochure Service Authority";
assertion.Conditions = new Conditions(
                           DateTime.MinValue,DateTime.MaxValue);
```

The rest of the assertion is similar to what we've seen before, except this time the Evidence is HP's authorization certificate, and Bob's print limit is set to press runs of no more than 500.

```
AuthenticationStatement authenticationStatement =
      new AuthenticationStatement(
         AuthenticationStatement.AuthenticationMethods.X509PublicKey);
NameIdentifier nameIdentifier = new NameIdentifier("",
     NameIdentifier.Formats.X509SubjectName, x509Certificate.Subject);
SubjectConfirmation subjectConfirmation = new SubjectConfirmation(
     ConfirmationMethod.Methods.Bearer);
string result = Convert.ToBase64String(certificate);
subjectConfirmation.SubjectConfirmationData = new
     SubjectConfirmationData(result);
authenticationStatement.Subject = new Subject(nameIdentifier,
     subjectConfirmation);
assertion.Statements.Add(authenticationStatement);
AuthorizationDecisionStatement authorizationDecisionStatement =
     new AuthorizationDecisionStatement();
authorizationDecisionStatement.Resource =
     "http://www.zebracopy.com/services/BrochureService.asmx";
Action print = new Action(
   "http://www.zebracopy.com/services/BrochureService.asmx", "Print");
Action revoke = new Action(
  "http://www.zebracopy.com/services/BrochureService.asmx", "Revoke");
authorizationDecisionStatement.Actions.Add(print);
authorizationDecisionStatement.Actions.Add(revoke);
authorizationDecisionStatement.Decision = Decision.Permit;
authorizationDecisionStatement.Subject =
authenticationStatement.Subject;
authorizationDecisionStatement.Evidence =  new Evidence(
     incoming_assertion);
assertion.Statements.Add(authorizationDecisionStatement);
AttributeStatement attributeStatement = new AttributeStatement();
```

```
attributeStatement.Subject = authenticationStatement.Subject;
attributeStatement.Attributes.Add(
     new ComponentSpace.SAML.Assertions.Attribute("PrintLimit",
        "urn:zebra:copy:brochure_service:Print", "500"));
assertion.Statements.Add(attributeStatement);
return assertion;
```

After returning to GetTokenFromHP, we make sure access was allowed. If it was, we sign the certificate with HP's private key and return the result to the ObtainServiceCapability method of the HP web service. That method converts the certificate to XML and returns it to the GetBrochureServiceToken in Users.cs.

```
if (resultAssertion == null)
    return null;
else return (resultAssertion.ToXml());
```

That means we've gone from the user application to the HP web service to the Zebra Copy Corporate web service to the surrogate of the Brochure web service and back to Users.cs. We write a console message and convert the XML form to an object.

```
if (hpServiceToken == null) return null;
string userName = _x509Cert.Subject;
System.Console.WriteLine(userName +
     " obtained a copy service certificate from HP...");
Assertion hpServiceAssertion = new Assertion(hpServiceToken);
return hpServiceAssertion;
```

We are now back at the second executable statement in Main of UserApplication.cs where we write the certificate to a file, display it in a browser, and pause execution so you have time to examine the certificate.

```
if (null != bobBrochureAuthorization)
{
  AuthZUtilities.OutputAssertionToFile(
           bobBrochureAuthorization,
          "Bob's Authorization to Use Zebra Copy's Brochure Service",
          "BobBrochureService.xml");
  Show("BobBrochureService.xml");
```

The local method Show displays the contents of the specified file in a browser using the system diagnostic class ProcessStartInfo.

```
static void Show(string fname)
{
   string baseDir = KeyPair.baseDir;
   string filename = baseDir + fname;

   ProcessStartInfo startInfo = new ProcessStartInfo("iexplore.exe");
   startInfo.FileName = filename;
   startInfo.Verb = "Open";
   Process.Start(startInfo);
}
```

Next we pause execution so you have time to examine the certificate.

```
Pause();
```

The Pause method simply waits for console input. Pressing Enter continues the execution of the program.

```
static void Pause()
{
    System.Console.WriteLine("Press Enter to continue ...");
    System.Console.ReadLine();
}
```

The next step is for Bob to get an authorization to read and write a file provided by the HP File web service. In this case, we hardwire the name of the file.

```
string fileName = "FederatedAccessManagement.pdf";
```

We step into GetFileContentServiceToken, of Users.cs, which takes the name of the file as an argument. The first thing we do is create a service proxy to interact with the HP File web service.

```
public Assertion GetFileContentServiceToken(string filename)
{
  ServiceClassLibrary.HPFileContentWs.HPFileContentService fs =
        new ServiceClassLibrary.HPFileContentWs.HPFileContentService();
```

That takes us to a system generated file that sets up some configuration information.

```
[global::System.Configuration.ApplicationScopedSettingAttribute()]
[global::System.Diagnostics.DebuggerNonUserCodeAttribute()]
[global::System.Configuration.SpecialSettingAttribute(
      global::System.Configuration.SpecialSetting.WebServiceUrl)]
[global::System.Configuration.DefaultSettingValueAttribute(
      "http://localhost:1499/HPWebSite/HPFileContentService.asmx")]
public string ServiceClassLibrary_HPFileContentWs_HPFileContentService
{
    get {return ((string)
 (this["ServiceClassLibrary_HPFileContentWs_HPFileContentService"]));}
}
```

Back in Users.cs, we provide the information needed to sign the SOAP messand and set a long timeout on the service so we can single step the program. Bob serializes his X.509 certificate to a byte array and invokes the GetFileAccessCapability method of the HP File web service.

```
fs.Timeout = 3600000;
byte[] myCertificate = this._keyPair.Certificate.Export(
        X509ContentType.Cert);
XmlElement hpFSToken = fs.GetFileAccessCapability(
        filename, myCertificate);
```

Method HPFileContentService.cs is part of the web service hosted by the HP web site. Don’t forget the blank line needed between the constructor and [WebMethod].

```
[WebService(Namespace = "http://www.hp.com/services")]
[WebServiceBinding(ConformsTo = WsiProfiles.BasicProfile1_1)]
public class HPFileContentService : System.Web.Services.WebService
{
    private string service =
        "http://www.hp.com/services/HPFileContentService.asmx";
    public HPFileContentService() {}
```

In the GetFileAccessCapability method we instantiate a local file service object as the local surrogate for the corresponding web service. As before, we do this here for the purposes of this tutorial. In reality, the HP web site would start the service as a Singleton for us.

```
[WebMethod]
public XmlElement GetFileAccessCapability(string filename,
      byte[] certificate)
{
    HPFileContent fc = new HPFileContent();
```

As in the HP web service, we have a very simple policy that states that subjects who can use Bob’s or Alice’s private key may get a certificate for any file they name. HPFileContent also has a member for the name space manager needed to parse the certificates and one to hold a reference to the HP web service, which should also be a Singleton started by the web server.

```
public class HPFileContent
{
  private string bobKey   = "30818902818100856E32A...";
  private string aliceKey = "308189028181008F9C6DA...";
  private static HP hp;
  private XmlNamespaceManager nsm =
          new XmlNamespaceManager(new NameTable());
  public HPFileContent()
  {
    if (hp == null) { hp = new HP(); }
  }
  nsm.AddNamespace("saml", "urn:oasis:names:tc:SAML:1.0:assertion");
  nsm.AddNamespace("dsig", SignedXml.XmlDsigNamespaceUrl);
}
```

The next step back in GetFileAccessCapability in HPFileContentService.cs is to create the actual certificate.

```
Assertion assertion = fc.CreateFileAccessAssertion(
            filename, certificate);
```

In the CreateFileAccessAssertion of HPFileContent.cs we see that only subjects who can use Alice's or Bob's private key may use this service under our trivial access policy.

```
public Assertion CreateFileAccessAssertion(string filename,
      byte[] pubkey)
{
  X509Certificate x509Certificate = new X509Certificate(pubkey);
  string pubkeystr = x509Certificate.GetPublicKeyString();
  if (!( pubkeystr == bobKey || pubkeystr == aliceKey ) ) return null;
```

Since Bob's X.509 certificate was provided, we'll create the certificate. Next we see another aspect of the policy. Bob will get a certificate that doesn't expire, but everyone else gets one that expires in 30 days. Again, such policies should be provided by some external policy engine.

```
Assertion assertion = new Assertion();
assertion.Issuer = "HP File Content Service Authority";
if (pubkeystr == bobKey)
  assertion.Conditions = new Conditions(
         DateTime.MinValue, DateTime.MaxValue);
else
  assertion.Conditions = new Conditions(new TimeSpan(720, 0, 0));
```

The rest of this method is similar to the others. Note that we include WriteFile and Revoke even though we don't invoke those methods or implement them. That's just to avoid having any dead code in the sample application.

```
NameIdentifier nameIdentifier =
        new NameIdentifier("", NameIdentifier.Formats.X509SubjectName,
                 x509Certificate.Subject);
SubjectConfirmation subjectConfirmation =
        new SubjectConfirmation(ConfirmationMethod.Methods.Bearer);
string result = Convert.ToBase64String(pubkey);
subjectConfirmation.SubjectConfirmationData =
        new SubjectConfirmationData(result);
Subject subject = new Subject(nameIdentifier, subjectConfirmation);
AuthorizationDecisionStatement authorizationDecisionStatement =
        new AuthorizationDecisionStatement();
authorizationDecisionStatement.Resource =
       "http://www.hp.com/services/HPFileContentService.asmx";
Action readAction = new Action(
       "http://www.hp.com/services/HPFileContentService.asmx",
       "ReadFile");
Action writeAction = new Action(
       "http://www.hp.com/services/HPFileContentService.asmx",
       "WriteFile");
Action revoke = new Action(
       "http://www.hp.com/services/HPFileContentService.asmx",
       "Revoke");
authorizationDecisionStatement.Actions.Add(readAction);
authorizationDecisionStatement.Actions.Add(writeAction);
authorizationDecisionStatement.Actions.Add(revoke);
authorizationDecisionStatement.Decision = Decision.Permit;
authorizationDecisionStatement.Subject = subject;
```

```
assertion.Statements.Add(authorizationDecisionStatement);
AttributeStatement attributeStatement = new AttributeStatement();
attributeStatement.Subject = authorizationDecisionStatement.Subject;
attributeStatement.Attributes.Add(
      new ComponentSpace.SAML.Assertions.Attribute("File",
       "http://www.hp.com/services/HPFileContentService.asmx/ReadFile",
        filename));
attributeStatement.Attributes.Add(
     new ComponentSpace.SAML.Assertions.Attribute("File",
      "http://www.hp.com/services/HPFileContentService.asmx/WriteFile",
      filename));
assertion.Statements.Add(attributeStatement);
return assertion;
```

We return the assertion to the GetFileAccessCapability method of HPFileContentService.cs and sign it with the HP private key if access was allowed. We could also sign it with a key specific to the HP File web service. We've done it this way to illustrate two subjects, HP Service and HP File Content Service, sharing an identity via a private key.

```
if (null == assertion)
{
    return null;
}
Assertion result = fc.SignFileAccessAssertion(assertion);
```

We return the XML version of the certificate back to the user code to GetFileContentServiceToken of Users.cs.

```
return (result.ToXml());
```

If all went well, that code converts it into an object which gets returned to UserApplication.cs, where we write a status message, convert the XML form of the assertion to an object and return.

```
System.Console.WriteLine(
     "Obtained a file service certificate from HP...");
Assertion hpFSAssertion = new Assertion(hpFSToken);
return hpFSAssertion;
```

Back in UserApplication.cs, we write the certificate to a file and display it in a browser.

```
AuthZUtilities.OutputAssertionToFile(
     bobHpFileContentServiceAssertion,
     "Bob's Authorization to Read/Write a File at the HP File Service",
     "BobFileService.xml");
Show("BobFileService.xml");
Pause();
```

From the WSDL specification of the Brochure service, Bob knows that the service will need the authorization to read his file. He can allow that by delegating to the Brochure service his read permission. He knows the service's public key because it is in the certificate granting Bob the right to use the service.

```
Assertion bobDelegatedHPFileContentServiceAssertion =
      bob.DelegateCertificateOnFileService(
             bobHpFileContentServiceAssertion,
             bobBrochureAuthorization, fileName);
```

That delegation is done in the method DelegateCertificateOnFileService of Users.cs. The arguments are the delegator's certificate authorizing use of the file and the delegator's certificate authorizing use of the service to be granted access to the file. Note that we don't specify the name of the file, which will be taken from the delegatorToken.

```
public Assertion DelegateCertificateOnFileService(
    Assertion delegatorToken, Assertion serviceToken)
```

The class MyAuthZCertDelegator, which has a null constructor, contains methods for delegating and signing certificates, so we ask it to delegate access, supplying the delegator's X.509 certificate in addition to the two input assertions.

```
MyAuthZCertDelegator delegator = new MyAuthZCertDelegator();
Assertion token = delegator.DelegateFileCertificate (
     delegatorToken, serviceToken, this.keyPair.Certificate);
```

The DelegateCertificateOnFileService method of MyAuthZCertDelegator starts by extracting the name of the file from the user's authorization certificate. Note that one or more of the delegations could have incorrectly specified a different filename. Not to worry. We'll check for that error at access time.

```
public Assertion DelegateFileCertificate (
        Assertion delegatorToken, Assertion serviceToken,
        X509Certificate delegatorCertificate)
{
  XmlNamespaceManager nsm = new XmlNamespaceManager(new NameTable());
  nsm.AddNamespace("saml", "urn:oasis:names:tc:SAML:1.0:assertion");
  XmlElement token = delegatorToken.ToXml();
  XmlNodeList chainedAssertions =
       token.SelectNodes("//saml:AttributeValue", nsm);
  string fileName = chainedAssertions[0].InnerText;
```

The next step is to get the Brochure service's X.509 certificate out of Bob's authorization to use the service, which is done in GetServicePublicKeyCertificate in AuthZUtilities.cs in the Authorization project.

```
X509Certificate delegateeCertificate =
      AuthZUtilities.GetServicePublicKeyCertificate(serviceToken);
```

There we convert Bob's authorization certificate to XML and define a namespace manager that understands the SAML assertion schema.

```
 public static X509Certificate GetServicePublicKeyCertificate(
      Assertion certificate)
{
  XmlElement xmlElement = certificate.ToXml();
  X509Certificate embeddedCertificate  = null;
  XmlNamespaceManager nsm = new XmlNamespaceManager(new NameTable());
  nsm.AddNamespace("saml", "urn:oasis:names:tc:SAML:1.0:assertion");
```

Next, we extract all Assertion nodes from the certificate. The innermost certificate, the one that belongs to the service, is the last one in the sequence.

```
XmlNodeList chainedAssertions = xmlElement.SelectNodes(
            "//saml:Assertion", nsm);
int last = chainedAssertions.Count - 1;
return AssertionSignature.GetCertificate(
                (XmlElement)chainedAssertions[last]);
```

Back in the DelegateFileCertificate method of MyAuthZCertDelegator.cs, we build the certificate. There are two things to note. First, we set the authority to end on the first day of the trade show the brochures are for. That means the Brochure service can use the same authorization to read all versions, and there is no risk if nobody remembers to revoke access. Second, we only grant read and revoke authority, showing how easily we can enforce least privilege.

```
assertion.Issuer = delegatorCertificate.Subject;
DateTime end = DateTime.Now.AddDays(45);
assertion.Conditions = new Conditions(DateTime.MinValue, end);
AuthenticationStatement authenticationStatement =
               new AuthenticationStatement(
AuthenticationStatement.AuthenticationMethods.X509PublicKey);
NameIdentifier nameIdentifier =
     new NameIdentifier("",NameIdentifier.Formats.X509SubjectName,
              delegateeCertificate.Subject);
SubjectConfirmation subjectConfirmation =
     new SubjectConfirmation(ConfirmationMethod.Methods.Bearer);
byte[] exportResult =
             delegateeCertificate.Export(X509ContentType.Cert);
string result = Convert.ToBase64String(exportResult);
subjectConfirmation.SubjectConfirmationData =
     new SubjectConfirmationData(result);
authenticationStatement.Subject = new Subject(
            nameIdentifier, subjectConfirmation);
assertion.Statements.Add(authenticationStatement);
AuthorizationDecisionStatement authorizationDecisionStatement =
     new AuthorizationDecisionStatement();
authorizationDecisionStatement.Resource =
            "http://www.hp.com/services/HPFileContentService.asmx";
Action read = new Action(
            "http://www.hp.com/services/HPFileContentService.asmx",
            "ReadFile");
Action revoke = new Action(
            "http://www.hp.com/services/HPFileContentService.asmx",
            "revoke");
authorizationDecisionStatement.Actions.Add(read);
authorizationDecisionStatement.Actions.Add(revoke);
authorizationDecisionStatement.Decision = Decision.Permit;
authorizationDecisionStatement.Subject =
            authenticationStatement.Subject;
authorizationDecisionStatement.Evidence = new Evidence(delegatorToken);
assertion.Statements.Add(authorizationDecisionStatement);
AttributeStatement attributeStatement = new AttributeStatement();
attributeStatement.Subject = authenticationStatement.Subject;
attributeStatement.Attributes.Add(
     new ComponentSpace.SAML.Assertions.Attribute(
       "File",
       "http://www.hp.com/services/HPFileContentService.asmx/ReadFile",
       fileName));
assertion.Statements.Add(attributeStatement);
return assertion;
```

Back in the DelegateFileCertificate method of Users.cs we sign the certificate with Bob's private key and return to UserApplication.

```
Assertion delegatedToken =
            AuthZUtiliites.SignDelegateCertificate(token,
               this._keyPair.Certificate, this._keyPair.PrivateKey);
return delegatedToken;
```

Back in UserApplication.cs, we write the certificate to disk and display it so you can examine it.

```
AuthZUtilities.OutputAssertionToFile(
         bobDelegatedHPFileContentServiceAssertion,
         "Bob's Delegated File Certificate to Brochure Service",
         "BrochureFileService.xml");
```

In a real implementation, Bob will probably start a new process to invoke the brochure service. Here, we'll just invoke it in-line. However, because we're using FAccM, Bob can enforce least privilege. In this case, he wants to print 37 copies. He could invoke the Brochure service directly, but an error or a virus in his program could print up to Bob's limit of 500. Instead, what Bob does is create a dummy user, bob2, to act as his agent.

```
User bob2 = new User("bob2.pfx", "password");
```

Bob2 has its own private key, so Bob delegates a subset of his rights to bob2.

```
Assertion bobToBob2Token = bob.DelegateBrochureCertificate (
      bobBrochureAuthorization, bob2, 37);
```

Step into DelegateCertificateOnCopyService in Users.cs. We see that it creates a delegator and invokes its DelegateCertificate on Copy service. The arguments are the delegator's right to use the service, the user to be granted the right, and the limit on the size of the maximum press run in the resulting authorization. We next instantiate an object to do the delegation and pass it the certificate to be delegated, Bob's X.509 certificate, bob2's X.509 certificate, and the maximum authorized press run.

```
public Assertion DelegateCertificateOnCopyService(
     Assertion serviceToken, User other, int limit)
{
  MyAuthZCertDelegator delegator = new MyAuthZCertDelegator();
  Assertion token = delegator.DelegateBrochureCertificate (
               serviceToken, this.keyPair.Certificate,
               other.KeyPair.Certificate, limit);
```

We create the new certificate in DelegateBrochureCertificate in MyAuthZCertDelegator in the ServiceClassLibrary project.

```
public Assertion DelegateCertificateOnCopyService(
       Assertion delegatorToken, X509Certificate delegatorCertificate,
       X509Certificate delegateeCertificate, int pageLimit)
{
  Assertion assertion = new Assertion();
  assertion.Issuer = delegatorCertificate.Subject;
```

Bob doesn't care about the time interval the certificate is valid because he's going to revoke it once the service invocation returns.

```
assertion.Conditions = new Conditions(
              DateTime.MinValue, DateTime.MaxValue);
```

The rest of the certificate creation is similar to what we've seen before.

```
AuthenticationStatement authenticationStatement =
     new AuthenticationStatement(
          AuthenticationStatement.AuthenticationMethods.X509PublicKey);
NameIdentifier nameIdentifier =
         new NameIdentifier ("",
            NameIdentifier.Formats.X509SubjectName,
            delegateeCertificate.Subject);
SubjectConfirmation subjectConfirmation =
         new SubjectConfirmation(ConfirmationMethod.Methods.Bearer);
byte[] exportResult =
         delegateeCertificate.Export(X509ContentType.Cert);
string result = Convert.ToBase64String(exportResult);
subjectConfirmation.SubjectConfirmationData =
         new SubjectConfirmationData(result);
authenticationStatement.Subject =
         new Subject(nameIdentifier, subjectConfirmation);
assertion.Statements.Add(authenticationStatement);
AuthorizationDecisionStatement authorizationDecisionStatement =
         new AuthorizationDecisionStatement();
authorizationDecisionStatement.Resource =
         "http://www.zebracopy.com/services/BrochureService.asmx";
Action print = new Action(
         "http://www.zebracopy.com/services/BrochureService.asmx",
         "Print");
Action revoke = new Action(
         "http://www.zebracopy.com/services/BrochureService.asmx",
         "Revoke");
```

```
authorizationDecisionStatement.Actions.Add(print);
authorizationDecisionStatement.Actions.Add(revoke);
authorizationDecisionStatement.Decision = Decision.Permit;
authorizationDecisionStatement.Subject =
         authenticationStatement.Subject;
authorizationDecisionStatement.Evidence = new Evidence(delegatorToken);
assertion.Statements.Add(authorizationDecisionStatement);
AttributeStatement attributeStatement = new AttributeStatement();
attributeStatement.Subject = authenticationStatement.Subject;
```

Note that there are two permitted actions, print and revoke. We use the namespace to determine which action the PrintLimit attribute belongs to.

```
attributeStatement.Attributes.Add(
         new ComponentSpace.SAML.Assertions.Attribute(
             "PrintLimit",
             "urn:zebra:copy:brochure_service:Print",
             limit.ToString()));
assertion.Statements.Add(attributeStatement);
return assertion;
```

Back in Users.cs, we sign the certificate and return it if all went well.

```
Assertion delegatedToken = AuthZUtilities.SignAssertion(
              token, keyPair.Certificate, keyPair.PrivateKey);
return delegatedToken;
```

Back in UserApplication.cs we write bob2's certificate to a file so you can examine it.

```
AuthZUtilities.OutputAssertionToFile(
               bobToBob2Token,
               "Bob2's Brochure Service Authorization",
               "Bob2BrochureService.xml");
```

We are now ready for bob2 to invoke the service. First we set up the proxy to handle communication with the service and a local object to represent the remote service.

```
ServiceClassLibrary.BrochureWS.BrochureService srvProxy =
        new ServiceClassLibrary.BrochureWS.BrochureService();
```

That takes us to some system generated code that configures the proxy.

```
[global::System.Configuration.ApplicationScopedSettingAttribute()]
[global::System.Diagnostics.DebuggerNonUserCodeAttribute()]
[global::System.Configuration.SpecialSettingAttribute(
       global::System.Configuration.SpecialSetting.WebServiceUrl)]
[global::System.Configuration.DefaultSettingValueAttribute(
       "http://localhost:1502/ZebraCopyWebSite/BrochureService.asmx")]
public string ServiceClassLibrary_BrochureWS_BrochureService {
    get {return ((string)
         (this["ServiceClassLibrary_BrochureWS_BrochureService"]));}
}
```

Back in UserApplication.cs we provide the information needed to sign the SOAP request and set a long timeout on the service. Next, bob2 invokes the Brochure service with the authorization delegated from Bob, the authorization to read the file that Bob delegated to the Brochure service, and the size of the press run. Note that bob2 does not need nor get an authorization to read the file being printed.

```
ServiceClassLibrary.BrochureWS.ServiceProcessingResult result =
          srv.Print(bobToBob2Token.ToXml(),
          bobDelegatedHPFileContentServiceAssertion.ToXml(), 37);
srvProxy.Timeout = 3600000;
```

This invocation takes us to the Print method in BrochureService in the ZebraCopyWebSite project. The first member points to the page providing the service, and the second will hold the information needed to verify the signature on the SOAP message.

```
[WebService(Namespace = "http://www.zebracopy.com/services")]
[WebServiceBinding(ConformsTo = WsiProfiles.BasicProfile1_1)]
public class BrochureService : System.Web.Services.WebService
{
    private string service =
           "http://www.zebracopy.com/services/BrochureService.asmx";

    public BrochureService(){}
```

That takes us to the Print method in BrochureService.cs, which instantiates the already instantiated Brochure service, converts the authorization token from XML to a local object, and validates the token.

```
[WebMethod]
public ServiceProcessingResult Print(XmlElement serviceToken,
        XmlElement fileAuthorization, string jobName, int pressRun)
{
 Brochure copyService = new Brochure();
 Assertion assertion = new Assertion(serviceToken);
 ServiceProcessingResult processResult = brochureService.ValidateToken(
              assertion, service, "Print", pressRun);
```

The ValidateToken method in Brochure.cs that validates the authorization takes the SAML authorization, the service being invoked, the method of that service, and the requested number of copies. It then invokes a local method to do application independent checks of the certificate.

```
public ServiceProcessingResult ValidateToken(Assertion signedAssertion,
            string service, string method, int pressRun)
{
   ServiceProcessingResult result =
        ValidateToken(signedAssertion, service, method);
```

This ValidateToken method, also in Brochure.cs, runs through a series of tests. The verification is done using methods inAuthZCertVerifier in the ZebraCopyWebSite project.

```
public ServiceProcessingResult ValidateToken(
       Assertion signedAssertion, string service, string method)
{
   bool verified = false;
   ServiceProcessingResult result = new ServiceProcessingResult();
  MyAuthZCertVerifier certVerifier = new MyAuthZCertVerifier();
```

Although the constructor for MyAuthZCertVerifier is empty, the constructor for its superclass, AuthZCertVerifier defines the name space manager for parsing the XML representation of the SAML certificates. We also put the revocation list here for convenience. In a real implementation, we'd use a persistent database to hold the reovkation list.

```
public class AuthZCertVerifier
{
  protected XmlNamespaceManager nsm;
  private static Hashtable revocationList = new Hashtable();
  public AuthZCertVerifier()
  {
    nsm = new XmlNamespaceManager(new NameTable());
    nsm.AddNamespace("saml", "urn:oasis:names:tc:SAML:1.0:assertion");
    nsm.AddNamespace("dsig", SignedXml.XmlDsigNamespaceUrl);
  }
```

Back in Brochure.cs, we initialize the status flag and check each signature on the delegation chain. Note that these checks may throw exceptions, but we won't catch them in the sample implementation

```
result.status = true;
verified = certVerifier.CheckSignature(signedAssertion);
```

The CheckSignature method in AuthZcertVerifier.cs converts the certificate to XML, and extracts all the Assertions.

```
public bool CheckSignature(Assertion  assertion)
{
  XmlElement xmlElement = assertion.ToXml();
  bool verified = false;
  XmlNodeList chainedAssertions =
          xmlElement.SelectNodes("//saml:Assertion", nsm);
```

Next, it steps through the assertions and verifies that each is correctly signed. Since the Evidence that a delegation is valid is the Assertion authorizing the delegator, this step catches all attempts to forge an authorization. We return false if the certificate didn't pass the verificaton test.

```
foreach (XmlNode anode in chainedAssertions)
{
    XmlElement assertionNode = (XmlElement)anode;
    verified = AssertionSignature.Verify(assertionNode);
    if (!verified) break;
    return verified;
}
```

We return to ValidateToken in Brochure.cs. If verification failed, we specify the reason and return. If it succeeded, we provide the status message and check the expiration date.

```
if (!verified)
{
  result.status = false;
  result.statusMsg = "Brochure: SAML token signature is incorrect" +
      Environment.NewLine;
  return result;
} else result.statusMsg = "Brochure: SAML token signature verified" +
         Environment.NewLine;
verified = certVerifier.CheckExpiration(signedAssertion);
```

The CheckExpiration method in AuthZCertVerifier.cs should look familiar.

```
public bool CheckExpiration(Assertion assertion)
{
    XmlElement xmlElement = assertion.ToXml();
    bool verified = false;
    XmlNodeList chainedAssertions =
           xmlElement.SelectNodes("//saml:Assertion", nsm);
    foreach (XmlNode anode in chainedAssertions)
    {
        XmlElement assertionNode = (XmlElement)anode;
        verified = CheckExpiration(assertionNode);
```

The CheckExpiration method that examines an XML node returns true if the current time falls between the beginning and ending times specified in the certificate. Otherwise, it returns false.

```
private bool CheckExpiration(XmlElement element)
{
  bool verified = false;
  XmlNode notAfterNode = element.SelectSingleNode
            ("saml:Conditions/@NotOnOrAfter", nsm);
  XmlNode notBeforeNode = element.SelectSingleNode
            ("saml:Conditions/@NotBefore", nsm);
  DateTime begTime = DateTime.Parse(notBeforeNode.InnerText);
  DateTime endTime = DateTime.Parse(notAfterNode.InnerText);
  DateTime now = DateTime.Now;
  return (DateTime.Compare(endTime, now) <= 0 ||
          DateTime.Compare(now,begTime) < 0 );
}
```

Back in the CheckExpiration method that takes an assertion as an argument, we set the status and return.

```
    if (!verified) break;
```

```
}
return verified;
```

Back in ValidateToken of Brochure.cs, we set the status and make sure the issuer is in the delegation chain.

```
if (!verified)
{
  result.status = false;
  result.statusMsg += "Brochure: SAML token is expired" +
         Environment.NewLine;
  return result;
} else result.statusMsg += "Brochure: SAML token is current" +
         Environment.NewLine;
verified = certVerifier.CheckIssuer(signedAssertion);
```

The CheckIssuer in AuthZCertVerifier cs. method verifies that the issuer of the certificate is the subject of the Evidence field. It extracts all the assertion nodes, passing each to the CheckIssuer method that takes an XML node as an argument.

```
public bool CheckIssuer(Assertion certificate)
{
  XmlElement xmlElement = certificate.ToXml();
  bool verified = false;
    XmlNodeList chainedAssertions = xmlElement.SelectNodes(
         "//saml:Assertion", nsm);
    foreach (XmlNode anode in chainedAssertions)
    {
        XmlElement assertionNode = (XmlElement)anode;
        verified = CheckIssuer(assertionNode);
```

This CheckIssuer method extracts the Evidence node. If there is none, then we’re dealing with the innermost assertion, and there is nothing to check.

```
private bool CheckIssuer(XmlElement element)
{
  X509Certificate innerSubjectCertificate = null;
  bool verified = false;
  try
  {
    XmlNode evidenceNode = element.SelectSingleNode(
          "saml:AuthorizationDecisionStatement/saml:Evidence", nsm);
    if (evidenceNode == null)
      verified true;
```

Otherwise, we extract the Subject from the Evidence and make sure it matches the issuer of the delegation. If there is a Subject, then something we’ll extract its info.

```
else {
    XmlElement evidenceElement = (XmlElement)evidenceNode;
    XmlNode subjectNode = evidenceElement.SelectSingleNode(
        "saml:Assertion/saml:AuthorizationDecisionStatement/"+
        "saml:Subject", nsm);
    XmlElement subjectElement = (XmlElement)subjectNode;
```

```
   XmlNode x509CertificateNode = subjectElement.SelectSingleNode(
        "saml:SubjectConfirmationsaml:SubjectConfirmationData", nsm);
    innerSubjectCertificate =
        RecoverX509Certificate(x509CertificateNode.InnerText);
```

RecoverX509Certificate converts the string representation of the issuer's X.509 certificate to an object.

```
private X509Certificate RecoverX509Certificate(string encodedText)
{
  X509Certificate x = new X509Certificate();
  byte[] bytes= Convert.FromBase64String(encodedText);
  X509Certificate certificate = new X509Certificate();
  certificate.Import(bytes);
  return certificate;
}
```

Back in CheckIssuer, we and verify that the Subject in the Evidence signed the delegation.

```
 verified = AssertionSignature.Verify(element,innerSubjectCertificate);
}
return verified;
```

Back in the CheckIssuer method that takes an assertion as an argument, we're done if the verification failed.

```
    verified = CheckIssuer(assertionNode);
    if (!verified) break;
}
return verified;
```

Next, the ValidateToken in the Brochure service checks to see if any of the certificates in the delegation chain have been revoked.

```
if (!verified)
{
  result.status = false;
  result.statusMsg += "Brochure: SAML token chain issuer mismatch" +
         Environment.NewLine;
  return result;
} else result.statusMsg += "Brochure: SAML CheckIssuer verified" +
         Environment.NewLine;
verified = certVerifier.CheckRevocationList(signedAssertion);
```

As before, we start in a method that takes an assertion and do the actual checking in a method that looks at an individual XML node. We use the AssertionID as the key into the hash table of revoked certificates. In a real application, we'd use a persistent store, such as a database, to hold the list of revoked certificates along with additional metadata. Note that we use the poor coding practice of returning a double negative.

```
public bool CheckRevocationList(Assertion assertion)
{
  bool notVerified = true;
  XmlElement xmlElement = assertion.ToXml();
  XmlNodeList chainedAssertions = xmlElement.SelectNodes(
          "//saml:Assertion", nsm);
  foreach (XmlNode anode in chainedAssertions)
  {
     string assertionID =
           ((XmlElement)anode).Attributes["AssertionID"].InnerText;
     notVerified = revocationList.Contains(assertionID);
     if (notVerified) break;
  }
  return !notVerified;
}
```

Back now to Brochure.cs, we verify that the requester is authorized to invoke this method of this service.

```
if (!verified)
{
  result.status = false;
  result.statusMsg +=
     "Brochure: SAML token chain revocation checking failed" +
        Environment.NewLine;
  return result;
} else result.statusMsg +=
     "Brochure: SAML token chain revocation checking succeeded" +
        Environment.NewLine;
}
verified = certVerifier.CheckMethod(signedAssertion, service, method);
```

This method verifies that all delegations are for this service and include the method being invoked. We extract the chain of assertions and check each one.

```
public bool CheckMethod(
          Assertion assertion, string service, string method)
{
    XmlElement xmlElement = assertion.ToXml();
    bool verified = false;
    XmlNodeList chainedAssertions =
          xmlElement.SelectNodes("//saml:Assertion", nsm);
    foreach (XmlNode anode in chainedAssertions)
    {
        XmlElement assertionNode = (XmlElement)anode;
        verified = CheckMethod(
             (XmlElement)assertionNode, service, method);
```

The assertion node specifies the service and method on that service that it authorizes. If the service matches and the request is for any of the authorized services, we return true.

```
private bool CheckMethod(
           XmlElement assertionNode, string service, string method)
{
    bool verified = false;
    XmlNode serviceNameNode = assertionNode.SelectSingleNode(
              "saml:AuthorizationDecisionStatement/@Resource", nsm);
    if (serviceNameNode.InnerText == service)
    {
        XmlNodeList actionNodes = assertionNode.SelectNodes(
              "saml:AuthorizationDecisionStatement/saml:Action", nsm);
        foreach (XmlNode anode in actionNodes)
        {
            string statedActionName = anode.InnerText;
            if (statedActionName == method)
            {
                verified = true;
                break;
            }
        }
    }
    return verified;
}
```

Back in the loop over assertions, we quit if the assertion we just examined doesn't authorize this method on this service.

```
        if (!verified) break;
    }
    return verified;
}
```

Back now to Brochure.cs, we're done with all the checks, so we can return the status.

```
if (!verified)
{
    result.status = false;
    result.statusMsg += "Brochure: Method not authorized" +
          Environment.NewLine;
    return result;
}
else result.statusMsg += "Brochure: Method authorized" +
                        Environment.NewLine;
return result;
```

Now, back in the ValidateToken method that takes four arguments, we do the service specific validation if we passed the previous tests.

```
if (result.status)
{
    MyAuthZCertVerifier certVerifier = new MyAuthZCertVerifier();
    bool verified = certVerifier.CheckResourceConstraints(
                         signedAssertion, pressRun);
```

In CheckResourceConstraints in MyAuthZCertVerifier.cs, we convert the SAML assertion to XML and intialize some constants..

```
public override bool CheckResourceConstraints(
       Assertion certificate, int pressRun)
{
  XmlElement xmlElement = certificate.ToXml();
  bool validated = true;
  int before = -1;
  int limit = -1;
```

Next, we extract the assertions and step through them. For each we'll extract the resource limit.

```
XmlNodeList chainedAssertions = xmlElement.SelectNodes(
             "//saml:Assertion", nsm);
foreach (XmlNode anode in chainedAssertions)
{
  XmlElement assertionNode = (XmlElement)anode;
  XmlNode printLimitNode = assertionNode.SelectSingleNode(
           "saml:AttributeStatement/saml: " +
           "Attribute[@AttributeName='PrintLimit']", nsm);
XmlNode valueNode = printLimitNode.SelectSingleNode(
           "saml:AttributeValue", nsm);
int number = Int32.Parse(valueNode.InnerText);
```

First we make sure that nobody has delegated a larger press run than that party is authorized to order.

```
    if (0 < number)
    {
       if (before < 0)
       {
           before = number;
         limit = number;
       } else
           if (before > number)
           {
               validated = false;
               break;
           }
       before = number;
   }
}
```

If we pass that test, we make sure the order is for fewer copies than the limit set in the outermost certificate and return the validation status.

```
if (validated)
    if (pressRun > limit ) validated = false;
return validated;
```

Now we return to Brochure.cs where we set the status and return to the Brochure web service.

```
if (!verified)
{
    result.status = false;
    result.statusMsg +=
        "Brochure: Resource Access Constraints Violated" +
        Environment.NewLine;
} else result.statusMsg +=
        "Brochure: Resource Access Contraints Checked" +
        Environment.NewLine;
return result;
```

After a long digression, we’re back in the Brochure web service in BrochureService.cs where we invoke method that implements the service. We first convert the XML form of the file authorization into an object and then invoke the Print method of the local surrogate for the Brochure service.

```
if (processResult.status)
{
   Assertion fileAssertion = new Assertion (fileAuthorization);
   bool status = brochureService.Print(fileAssertion, pressRun);
```

The Print method in Brochure.cs invokes the HP File Content service. First it sets up a local proxy, provides the information needed to sign the SOAP request, and then invokes the ReadFile method of the service. Note that in the real world, we would extract the service destination from the authorization certificate when the service endpoint reference is encoded with something like WS-Addressing [30] from the Windows Communication Framework (WCF). Notice that we don’t specify the name of the file to be printed. The HP File Content service will extract it from the authorization certificate.

```
public bool Print(Assertion callbackToken, int pressRun)
{
  HPFileContentServiceWs.HPFileContentService fileWs =
          new HPFileContentServiceWs.HPFileContentService();
  HPFileContentServiceWs.FileServiceProcessingResult processingResult =
          fileWs.ReadFile(callbackToken.ToXml());
```

This invocation takes us to the ReadFile method of HPFileContentService on the HP Web site.

```
[WebService(Namespace = "http://www.hp.com/services")]
[WebServiceBinding(ConformsTo = WsiProfiles.BasicProfile1_1)]
public class HPFileContentService : System.Web.Services.WebService
{
    private string service =
          "http://www.hp.com/services/HPFileContentService.asmx";
    public HPFileContentService() {}
```

We instantiate the file service, convert the XML form of the authorization, and validate the authorization certificate. The fourth parameter to the ValidateToken method indicates that there are application specific checks to be done.

```
[WebMethod]
public FileServiceProcessingResult ReadFile(XmlElement token)
{
  HPFileContent fc = new HPFileContent();
  Assertion assertion = new Assertion(token);
  ServiceProcessingResult validatedResult =
       fc.ValidateToken(assertion, service, "ReadFile", true);
```

The validation routine does the same generic checks we just saw for the Brochure service.

```
public ServiceProcessingResult ValidateToken(Assertion signedAssertion,
          string service, string method, bool appSpecificConstraints)
{
    bool verified = false;
    ServiceProcessingResult result = new ServiceProcessingResult();
    MyFileServiceAuthZCertVerifier certVerifier = new
          MyFileServiceAuthZCertVerifier();
    verified = certVerifier.CheckSignature(signedAssertion);
    result.status = verified;
    if (!verified)
    {
        result.statusMsg = "HP: SAML token signature is incorrect" +
                        Environment.NewLine;
        return result;
    }
    else result.statusMsg = "HP: SAML token signature verified" +
                        Environment.NewLine;
    verified = certVerifier.CheckExpiration(signedAssertion);
    result.status = verified;
    if (!verified)
    {
        result.statusMsg += "HP: SAML token is expired" +
                    Environment.NewLine;
        return result;
    }
    else result.statusMsg += "HP: SAML token is current" +
                    Environment.NewLine;
    verified = certVerifier.CheckIssuer(signedAssertion);
    result.status = verified;
    if (!verified)
    {
        result.statusMsg += "HP: SAML token chain issuer mismatch" +
                    Environment.NewLine;
        return result;
    }
    else result.statusMsg += "HP: SAML CheckIssuer verified" +
                    Environment.NewLine;
    verified = certVerifier.CheckRevocationList(signedAssertion);
    result.status = verified;
    if (!verified)
    {
        result.statusMsg +=
              "HP: SAML token chain revocation checking failed" +
                    Environment.NewLine;
        return result;
    }
    else result.statusMsg +=
              "HP: SAML token chain revocation checking succeeded" +
                    Environment.NewLine;
    verified = certVerifier.CheckMethod(signedAssertion,
             service, method);
    result.status = verified;
    if (!verified)
    {
        result.statusMsg += "HP: Method not authorized" +
                        Environment.NewLine;
        return result;
    }
    else result.statusMsg += "HP: Method authorized" +
                        Environment.NewLine;
```

Now we check any application specific constraints, if there are any. In this case we want to make sure that all delegations refer to the same file. We could simply take the file name from the innermost certificate, but that might lead to errors.

```
if (appSpecificConstraints)
{
   verified = certVerifier.CheckResourceConstraints(signedAssertion);
```

The method in MyFileServiceAuthZCertVerifier.cs in the HPWebSite project steps through all the nodes in the authorization to use the file and returns true only if all the AttributeValues specify the same file name.

```
public override bool CheckResourceConstraints(Assertion certificate)
{
    XmlElement token = certificate.ToXml();
    XmlNodeList chainedAssertions =
           token.SelectNodes("//saml:AttributeValue", nsm);
    string fileName = "";
    foreach (XmlNode anode in chainedAssertions)
    {
        string fn = anode.InnerText;
        if ("" == fileName)
          fileName = fn;
        else
          if (fileName != fn) return false;
    }
    return true;
}
```

Back in HPFileContent.cs we set up the status and return.

```
result.status = verified;
if (!verified)
    result.statusMsg +=
            "ZebraCopy: Resource Access Constraints Violated" +
            Environment.NewLine;
else
    result.statusMsg +=
         "ZebraCopy: Resource Access Contraints Checked" +
         Environment.NewLine;
return result;
```

Back in HPFileContentService.cs, we'll read the file into a byte array and return the result to the Brochure service.

```
if (validatedResult.status)
{
  byte[] fileAccessResult  = null;
    fileAccessResult = fc.ReadFile(token);
```

The ReadFile method first asks for the name of the file to be read. For this sample code, we don't actually read the file. Instead, we return a null byte array.

```
public byte[] ReadFile(XmlElement token)
{
    XmlNodeList attributes = token.SelectNodes(
          "//saml:AttributeValue", nsm);
    string fn = attributes[0].InnerText;
    byte[] content = null;
    return content;
}
```

The last step in the HP File Content web service is to set the status, which includes the data.

```
FileServiceProcessingResult returnedResult =
      new FileServiceProcessingResult(true, null, fileAccessResult);
```

The FileServiceProcessingResult consists of the status, status message, and the result of the read.

```
public FileServiceProcessingResult (bool st, string msg, byte[] r)
{
    this.status = st;
    this.statusMsg = msg;
    this.content = r;
}
```

Finally, we return the result to the Brochure service.

```
return returnedResult;
```

The Brochure service can now print the job. The method DoProcessingContent is empty for this example.

```
if (readStatus.status)
{
  byte[] content = processingResult.content;
  DoProcessingContent(content,pressRun);
}
return readStatus.status;
```

The return takes us back to BrochureService.cs, where we set the status and return to the user application. The result shows that Bob's agent successfully invoked the Brochure service.

```
Show("Bob's agent performed service", result);
```

Then, since the job is done, Bob revokes bob2's right to use the service.

```
result = bob.RevokeBrochureServiceCertificate(
             bobBrochureAuthorization,bobToBob2Token);
```

The revocation method in Users.cs takes two SAML assertions as arguments. The first is the one Bob delegated from. The second is the delegation certificate to be revoked. This method sets up the proxy for the remote service, which includes providing the data needed to sign the SOAP message, and invokes its Revoke method.

```
public ServiceClassLibrary.BrochureWS.ServiceProcessingResult
        RevokeBrochureServiceCertificate(
             Assertion serviceToken,
             Assertion toBeRevoked)
{
    ServiceClassLibrary.BrochureWS.BrochureService srv =
        new ServiceClassLibrary.BrochureWS.BrochureService();
    srv.Timeout = 3600000;
    return srv.RevokeToken(serviceToken.ToXml(), toBeRevoked.ToXml());
```

We are now back in the Brochure web service at the Zebra Copy web site in BrochureService.cs. The method instantiates the already instantiated the local Brochure service surrogate and converts the certificate authorizing the right to revoke to an object. It then validates this authorization. If authorized, it then invokes the RevokeToken method.

```
[WebMethod]
public ServiceProcessingResult RevokeToken(
        XmlElement token, XmlElement toBeRevoked)
{
    Brochure brochureService = new Brochure();
    Assertion assertion = new Assertion(token);
    ServiceProcessingResult result =
           brochureService.ValidateToken(
                assertion, service, "Revoke");
    if (result.status)
    {
        Assertion toBeRevokedAssertion = new Assertion(toBeRevoked);
        result.status = false;
        result = brochureService.RevokeToken(assertion,
                   toBeRevokedAssertion);
```

The RevokeToken method in Brochure.cs initializes the status report and the object that does application specific validation. Finally, we make sure that this is a valid revocation request.

```
public ServiceProcessingResult RevokeToken(
           Assertion token, Assertion toBeRevoked)
{
    ServiceProcessingResult result = new ServiceProcessingResult();
    AuthZCertVerifier certVerifier = new MyAuthZCertVerifier();
    bool verified = certVerifier.ValidateRevocationRequest(
            token, toBeRevoked);
```

Next we enter the ValidateRevocationRequest method in AuthZCertVerifier.cs in the Authorization project. It extracts the AssertionID, a string unique to each Assertion, and tests to see if the AssertionID of the authorization to revoke is the same as the AssertionID in the outermost Evidence field. If they are, we know that the authorization to be revoked was delegated from the certificate used as an authorization to revoke.

```
public bool ValidateRevocationRequest(
        Assertion token, Assertion toBeRevoked)
{
    string tokenID = token.AssertionID;
    XmlElement xmlElement = toBeRevoked.ToXml();
    XmlNode node = xmlElement.SelectSingleNode(
        "/saml:Assertion/saml:AuthorizationDecisionStatement/saml: " +
        "Evidence/saml:Assertion", nsm);
    string toBeRevokedID = node.Attributes["AssertionID"].InnerText;
    return tokenID == toBeRevokedID;
}
```

Back in the RevokeToken method of Brochure.cs we set the status message and invoke the method that does the actual revocation.

```
if (!verified)
{
    result.statusMsg += "Token Revocation Request Denied" +
                     " Due To Invalid Client Certificate";
    result.status = false;
}
else
{
    result.status = true;
    result.statusMsg = null;
}
if (verified)
{
    string resourceName = "";
    certVerifier.RevokeCertificate(toBeRevoked, resourceName);
```

The RevokeCertificate method of AuthZCertVerifier.cs in the Authorization project extracts the AssertionID of the supplied certificate and adds it to the revocation list. Note that the label is used as a placeholder for other data that we might want to keep with the AssertionID. For example, we might want to keep the expiration date of the certificate so we can delete the entry after the certificate has expired.

```
public void RevokeCertificate(Assertion assertion, string label)
{
    XmlElement element = assertion.ToXml();
    string assertionID = element.Attributes["AssertionID"].InnerText;
    revocationList.Add(assertionID, label);
}
```

Back in Brochure.cs we return the status report.

```
return result;
```

The return via BrochureService.cs and Users.cs takes us all the way back to UserApplication.cs.

```
Show("Bob revokes Bob2's certificate", result);
Pause();
```

The rest of the code involves Bob’s delegation to Alice and her attempted uses and misuses of the Brochure service. We’ve seen all the methods that will be invoked, so from now on we’ll just step through UserApplication.cs.

Normally, Alice would be running her own process, but here we just run her with Bob. Once Alice is instantiated, Bob delegates to her the right to order press runs of up to 100 copies. We write the certificate to a file so you can examine it.

```
User alice = new User("alice.pfx","password");
int copyLimit = 100;
Assertion bobToAliceToken = bob.DelegateBrochureCertificate (
     bobBrochureAuthorization, alice, copyLimit);
AuthZUtilities.OutputAssertionToFile(
     bobToAliceToken,
     "Bob's Delegation of Brochure Service to Alice",
     "AliceBrochureService.xml");
Show("AliceBrochureService.xml");
Pause();
```

Alice next gets her own certificate to the file to be printed.

```
Assertion aliceHpFileContentServiceAssertion =
                alice.GetFileContentServiceToken(fileName);
AuthZUtilities.OutputAssertionToFile(
   aliceHpFileContentServiceAssertion,
   "Alice's Authorization to Read/Write a File at the HP File Service",
      "AliceFileService.xml");
```

Alice delegates her right to read the file to the Brochure service. The bobToAliceToken contains the X.509 certificate of the Brochure service.

```
Assertion aliceDelegatedHPFileContentServiceAssertion =
    alice.DelegateFileServiceCertificate(
         aliceHpFileContentServiceAssertion, bobToAliceToken);
AuthZUtilities.OutputAssertionToFile(
     aliceDelegatedHPFileContentServiceAssertion,
     "Alice's Delegation to Read a File to Brochure Service",
     "AliceFileServiceToBrochure.xml");
```

Alice next orders 28 brochures. Note that Alice is being careless. Although she only wants to print 28 copies, the process she runs will be able to order up to 100. She should do what Bob did earlier and create an authorization good for the exact number of copies she wants to make.

```
result = srv.Print(bobToAliceToken.ToXml(),
            aliceDelegatedHPFileContentServiceAssertion.ToXml(), 28);
Show("Alice performed service (with 28 copies)", result);
```

That worked, so Alice tries ordering 150. Notice that the information needed to sign her SOAP request doesn't need to be reset because it is part of her messaging layer.

```
result = srv.Print( bobToAliceToken.ToXml(),
       aliceDelegatedHPFileContentServiceAssertion.ToXml(), 150);
Show("Alice performed service (with 150 copies)", result);
```

That fails when the validation detects that she is ordering more copies than specified in her authorization. Alice now wants to continue to order brochures after Bob revokes her authorization, so she creates an agent, alice2, and delegates her rights to it.

```
User alice2 = new User("alice2.pfx","password");
Assertion aliceSelfDelegatedCertificate =
  alice.DelegateBrochureCertificate (bobToAliceToken, alice2, 100);
AuthZUtilities.OutputAssertionToFile(
      aliceSelfDelegatedCertificate,
      "Alice's Delegated File Certificate to Alice2",
      "AliceToAliceBrochure.xml");
```

Sure enough, Bob revokes her authorization and verifies that the revocation request was accepted.

```
result = bob.RevokeBrochureServiceCertificate(bobToAliceToken);
Show("Bob revokes Alice's certificate", result);
```

Alice tries to use her certificate and finds that it has been revoked. That attempt fails because the delegation chain is terminated at the first revoked certificate.

```
result = srv.Print( bobToAliceToken.ToXml(),
            aliceDelegatedHPFileContentServiceAssertion.ToXml(), 100);
Show("Alice performed service (certificate revoked already)", result);
Pause();
```

That didn't work, so Alice tries with the certificate she delegated to alice2, but that fails, too. Note that Alice had to reset the SOAP signing information to use alice2's private key.

```
result = srv.Print( aliceSelfDelegatedCertificate.ToXml(),
         aliceDelegatedHPFileContentServiceAssertion.ToXml(), 100);
Show("Alice performed service with self-delegated certificate ",
    result);
```

In a final attempt to get access to the Brochure service, Alice asks HP for the right. Since Alice isn't in the approved list, HP denies the request.

```
System.Console.WriteLine(
      "Alice attempts to get brochure service authorization.");
Assertion aliceBrochureAuthorization = alice.GetBrochureServiceToken();
if (null != aliceBrochureAuthorization)
{
  AuthZUtilities.OutputAssertionToFile(
     aliceBrochureAuthorization,
     "Alice’s Right to Use the Zebra Copy Brochure Service",
     "AliceBrochureService.xml");
  Show("AliceBrochureService.xml");
} else System.Console.WriteLine("Access denied.");
Pause();
```

That ends the detailed description of the sample application. At this point, you should be able to start building your own infrastructure based on Federated Access Management.

Appendix E. SOAP Message Layer Code

There is a problem with the sample code described in Appendix D; there are no restrictions on who can use an authorization. For example, Alice could submit Bob’s authorization if hers had been revoked. That’s because we didn’t show how to sign the SOAP messages and coordinate that signature with the SAML Assertions that it includes. We’d have to do the same thing if we were using the SAML certificates for authentication or to carry attributes, but it’s worth seeing how it is done.

Sample code including this verification step is available for download from HP [17]. There are two reasons we are providing different versions. The version with message signing doesn’t single step between processes as does the code described in Appendix D. We also found that the hidden interaction with the messaging layer makes understanding the control flow more difficult. Hence, we decided to present the basic mechanism without message authentication and describe the additions separately.

Microsoft provides the Web Services Extensions (WSE) library for dealing with many aspects of SOAP messages, including signing them and verifying the signatures. We did not use this library because we want our sample code to be more platform independent. Also, the latest WSE 3.0 does not provide security token classes to support directly file-based certificate stores, such as the *.pfx files used in the sample code.

In this Appendix, we’ll describe two classes that implement just the WSE functionality we need. The first class has no methods, just data members. MessageSigningHeader.cs in the SoapExtensionLib project is used to communicate signing information between the application and the SOAP messaging layer. The first two variables contain the file name of the user’s key store and the password needed to extract the user’s private key. Next we have a variable to hold the signer’s X.509 certificate, which contains the signer’s public key. Finally, there is a place to store the actual signature and the result of verifying it against the received message.

```
public class MessageSigningHeader :
      System.Web.Services.Protocols.SoapHeader
```

```
{
        public string MessageSigningCertificateStore;
        public string MessageSigningStorePassword;
        public string MessageSigningCertificate;
        public XmlElement MessageSigningSignature;
        public bool VerificationResult;
```

In order to understand the other class, we need to see how the client and service use this header and how the user gets access to the messaging layer.

E.1. Client and Server Changes

Some changes are needed to the code shown in Appendix D in order to set up the data needed to manage the SOAP messages. Refer back to the point in UserApplication.cs where bob2 is about to invoke the Brochure service. Before, bob2 simply instantiated the service proxy and set a long timeout. In order to properly sign the SOAP request, bob2 now also instantiates a MessageSigningHeader and provides the file name of a key store and the password needed to extract the private key.

```
ServiceClassLibrary.BrochureWS.BrochureService srv =
      new ServiceClassLibrary.BrochureWS.BrochureService();
ServiceClassLibrary.BrochureWS.MessageSigningHeader header =
      new ServiceClassLibrary.BrochureWS.MessageSigningHeader();
header.MessageSigningCertificateStore = "bob2.pfx";
header.MessageSigningStorePassword = "password";
srv.MessageSigningHeaderValue = header;
srv.Timeout = 3600000;
```

Some changes are also needed on the service side. The class embodying the service needs a place to hold the message signing header. So, for example, the BrochureService class adds a member

```
public MessageSigningHeader msgSigningHeader;
```

and each method adds a web service declaration, *i.e.*,

```
 [WebMethod]
 [SoapHeader("msgSigningHeader", Direction = SoapHeaderDirection.In)]
 public ServiceProcessingResult Print(XmlElement serviceToken,
        XmlElement fileAuthorization, int pressRun)
```

These declarations let the server access the signature the client put into the SOAP header. The server also needs to verify that the authorization presented was delegated to the public key corresponding to the private key used to sign the SOAP request. We enable that check by adding the signature to the parameters passed into the validation method. For example, BrocureService.cs is changed to

```
ServiceProcessingResult processResult = brochureService.ValidateToken(
    assertion, service, "Print", pressRun, msgSigningHeader);
```

which eventually takes us to the application independent validation method in Brochure.cs. We see that the MessageSigningHeader argument has been added and that we check the signature verification. We'll see later how we verified the signature in a service independent way by putting an interceptor in the messaging layer.

```
public ServiceProcessingResult ValidateToken(
         Assertion signedAssertion, string service, string method,
         MessageSigningHeader header)
{
    bool verified = false;
    ServiceProcessingResult result = new ServiceProcessingResult();
    MyAuthZCertVerifier certVerifier = new MyAuthZCertVerifier();
    result.status = true;
    verified = header.VerificationResult;
    if (!verified)
    {
        result.status = false;
        result.statusMsg = "Brochure: Client request signature " +
                "checking failed" + Environment.NewLine;
        return result;
    } else {
        result.statusMsg = "Brochure: Client request signature " +
               "checking verified" + Environment.NewLine; }
```

Once we're sure the SOAP message was properly signed, we can check that the signer's public key is the one the authorization was issued to. We do that by extracting the signer's X.509 certificate from the message header and invoking the check method.

```
byte[] cert = System.Text.Encoding.Default.GetBytes(
                    header.MessageSigningCertificate);
X509Certificate certFromClient = new X509Certificate(cert);
verified = certVerifier.CheckIssuserWithRequest(
         signedAssertion, certFromClient);
```

The verfication is done by the CheckIssuerWithRequest method which we added to AuthZCertVerifier.cs in the Authorization project. This method extracts the Subject field from the authorization assertion and gets the delegatee's X.509 certificate from the SubjectConfirmationData. The public key in that certificate is then compared with the public key in the X.509 certificate in the SOAP message header.

```
public bool CheckIssuerWithRequest(
        Assertion assertion, X509Certificate certFromClient)
{
    XmlElement xmlElement = assertion.ToXml();
    XmlNode subjectNode = xmlElement.SelectSingleNode(
        "/saml:Assertion/saml:AuthenticationStatement/saml:Subject",
        nsm);
    XmlElement subjectElement = (XmlElement)subjectNode;
    XmlNode x509DataNode = subjectElement.SelectSingleNode(
        "saml:SubjectConfirmation/saml:SubjectConfirmationData",
        nsm);
    X509Certificate innerSubjectCertificate =
            RecoverX509Certificate(x509DataNode.InnerText);
    bool verified = AuthZUtilities.CompareByteArray(
                              innerSubjectCertificate.GetPublicKey(),
                              certFromClient.GetPublicKey());
    if (verified) return true;
    return false;
}
```

In a real implementation, the server would sign its messages to the client in the service response path, and the client would verify those signatures. We didn't do that for this sample code because it isn't needed to demonstrate using SAML certificates as authorizations. Hence, the changes you've just seen are all that is needed for the client and server. Everything else is done below the application layer.

E.2. Message Interception Basics

The bulk of the work is done in WebServiceSOAPExtension.cs. We followed the instructions provided by Microsoft [20] to write this class. That article describes what is happening in terms of an IIS web server and the .NET framework. However, the bulk of the code in this file is independent of them.

This WebServiceSoapExtension class works in conjunction with the MessageSigningHeader introduced in Appendix E.1. The MessageSigningHeader allows the client to specify the settings per invocation request required for message processing at the SOAP message level. On the receiving side, the MesageSigningHeader allows the server to retrieve the result from the message processing done at the SOAP message level.

To get started, we need to understand what is happening underneath the application at the messaging layer. The user invokes a web service by specifying a method on a proxy. We saw this when bob2 invoked the Brochure service with

```
ServiceClassLibrary.BrochureWS.ServiceProcessingResult result =
        srv.Print(bobToBob2Token.ToXml(),
           bobDelegatedHPFileContentServiceAssertion.ToXml(), 37);
```

The proxy converts this method invocation into a SOAP request that is sent as an HTTP message to the web server. Based on the SOAPAction header or the request element's name, the server decides to which method of which class in the application to dispatch the message and deserialized parameters.

The Extensions library sits between the client and web server and between the web server and service as shown in Figure A3. That means we won't see the normal call return path. Instead, magical things seem to happen when single stepping the code. Magically, we enter some routine. Magically, a return takes us to some other strange place. Of course, it's not really magic. It's just smoke and mirrors. Under the covers, we are bouncing back and forth between the messaging layer and the code visible to us.

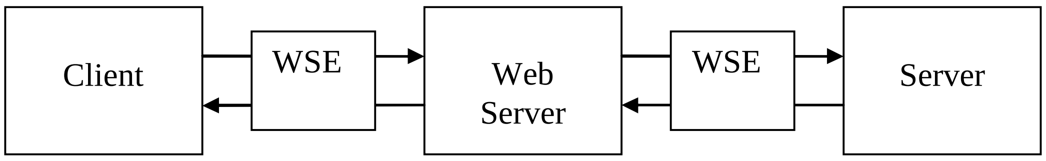

Figure A3. Message pipeline with SOAP Extension hooks.

Let's look at the flow between the messaging layer and the extension code as shown in Figure A4, which we copied from the article we used when writing this code. In this figure, code in the extension is in bold. The gray boxes are message stages that code in the ProcessMesssage method uses in a switch statement.

The interception process gets started when the client invokes a method on the service proxy. The client is given the chance to do some setup of the interception before the SOAP request reaches the messaging layer by invoking an implemention of the abstract methods in the initialization phase. The messaging layer then invokes the ChainStream method, which enables the actual interception by providing a Stream from the messaging layer to the extension code and one from the extension code back into the messaging layer. The client is given two chances to interact with the SOAP request on the way out, before and after serialization. The serialized message is in the form of an XmlDocument object. The serialized message is a UTF8 document consisting of the XML representing the SOAP request, including the SOAP headers and soap:Body. Once the client-side code is finished with the serialized message, the messaging layer puts the SOAP request on the wire for delivery to the server.

The service has a chance to do some setup of its message interception immediately after the message comes in over the wire. As with the client, in ChainStream the service sets up the Streams that enable the interception. The service gets two chances to interact with the message before the actual service is invoked, before deserialization and after deserialization. Once the post-deserialization processing is done, the actual service in the application layer gets invoked.

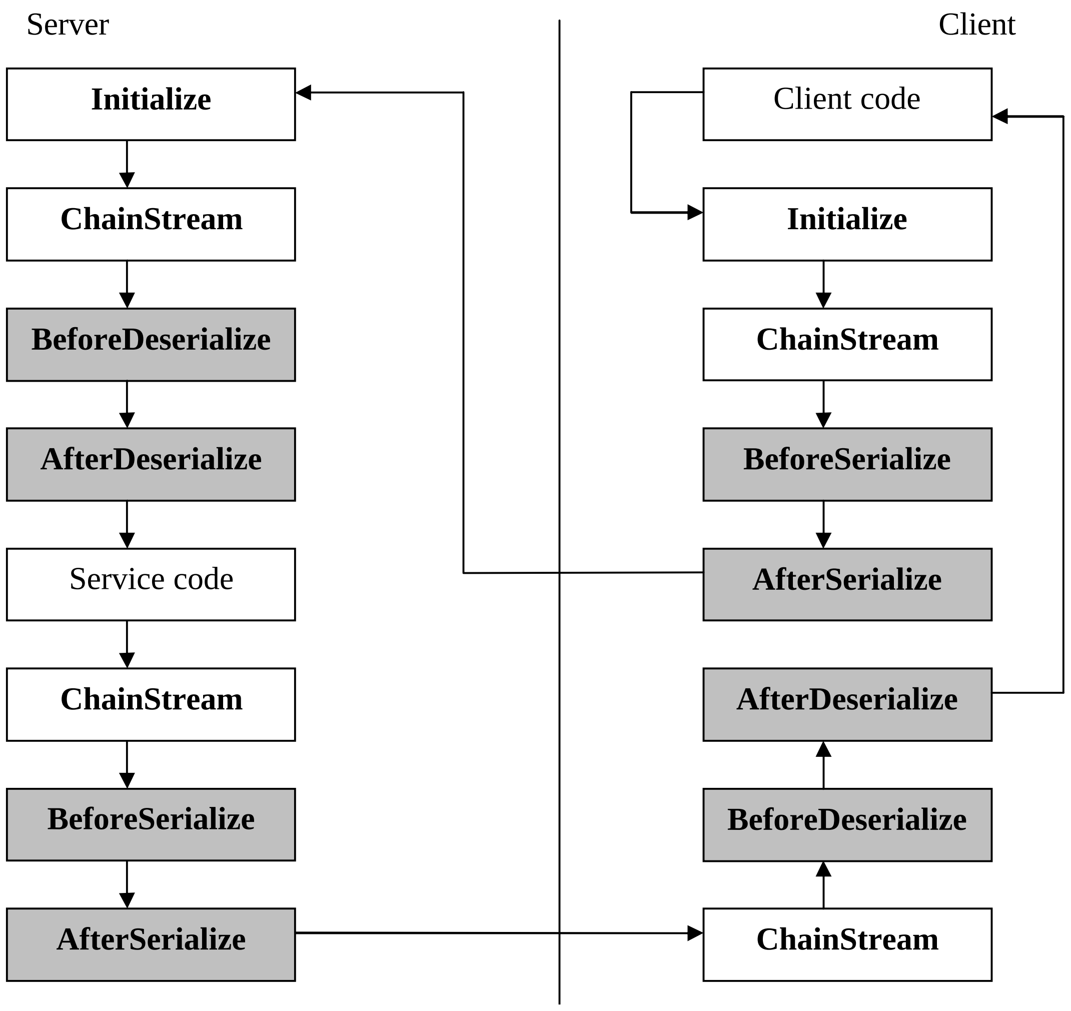

Figure A4. Flow between the message layer and the interception code.

Any response from the service is sent back to the client. What makes the invocation RPC-like is the asymmetry in the message handling. Note that the return is just a SOAP message, but it goes directly to the client's ChainStream method instead of passing through the initialization code. That makes it easier to correlate the return with the request, but it complicates processing of multiple requests outstanding at the same time.

E.3. Implementing Interception

In this Section, we'll see the actual code we wrote for signing the SOAP request and verifying the signature. We only implement these methods for the client request. In a real system, the server would also sign its messages, and the client would validate the signature. Another oddity of this reference implementation is that we use the same code for both the client and the service. It's just a convenience for this sample implementation. Also, in Appendix D we walked through the code in execution order. The complicated interaction between the messaging layer and interception code makes this approach less useful. So, here we’ll present the code more or less in the order it appears in the source files.

You plug into the message stream by overriding methods in the SoapExtension class. Before the message enters the messaging layer, you get the opportunity to do some setup by providing a concrete implementation of some abstract classes. Here we simply do nothing.

```
public override object GetInitializer(Type serviceType)
{ return null; }
public override object GetInitializer(
   LogicalMethodInfo methodInfo, SoapExtensionAttribute attribute)
{ return null; }
public override void Initialize(object initializer)
{ return; }
```

Even though there are four lines entering an leaving each of the SOAP Extensions boxes in Figure A3, we’ll only be dealing with one message at a time. Hence, we define a Stream for the message coming into the interceptor and one for the processed message going out.

```
public class WebServcSoapExt : SoapExtension
{
      private Stream outwardStream;
      private Stream inwardStream;
```

The actual interception starts by overriding the ChainStream method, which gives us the opportunity to store the message in a Stream variable. The argument is the SOAP message. Here we see our first bit of magic; the return takes us to the ProcessMessage method. That’s not really what happened, of course. We really went from ChainStream into the messaging layer which called ProcessMessage.

```
public override Stream ChainStream(Stream stream)
{
      inwardStream = stream;
      outwardStream = new MemoryStream();
      return outwardStream;
}
```

ProcessMessage gets called two times on each of the sender and the receiver sides. On the sending side, ProcessMessage is first invoked before the XML document containing the SOAP request has been serialized. That gives you the opportunity to modify the request at the object level, perhaps to add some attribute. We won’t be doing anything, so we simply leave the switch block.

```
public override void ProcessMessage(
          System.Web.Services.Protocols.SoapMessage message)
{
      string soapMsg1;
      StreamReader readStr;
      StreamWriter writeStr;
      XmlDocument xDoc = new XmlDocument();
      xDoc.PreserveWhitespace = false;
      switch (message.Stage)
      {
            case SoapMessageStage.BeforeSerialize:
                  break;
```

Next, ProcessMessage is invoked on the sending side after serialization. At this point, you have an XML document and can do such things as encrypt or sign the request. The message is defined as a stream provided by ChainStream, which we convert to a string.

```
case SoapMessageStage.AfterSerialize:
      inwardStream.Position = 0;
      readStr = new StreamReader(outwardStream);
      writeStr = new StreamWriter(inwardStream);
      soapMsg1 = readStr.ReadToEnd();
```

Since we're using the same code for the client and server, we need to figure out which this invocation is for. In this case, it's the client, so we'll sign the SOAP message. We'll start by turning the serialized message into an XmlDocument object. It seems strange to convert the serialized message into an object instead of just signing the message before serialization. The difference is that serialization wraps the body of the message in a soap:Body tag. That lets us extract the entire body and sign it as a unit. Before serialization, we would have to extract the elements in turn and sign them individually.

Once we have the XmlDocument object, we'll extract the file name of the key store and the corresponding password. Recall, these items were put into the SOAP header before the service proxy was invoked.

```
if ( message is System.Web.Services.Protocols.SoapClientMessage)
{
    xDoc.LoadXml(soapMsg1);
    XmlNodeList xPassword =
          xDoc.GetElementsByTagName("MessageSigningStorePassword");
    string password = xPassword[0].InnerXml;
    XmlNodeList xCertificateStore =
          xDoc.GetElementsByTagName("MessageSigningCertificateStore");
    string certificateStoreFileName = xCertificateStore[0].InnerXml;
```

Next, we extract the client's public and private keys as well as the corresponding X.509 certificate. Then we're ready to sign the message.

```
KeyPair keyPair = new KeyPair(certificateStoreFileName, password);
byte[] myCertificate =
        keyPair.Certificate.Export(X509ContentType.Cert);
string certificateString = Convert.ToBase64String(myCertificate);
XmlElement  sig = ComputeSignature(keyPair, xDoc);
```

Once that's done, we add the client's X.509 certificate and the message signature into the SOAP header. We don't need the key store and password for the new header.

```
MessageSigningHeader header = new MessageSigningHeader();
header.MessageSigningCertificate = certificateString;
header.MessageSigningSignature = sig;
```

We can now add the signature to the SOAP header. After serializing the header object into XML, we import it to the XmlDocument object containing the SOAP request. The import works by cloning the XmlElement specified by the first argument. The second argument says to clone the full object graph, not just the top level. The return value is the clone of the input argument.

```
XmlElement result = SerializedToXml(header);
XmlElement importedResult = (XmlElement)xDoc.ImportNode(result, true);
```

We don't want to transmit the signing information to the service, especially the client's private key, so we remove them.

```
XmlNodeList messageSigningHeader =
   xDoc.GetElementsByTagName("MessageSigningHeader");
messageSigningHeader[0].RemoveChild(xPassword[0]);
messageSigningHeader[0].RemoveChild(xCertificateStore[0]);
```

You'd think we'd be done, but there is a glitch. We need to add the parent node's namespace to the node containing the signature. If we don't, the resulting message contains an empty specification of "" for xmlns, which causes deserialization problems. RemoveAttributes doesn't remove the empty specification for xmlns.

```
XmlNode node1 = importedResult.SelectSingleNode(
        "//MessageSigningCertificate");
XmlNode node1_clone = xDoc.CreateElement(
        "MessageSigningCertificate",
        messageSigningHeader[0].NamespaceURI);
node1_clone.InnerText = node1.InnerText;
messageSigningHeader[0].AppendChild(node1_clone);
XmlNode node2 = importedResult.SelectSingleNode(
        "//MessageSigningSignature");
XmlNode node2_clone = xDoc.CreateElement("MessageSigningSignature",
messageSigningHeader[0].NamespaceURI);
node2_clone.AppendChild(node2.FirstChild);
messageSigningHeader[0].AppendChild(node2_clone);
```

We now put the XML document containing the serialized SOAP message into the output stream and exit the switch block. Since there is no more code, the return is into the messaging layer.

```
soapMsg1 = xDoc.InnerXml;
writeStr.Write(soapMsg1);
writeStr.Flush();
break;
```

The messaging layer sends the SOAP request to the server's web server for forwarding to the server. On the way from the web server to the server, the message passes through the extension library, which calls Initialize and ChainStream. In this sample implementation, that's the same code we saw on the client side. Next, ProcessMessage gets called before deserialization. As before, we initialize a couple of Streams to handle the message on the way in and on the way out and convert the request to a string.

```
case SoapMessageStage.BeforeDeserialize:
      readStr = new StreamReader(inwardStream);
      writeStr = new StreamWriter(outwardStream);
      soapMsg1 = readStr.ReadToEnd();
```

Since we're using the same library code for both the server and the client, we need to figure out which one we've got this time. It turns out to be the server, so we convert the XML document into an object and extract the signing header. Those arguments get sent to the VerifySignature method.

```
else if( message is System.Web.Services.Protocols.SoapServerMessage)
{

    xDoc.LoadXml(soapMsg1);
    XmlNodeList headers =
        xDoc.GetElementsByTagName("MessageSigningHeader");
    bool result = VerifySignature(xDoc, (XmlElement)headers[0]);
```

In the code we saw earlier, we simply returned the verification result. We can't do that here because logically we're in the messaging layer. Instead, we put the verification result into the SOAP header where the application layer verification method can get to it. The implementation is straightforward except for one thing. Only a lower case "false" gets through the conversion process unmodified. "False" is not a legitate boolean token when parsed by the SOAP message deserilizer. The last step simply makes sure the soap:Body conforms to the schema expected by the SOAP message's deserializer by removing any attributes added for the purpose of signature construction (at the client side) and verification process (at the server side).

```
XmlNodeList verificationResultNodes =
       xDoc.GetElementsByTagName("VerificationResult");
verificationResultNodes[0].InnerText = result.ToString().ToLower();
XmlNamespaceManager nsm = new XmlNamespaceManager(new NameTable());
nsm.AddNamespace("soap", "http://schemas.xmlsoap.org/soap/envelope/");
nsm.AddNamespace("dsig", SignedXml.XmlDsigNamespaceUrl);
XmlNode node = xDoc.SelectSingleNode("//soap:Body", nsm);
XmlElement soapNode = (XmlElement)node;
soapNode.RemoveAllAttributes();
```

As before, the result is returned as a Stream to the messaging layer.

```
soapMsg1 = xDoc.InnerXml;
writeStr.Write(soapMsg1);
writeStr.Flush();
outwardStream.Position = 0;
break;
```

We’ve already seen how the service verifies that the signer of the SOAP request is the rightful holder of the authorization to use the service. Now let’s look at the return path. The return is simpler because the service doesn’t sign the reply, and the client knows not to check it.

The reply goes directly from the server’s AfterSerialize into the client’s ChainStream. The messasging layer then invokes the client’s ProcessMethod before deserialization. The client does no work in this case, so we just prepare the message to be returned to the message layer. We saw the code in the else if block when the server checked the signature on the SOAP request. It’s shown here just as a reminder.

```
case SoapMessageStage.BeforeDeserialize:
   readStr = new StreamReader(inwardStream);
   writeStr = new StreamWriter(outwardStream);
   soapMsg1 = readStr.ReadToEnd();
   if (message is System.Web.Services.Protocols.SoapClientMessage)
      xDoc.LoadXml(soapMsg1);
   else if (message is System.Web.Services.Protocols.SoapServerMessage)
   { ... }
```

The message is put back into the messaging layer with code we saw previously.

```
soapMsg1 = xDoc.InnerXml;
writeStr.Write(soapMsg1);
writeStr.Flush();
outwardStream.Position = 0;
break;
```

ProcessMessage gets invoked one more time after deserializing the reply, but there is nothing for the client to do, so we exit the switch block and return to the messaging layer.

```
case SoapMessageStage.AfterDeserialize:
    break;
```

The client next gets control at the statement immediately following the invocation of the service proxy. That’s the complete round trip, but we still need to see the code for signing the request and for verifying the signature.

The method that computes the signature starts by defining an namespace manager for parsing the document and extracts the SOAP body, the part of the message that gets signed.

```
private XmlElement ComputeSignature(
           KeyPair keyPair, XmlDocument document)
{
   XmlNamespaceManager nsm = new XmlNamespaceManager(new NameTable());
   nsm.AddNamespace("soap",
             "http://schemas.xmlsoap.org/soap/envelope/");
   nsm.AddNamespace("dsig", SignedXml.XmlDsigNamespaceUrl);
   XmlNode node = document.SelectSingleNode("//soap:Body", nsm);
   XmlElement soapNode = (XmlElement)node;
```

Next, we create a new attribute “signer” as the reference ID for constructing the XML signature.

```
XmlAttribute attribute = (XmlAttribute)document.CreateNode(
            XmlNodeType.Attribute, "ID", "");
attribute.InnerText = "signer";
soapNode.Attributes.Append(attribute);
SignedXml signedXml = new SignedXml(soapNode);
```

We now extract the key to be used to sign the message, add it to the state of the signing object, and specify the reference (ID “#signer”) to the part of the XML document that will be used to construct the signature. Here that part is the entire SOAP body.

```
X509Certificate _x509Certificate = keyPair.Certificate;
RSA key = keyPair.PrivateKey;
signedXml.SigningKey = key;
Signature XMLSignature = signedXml.Signature;
Reference reference = new Reference("");
reference.Uri = "#signer";
XmlDsigEnvelopedSignatureTransform env = new
       XmlDsigEnvelopedSignatureTransform();
reference.AddTransform(env);
XmlDsigExcC14NTransform env2 = new XmlDsigExcC14NTransform();
reference.AddTransform(env2);
XMLSignature.SignedInfo.AddReference(reference);
```

We add the client’s public key information and invoke the method that does the actual signing computation.

```
KeyInfo keyInfo = new KeyInfo();
keyInfo.AddClause(new KeyInfoX509Data(_x509Certificate));
XMLSignature.KeyInfo = keyInfo;
signedXml.ComputeSignature();
```

Finally, we return the signature as an XmlElement that can be include in the SOAP header.

```
XmlElement xmlDigitalSignature = signedXml.GetXml();
return xmlDigitalSignature;
```

The SOAP header now has a signature that can be used to verify the integrity of the SOAP body. This signature also contains the public key corresponding to the private key used to sign the message.

When the message reaches the server, we verify the signature before the message is deserialized. That guarantees that the message hasn’t been tampered with. Note that we are not checking for replay attacks. To do that, we would need to keep some state on the server side, such as the time stamp of the last message seen from that client.

We start the verification by defining a name space manager for parsing the SOAP envelope and extracting the MessageSigningHeader that the client put into the SOAP header. Note that we're not testing for missing or invalid headers in this sample code. That means an incorrectly formatted message will crash the service.

```
private bool VerifySignature(XmlDocument document, XmlElement header)
{
    XmlNamespaceManager nsm = new XmlNamespaceManager(new NameTable());
    nsm.AddNamespace("soap",
             "http://schemas.xmlsoap.org/soap/envelope/");
    nsm.AddNamespace("dsig", SignedXml.XmlDsigNamespaceUrl);

    XmlNodeList signingHeaderNodes= document.GetElementsByTagName (
           "MessageSigningHeader");
    XmlNode theSigningHeaderNode = signingHeaderNodes[0];
```

We next step through the nodes in the header looking for the one containing the signature. Again, we're not checking to make sure that one exists, something you'd do in a real implementation.

```
XmlNode signatureNode = null;
for (int i = 0; i < theSigningHeaderNode.ChildNodes.Count; i++)
{
   XmlNode node = theSigningHeaderNode.ChildNodes[i];
   if (node.Name == "MessageSigningSignature")
   {
      signatureNode = node.FirstChild;
      break;
   }
}
```

We now extract the soap body from the message and append the XML signature carried in the SOAP header to it. We also instantiate an object that has the method that checks the signature and supply the signature to be checked to it.

```
XmlNode snode = document.SelectSingleNode("//soap:Body", nsm);
XmlElement soapNode = (XmlElement)snode;
soapNode.AppendChild(signatureNode);
SignedXml signedXml = new SignedXml(soapNode);
signedXml.LoadXml((XmlElement)signatureNode);
```

There are two checks we can do. The first makes sure that the message hasn't been modified and that it was properly signed. The second also verifies that the private key used to sign the message corresponds to a particular public key. We need the latter. The public key we're checking comes from the X.509 certificate that the client put into the header.

```
byte[] certificate = System.Text.Encoding.Default.GetBytes(
         header.GetElementsByTagName (
             "MessageSigningCertificate")[0].InnerText);
X509Certificate _x509Certificate = new X509Certificate(certificate);
X509Certificate2 certificate2 = new X509Certificate2(_x509Certificate);
```

Finally, we can check the validity of the signature. The second argument says to check the validity of the signature but not that of the X.509 certificate.

```
result = signedXml.CheckSignature(certificate2, true);
return result;
```

There are two utility methods in WebServiceSOAPExtension.cs that are included here for completeness. No explanation is needed. The first creates an XmlElement representation of a MessageSigningHeader.

```
private XmlElement SerializedToXml(MessageSigningHeader header)
{
   MemoryStream memoryStream = new MemoryStream();
   XmlTextWriter xmlTextWriter = new XmlTextWriter(
         memoryStream, Encoding.UTF8);
   XmlSerializer xs = new XmlSerializer(typeof(MessageSigningHeader));
   xs.Serialize(xmlTextWriter, header);
   memoryStream = (MemoryStream)xmlTextWriter.BaseStream;
   string xmlizedString = (new UTF8Encoding()).GetString(
        memoryStream.ToArray());
   int start = xmlizedString.IndexOf("<MessageSigningHeader");
   int end = xmlizedString.IndexOf("</MessageSigningHeader>");
   string content = xmlizedString.Substring(start, (end - start) +
        "</MessageSigningHeader>".Length);
   XmlDocument doc = new XmlDocument();
   doc.LoadXml(content);
   return (doc.DocumentElement);
}
```

The other writes and XmlElement object to a file. It is very similar to the OutputAssertionToFile method we saw in AuthZUtilities.cs.

```
public static void OutputXmlElementToFile(
      XmlElement xmlElement, string label, string fn)
{
   string baseDir = KeyPair.baseDir;
   string filename = baseDir + fn;
   XmlDocument dom = new XmlDocument();
   FileStream fs = null;
   if (File.Exists(filename))
       File.Delete(filename);
   fs = new FileStream(filename, FileMode.Create);
   XmlDeclaration xmlDecl =
                dom.CreateXmlDeclaration("1.0", "utf-8", null);
   dom.InsertBefore(xmlDecl, dom.DocumentElement);
   XmlNode rootNode = dom.CreateNode(XmlNodeType.Element,
                "document-log", String.Empty);
   dom.AppendChild(rootNode);
   XmlNode timeAttributeNode =
      dom.CreateNode(XmlNodeType.Attribute, "timestamp", String.Empty);
   timeAttributeNode.InnerText = DateTime.Now.ToString();
   rootNode.Attributes.Append((XmlAttribute)timeAttributeNode);
   XmlNode labelAttributeNode =
      dom.CreateNode(XmlNodeType.Attribute, "label", String.Empty);
   labelAttributeNode.InnerText = label;
   rootNode.Attributes.Append((XmlAttribute)labelAttributeNode);
   XmlNode newNode = dom.ImportNode(xmlElement, true);
   rootNode.AppendChild(newNode);
   fs.Position = 0;
   dom.Save(fs);
   fs.Close();
}
```

That ends the description of the message interception code used to sign and verify the signature of the SOAP messages. If you find any errors or a better way of doing something, please contact the authors.