****

**Redbox**

Redbox.com Federated Authentication Integration

**Author**: *Gary Strasdin*

**Document Title**: *Redbox.com Federated Authentication Integration*

**Updated**: *6/14/2011 3:51 PM*

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Reviewer and Approvers** | | | | |
| **Name** | **Title/Role** | **Reviewer/**  **Approver** | **Approval**  **Date** | **Approval Method** |
| Saad Rehmani | Director of Architecture | Approver |  |  |
| Gunhan Tatman | VP of Development | Approver |  |  |
| Bart Dolega | Director of Platform Services | Approver |  |  |
|  | Product Manager | Reviewer |  |  |
|  |  |  |  |  |
|  |  |  |  |  |
| Ratnakar Lavu | CTO | Approver |  |  |

[1. Introduction 3](#_Toc295979723)

[1.1 Project Description 3](#_Toc295979724)

[1.2 Description and Scope of this Document 3](#_Toc295979725)

[2. Overview 3](#_Toc295979726)

[2.1 Federated Authentication and Claims-Based Identity 3](#_Toc295979727)

[2.2 Project Goals 4](#_Toc295979728)

[3. Infrastructure 5](#_Toc295979729)

[4. Federated Authentication 6](#_Toc295979730)

[4.1 Essential Steps 6](#_Toc295979731)

[4.2 Active vs. Passive Authentication 7](#_Toc295979732)

[5. Logging in to Redbox.com 7](#_Toc295979733)

[5.1 Redbox Identity Provider Token Processing 10](#_Toc295979734)

[5.2 ACS Federated Provider Token Processing 10](#_Toc295979735)

[5.3 Redbox Relying Party Token Processing 10](#_Toc295979736)

[5.4 Claim Authentication Managers and Claim Transformations 11](#_Toc295979737)

[6. Redbox.com Integration 12](#_Toc295979738)

[6.1 Organization of Code 12](#_Toc295979739)

[6.2 Configuring Redbox.com as a Relying Party 13](#_Toc295979740)

[6.3 Building the Redbox Identity Provider 16](#_Toc295979741)

[6.4 Establishing a Windows Azure Access Control Service 26](#_Toc295979742)

[6.5 Enhancing Redbox.com to Support Federated Authentication 27](#_Toc295979743)

[7. Appendices 30](#_Toc295979744)

[7.1 Federated Authentication 30](#_Toc295979745)

[7.2 Windows Identity Foundation 30](#_Toc295979746)

[7.3 Identity Developer Training Kit 30](#_Toc295979747)

[7.4 Windows Azure Access Control Service 30](#_Toc295979748)

[7.5 Thinktecture Identity Server 31](#_Toc295979749)

[8. Change Log 32](#_Toc295979750)

# Introduction

## Project Description

The purpose of this project was to determine how Federated Authentication could be integrated into the Redbox.com website with minimal impact to existing code.

## Description and Scope of this Document

This document describes the steps that were taken to perform this integration. Some of these steps are essential while others only demonstrate a proof-of-concept with the intent that the reader implements them in a manner they deem appropriate. This document provides an overview of Federated Authentication, but only to the extent necessary to provide the proper context to support the relevant code changes. Many of the steps needed to perform this integration are rich in details and it is not practical to capture each one. Some are described at a conceptual level with references made to additional resources for the reader to consider. The appendix contains a list of all resources and supporting documents. The author also makes himself available to provide any additional information that may have been overlooked.

# Overview

## Federated Authentication and Claims-Based Identity

Federated Authentication is concerned with externalizing a user’s identity so that it may be shared among different domains. In practice, this is accomplished by building a Security Token Service, STS, that provides assertions about a supplied security principal. An STS may act as an Identity Provider, IP, which is used to authenticate credentials or a Federated Provider, FP, which is used for centralizing trust relationships and performing transformations on assertions. Regardless, these assertions are packaged into a Security Token and passed across a Trust Boundary to be consumed by a Relying Party, RP, in a way that protects the integrity of the assertions. The RP accepts these assertions as de facto because they were issued by an STS that it trusts. The assertions are made in the form of Claims which represent information about the identity proffered by the original security principal. Collectively, these claims form the basis of Claims-Based Identity. The OASIS consortium defines an industry standard protocol named WS-Federation that describes this interaction. Windows Identity Foundation, WIF, is a .NET API that implements the WS-Federation protocol.

Federated Authentication is better understood by analogy (see diagram below).

1. A patron enters a car rental facility to rent a vehicle, but the clerk first needs to verify the patron’s age.
2. The patron visits the Department of Motor Vehicles, DMV, to obtain a driver’s license and presents some other identifiable information such as a birth certificate.
3. The DMV issues the patron a driver’s license.
4. The patron returns to the store and presents the driver’s license. Since the clerk trusts the DMV, the date of birth listed on the driver’s license is assumed to be accurate and the rental can be made.



Figure : Analogy of Federated Authentication

By analogy, the patron is a user requesting access to a protected resource (the car rental). The store is a relying party that has previously established a trust relationship with an Identity Provider (the DMV). The user presents a security principal which contains some form of credentials (the birth certificate) to the IP. The IP validates the credentials and supplies a security token (the driver’s license). The security token contains a number of assertions about the user’s identity just as a driver’s license lists a subject’s name, address, date of birth, etc. The user presents the security token to the relying party where it is validated (by verifying the token’s digital signature) and grant’s access to the protected resource.

Federated Authentication has a number of advantages:

* User authentication is externalized which reduces overhead and creates a more scalable infrastructure.
* The process of authentication is opaque to the relying party – it only needs to trust the STS. This means that the STS can be maintained separately from the relying party and any number or combination of STS’s can be used to authenticate a user.
* Federated Authentication makes possible Single Sign On behavior.
* WS-Federation is an industry standard which means it can easily interoperate with different technologies.

## Project Goals

Integrating Federation Authentication into Redbox.com has a number of distinct goals:

1. Convert the application to make use of Federation Authentication and hence claims-based identity in place of Forms Authentication.
2. Provide backwards compatibility with the existing use of web accounts, roles, authorization rules, etc.
3. Provide backwards compatibility with the mobile applications which rely of services exposed by Redbox.com.
4. Make use of the Windows Azure Access Control Service to act as Federated Provider to delegate authentication to any number of different Identity Providers.
5. Demonstrate the use of different IPs to log into Redbox.com including Facebook and a third party digital partner.
6. Make use of the existing log in page on Redbox.com to authenticate users with their Redbox account.

# Infrastructure

Access Control Service, ACS, is a cloud-based SAS provided by Microsoft Azure which acts as a centrally trusted Federated Provider. The Redbox.com website, the future Digital Partner website and OpenAPI will be configured as Relying Parties and thus enter into a trust relationship with ACS. This permits security tokens (and the user identities they represent) to be exchanged across any of these domains. In turn, a new Redbox Identity Provider, a Digital Partner IP and Facebook will be configured within ACS. This allows users visiting Redox.com and the Digital Partner site or those accessing OpenAPI services to potentially authenticate with any of these providers and gain access to any of them. If future partners or other IPs are added they simply need to be configured within ACS without any changes to any Relying Party.



Figure : Infrastructure to Support Federated Authentication among Various Relying Parties

To facilitate this proof of concept, the Redbox Identity Provider code was integrated into the same codebase as the Redbox Relying Party (Redbox.com). Since IPs are responsible for authenticating users they also render the log in pages used for authentication. This centralization of code was motivated by the desire to keep the existing log in page intact and not duplicate code across different systems (and thus minimize future maintenance efforts). The choice of where this IP will eventually be located still needs to be determined, but the architectural goals of OpenAPI (such as support for OAuth) and the development goals of the Web Team are influencing factors. For purposes of this discussion bear in mind this centralization and reuse of code.

As an alternative solution, some open source Identity Providers exist for reuse and extension. One such offerimg is the Thinktecture Identity Server which is currently a beta release. The appendix describes the steps needed to set up and build the solution.

# Federated Authentication

## Essential Steps

To gain access to a protected resource using Federated Authentication a number of distinct token processing steps occur. Each step is concerned with passing a user’s identity across a Trust Boundary which is typically a different domain. This trust relationship is ensured by using a certificate to sign tokens and later verify their signatures. First, a Security Token is generated by an Identity Provider in response to user authentication. If the Relying Party directly trusts the Identity Provider, that token can be immediately consumed. In this design, however, the token is next passed to a Federated Provider to be exchanged for another Security Token before it is sent to the Relying Party. This step ensures that each RP only needs to trust a single STS.

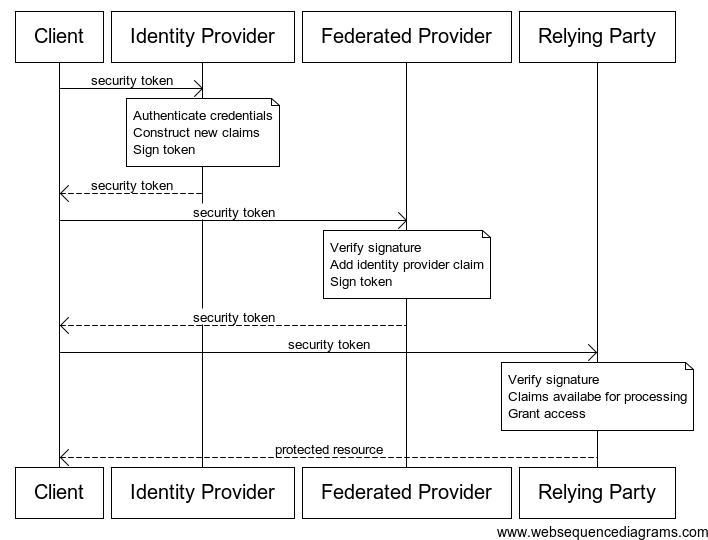


Figure : Essential Token Processing Sequence

## Active vs. Passive Authentication

The orchestration of these steps can be performed either actively or passively. Active authentication involves the use of a client such as .NET or Java to directly communicate with various STS’s to process security tokens. The typical use of active authentication is the scenario in which a client wishes to invoke a web service (such as one hosted on the Redbox RP). The client would construct a security token which contains a pair of *UserName* and *Password* claims. It would then pass this token to the Redbox IP which would authenticate the credentials and construct a new set of claims (one of which would be a *Name Identifier* claim which uniquely identifies the user). The client would then exchange this token with ACS for another one which includes the *Identity Provider* claim. This token could then be sent to the Redbox RP when invoking the web service. The RP would authenticate the request using the token and grant access. Note that WIF and WCF abstract these details from the developer. A WCF client would issue a request using a UserNameWSTrustBinding.

In contrast, passive authentication involves the use of a browser to communicate with various STS’s to acquire a security token. This scenario relies on browser redirects and JavaScript-induced POSTs to pass security tokens across domains. For example, if a user wishes to log into the Redbox RP, they are first redirected to ACS, then to the Redbox IP. After logging in, a series of POSTs occur to pass the token to ACS then back to the Redbox RP. These redirects and POSTs are transparent to the user and abstracted from the developer by the WIF API. The behavior is configured using the <Microsoft.IdentityModel> configuration section.

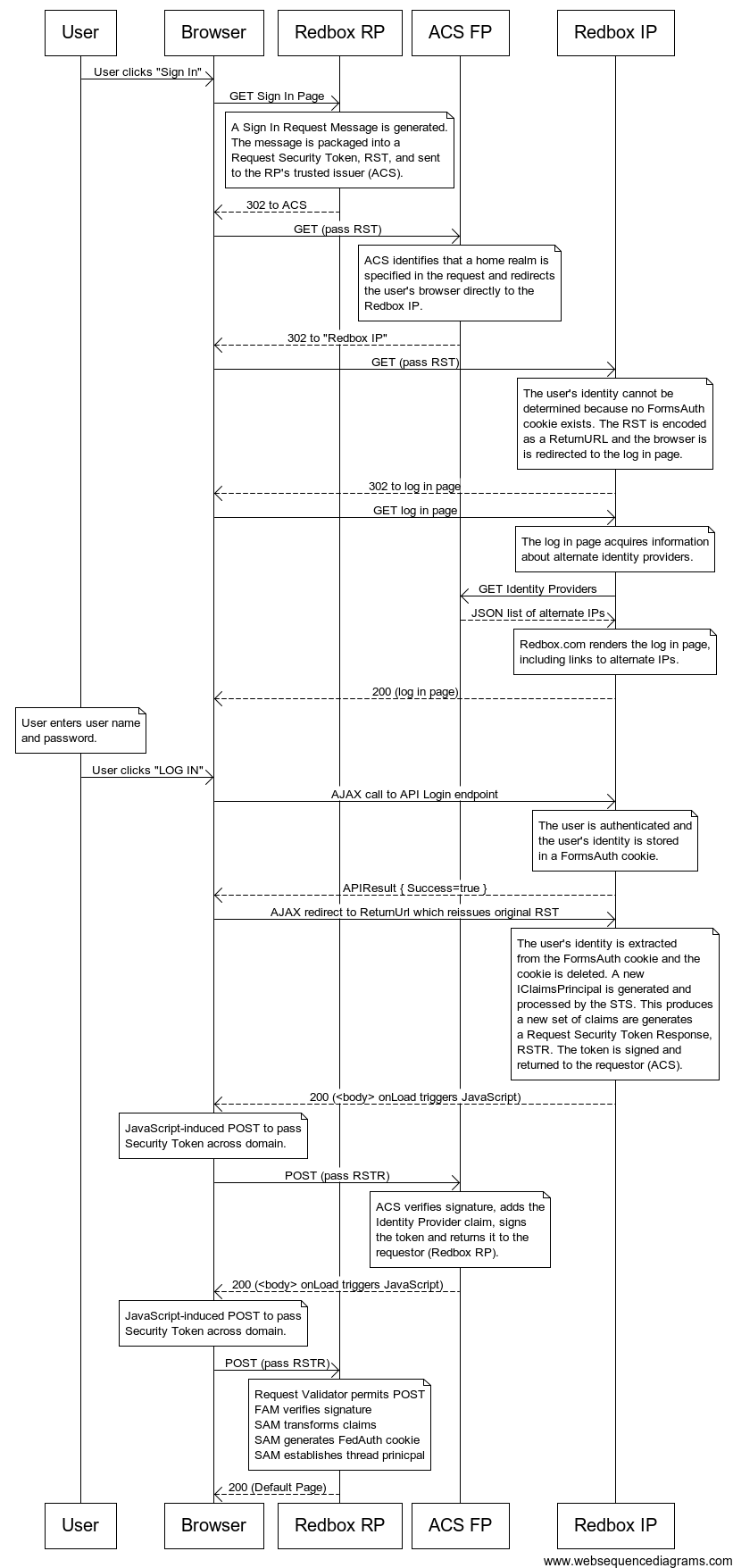
# Logging in to Redbox.com

This POC is primarily concerned with demonstrating how a user can log into Redbox.com using passive authentication with different Identity Providers. The diagram below depicts the claims processing that occurs with each step.



Figure : Redbox Claims Transformations

The remainder of this discussion is concerned with how Redbox.com can be adapted to the use of passive authentication. The following diagram illustrates the typical sequence that would occur when an anonymous user clicks “Sign In” to log into Redbox.com. Note that the user experience remains unchanged.



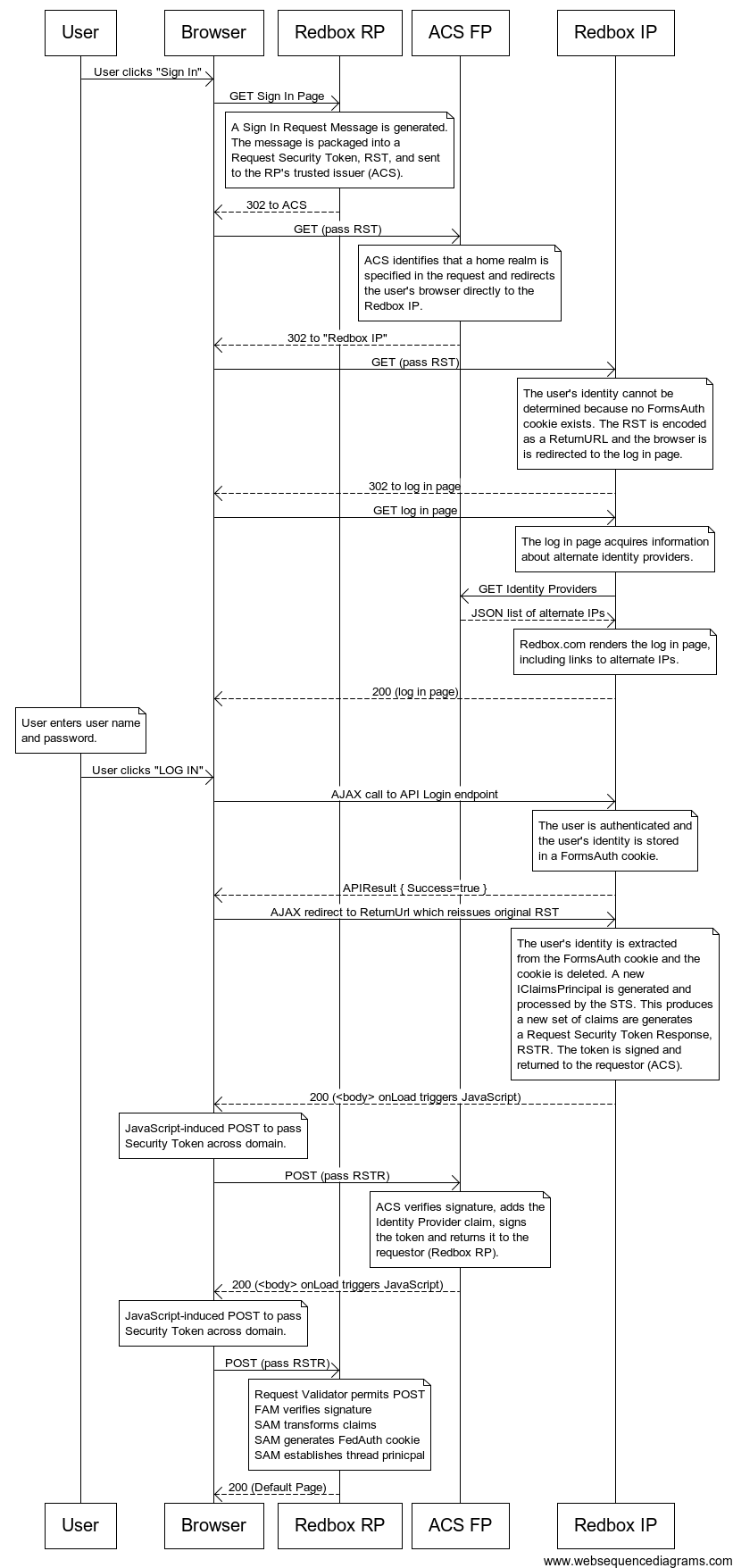


Figure : Passive Federated Authentication in Redbox.com

## Redbox Identity Provider Token Processing

First, a user must provide a security principal to an Identity Provider. In this scenario the IP is the Redbox IP, but in practice it could also be Facebook or a Digital Partner. That security principal will contain a pair of claims: a user name and password. In terms of claims-based identity, these claims represent the user’s identity. Alternatively, the user could present a certificate claim representing their identity. Support for certificate validation is not demonstrated in this POC and will not be part of the final architectural design as each user would require a unique certificate.

The IP validates the supplied user name and password and (if valid) constructs a new set of claims which include: a *Name Identifier* (a value known to the IP which uniquely identifies the user – in this case the web account id), a *Given Name* (a human-readable name identifying the user such as their first and last name) and *Email*. Note that the given name and email are derived from the name identifier. The choice of these three specific claims is central to the design of this system and at present is put forth as a requirement. The claims are then packaged into a security token, signed and returned to the user.

*Note that this design makes use of SAML 2.0 (Security Assertion Markup Language) tokens. Version 1.2 of the WS-Federation standard makes Name Identifiers a required claim for SAML 2.0 tokens. WS-Federation defines many other industry standard claim types which are identified by namespace-qualified string constants.*

## ACS Federated Provider Token Processing

Next, the user passes the token received from the IP to ACS. ACS verifies the signature of the token and determines that it originated with the Redbox IP. Since a trust relationship was previously established between the Redbox IP and ACS, the token is trusted and ACS accepts the assertions that it makes. ACS can be configured to perform simple claim transformations but none are needed for this design. Instead, it simply injects one additional claim, *Identity Provider*, which the relying party can use to identify the issuer of the original token. As in the previous step, the token is signed and returned to the user.

*Note that the Identity Provider claim is specific to Windows Azure and is not part of the WS-Federation standard. Extension of the standard with custom claim types is common. In fact, the Redbox.IdentityModel library defines new claim types that are specific to Redbox.*

## Redbox Relying Party Token Processing

### Request Validation

Finally, the user presents the token it received from ACS to the Redbox RP. Since the token is constructed using SAML, ASP.NET would normally reject the POST because it contains XML markup. ASP.NET 4.0 provides a new feature called a RequestValidator which allows for custom evaluation of the posted content to make that determination instead. A class library developed for this POC, Redbox.IdentityModel, contains a class named FederatedAuthenticationRequestValidator which determines if the posted content is a Security Token and permits the POST to proceed.

### The Federated Authentication Module

Since the website configuration has been modified to support WIF (and hence use Federated Authentication instead of Forms Authentication), two new HTTP Modules surface to process the incoming token. First, the WSFederatedAuthenticationModule, FAM, receives the token and verifies the token’s signature. Once satisfied that the token can be trusted it knows that the originating issuer (Redbox IP) can also be trusted and accepts all of the assertions made by the token. Technically, the IP is completely opaque to the RP since the token was first passed through ACS, but transitively the assertions are accepted. Next, the FAM packages the claims into an IClaimsPrincipal and passes it to the SessionAuthenticationModule, SAM.

*Note that the incoming principal is representative of the original principal issued by the user in the form of a user name and password. Since the user could’ve authenticated using Facebook (or any IP other than Redbox), this illustrates how ACS acts as an FP (or intermediary) to share security principals across different domains using the various trust relationships that have been established.*

### Session Authentication Module

Next, the SAM performs a final transformation of the supplied claims before the token is stored in a FedAuth cookie. The SAM then generates an IClaimsPrincipal and sets it to the current thread’s principal. Note that the *Role* claims are used support the IPrincipal.IsInRole() method. On subsequent requests, the SAM deserializes the FedAuth cookie to recover the security token and reestablish the thread’s principal. The SAM offers two extensibility points: Claims Authentication Managers and Session Security Token Handlers. Claims Authentication Managers perform claim transformations. They only perform these transformations once when the token is first posted to the RP and processed by the FAM. Session Security Token Handlers are responsible for serializing and deserializing security tokens to and from cookies. A custom Claims Authentication Manager was written for this POC, WebAccountClaimsAuthenticationManager, and is presented as a starting point for all claim transformations required by this design.

*A custom Session Security Token Handler was not written for this POC but may be considered since SAML markup is verbose and produces sizeable cookies.*

## Claim Authentication Managers and Claim Transformations

By default all claims are simply passed through by the SAM without any transformations. The design of this POC defines four claims for use by the Redbox RP: *Web Account ID*, *Given Name*, *Email*, and a collection of *Role* claims. The claims that are required by the Redbox RP include: *Identity Provider*, *Name Identifier*, *Given Name* and *Email.* The WebAccountClaimsAuthenticationManager transforms these claims in the following way:

1. The *Identity Provider* claim and *Name Identifier* claim are used to locate the user’s web account and generate a *Web Account ID* claim.
   1. If the Identity Provider is the Redbox IP then the Name Identifier is treated as a web account id.
   2. Otherwise, the Customer Profile API should be used to locate the user’s web account.   
        
      *This portion of code is not implemented and a hardcoded value is currently returned as a proof of concept. Also, consider that the user’s account can only be located if it has been previously provisioned within Customer Profile.   
        
      Depending on how Facebook integration will be achieved, it may be necessary to auto-provision a web account and associate it with the Facebook user’s account if one doesn’t already exist. It is important to make this association between Redbox’s web account id and Facebook’s name identifier instead of the user’s email address. Otherwise someone could potentially gain access to the user’s email account, set up a Facebook account using that email address and associate it with an existing Redbox account and hence gain access to the user’s Redbox account. The claims provided by Facebook don’t provide all the necessary information to provision a web account; it may only be possible to prepopulate a portion of an account registration form. The requirements for this use case and the implicit customer experience hasn’t been determined and requires further analysis.*
   3. If the user’s web account cannot be located, an anonymous claims identity is generated.
2. If the user has been identified the web account is used to generate a set of *Role* claims.
3. If the *Given Name* claim is available (and the user has been identified) this claim is passed through.
4. If the *Email* claim is available (and the user has been identified) this claim is also passed through.

# Redbox.com Integration

## Organization of Code

The reusable portions of code that were developed for this POC were refactored into a separate class library named Redbox.IdentityModel. It is stored in Vault’s Innovation repository at *$/Redbox Federated Authentication/Redbox.IdentityModel.* It contains the following classes:

**{ } Redbox.IdentityModel**

ClaimTypes   
*Used to define constants that represent Redbox-specific claims such as WebAccountID and PartnerID*.

RedboxSecurityTokenService   
*The class used to host the core Redbox IP functionality.*

RedboxSecurityTokenServiceConfiguration  
*The class used to configure and instantiate an instance of a RedboxSecurityTokenService class.*

RedboxSecurityTokenServiceFederationMetadataGenerator  
*The class used to generate the metadata for the RedboxSecurityTokenService.*

**{ } Redbox.IdentityModel.Web**

FederatedAuthenticationRequestValidator  
*The class used to permit the HTTP POSTs of SAML tokens to an ASP.NET web application.*

The changes made to the Redbox.com website are located in the Innovation Repository at *$/Redbox Federated Authentication/Web Systems/Redbox.Web.* The starting point for this code was release 2.8.0.x of Redbox.Web.

**{ } redbox.Web.Controllers**

FedAuthController  
*The MVC controller used to process all WS-Federation related requests.*

**{ } redbox.Web.IdentityModel**

Certificates  
*A static class holding references to the certificates used for signing and encrypting security tokens.*

CertificateUtility  
*A helper class used to load X.509 certificates from a certificate store.*

IdentityProvider  
*A domain object used for deserializing data from ACS that represents alternate home realms users can use for authentication.*

OriginalIssuers  
*A static class holding constants that can be used to identify the original issuers of a security token.*

WebAccountClaimsAuthenticationManager  
*The class responsible for performing claims transformations on the Redbox RP prior to serializing s security token into a FedAuth cookie.*

**{ } redbox.Web.Mvc**

DynamicRouteConstraint  
*An MVC route constraint that allows for flexible rules evaluation which is used to route API requests from mobile devices.*

## Configuring Redbox.com as a Relying Party

The first step in configuring a website as a relying party is to install WIF. Refer to the appendix for download information.

Once installed, a new context menu option will appear in Visual Studio for websites and web applications. If the user right-clicks on the project and selects “Add STS Reference…” the FedUtil wizard will appear to guide the user through a series of steps that make changes to the project’s web.config file. It is recommended that the user does not utilize this wizard. If the proper choices aren’t made, subsequent attempts will clutter the configuration and potentially make it unusable and more difficult to understand. Instead, the exercise of modifying the web.config file directly will produce a correct configuration and a deeper understanding of how to maintain it. Refer to the appendix for further information on the FedUtil wizard.

### Manually Configuring WIF

1. Add a Reference to the Microsoft.IdentityModel config section and assembly

<configSections>

<section name="microsoft.identityModel" type="Microsoft.IdentityModel.Configuration.MicrosoftIdentityModelSection, Microsoft.IdentityModel, Version=3.5.0.0, Culture=neutral, PublicKeyToken=31bf3856ad364e35" />

</configSections>

…

<system.web>

<compilation debug="true" targetFramework="4.0">

<assemblies>

<add assembly="Microsoft.IdentityModel, Version=3.5.0.0, Culture=neutral, PublicKeyToken=31BF3856AD364E35" />

</assemblies>

</compilation>

…

</system.web>

1. Register the FAM and SAM modules

<system.webServer>

…

<add name="WSFederationAuthenticationModule" type="Microsoft.IdentityModel.Web.WSFederationAuthenticationModule, Microsoft.IdentityModel, Version=3.5.0.0, Culture=neutral, PublicKeyToken=31bf3856ad364e35" preCondition="managedHandler" />

<add name="SessionAuthenticationModule" type="Microsoft.IdentityModel.Web.SessionAuthenticationModule, Microsoft.IdentityModel, Version=3.5.0.0, Culture=neutral, PublicKeyToken=31bf3856ad364e35" preCondition="managedHandler" />

</modules>

…

</system.webServer>

1. Remove support for Forms authentication

<authentication mode="None" />

1. Register the new custom request validator to support POSTs containing SAML tokens

<httpRuntime requestValidationMode="2.0" requestValidationType="Redbox.IdentityModel.Web.FederatedAuthenticationRequestValidator, Redbox.IdentityModel" />

1. Add the WIF configuration. In this POC the WIF configuration was placed in an isolated file

<microsoft.identityModel configSource="IdentityModel\wif.config" />

wif.config:

<microsoft.identityModel>

<service>

<audienceUris>

<add value="https://gstrasdin-dt.corp.coinstar.com/" />

</audienceUris>

<federatedAuthentication>

<wsFederation

passiveRedirectEnabled="true"

issuer="https://rb-dev.accesscontrol.windows.net/v2/wsfederation"

realm="https://gstrasdin-dt.corp.coinstar.com/"

homeRealm="Redbox"

requireHttps="true" />

<cookieHandler requireSsl="false" />

</federatedAuthentication>

<claimsAuthenticationManager type="redbox.Web.IdentityModel.WebAccountClaimsAuthenticationManager, redbox.Web"/>

<applicationService>

<claimTypeRequired>

<claimType type="http://schemas.microsoft.com/accesscontrolservice/2010/07/claims/identityprovider" optional="false" />

<claimType type="http://schemas.xmlsoap.org/ws/2005/05/identity/claims/nameidentifier" optional="false" />

<claimType type="http://schemas.xmlsoap.org/ws/2005/05/identity/claims/givenname" optional="true" />

<claimType type="http://schemas.xmlsoap.org/ws/2005/05/identity/claims/emailaddress" optional="true"/>

</claimTypeRequired>

</applicationService>

<issuerNameRegistry type="Microsoft.IdentityModel.Tokens.ConfigurationBasedIssuerNameRegistry, Microsoft.IdentityModel, Version=3.5.0.0, Culture=neutral, PublicKeyToken=31bf3856ad364e35">

<trustedIssuers>

<add thumbprint="E23CF648B1BDEDCD6EE75F1A67DCEC66781B9C9F" name="https://rb-dev.accesscontrol.windows.net/" />

</trustedIssuers>

</issuerNameRegistry>

<certificateValidation certificateValidationMode="None" />

</service>

</microsoft.identityModel>

### Discussion of WIF Configuration

**<audienceUris/>** - These are the URIs that represent the Relying Party. In practice, only one is needed. Note that the fully qualified URI is required. Since Federated Authentication is performed over SSL, a self-signed certificate was created to secure the local instance of IIS. ACS will only issue a token to a registered relying party and the URI of the relying party must match the URI of the SSL certificate.

**<wsFederation/>** - This element configures the FAM module.

**passiveRedirectEnabled**  - Must be set to true to enable authentication using a browser.

**issuer** - The WS-Federation issue endpoint exposed by ACS.

**realm** - The relying party’s realm (same as audience URI).

**homeRealm** - If this attribute is not specified, ACS will display a home realm discovery page where the user selects an identity provider with which to authenticate. Specifying a value here allows that page to be skipped by preselecting the home realm. In this POC the Redbox.com log in page displays a list of alternate Identity Providers and, in effect, function as both a log in and home realm discovery page without losing the look and feel of the site (discussed later). Note that this value must match the Redbox IP’s entity ID (discussed later).

**requiresHttps** - Must be set to true to enable interoperability with ACS.

**<cookieHandler/>** - Configures the Session Security Token Handler.

**requireSsl** - Set to false to allow authenticated browsing of unsecured pages after logging in.

**<claimsAuthenticationManager/>** - Specifies the custom Claims Authentication Manager described earlier.

**<claimTypeRequired/>** - This section is completely optional and does not influence run-time behavior. It is used by the FedUtil to generate a FederationMetadata document for a Relying Party. This document can be used to configure ACS but is not required.

**<issuerNameRegistry/>** - Specifies the registry used to verify token signatures. The type specified is used to verify signatures using certificate thumbprints.

**<trustedIssuers/>** - Lists the various STS’s that this Relying Party trusts. Since ACS is functioning as a Federated Provider in this design and all Identity Providers are opaque to the Redbox RP, ACS is the only trusted issuer that needs to be listed.

**thumbprint** - The issuer’s thumbprint. In ACS this is listed under the “Token Signing Certificates” section of the Relying Party configuration. Click on the name of the certificate to view its details.

**name** - The issuer’s name. This value must match certificate’s subject. Again, click on the name of the certificate to view its details.

**certificateValidation** - Controls how the certificate that is used to verify the token signatures is validated.

**certificateValidationMode** - Set to “None” to indicate that the certificate is not located in a store and does not need to be validated.

### IIS Configuration

Before the Redbox RP can verify a security token’s signature it must be able to perform cryptographic operations within the context of the current security principal. This requires that the application pool be permitted to impersonate the user issuing the request. To enable this within IIS 7, the *Load User Profile* setting needs to be set to *true*. Do this by modifying the application pool’s advanced settings.

Application Pool → Advanced Settings → Process Model → Load User Profile → True

## Building the Redbox Identity Provider

### Building a Security Token Service Using the WIF API

A Security Token Service is responsible for issuing security tokens. It is at the heart of all Identity Providers and Federated Providers. Building an STS using the WIF API is achieved by inheriting from the SecurityTokenService class and mapping a web service endpoint to the Issue() method. This method invokes several other methods which compose the Claims Issuance Pipeline (see appendix). The pipeline invokes calls to two abstract methods, GetScope() and GetOutputClaimsIdentity() which must be implemented in your subclass. GetScope() is responsible for generating a descriptor that controls the behavior of the pipeline. GetOutputClaimsIdentity() is where the actual claim transformations occur.

The Redbox.IdentityModel library includes a RedboxSecurityTokenService class that provides a default implementation that was used to build the Redbox IP. The GetScope() implementation identifies the signing and encrypting certificates that are used by the pipeline. WS-Federation requires that all token be signed, so an X.509 certificate is required to instantiate the class. Token encryption is optional. The GetOutputClaimsIdentity() implementation constructs the three claims mentioned earlier: *Name Identifier*, *Name* and *Email.*

The Redbox.IdentityModel library also defines a claim type that is unique to Redbox: *Web Account ID*. The current implementation of GetOutputClaimsIdentity() requires this claim to be present in the incoming claims principal. It does not perform any user name and password validation; it simply uses the web account id to locate the referenced web account using the web account service. To support active authentication this implementation will need to be extended to generate output claims using either a *Web Account ID* claim or a pair of *UserName* and *Password* claims. Since the existing Login API endpoint in Redbox.com already authenticates a user and returns a web account id, it was only necessary to demonstrate an STS behavior driven purely by a *Web Account ID* claim for this POC.

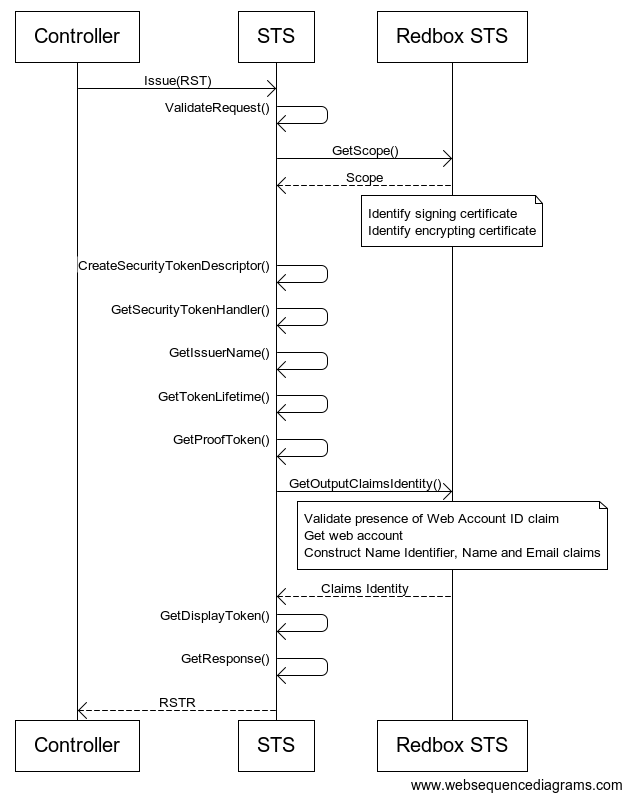


Figure : Implementation of RedboxSecurityTokenService within Claims Issuance Pipeline

### Creating a Self-Signed Certificate

This POC makes use of a self-signed certificate for both signing security tokens and providing SSL encryption. A new certificate can be generated in the Server Certificates panel in IIS. The certificate will be placed in the Personal Certificates store on the Local Computer.

### Granting the ASP.NET Application Pool Access to the Certificate

Before the Redbox IP can sign a security token the application pool must have access to the certificate. This requires two steps. First, (as described in section 6.2.3) the *Load User Profile* in IIS must be set to *true*. Do this by modifying the application pool’s advanced settings.

Application Pool → Advanced Settings → Process Model → Load User Profile → True

Next, the application pool must be granted read access to the certificate. Microsoft provides a tool named FindPrivateKey which can be used to locate the physical file location of the certificate (see appendix). Once located, modify the file’s permissions to grant the application pool access. To determine the proper user name use the syntax “IIS AppPool\*[application pool name]*” which is the IIS 7 nomenclature for representing Application Pool identities (see appendix).

### Using the Certificate to Sign the Security Token

To instantiate the RedboxSecurityTokenService class an instance of an X509Certificate is needed. A helper class, CertificateUtility, was written for this POC which can be used to acquire the certificate using the store name, location and subject. If the desired certificate was created using IIS as described above, the store name will be “My” and the location will be “LocalMachine”. The subject can be determined by viewing the certificate properties in IIS or the Microsoft Management Console Certificate Snap-In. The syntax for the subject name is “CN=*[subject]*”.

### The FedAuthController

A new MVC controller was created for the Redbox.com site named FedAuthController. It encapsulates most of the new logic introduced for this POC. Its primary purpose is to orchestrate passive WS-Federation token requests, but it is also used to generate the Federation Metadata document that ACS needs when configuring a new Identity Provider. The Redbox IP will also need to support active authentication for OpenAPI. The FedAuthController is a logical place to add that support.

### Configuring MVC Routes

Three new MVC routes are used to support the FedAuthController. The first supports WS-Federation requests.

// Route all WS-Federation requests to the FedAuthController

routes.MapRoute(

"wsfederation",

"wsfed/{action}",

new { controller = "FedAuth" }

);

The second route is used to generate the Federation Metadata document. This route is purely optional, but was used to provide a URL consistent with industry standards.

// Provide a user-friendly route for generating sts federation metadata

routes.MapRoute(

"FederationMetadata",

"FederationMetadata/2007-06/FederationMetadata.xml",

new { controller = "FedAuth", action = "GenerateMetadata" }

);

Note that this route required an additional, supporting web.config change. An existing HTTP handler was registered to pass requests for sitemap.xml to the ASP.NET ISAPI module.

<handlers>

<add name="xml" path="sitemap.xml" verb="\*" type=…

</handlers>

This was changed to process all xml requests, but a more granular configuration may be needed to properly serve up 404 responses.

<handlers>

<add name="xml" path="\*.xml" verb="\*" type=…

</handlers>

Finally, a third route was configured to intercept API requests from mobile devices. This is described later.

### Extending the Existing Login Sequence

The FedAuthController.SignIn() method is used to intercept requests that were originally destined for the login page and kick off a passive authentication sequence. It generates a SignInRequestMessage and redirects the browser to the Redbox IP for processing (refer to the sequence diagram in section 5). Before doing so; however, it captures the original ReturnUrl that was specified in the request and packages it into the SignInRequestMessage. When the sequence is complete this instructs the FAM to redirect the browser back to the original page the user was visiting when they clicked “Sign In” (or the protected page they were requesting). This produces a log in behavior consistent with the existing user experience.

To properly intercept these requests, the Welcome.ascx file was updated to instruct the AJAX client where to send the user in response to the sign in click event. The code was changed from…

string loginUrl = Settings.ResolveServerUrlByKey("Register", Settings.UrlType.Secure);

…to:

string loginUrl = Settings.ResolveServerUrl("/wsfed/signin", Settings.UrlType.Secure);

Since the Redbox IP is hosted in the same codebase as the Redbox RP, the SignInRequestMessage is received by the FedAuthController.Issue() method. The URL is examined to determine if the message is requesting a WS-Federation Sign In or Sign Out operation.

If a Sign In request was issued, a FormsAuthenticationTicket is extracted from the request’s cookies collection. If the ticket contains a useable security principal it is fed into the RedboxSecurityTokenService to generate a response and continue the passive WS-Federation token issuance process. If a ticket does not exist, the user is redirected to the log in page to acquire the principal. A return URL is first constructed which instructs the AJAX client to reissue the original Sign In request if authentication succeeds. Constructing the return URL uncovered an issue during this POC.

The Website.GetReturnUrl() helper method uses the Uri.ToString() method to construct the return URL:

public static string GetReturnUrl()

{

…

string returnFullPath = HttpContext.Current.Request.Url.ToString();

…

returnUrl = "?ReturnUrl=" + HttpUtility.UrlEncode(returnFullPath);

…

}

The UriToString() method encodes some of the URI’s components. When the returnFullPath string is later encoded as a query parameter, any components in the original URI that were originally encoded become doubly encoded. Instead, the Uri.AbsoluteUri property should be used to preserve the integrity of the original URL.

public static string GetReturnUrl()

{

…

string returnFullPath = HttpContext.Current.Request.Url.AbsoluteUri;

…

}

A regression should be performed to confirm that other uses of this helper method are not impacted by this change. If the original URL does not contain any encoded components, they should not be affected.

An important point to consider is how Forms Authentication is used in this POC. Since the website was enhanced to support Federated Authentication, Forms Authentication is no longer needed. Federated Authentication provides all the features of Forms Authentication plus some additional ones which will become apparent later in this document. However, since the existing authentication mechanism (provided by the API Login endpoint) is based on Forms Authentication, it has been repurposed to support passive WS-Federation token requests. In other words, the FormsAuth cookie that is generated when a user logs in is no longer used to authenticate the user with subsequent requests; it is used purely as a mechanism to deliver a security principal back into the WS-Federation token issuance process. Once the FormsAuthenticationTicket is recovered from the cookie, the cookie is destroyed. This has an important advantage: the existing log in page and API login endpoint do not require any changes.

It is recommended, however, that the CreateTicket() helper method used to construct the FormsAuth cookie be updated. The existing code generates a ticket, packages roles into the payload and establishes the thread’s security principal.

public static void CreateTicket(WebAccountDomainObjects.WebAccount webAccount, bool persist)

{

…

FormsAuthenticationTicket ticket = new FormsAuthenticationTicket(1, webAccount.WebAccountID.ToString(),

DateTime.Now, DateTime.Now.AddMinutes(timeout), persist, webAccount.Roles.JoinToString(","),

FormsAuthentication.FormsCookiePath);

HttpCookie cookie = new HttpCookie(FormsAuthentication.FormsCookieName, FormsAuthentication.Encrypt(ticket));

…

MvcApplication.CurrentContext.Response.Cookies.Add(cookie);

FormsIdentity formsIdentity = new FormsIdentity(ticket);

MvcApplication.CurrentContext.User = new GenericPrincipal(formsIdentity, webAccount.Roles.ToArray());

}

Roles no longer need to be persisted in the cookie since authorization will automatically be handled by the SAM (The claims persisted in a FedAuth cookie will contain *Role* claims). Also, it is no longer necessary to establish a security principal in this helper method since the act of logging in will always result in a browser redirect to ACS to continue the passive authentication sequence and upon return to Redbox.com the SAM will automatically generate the thread’s security principal. Note the following changes.

public static void CreateTicket(WebAccountDomainObjects.WebAccount webAccount, bool persist)

{

…

FormsAuthenticationTicket ticket = new FormsAuthenticationTicket(1, webAccount.WebAccountID.ToString(),

DateTime.Now, DateTime.Now.AddMinutes(timeout), persist, String.Empty, FormsAuthentication.FormsCookiePath);

HttpCookie cookie = new HttpCookie(FormsAuthentication.FormsCookieName, FormsAuthentication.Encrypt(ticket));

…

MvcApplication.CurrentContext.Response.Cookies.Add(cookie);

// FormsIdentity formsIdentity = new FormsIdentity(ticket); // deprecated

// MvcApplication.CurrentContext.User = new GenericPrincipal(formsIdentity, webAccount.Roles.ToArray()); // deprecated

}

### Supporting Alternate Home Realms

By default, ACS will display a home realm discovery page that allows the user to choose an Identity Provider with which to authenticate. Developers have very little control over the appearance of this page.

The home realm can be preselected through configuration. This allows the ACS to skip over this page and send the user directly to the Redbox IP log in page. To gain access to the log in pages provided by the other identity providers, ACS provides an endpoint that returns a JSON object array listing information about each provider. This information can be used to augment the appearance of the Redbox IP log in page to provide links to the other identity providers.

string acs = "https://rb-dev.accesscontrol.windows.net:443";

string realm = Settings.ResolveServerUrl("/", true);

string replyTo = realm;

string context = String.Empty;

string requestId = String.Empty;

decimal version = 1.0m;

string callback = String.Empty;

string identityProvidersUrl =

String.Format("{0}/v2/metadata/IdentityProviders.js?protocol=wsfederation&realm={1}&replyTo={2}&context={3}&request\_id={4}&version={5}&callback={6}",

acs,

HttpUtility.UrlEncode(realm),

HttpUtility.UrlEncode(replyTo),

HttpUtility.UrlEncode(context),

HttpUtility.UrlEncode(requestId), version.ToString("F1"),

HttpUtility.UrlEncode(callback)

);

HttpWebRequest request = (HttpWebRequest)WebRequest.Create(identityProvidersUrl);

HttpWebResponse response = (HttpWebResponse)request.GetResponse();

using (Stream stream = response.GetResponseStream())

{

DataContractJsonSerializer serializer = new DataContractJsonSerializer(typeof(IdentityProvider[]));

IdentityProvider[] identityProviders = serializer.ReadObject(stream) as IdentityProvider[];

if (identityProviders != null)

{

IEnumerable<string> divs = (

from ip in identityProviders

where !ip.Name.Contains("redbox")

select String.Format("<div><a href=\"{0}\"><img src=\"{1}\" alt=\"{2}\" border=\"0\"/></a></div>", ip.LoginUrl,

ip.ImageUrl, ip.Name)

);

return String.Join("\r\n", divs);

}

}

This call is being made from the server with every request for the log in page. For scalability reasons the call could be made from the browser so that changes within ACS are immediately recognized, or the call could be made from the server during application start up and cached during the application’s lifetime. This detail is left to the Web Team’s discretion.

Another point to consider from the earlier code example is that the realm always represents the Redbox IP. This may not be appropriate in all scenarios. Consider what would happen if a user on a digital partner site wanted to log in using their Redbox credentials. When they reach the Redbox log in page they would be presented with links that allow them to log into Redbox.com instead. To build these links properly, the request made to ACS to get the list of identity providers needs to be constructed using the realm of the digital partner site instead. This means the FedAuthController needs to pass additional contextual information to the log in page so the proper realm can be determined. When the RST is delivered to the Issue() method the realm can be identified. So, in addition to constructing a ReturnUrl query string parameter, another one could pass the realm.

### Generating Federation Metadata

Before configuring ACS to use an Identity Provider, a Federation Metadata document is needed which describes (among other things) the claim types offered, the passive and active STS endpoints and the certificate used for token validation. A full discussion of this metadata is beyond the scope of this document but there are three important things to consider.

First, the entity id declared in the document identifies the name of the IP’s home realm. If the RP is configured to preselect the home realm (as described above) these strings must match.

Next, the claim types that are offered by the metadata are what ACS uses to configure claim transformation rules. If ACS is configured to pass through all claims this is not an issue, but if the design calls for specific transforms the metadata document must provide the correct types to drive the UI in ACS that is used to configure the rules.

Finally, the WS-Federation standard requires that the metadata document list both passive and active endpoints. The passive endpoint, “/wsfed/issue”, is mapped using the MVC route to the Issue() method on the FedAuthController. This POC does not provide an active endpoint, but in practice one will be needed to support OpenAPI and eventually mobile. Some clients will need to issue a token request to the Redbox IP and exchange the resulting security token with ACS to secure web service calls to OpenAPI or the Redbox API. Since this design is not yet complete, this POC was focused purely on passive authentication and a ficticous active endpoint was supplied in the metadata document to facilitate proper metadata generation.

The Redbox.IdentityModel library contains a metadata generator, RedboxSecurityTokenServiceFederationMetadataGenerator, which is closely tied to the implementation of the RedboxSecurityTokenService class. If no changes are needed to this implementation, the generator can be used as-is. See the FedAuthController.GenerateMetadata() method to see how the generator is used.

### Compatibility Issues with Mobile Applications

Supporting Federated Authentication within the mobile applications presents a number of challenges. First, mobile is well suited to perform active authentication, but this would require that all mobile applications be updated to orchestrate WS-Federation calls. Additionally it would require that some API endpoints in Redbox.com be updated to support an authentication mechanism that relies on authentication headers in place of FedAuth cookies. This change would break all the calls made by the AJAX client. Additionally, the API Login endpoint performs other steps such as creating a Pluck cookie and saving personalization data. If changes to the mobile applications are not possible, then the existing API Login endpoint needs to perform the active authentication on its behalf in order to generate the FedAuth cookie to support authentication with future requests, but this would increase code complexity and load on the web server. Clearly, there is no clear cut solution and a migration path is needed to eventually support active authentication.

The first step in that migration would be to provide backwards compatibility so that no changes are needed are needed in mobile. These changes are demonstrated in this POC.

The next step in the migration would be to break apart the different responsibilities of the API Login endpoint. In other words, if the mobile application authenticates using the active Redbox IP endpoint it needs a way to obtain the Pluck cookie. This can be achieved by moving the Pluck cookie generation to a WSFederationAuthenticationModule.SignedIn event handler. In this way the Pluck cookie would be generated during both active and passive authentication. The active authentication endpoint could then be implemented and the mobile applications updated.

### Providing Backwards-compatibility with Mobile Applications

To provide this compatibility a new method was added to the FedAuthController, but MVC still needs a way to route API Login requests to a different endpoint if the request originated with a mobile application. This was achieved by developing a custom route constraint. The DynamicRouteConstraint class permits MVC routes to be declared in a way that provides flexible in-line rule evaluation.

// Map login API calls from mobile devices to a mobile-specific login action

routes.MapRoute(

"mobileApiHandler",

"api/Account/Login",

new { controller = "FedAuth", action = "MobileLogin" },

new { constraint = new DynamicRouteConstraint(r => r.UserAgent.Contains("Chrome")) }

);

The DynamicRouteConstraint class is constructed using a function of type Func<HttpRequestBase, bool>. This allows the route to be evaluated using the full Request object and not just the URL. In the example above, requests are routed to the FedAuthController.MobileLogin() method if the request was issued from a Chrome browser. Obviously, a different rule is needed to support mobile, but this was done for testing purposes.

Once the request arrives at the MobileLogin() method an active authentication gets kicked off. A SignInRequestMessage is sent to the Redbox IP using a UserNameWSTrustBinding. The response includes a security token which is passed to ACS using an IssuedTokenWSTrustBinding. The token that is returned is then POSTed to the Redbox RP to obtain the FedAuth cookie. That cookie is then copied over to the current HTTP response so that it can be fed back to the mobile device. This solution is intended to illustrate how backwards compatibility can be achieved and to demonstrate the steps the mobile applications will eventually need to perform. Further examination and testing of this code is warranted.

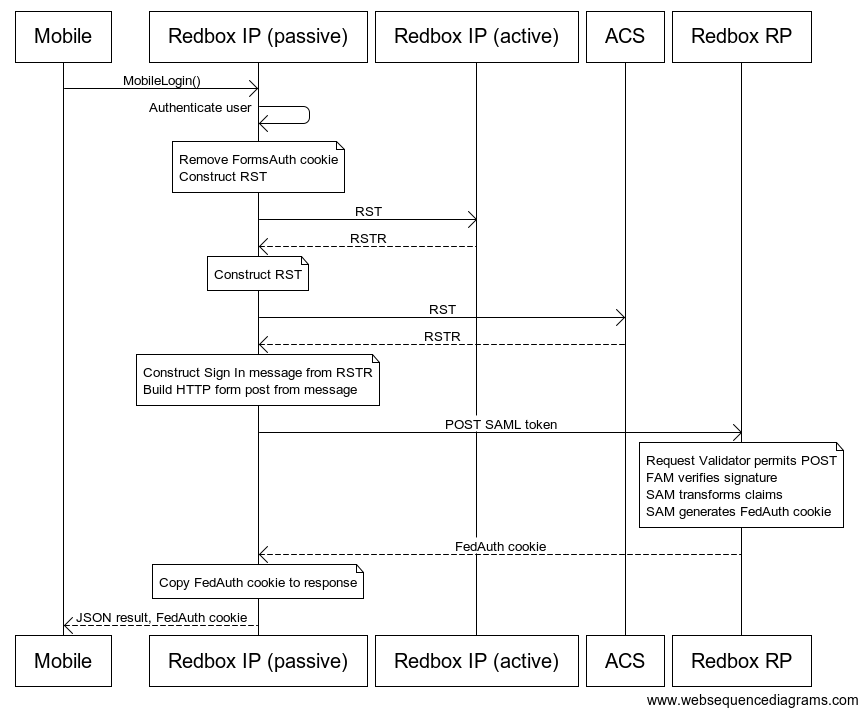


Figure : Active Authentication on Behalf of Mobile to Maintain Backward Compatibility

## Establishing a Windows Azure Access Control Service

A full discussion of ACS is out of scope for this document. A few points are highlighted below. For further information refer to the appendix. Windows Azure provides a production environment at <http://windows.azure.com> as well as a development environment at <http://portal.appfabriclabs.com>. A development environment was originally provisioned for this POC, but a production environment was eventually needed because of feature inconsistencies with regard to OAuth support.

A namespace must first be created before ACS can be used. Relying Parties, Identity Providers and claim transformation rules are all configured through the administration portal associated with the namespace. A number of preexisting Identity Providers exist including Facebook, Windows Live, Google and Yahoo. Configuring a Facebook IP requires that an application first be configured within Facebook to obtain an application id and secret. Configuring a custom Identity Provider requires a Federation Metadata document. In practice, ACS was unable to pull the document from the URL because the site was secured with a self-signed certificate. Instead the XML was saved to a file and uploaded directly. This process only worked, however, when the XML was saved using Notepad.

## Enhancing Redbox.com to Support Federated Authentication

With the changes in place to support the Redbox IP very few changes are actually needed to Redbox.com (the Redbox RP) to support Federated Authentication.

### Accessing a Web Account

The most important change is to the Website.AuthWebAccountID property getter which currently relies on the IIdentity.Name to be populated with the user’s web account id. Since, that information is now packaged as a *Web Account ID* claim, it needs to be resolved in a different manner.

int webAccountID;

IClaimsIdentity identity = MvcApplication.CurrentContext.User.Identity as IClaimsIdentity;

if (identity != null)

{

string webAccountIdString = (

from claim in identity.Claims

where claim.ClaimType.Equals(ClaimTypes.WebAccountID, StringComparison.OrdinalIgnoreCase)

select claim.Value

).FirstOrDefault();

if (!String.IsNullOrWhiteSpace(webAccountIdString) && Int32.TryParse(webAccountIdString, out webAccountID))

{

return webAccountID;

}

}

Alternatively, the WebAccountClaimsAuthenticationManager could be changed to package the web account id as a *Name* claim instead of a *Web Account ID* claim. The SAM uses the *Name* claim to set the IIdentity.Name property which would fully support the current implementation of Website.AuthWebAccountID. But, the *Web Account ID* claim was used instead to illustrate the use of claims and how the existing code can be modified to become claims aware. The final approach is left to the Web Team’s discretion.

### Making Redbox.com Claims-Aware

As a general note, using claims-based identity doesn’t add any value if the claims associated with the identity aren’t in some way utilized. The *Web Account ID* claim is used to identify the user, but including additional claims such as *Given Name* and *Email* are unnecessary if that same information is available in a different form. The choice of persisting user attributes in a cookie (represented as claims) or acquiring them in elsewhere for each request is a design decision to be considered. The author doesn’t make any recommendations other than discouraging the use of a mixed model.

### Restricting Access

The ViewUserControl which serves as the superclass to YourAccount.ascx control contains a method named EnsureLogin() which forces a redirect to the login page if the user is not authenticated. Since Federation Authentication is being used instead of Forms Authentication, this redirect logic requires a change. The original code…

if (!HttpContext.Current.Request.IsAuthenticated)

{

System.Web.Security.FormsAuthentication.RedirectToLoginPage();

HttpContext.Current.Response.End();

}

… was changed to:

if (!HttpContext.Current.Request.IsAuthenticated)

{

FederatedAuthentication.WSFederationAuthenticationModule.RedirectToIdentityProvider(null, String.Empty, true);

}

Note that RedirectToIdentityProvider() internally calls HttpApplication.CompleteRequest() which ends the current request.

### Signing Out

The helper method used to sign out of the website, Website.SignOut(), is used to delete cookies and reset the current thread’s security principal.

public static void SignOut()

{

FormsAuthentication.SignOut();

CreatePluckCookie(false, true);

MvcApplication.CurrentContext.User = null;

}

This method was modified to remove the FedAuth cookie in place of the FormsAuth cookie.

public static void SignOut()

{

FederatedAuthentication.SessionAuthenticationModule.SignOut();

CreatePluckCookie(false, true);

MvcApplication.CurrentContext.User = null;

}

Note that the earlier discussion regarding separating the responsibilities of logging in and managing other resources like pluck cookies applies here as well. It is recommended that the CreatePluckCookie(false, true); method call be moved to a SessionAuthenticationModule.OnSignedOut event handler.

Another point to consider is Single Sign Out behavior. The FedAuthController.Issue() method also processes Sign Out requests. But to sign out of multiple domains, The WSFederationAuthenticationModule contains a method called GetFederationPassiveSignOutUrl(). This URL would need to be requested when the user clicks on “Sign Out” if Single Sign Out behavior is desired. This will kick off a sequence of steps that sends Sign Out messages to each domain where the user is currently authenticated.

### Deprecated Code

Finally, the Website.AssignRoles() method was used to establish the thread’s security principal using the roles extracted from the FormsAuth cookie.

public static void AssignRoles()

{

if (IsAuthenticated)

{

FormsIdentity formsIdentity = (FormsIdentity)HttpContext.Current.User.Identity;

FormsAuthenticationTicket ticket = formsIdentity.Ticket;

if (ticket.UserData.Length != 0)

{

string[] userData = ticket.UserData.Split(',');

HttpContext.Current.User = new GenericPrincipal(formsIdentity, userData);

}

}

}

This method is now obsolete because the SAM will automatically generate the security principal using the *Role* claims in the FedAuth cookie. It was marked obsolete for illustrative purposes.

[Obsolete("This code is deprecated in favor of Federated Authentication.")]

public static void AssignRoles()

{

throw new NotSupportedException("This code is deprecated in favor of Federated Authentication.");

}

# Appendices

## Federated Authentication

Web Services Federation Language (WS-Federation) Version 1.2 OASIS Specification:

<http://docs.oasis-open.org/wsfed/federation/v1.2/os/ws-federation-1.2-spec-os.html>

Security Assertion Markup Language, SAML, version 2.0 Specification:

<http://saml.xml.org/saml-specifications>

Microsoft Patterns & Practices book: *A Guide to Claims-based Identity and Access Control*

<http://msdn.microsoft.com/en-us/library/ff423674.aspx>

## Windows Identity Foundation

Main product page: <http://msdn.microsoft.com/en-us/security/aa570351.aspx>

Download link: <http://www.microsoft.com/download/en/details.aspx?displaylang=en&id=17331>

Main documentation: <http://msdn.microsoft.com/en-us/library/ee748484.aspx>

Technology overview: <http://msdn.microsoft.com/en-us/library/ee517276.aspx>

Class library reference: <http://msdn.microsoft.com/en-us/library/idfx_mref_reference_home.aspx>

Configuration reference: <http://msdn.microsoft.com/en-us/library/gg607699.aspx>

FedUtil reference: <http://msdn.microsoft.com/en-us/library/ee517284.aspx>

Claims Issuance Pipeline: <http://msdn.microsoft.com/en-us/library/ee748501.aspx>

Overview of *Programming Windows Identity Foundation* book with helpful links. Physical copy also available.

<http://social.technet.microsoft.com/wiki/contents/articles/windows-identity-foundation-wif-and-azure-appfabric-access-control-service-acs-content-map.aspx>

Getting access to an X.509 certificate to sign a security token:

<http://www.punkouter.com/2010/04/how-to-resolve-keyset-does-not-exist-error/>

FindPrivateKey command-line tool to retrieve a private key from a certificate store:

<http://msdn.microsoft.com/en-us/library/ms732026.aspx>

## Identity Developer Training Kit

The Identity Developer Training Kit contains a set of hands-on labs, documents and references that will help you to learn how to take advantage of Microsoft's latest identity and access control developer's products and services.

<http://www.microsoft.com/downloads/en/details.aspx?FamilyID=C3E315FA-94E2-4028-99CB-904369F177C0>

## Windows Azure Access Control Service

Main product page (production version): <http://windows.azure.com>

Main product page (development portal): <http://portal.appfabriclabs.com>

Main documentation: <http://msdn.microsoft.com/en-us/library/gg429786.aspx>

ACS error codes: <http://acs.codeplex.com/wikipage?title=ACS%20Error%20Codes&version=8>

## Thinktecture Identity Server

An open source STS with full support for passive and active authentication, OAuth and SAML & SWT token formats.

<http://identityserver.codeplex.com/>

### Steps to build the Identity Server solution within Visual Studio

1. Download Thinktecture Identity Server  
   <http://identityserver.codeplex.com/releases/view/67389>
2. Install Windows Identity Foundation  
   <http://www.microsoft.com/downloads/en/details.aspx?FamilyID=EB9C345F-E830-40B8-A5FE-AE7A864C4D76>
3. Install Windows Identity Foundation SDK  
   <http://www.microsoft.com/downloads/en/details.aspx?FamilyID=c148b2df-c7af-46bb-9162-2c9422208504&displaylang=en>
4. Install WebMatrix  
   <http://www.microsoft.com/web/webmatrix/>
5. Install WCF REST Starter Kit Preview 2  
   <http://aspnet.codeplex.com/releases/view/24644>
6. Install MVC 3  
   <http://www.microsoft.com/downloads/en/details.aspx?FamilyID=d2928bc1-f48c-4e95-a064-2a455a22c8f6&displaylang=en>
7. Install Windows Azure SDK and hotfixes  
   <http://www.microsoft.com/windowsazure/windowsazuresdk+tools/>
8. Install Visual Studio 2010 SP1  
   <http://www.microsoft.com/downloads/en/details.aspx?FamilyID=75568aa6-8107-475d-948a-ef22627e57a5&displaylang=en>
9. Install SQL Server Compact 4.0 for Windows  
   <http://www.microsoft.com/downloads/en/details.aspx?FamilyID=033cfb76-5382-44fb-bc7e-b3c8174832e2>
10. Install Visual Studio 2010 SP1 Tools for SQL Server Compact 4.0  
    <http://www.microsoft.com/web/gallery/install.aspx?appid=SQLCE_4_0%3BSQLCEVSTOOLS_4_0%3BWDeploy>
11. Install Microsoft Code Contracts  
    <http://msdn.microsoft.com/en-us/devlabs/dd491992.aspx>
12. In the "Code Contract" property page on the Thinktecture.IdentityServer class library, enable "Runtime Checking" and set the "Perform Runtime Contract Checking" option to "None".

# Change Log

|  |  |  |  |
| --- | --- | --- | --- |
| **Date** | **Version** | **Author** | **Description** |
| 6/7/2011 | 1.0 | Gary Strasdin | Initial Draft |
| 6/14/2011 | 1.1 | Gary Strasdin | Changes made after initial feedback/ added Appendix information |
| 6/16/2011 | 1.2 | Gary Strasdin | Added supporting diagrams for claims issuance and mobile compatibility |
|  |  |  |  |