|  |  |
| --- | --- |
| **mslogo-1.jpg** | PluralsightColorLogoLgTrans.png |



A Developer’s Guide Access Control in the Windows Azure platform AppFabric

Access control in the cloud

Keith Brown, Pluralsight

November 2009

*All information and code samples are based on the November 2009 CTP release.*

Contents

[Abstract 4](#_Toc246088998)

[An Overview of Windows Azure platform AppFabric Access Control 4](#_Toc246088999)

[Where can I use Access Control Today? 4](#_Toc246089000)

[Identity Challenges 5](#_Toc246089001)

[A Better Solution: Windows Azure platform Access Control 6](#_Toc246089002)

[Introduction to Claims-based Identity 7](#_Toc246089003)

[Claims-based Identity Terminology 7](#_Toc246089004)

[Identity 7](#_Toc246089005)

[Claim 7](#_Toc246089006)

[Security Token 7](#_Toc246089007)

[Issuing Authority & Identity Provider 8](#_Toc246089008)

[Security Token Service (STS) 9](#_Toc246089009)

[Relying Party (RP) 9](#_Toc246089010)

[Hello Access Control 10](#_Toc246089011)

[Standards 10](#_Toc246089012)

[Simple Web Token (SWT) 11](#_Toc246089013)

[Web Resource Authorization Protocol (WRAP) 11](#_Toc246089014)

[Chained Issuers 12](#_Toc246089015)

[Identity Federation Across Security Realms 12](#_Toc246089016)

[Federation and Cross-Platform Interoperability 14](#_Toc246089017)

[Understanding Windows Azure platform AppFabric Access Control 15](#_Toc246089018)

[Getting Started with Access Control 15](#_Toc246089019)

[Access Control in Action: Setting up the StringReverser Sample 15](#_Toc246089020)

[Using ACM.EXE to Configure AC 16](#_Toc246089021)

[Discovering the Management Key and Endpoints for a Service Namespace 16](#_Toc246089022)

[Creating an Issuer 17](#_Toc246089023)

[Creating a Token Policy 19](#_Toc246089024)

[Creating a Scope 20](#_Toc246089025)

[Creating a Rule 21](#_Toc246089026)

[ACM in Action: A Simple Client 22](#_Toc246089027)

[Audience and Applies\_To 23](#_Toc246089028)

[How AC Processes the Token Request 23](#_Toc246089029)

[AC in Action: The StringReverser Service 25](#_Toc246089030)

[Validating an AC Token 26](#_Toc246089031)

[Key Rollover 28](#_Toc246089032)

[Implementing Chained Issuers with AC 29](#_Toc246089033)

[Exchanging a SWT for a SWT 30](#_Toc246089034)

[Exchanging a SAML Token for a SWT 30](#_Toc246089035)

[Delegating with AC 32](#_Toc246089036)

[More on Scope Matching, applies\_to, and Audience 32](#_Toc246089037)

[A Note on Case Sensitivity 33](#_Toc246089038)

[The Beauty of Claims, Manifested 33](#_Toc246089039)

[Summary 34](#_Toc246089040)

[Additional Resources 34](#_Toc246089041)

[About the Author 34](#_Toc246089042)

[Acknowledgements 35](#_Toc246089043)

# Abstract

The goal of this whitepaper is to show developers how to use a claims-based identity model and the Access Control feature of the Windows Azure platform AppFabric to implement single sign-on, federated identity, and role based access control in REST Web Services and clients today, with support for WS-\* and single sign on for web applications coming in the future.

*Note: The Service Bus and Access Control services that were once collectively known as the .NET Services now run directly within Windows Azure. Windows Azure now provides secure connectivity natively via Service Bus and Access Control, in much the same way that it also provides compute and storage as a cloud service. To reflect this change, these capabilities are now branded Windows Azure platform AppFabric, and you will see these changes take effect as we transition from a CTP to a business.*

# An Overview of Windows Azure platform AppFabric Access Control

The Service Bus and Access Control are highly scalable developer-oriented features of the Windows Azure™ platform. These features provide developers with common building blocks and infrastructure support for cloud-based and cloud-aware applications. Just like you rely on the .NET Framework for common building blocks when developing on-premise software, you will rely on these services as building blocks in your cloud applications. Service Bus also relies on Access Control (AC) to handle authentication and help with authorization.[[1]](#footnote-1)

This whitepaper focuses specifically on how to implement claims-based identity in a REST Web service, leveraging Access Control for authentication, authorization, and to a certain degree, personalization. This paper explains claims and other terms and concepts you'll need to know. With that in place, the paper will show how AC fits in to the picture. The next section presents a a roadmap of what you can do today with AC, and what you'll be able to do in the future, with the rationale behind it all.

## Where can I use Access Control Today?

Over the past decade there have been many initiatives to bring claims-based identity and single sign on to browser-based web applications and SOAP Web services. But REST Web services, which are quickly gaining momentum, lack this fundamental support for identity. For this reason, the AC team reprioritized their work to address this need. When AC is first released, it will issue simple tokens that clients and REST Web Services can easily request and consume on any platform, without the requirement for libraries such as WIF (Windows Identity Foundation) or WCF (Windows Communication Foundation). Indeed, the service is equally usable from any language or platform.

With its first release, AC will directly authenticate simple clients using a symmetric key (similar to the familiar user name and password) and broker authentication for enterprise clients that use ADFS v2 (Active Directory Federation Services).

In future releases, AC will implement single-sign on for browser-based web applications as well as SOAP Web services, and will accept security tokens from many other identity frameworks such as Facebook Connect, Google Accounts, Windows Live ID, and others.

In short, the first release of AC will provide basic identity services for REST Web services, and in the future, AC will evolve to become a cloud-based identity management framework for all types of clients, applications, and services.

The next section explains these terms and concepts, and explores the reasons why claims-based identity management is such an exciting paradigm.

## Identity Challenges

The first two questions most applications and services have to answer these days are related to identity: who is the user, and what is she allowed to do? This need for authentication and authorization are common across many different types of systems, from Web services and browser-based applications, to rich Windows desktop applications, and console command line applications. But despite the common need for these features, many services require with solutions. Most developers are not security experts, and many feel uncomfortable being given the job of authenticating and authorizing users. This is not a subject that has been traditionally taught in computer science curriculum, and there’s a long history of ignoring it until late in the development lifecycle.

It’s not surprising to see a single company with tens or hundreds of applications and services, many of which have their own private silo for user identities, and most of which are hardwired to use one particular means of authentication. Developers know how tedious it is to build identity support into each service, and IT professionals know how expensive it is to manage the resulting set of services.

The rampant use of passwords has lead to widespread phishing attacks[[2]](#footnote-2). With so many services building in custom solutions for authentication and authorization, it’s often difficult to implement single-sign on across them, or to federate identity across security realms.

The number of solutions to these problems is growing every year. With so many competing protocols and identity providers, it can be very difficult to support them all, which is where AC provides value. As AC evolves to support more and more of these protocols and providers, your REST Web services that use AC will gain broader access to users in many different security realms, regardless of platform.

## A Better Solution: Windows Azure platform Access Control

The identity solution that the industry has been moving toward over the last decade is based on claims[[3]](#footnote-3). A claims-based identity model allows the common features of authentication and authorization to be factored out of your code. Such logic can then be centralized into external services that are written and maintained by subject matter experts in security and identity. This is beneficial to all involved.

Access Control is a cloud-based service that does exactly that. Instead of writing your own custom user account and role database, you can let AC orchestrate the authentication and much of the authorization of your users. With a single code base in your service, you can authorize access to both enterprise clients as well as simple clients. Enterprise clients can leverage ADFS v2 to allow users to authenticate using their Active Directory logon credentials. Simple clients can establish a shared secret (essentially a very long password) with AC to authenticate directly with AC. As mentioned earlier, in the future support will be provided for more existing identity systems such as Facebook Connect, Google Accounts, Windows Live ID, as well as other enterprise identity providers.

That bears repeating. With a single, simple code base in your REST Web Service and clients, over time as AC evolves, you'll be able to authorize access to more and more users without having to change your server-side code base.

When your service uses AC, the user must obtain a security token from AC in order to log in to your service. This token is a lot like a signed email message from AC to your service with a set of claims about the user's identity. AC does not issue a token unless the user first proves his or her identity: either by authenticating with AC directly or by presenting a security token from another trusted issuer (ADFS for example) that has authenticated that user. Therefore, by the time the user presents a token to your service, assuming you can validate the token (more on that later), you can trust the claims in the token and begin processing the user's request (see Figure 1).



Figure 1: User Presents Claims

Under this model, single sign-on is easier to achieve, and your service is **no longer responsible** for:

* Authenticating users
* Storing user accounts and passwords
* Calling to enterprise directories to look up user identity details
* Integrating with identity systems from other platforms or companies
* Delegation of authentication (a.k.a. federation) with other security realms

This model allows your service to make identity-related decisions based on claims about the user made by a trusted issuer like AC. This could be anything from simple service personalization with the user’s first name, to authorizing the user to access higher-valued features and resources in your service.

# Introduction to Claims-based Identity

This section of the paper introduces some terminology and concepts so that can help you, as a developer, understand this identity architecture.

## Claims-based Identity Terminology

There are several terms commonly used to describe claims-based identity, and it is important to clearly define these terms before diving into the details of AC.

### Identity

The word “identity” is a much overloaded term. So far this paper has used it to describe the problem space that includes authentication, authorization, and so on. But for the purposes of describing AC, the term identity will be used to refer to a set of claims made by a trusted issuer about the user.

### Claim

You can think of a claim as a bit of identity information, such as name, email address, age, membership in the sales role, and so on. The more claims your service receives, the more you’ll know about the user who is making the request. The word “claim” is used, rather than the more traditional “attribute”, commonly used in the enterprise directory world. The reason has to do with the delivery method – in this model your service does not look up user attributes in a directory. Instead, the user delivers claims to your service, and you examine them with a certain measure of doubt. Claims are signed by an *issuer*, and you trust a set of claims only as much as you trust that issuer. Part of accepting a claim is verifying that it came from an issuer that you trust. With AC this verification is easy to achieve using a cryptographic technique called Hashed Message Authentication Code (HMAC). Code for computing an HMAC is found on a wide variety of platforms, including Microsoft .NET.

### Security Token

The user delivers a set of claims to your service piggybacked along with his or her request. In a REST Web service, these claims are carried in the Authorization header of the HTTP(S) request. Regardless of how they arrive, claims must somehow be serialized, and this is managed by security tokens. A *security token* is a serialized set of claims that is signed by the issuing authority. The signature is important – it gives you assurance that the user didn’t just fabricate a set of claims and send them to you.

Some may object to the term "signature" here because AC uses a shared key and HMAC to "sign" the token as opposed to a more traditional asymmetric key. Since the signing key is shared with the service so that it can validate the HMAC on incoming tokens, nothing stops the service from creating and signing a token in the same fashion. There's nothing wrong with a signature created and verified with a shared key, as long as you understand the limitations of this method and plan accordingly. For one thing, since a service could sign a token just like AC does, there's no way for the service to prove to a third party that a token was signed by AC and not by the service itself. This means that you cannot achieve a feature called "nonrepudiation" with AC-issued tokens. Secondly, you would never want to use the same signing key for two different services, because if an attacker compromises one of those services, he could use the service's copy of the token signing key to mint tokens to send to the other service, which would validate them without knowing that they did not come from AC. This can have a detrimental domino effect, toppling service after service if you're not careful to give each service its own key. Later in this paper when we cover configuring AC, you'll be reminded of this so that you don't forget.

### Issuing Authority & Identity Provider

An issuing authority has two main features. The first and most obvious is that it issues security tokens. The second feature is the logic that determines which claims to issue. This is based on the user’s identity, the resource to which the request applies, and possibly other contextual data such as time of day. This type of logic is often referred to as *policy*[[4]](#footnote-4).

There are many issuing authorities, including Windows Live ID, ADFS, PingFederate from Ping Identity (a product that exposes user identities from the Java world), Facebook Connect, and more.

Some authorities, such as Windows Live ID, know how to authenticate a user directly. Their job is to validate some credential from the user and issue a token with an identifier for the user's account and possibly other identity attributes. These types of authorities are called *identity providers* (sometimes shortened as IdP). It’s ultimately their responsibility to answer the question, “who are you?” and ensure that the user knows his or her password, is in possession of a smart card, knows the PIN code, has a matching retinal scan, and so on. There are many ways to authenticate a user, and by allowing an identity provider to do that heavy lifting, that’s one less hard problem for your service.

AC is also an issuing authority, but it is not designed to play the role of an identity provider (although you can make it do this for prototyping purposes or very simple systems, as shown later). The primary job of AC is to *provide a set of claims that are immediately useful to your service*. This means transforming the claims that come from a real identity provider like ADFS, Windows Live ID, or Facebook Connect into claims that have meaning to your service, which might include user name, email address, roles, and so on. By relying on AC to broker identity for your service, you can keep the code base for identity in your service relatively simple. As AC evolves over time you will not have to change that code to take advantage of new identity providers. Instead you'll simply reconfigure the settings in AC to select which identity providers you want AC to trust on your behalf.

A big part of AC is its administration interface for configuring settings such as keys, identity providers, and rules for claims issuance. AC exposes a REST interface for configuring these settings. The associated SDK includes a command-line client built on top of this REST interface called Acm.exe[[5]](#footnote-5), which is demonstrated later. A sample WPF client called the Management Browser is available on [CodePlex](http://www.codeplex.com/), and a Web browser interface is also planned for a future release of the developer portal.

### Security Token Service (STS)

A security token service (STS) is a technical term for the Web interface in an issuing authority that allows clients to request and receive a security token according to interoperable protocols that are discussed in the following section. This term comes from the WS-Trust standard, and is often used in the literature to refer to an issuing authority.

You'll see the term STS used on the developer portal page (Figure 2). This indicates the URL to use to request a token from an issuer. AC actually exposes two STS interfaces: one that clients must use to obtain a token to access your service, and a different one that AC management tools such as Acm.exe use to get a token with which to administer your AC settings via the management REST interface. It is interesting to note that AC itself uses claims: in order to program its management REST interface, you must obtain a token from its management STS.



Figure 2: STS Endpoints for AC

|  |  |
| --- | --- |
| STS Endpoint | This is the URL your clients use to obtain a token for your service. |
| Management Endpoint | This is the URL that Acm.exe uses to access the management REST interface of AC. Use this if you want to manage AC settings programmatically, instead of using a tool such as ACM.EXE. |
| Management STS Endpoint | This is the URL that Acm.exe uses to request a token to use the management REST interface of AC. You can use this if you plan on using the Management Endpoint as mentioned above. |

To avoid jargon wherever possible, this paper will use the term *issuer* to refer to AC and other issuing authorities, including identity providers such as Windows Live ID and Facebook Connect. But don't be confused if you read other literature that uses the term *STS* interchangeably.

### Relying Party (RP)

When you build a service that relies on claims, technically you are building a *relying party*. Some synonyms that you may have heard are *claims aware application*, or *claims-based application*. This definition is included here because you’ll find it in the literature, but since AC helps you build services that act as relying parties, this paper will simply refer to your service as just that: a service.

## Hello Access Control

Now some terminology is defined, here’s a high level view of how a claims-based system works. The first case discussed is the simplest : a client, an issuer (AC), and a service.



Figure 3: Requesting and Sending Claims

Figure 3 shows a simple interaction with AC. The client sends an HTTPS POST to AC that includes three things: the user's name, password, and the base URI of the service that the client wants to access. AC checks to see if an account is registered for the user, and if so, validates the password. If all is well, AC then runs a set of rules that take as input the user's name and the base URI of the web service that the client plans to access. The output of these rules is a set of claims that AC bundles into a token, signs, and sends back to the client.

The client then makes a request to the Web service, including the token in the HTTP Authorization header. The Web service checks for this header, finds the token, validates its signature, and extracts the claims that AC issued. It uses these claims to discover the user's name, roles, and whatever other claims you've configured in your AC rule set.

## Standards

Single sign on and claims-based identity have been evolving since the turn of the century, and as a result there are a lot of ways of doing it. There are competing standards for token formats, and competing standards for the protocols used to request those tokens and send them to services. Indeed, this is what makes AC so useful - over time as it evolves to support a broader range of these standards, your service will benefit from broader access to clients without having to know the details of all of these standards, or worry about trying to implement them correctly.

The first standard was Security Assertion Markup Language (SAML). SAML specified an XML format for tokens (SAML tokens) as well as protocols for performing Web App/Service single sign on using SAML tokens, sometimes referred to inside Microsoft as SAMLP (for the SAML protocol suite). WS-Federation and related WS-\* specifications also define a set of protocols for Web App/Service single sign on, but do not restrict the token format to SAML, although practically speaking, it is the most common format in use today.

### Simple Web Token (SWT)

While SAML and WS-\* are protocols designed to be used with SOAP, REST aims for a more minimalist approach. Thus, AC issues tokens in a format called Simple Web Token (SWT) developed jointly by Microsoft, Google, and Yahoo. A SWT token (pronounced *swat*) looks very much like the query string in a URL, and consequently is easy to parse by any REST Web service. Following is an example of the payload of a typical SWT token. The payload is URL-decoded so you can read it more easily:

firstname=Keith&email=keith@fabrikam.com&roles=staff,partner&issuer=https://foo.accesscontrol.windows.net/WRAPv0.8&Audience=http://fabrikam.com/svc&ExpiresOn=1256767172 &HMACHA256=0egc2SllR6RGb5lrM5EFyCLIuyBvz3gJn3bMgGD1z58=

Splitting this string on '&' reveals that it's a set of key/value pairs:

|  |  |
| --- | --- |
| firstname | Keith |
| email | keith@fabrikam.com |
| roles | staff,partner |
| Issuer | https://foo.accesscontrol.windows.net/WRAPv0.8 |
| Audience | http://fabrikam.com/svc |
| ExpiresOn | 1256767172 |
| HMACHA256 | 0egc2SllR6RGb5lrM5EFyCLIuyBvz3gJn3bMgGD1z58= |

The first three items are claims inserted by rules set up in AC. The last four items are always found in an AC-issued token, and are used to check the validity of the token. As mentioned earlier, you can think of a token as a signed email message carried by the user from the issuer to your service, and that is demonstrated here. The Issuer is who the message is from. The issuer URI in this example identifies AC as the issuer for a service namespace called "test". The Audience indicates to whom the message is directed. The Audience in this example is http://fabrikam.com/svc, the base URI for the service. ExpiresOn is the number of seconds since Epoch time[[6]](#footnote-6) and indicates an absolute time when the token expires. The last value, HMACHA256, which always comes at the very end of a SWT payload, is the signature. If any of these four values are invalid, your service should reject the token and the request should fail (typically by returning a 401 Unauthorized message).

Later, this paper discusses how the values of these items are computed by AC, but for now, focus on the simplicity of the SWT format and how it represents a message from an issuer to a service about a user.

### Web Resource Authorization Protocol (WRAP)

The protocol that AC uses to issue tokens is called Web Resource Authorization Protocol. WRAP is a REST convention (developed in conjunction with SWT) that is used to request tokens from issuers such as AC. As you might expect, this community-developed protocol is simple to use. To request a token, issue a POST command with your request to your issuer's WRAP endpoint (AC refers to this as its STS endpoint) with a content type of "application/x-www-form-urlencoded." In the .NET Framework, the **UploadValues** method in the **WebClient** class makes this a trivial operation.

If your request is accepted, the response includes a token (SWT) that you can send to the service. The request and response are shown in more detail later.

## Chained Issuers

The example above was somewhat contrived. While you can use AC in this way, essentially asking it to validate a user name and password in order to issue a token, in the long term that's not how AC will typically be used. The power of AC lies in its rules engine and its ability to federate with many identity providers. With its first release, the only identity provider that AC will support is ADFS v2, but in future releases it will also support Windows Live ID, Facebook Connect, Google Accounts, and more.

When you configure AC, you can choose which issuers you want it to trust. You could, for example, ask AC to trust an ADFS issuer from adatum.com by providing AC with the certificate[[7]](#footnote-7) corresponding to the ADatum token signing key. This would allow ADFS to validate tokens issued by the ADatum ADFS server.

Asking AC to trust another issuer in effect creates a chain of issuers, with a simple separation of concerns: AC does not want to be in the business of authenticating users, any more than you do. AC is designed to be a claims-transformation engine that you can use to implement role-based access control, basic personalization, and so on. It defers responsibility for authenticating users to whatever identity provider you choose.

Separation of concerns is only one benefit you get when you chain issuers together. The other benefit is identity federation, which is discussed in the next section.

## Identity Federation Across Security Realms

When you build claims-aware Web applications and services, you decouple yourself from any one user store. All you want to know is that an authority you trust has given you the identity details you need about the user of your service. You don’t have to worry about what domain or security realm that user happens to be part of. This makes it very natural to *federate identity* across security realms.

As follows is a concrete scenario to help explain this idea. A company called Fabrikam is in the business of manufacturing bicycles, and thousands of bike shops around the world carry their bikes. Fabrikam has a REST Web service that allows retailers to obtain information about Fabrikam bikes, make purchases, and so on.

In a traditional (non-claims based) system, when a retailer (Bob) wants to start selling Fabrikam’s bikes, he contacts Fabrikam, signs some legal agreements, and tells Fabrikam about his employees: who should be allowed to use Fabrikam’s purchasing service, who should be allowed to make purchases, and so on. Fabrikam then issues a user name and password for each of these users at Bob’s bike shop, and configures its website to grant those users different levels of access depending on their job.

Over time, Bob might conduct business with many other bike manufacturers, each with their own proprietary mechanism for purchasing. Some use the Web, while some rely on fax and phone calls. It’s easy for Bob to forget about all of these details when he’s doing his best just to sell bikes every day. So when Alice joins as a new employee, it takes Bob a while to remember that he has to call Fabrikam (and all of the other manufacturers) and let them know that Alice should be allowed to make purchases. Alice’s first few weeks on the job are a bit daunting as she learns all of the passwords she needs to know for the various systems she’ll be using, and she’ll be denied access to Fabrikam’s retailer website until Bob gets around to calling Fabrikam to add Alice as a user.

What happens when Alice’s role in Bob’s company changes, or even worse, if she leaves the company entirely? When does Fabrikam find out about this?

In this scenario, two companies have established a trust relationship, a covenant[[8]](#footnote-8), between one another. Fabrikam relies on Bob to indicate which employees in his shop should have access to Fabrikam’s resources, and what level of access each should have. Identity federation simply *automates this covenant*. Since Fabrikam already trusts Bob to tell the truth about his employees, it makes sense to let Bob’s system authenticate those employees and automatically give Fabrikam the details about each employee’s current role in the company. Bob can do this by using Active Directory with ADFS v2 as his identity provider (IdP), for example.

Once Bob’s systems are responsible for authenticating his staff, Fabrikam no longer has to issue user accounts for Bob’s employees. When Alice logs into her computer at Bob’s bike shop, that login can be used to tell Fabrikam who Alice is, and what role she plays in Bob’s organization. If Alice leaves the company, Bob disables her user account, and she will no longer be able to use Bob's computer systems, Fabrikam’s purchasing service, or any other service that federates with Bob’s identity provider. When Alice changes jobs, and Bob adjusts her group memberships in his directory, Fabrikam discovers that change the next time Alice logs on and uses Fabrikam’s purchasing service.

Identity federation enables single sign-on across organizations, and this is good for developers, IT pros, users, and shareholders alike.

Identity federation works by using chained issuers. Your service still trusts the same issuer it always did, and that issuer will continue to issue all of the tokens that your service needs. But now instead of authenticating all users directly, your issuer will be configured to accept security tokens from issuers at partner organizations (or from that Java-based division in your own company), leaving them to authenticate users in their own realm in a way that makes sense for them.



Figure 4: Bob's bike shop federates with Fabrikam

In Figure 4, the client is in one security realm (Bob’s shop), while the purchasing service is in another (Fabrikam). In this case, the client (for example, Alice) authenticates with Bob’s issuer and gets a security token that she can send to Fabrikam (step 1). This token indicates that Alice has been authenticated by Bob’s security infrastructure, and includes claims that specify what roles she plays in Bob’s organization.

The client sends this token to Fabrikam’s issuer, where it evaluates the claims, decides that Alice should be allowed to access the service in question, and issues a second security token that contains the claims that the purchasing service expects (step 2). The client sends this second token to the service (step 3), which validates it and obtains all of the identity details it needs about Alice, and can safely allow her to access the service according to the claims it obtained from Fabrikam’s issuer.

Note that the purchasing service did not have to concern itself with validating a security token from Bob’s shop. Fabrikam’s issuing authority did all of that work: making certain to issue security tokens only to employees of trusted partner companies that have previously established a relationship with Fabrikam. In this example, the service will always get tokens from its own issuer. If it sees a token from anywhere else, it will reject it. This keeps the service as simple as possible.

So where does AC fit in? In Figure 4, Fabrikam could use AC as its issuing authority, and Bob could use ADFS v2 to authenticate its clients and issue tokens for them. When you build services using AC, you have the option to accept users from other organizations without having to change your service code. If you want to personalize your service depending on the realm the user is from, AC can help you do that as well.

## Federation and Cross-Platform Interoperability

Even within a single company, federation can be useful. If you end up with a heterogeneous IT system, for example, one that uses Java and Microsoft .NET technologies, as long as your services are built to support federated identity, you have a clear path to achieve single sign-on and all of its benefits, because there are issuers that exist for Microsoft technologies as well as for Java.

While AC happens to be developed using Microsoft .NET, it could have been written using Java or some other technology. As long as all parties involved follow the standards for federated identity, clients, issuers, and services can be built using entirely different technologies and platforms. Once your service is built to rely on AC, you could switch to a different issuer if you found a competitor that you liked better, even if that issuer happened to be built on a non-Microsoft platform. In the world of federated identity, interoperability is extremely important, and the AC team is dedicated to helping fulfill that ideal.

# Understanding Windows Azure platform AppFabric Access Control

AC has three features that you can manage: a WRAP endpoint that issues security tokens, a management REST interface that allows you to configure settings that impact how those tokens are constructed, and the sample code from the SDK that shows how to use AC and provides tools to work with, such as Acm.exe.

## Getting Started with Access Control

In order to get started using Access Control, you will need to create a service namespace (the introductory white paper explains how this is done). Once you've done this, you’ve essentially created your own issuing authority in AC, complete with its own WRAP endpoint for issuing tokens, as well as a private management endpoint that you can use to configure your service namespace. The service namespace becomes part of the URL as you’ll see later, and is used for billing purposes. If you have three services, it makes sense to set up three corresponding service namespaces so that you'll be able to disambiguate AC usage charges for each service.

## Access Control in Action: Setting up the StringReverser Sample

The best way to understand AC is to see it work. This section provides a walkthrough of one of the samples packaged with the SDK, called the String Reverser. This sample uses AC to secure a simple ASP.NET REST Web service that takes some input and produces some output. While this is a very simple example, it shows how clients and services can interact with AC at runtime. The sample solution (which you can find in the SDK samples directory under AccessControl\GettingStarted) includes two projects. One is an ASP.NET REST Web Service that implements the string reverser, and the other is a console application that calls the service. In order to use the sample, you will need to configure AC via the service namespace, which in the example below is called "my-service-namespace".

Please note that for clarity, this example uses a slightly different naming convention for the AC resources in this walkthrough - as of this writing, this SDK sample comes with a setup script that uses the same name for all resources, "gettingstarted". As long as you run the included setup script to set up AC, you should be able to run the sample yourself without any difficulty. Because this example steps through the actions those scripts perform, so that you can begin learning how to configure AC for your own services, we will use more descriptive names.

The first thing to do is indicate to AC which identity providers to trust. This simple example does not use a separate identity provider. To keep things simple with only a single issuer (AC) involved, the client will act as its own issuer and authenticate the user to AC directly. This allows us to set up a shared secret (a symmetric key) between AC and the client. This secret essentially acts like a password, and as long as the client can prove knowledge of this "password", it'll be granted access to the String Reverser service. The console client application in the sample will ask you to supply this value when you run it.

### Using ACM.EXE to Configure AC

The first step is to set up an Issuer resource in AC in the service namespace. You can use ACM.EXE to do this, but first you must tell ACM three things:

* The DNS name of the AC service so that ACM can contact the AC management service
* The name of the service namespace so that ACM can compute the management service endpoint
* The management key, so that ACM can log on to the management service

### Discovering the Management Key and Endpoints for a Service Namespace

You can obtain all three of these values by logging into the developer portal with a Web browser and navigating to the service namespace (see Figure 2 and Figure 5).



Figure 5: Discovering the management key for a service namespace

While it's possible to specify these parameters via command line arguments, since this example demonstrates quite a few ACM commands to set things up, it is more convenient to add them to the ACM.EXE.config file instead, as shown in Figure 6.

<configuration>

 <appSettings>

 <add key="host" value="accesscontrol.windows.net"/>

 <add key="service" value="my-service-namespace"/>

 <add key="mgmtkey" value="UYQkjGhjgWtpSh70T14yBRyfxz6ceakmDS0WvhEL6uE="/>

 </appSettings>

</configuration>

Figure 6 : Configuring ACM.EXE

Now you can use ACM.EXE to set up the issuer. This tool will automatically generate random keys, which helps ensure that they can't easily be guessed by an attacker. Note that ACM.EXE is a thin wrapper around a REST Web Service (the AC management service). That helps to remember the commands it accepts, which are the typical resource management commands for a REST service:

* Get(All)
* Create
* Update
* Delete

### Creating an Issuer

There are several types of resources that you can manage in AC, and we'll start with Issuer. For this example, you must create an Issuer with a key that you can use in the client (the "password" mentioned earlier) to get a token from AC. First, you can query ACM.EXE to determine what information is required in order to create an Issuer resource (Figure 7).

>acm create issuer -?

ACS Tool - Microsoft Access Control Service Tool

Copyright (c) Microsoft Corporation. All Rights Reserved

create issuer -name:<name> -issuername:<issuername> (-key:<key> | -certfile |

-autogeneratekey) [-algorithm:<Symmetric256BitKey|X509>] [-service:<service>]

[-host:<host>] [-mgmtkey:<mgmtkey>] [-simpleout]

 name: Friendly name of the resource

 issuername: The Issuer Name as passed in the token (not display name)

 algorithm: Symmetric256BitKey or X509 (default: Symmetric256BitKey)

 key: The signing key for ACS issued tokens

 certfile: File location of the X509 (.cer) file

 autogeneratekey: Autogenerates a signing key

 service: Service namespace

 host: Host of the management service

 mgmtkey: Key used to authenticate with Managment Service

 simpleout: Displays only the ID, not the full message, when creating

Figure 7: Built-in help in ACM.EXE

You can get help on all commands by using this syntax (acm VERB RESOURCE -?), and if you forget the set of commands and resources, you can get a list of them by running ACM.EXE with no arguments. Note the parameters for the create issuer command shown in Figure 7. All resources in AC have a friendly name supplied by the creator and an ID supplied by AC, so you must specify a friendly name via the -name parameter. This example builds the command in steps, starting with the friendly name.

acm create issuer -name:my-issuer

You must also supply an issuername, which is the value of the Issuer claim that AC will run through its rules engine when you request a token using this issuer. For simplicity, this is named my-issuer as well.

acm create issuer -name:my-issuer -issuername:my-issuer

Now you must specify a key, or ask ACM.EXE to generate a random key on your behalf. Note that symmetric key is the default algorithm, so you don't need to specify the -algorithm parameter. This example has ACM generate a random key.

acm create issuer -name:my-issuer -issuername:my-issuer -autogeneratekey

You can omit the service, host, and mgmtkey parameters since these are already configured in ACM.EXE.config. When you run this command, ACM.EXE generates a random key and sends an HTTPS POST to the AC management endpoint that includes the details of the request. This POST is accompanied by an Authorization header with a token from the AC management STS, so there are actually a couple of round trips to AC occurring:

1. POST to AC management STS that sends the management key and retrieves a token for AC.
2. POST to AC management endpoint that sends the request along with the token retrieved in (1).

It's interesting to note that if you were coding up your own management tool instead of using ACM.EXE, you could reuse the token retrieved in step 1 until it expires.

AC validates the token and ensures that you are allowed to manage this namespace, then creates an Issuer resource using the key and the names that you specified, and returns the ID of the new object to ACM, which prints it out. Note that in all of the following examples, the command line to ACM is split across multiple lines for ease of reading. You must remove these line breaks if you run the command yourself.

>acm create issuer

 -name:my-issuer

 -issuername:my-issuer

 -autogeneratekey

Object created successfully (ID:'iss\_0beec7b5ea3f0fdbc95d0dd47f3c5bc275da8a33')

Figure 8: Creating an Issuer resource using ACM.EXE

Since ACM.EXE didn't print out the key, you must make one more call to obtain the key it generated for the issuer:

>acm getall issuer

Count: 1

id: iss\_0beec7b5ea3f0fdbc95d0dd47f3c5bc275da8a33

name: my-issuer

issuername: my-issuer

key: ueh0xnVLw7y53mEGHNeMmDu6vjz8FqMNiMBFdkc/6JM=

algorithm: Symmetric256BitKey

That's a pretty strong password! You need that in order to run the client application. Issuer resources in AC basically tell AC what to expect for input when a client requests a token.

### Creating a Token Policy

Now that AC knows what it'll be getting as input, the next step is to tell it how to output tokens. You do this by creating a TokenPolicy resource (Figure 9), which simply needs a name, a key, and a timeout in seconds. AC will use this timeout to compute the ExpiresOn time for any token that it issues using this policy. For this example, we specify 3600 seconds for the timeout, so that clients will be able to cache their token and reuse it for up to one hour before they need to go back to AC to get a new one.

>acm create tokenpolicy

 -name:my-token-policy

 -timeout:3600

 -autogeneratekey

Object created successfully (ID: tp\_25da58a8125344e9905972d89cf4a072')

>acm getall tokenpolicy

Count: 1

id: tp\_25da58a8125344e9905972d89cf4a072

name: my-token-policy

timeout: 3600

key: 6wxMs7Il7jyssRmEKK4X/MluLIVp5XCCYxqC9Yt9lBg=

Figure 9: Creating a TokenPolicy resource

The timeout setting is very important! It allows you to trade off security vs. performance. Users can continue to use their tokens until they expire, so if you revoke a user's permission by changing settings in AC, if that user already had a token cached, she can continue to use it until it expires. If you set the timeout to a very small value, clients will need to visit AC more frequently to obtain fresh tokens, but your services will get more up-to-date information about the user. Small values for the timeout also imply that your clocks will need to be carefully synchronized as well, unless your service allows for a certain amount of clock skew. In many systems it's totally reasonable to set these timeouts to be closer to a work day (8 hours), given that user identity details don't change that frequently. Fortunately you can make these decisions on a service by service basis, because you will generally use a distinct service namespace for each service (for billing purposes).

However you choose to set things up, you should be careful to ensure that each service shares a unique symmetric key with AC via a unique TokenPolicy resource. This prevents the domino attack I described earlier (see page 8).

The timeout is also important because it allows you to contain the window of opportunity for replay attacks, where an attacker has managed to steal a token (for example, by eavesdropping on the connection of a legitimate user, or by compromising a client application). The attacker can replay that token in order to impersonate the user until the token expires. Besides adjusting the timeout, another strong defense against replay attacks is to require HTTPS for all requests to your service, to prevent casual eavesdroppers from recording the token of a legitimate user.

### Creating a Scope

As mentioned earlier, each service you create will typically have its own service namespace for billing purposes. But within a single service, you might have different types of resources that need different authorization rules. The claims you need to authorize access to one type of resource may differ from the claims you need for another. The String Reverser is a simple service that only needs a single scope. Indeed it only needs one claim that says "this user is allowed to reverse strings".

When a client requests a token from AC, it must send a parameter called *applies\_to*, which is a URI that indicates the resource it is trying to access. For a service with multiple scopes, the client must be very specific with applies\_to, and provide the URI[[9]](#footnote-9) to the resource in question. AC will then use this to look up the correct scope, and will use the rules from that scope to figure out the claims for the client's token. The String Reverser only has one resource (the reverser), so you only need to create a single scope with an appliesto property that is the base URI of the service, which is http://localhost/ACSGettingStarted for this example.

A scope links to a RuleSet that tells AC how to issue claims (more on that later) and a TokenPolicy, which tells AC how it should sign the token that contains those claims. Note that you must use the ID (not the friendly name) of the TokenPolicy resource when you link it to the new scope. AC uses this convention across the board in its management service, so be sure to note the ID for any object that you create using a tool like ACM.EXE. If you forget to do this, you can always use the getall command to get a listing of all resources of a given type, and discover the ID for the one you want.

>acm create scope

 -name:my-scope

 -appliesto:http://localhost/ACSGettingStarted

 -tokenpolicyid:tp\_25da58a8125344e9905972d89cf4a072

Object created successfully (ID:'scp\_3fe16b4df862e0f361249656a62c60738206c3a2')

Figure 10: Creating a Scope

### Creating a Rule

You can sum up what AC does in four words: "Claims in, claims out." At its heart, AC is a claims transformation engine, and it works based on rules. Rules describe the logic for how AC transforms input claims into output claims. For the String Reverser service, the rules are very simple. If you know the password, you may use the service. Anyone who requests a claim and is able to provide the "password" for the issuer that I created earlier should be able to get a token to use the service.

Let's translate this into a real AC rule. Basically what you must say is, "If you can prove you know the key for my-issuer, I'll grant you permission to use the string reverser". In AC, rules are defined in terms of input claims and output claims. To reword this in AC terms, "If an input claim is found of type Issuer with a value of 'my-issuer', issue an output claim of type 'action' with a value of 'reverse'." This will instruct AC to issue a token with a single claim that the StringReverser service can validate, assuming the client requesting a token can supply the password for my-issuer. Note that StringReverser is indeed coded to look for a claim of type "action" with a value of "reverse".

Rules are technically created within RuleSet resources, and in a future version of AC you may be able to share one RuleSet across multiple scopes. But in this initial release, each scope has exactly one RuleSet, so ACM.EXE makes it easy for you to create rules in the default RuleSet for the scope.

>acm create rule

 -name:my-rule

 -scopeid:scp\_3fe16b4df862e0f361249656a62c60738206c3a2

 -inputclaimissuerid:iss\_0beec7b5ea3f0fdbc95d0dd47f3c5bc275da8a33

 -inputclaimtype:Issuer

 -inputclaimvalue:my-issuer

 -outputclaimtype:action

 -outputclaimvalue:reverse

Object created successfully (ID: rul\_d50d977a296c89d651d5687bcfcd9686156552f...)

Figure 11: Creating a Rule

Note that the input claim type is set to "Issuer" in this rule. Issuer is a special claim type that AC knows about intrinsically. Specifically, AC guarantees that any rule with an antecedent of type "Issuer" will test true if the client either proves knowledge of the issuer key directly (as in this simple example), or supplies a valid token from another issuer whose name matches the issuer resource referred to in your rule.[[10]](#footnote-10)

The last step to get this sample working is to create a virtual directory in IIS that points to the service. As mentioned earlier, the sample comes with a setup script that does all of this for you (sets up AC and adds a virtual directory), so you should use that to run the sample. This example has demonstrated what goes on inside of that script.

## ACM in Action: A Simple Client

Now that AC is set up so that it can issue tokens for the StringReverser service, it's time to examine the client code to see what it requires to request a token from AC. Given that this sample uses an AC issuer to act as a user name and password, it makes sense to simply call AC and supply this name and password along with the URI of the service (applies\_to) and expect to receive a token in return. AC supports this simple protocol, and it's surprisingly easy to code using the System.Net.WebClient class (Figure 12).

private static string GetTokenFromACS(string issuerKeySuppliedByCaller)

{

 // request a token from ACS

 WebClient client = new WebClient();

 client.BaseAddress = string.Format("https://{0}.{1}",

 serviceNamespace, acsHostName);

 NameValueCollection values = new NameValueCollection();

 values.Add("wrap\_name", "my-issuer");

 values.Add("wrap\_password", issuerKeySuppliedByCaller);

 values.Add("applies\_to", "http://localhost/ACSGettingStarted");

 byte[] responseBytes = client.UploadValues("WRAPv0.8", "POST", values);

 string response = Encoding.UTF8.GetString(responseBytes);

 return response

 .Split('&')

 .Single(value => value.StartsWith("wrap\_token=",

 StringComparison.OrdinalIgnoreCase))

 .Split('=')[1];

}

Figure 12: Requesting a Token from AC

First, note how the code computes the DNS name of the AC WRAP endpoint from two things: the host name of AC and the service namespace. The AC host name is accesscontrol.windows.net. The service namespace used when running this sample was my-service-namespace, so the full WRAP endpoint for it becomes:

https://my-service-namespace.accesscontrol.windows.net/WRAPv0.8

WebClient.UploadValues sends a POST to this endpoint, passing three parameters: wrap\_name, wrap\_password, and applies\_to. The name and password refer to the issuer name set up in AC, and its symmetric key (which ACM.EXE generated automatically). Note how the base URI of the service is sent via the applies\_to parameter, so that AC can find the scope that the example set up and execute the proper set of rules.

### Audience and Applies\_To

It may seem easy for the client to "cheat" and maliciously supply an incorrect URI for applies\_to. Remember that a service with multiple scopes can have multiple sets of rules, one per scope. Perhaps the client knows that she has more permissions in a scope for resource X, so what keeps him or her from getting a token for X (by setting applies\_to to a URI that will match the scope for X) and then sending that token with a request on a resource of type Y?

Recall that a Simple Web Token is a bit like an email message from the issuer to the service. Note that AC fills in the Audience parameter (which is similar to the To header in an email message) by copying the exact value that the client supplied for applies\_to in the token request. Thus, the service can see the exact URI for which the client requested a token, and as you'll see when we look at the StringReverser service, this is one of the things that determine the validity of a token. Getting a token with an invalid Audience parameter would be similar to finding an email message in your Inbox that was addressed to someone else. Tokens whose Audience doesn't match the resource being requested should be discarded.

### How AC Processes the Token Request

When AC receives this request, it performs the following steps:

1. Finds an issuer resource in the service namespace whose issuername property matches the value supplied via wrap\_name.
2. Verifies that the value supplied via wrap\_password matches the value of the issuer's key property[[11]](#footnote-11).
3. Finds the best matched[[12]](#footnote-12) scope in the service namespace based on the URI supplied via applies\_to.
4. Runs the RuleSet linked to the scope with a single Issuer claim whose value is copied from the issuer resource's issuername property.
5. Packages the resulting claims into a Simple Web Token and signs the token with the key in the Token Policy linked to the scope.
6. URL encodes the token and sends it back to the client.

Note that if any of these steps fail (for example, the issuer or scope cannot be found, or the issuer key doesn't match) AC rejects the request. If successful in the StringReverser example, the response will look something like this (the SWT in the response is abbreviated here as [TOKEN]):

wrap\_token=[TOKEN]&wrap\_token\_expires\_in=3600

The last lines of code in Figure 12 show how to parse the token out of this response. Keep in mind that the token is still URL encoded at this point, and will need to be decoded before being sent to the service.

Examining the token should reveal something similar to this:

|  |  |
| --- | --- |
| action | reverse |
| Issuer | https://my-service-namespace.accesscontrol.windows.net/WRAPv0.8 |
| Audience | http://localhost/ACGettingStarted |
| ExpiresOn | 1256767172 |
| HMACHA256 | 0egc2SllR6RGb5lrM5EFyCLIuyBvz3gJn3bMgGD1z58= |

The client is free to cache this token as long as it wants, bearing in mind that the service will stop accepting it once its ExpiresOn time has passed. In this simple example, the client program exits as soon as it's done making its request, so there's no opportunity for caching, but this is a great performance optimization that you should keep in mind when building your own longer-running clients. If you want to perform this sort of caching, you should avoid looking at the token contents directly, as AC will likely offer an option in the future to encrypt the token values so that only the relying party can read them. This is why the response includes a wrap\_token\_expires\_in parameter. This tells the client the approximate number of seconds in the future that the token will expire, and this parameter is never encrypted.

With the token in hand, the client is now ready to call the service. It simply needs to URL decode the token it received from AC and send it in the Authorization header along with the request to the service using the WRAPv0.8 authorization scheme. This sample uses HTTP to call the service, which means this token will go over the wire in cleartext, which is not a problem since it is only reversing a string. But you must use HTTPS if you want to prevent an eavesdropper from recording and replaying tokens to a more valuable service. It's interesting to note that AC requires HTTPS for all requests to any of its endpoints. This is a wise practice for any service that requires more than a trivial level of security.

private static string SendMessageToService(string token, string valueToReverse)

{

 WebClient client = new WebClient();

 client.BaseAddress = "http://localhost/ACSGettingStarted/Default.aspx";

 string headerValue = "WRAPv0.8" + " " + HttpUtility.UrlDecode(token);

 client.Headers.Add("Authorization", headerValue);

 NameValueCollection values = new NameValueCollection();

 values = new NameValueCollection();

 values.Add("string\_to\_reverse", valueToReverse);

 byte[] serviceResponseBytes = client.UploadValues(string.Empty, values);

 return Encoding.UTF8.GetString(serviceResponseBytes);

}

Figure 13: Sending the Request and Token to the Service

The vast majority of this code is a simple HTTP POST of a Web form to the StringReverser service. There's really not much of interest here except for the Authorization header, which is set to include the token sent from the server. Notice the call to UrlDecode - this is necessary because the token is returned in an URL-encoded format and must be decoded before being sent to the service in the Authorization header.

You may not have noticed this, but the client didn't have to perform a single cryptographic operation in this simple example. This is the simplest way to request a token from AC, and works well for very simple apps that may not have access to cryptographic primitives such as HMACHA256.

## AC in Action: The StringReverser Service

Next we will examine what it takes to secure a service using AC. When a request arrives the first task for the service is to check for the Authorization header. No header, no service! After pulling out the Authorization header (Figure 14), the service splits off the standard "WRAPv0.8 " prefix and validates the token.

protected void Page\_Load(object sender, EventArgs e)

{

 // get the authorization header

 string headerValue = Request.Headers.Get("Authorization");

 // check that a value is there

 if (string.IsNullOrEmpty(headerValue))

 {

 this.ReturnUnauthorized();

 return;

 }

 // check that it starts with 'WRAPv0.8'

 if (!headerValue.StartsWith("WRAPv0.8 ", StringComparison.OrdinalIgnoreCase))

 {

 this.ReturnUnauthorized();

 return;

 }

 // check there is a ' ' between 'WRAPv0.8' and the token

 // no other spaces allowed

 string[] tokenValues = headerValue.Split(' ');

 if (tokenValues.Length != 2)

 {

 this.ReturnUnauthorized();

 return;

 }

 // the token is always the 2nd value

 string token = tokenValues[1];

 // create a token validator

 TokenValidator validator = new TokenValidator(

 this.acsHostName,

 this.serviceNamespace,

 this.trustedAudience,

 this.trustedTokenPolicyKey);

 // validate the token

 if (!validator.Validate(token))

 {

 this.ReturnUnauthorized();

 return;

 }

 // ...

Figure 14: Extracting a Token from the Authorization Header

### Validating an AC Token

In the StringReverser example, a helper class called TokenValidator is used to validate the token. This class is convenient for .NET developers, although the code inside of it is quite easy to write yourself if you are using some other platform besides .NET to build your service. The Validate method (Figure 15) shows the four important checks you must make on any AC-issued token before accepting it as valid. The most complicated of these helper methods, IsHMACValid, is implemented in seven lines of code using the .NET Framework.

public bool Validate(string token)

{

 if (!this.IsHMACValid(token,

 Convert.FromBase64String(this.trustedSigningKey)))

 return false;

 if (this.IsExpired(token))

 return false;

 if (!this.IsIssuerTrusted(token))

 return false;

 if (!this.IsAudienceTrusted(token))

 return false;

 return true;

}

Figure 15: Validating a Token Issued by AC

The first test is a signature check, which gives the service confidence that the contents of the token were created by AC, the only other party privy to the shared signature key. This also ensures that the contents of the token haven't been tampered with since it was issued. IsHMACValid splits the HMACHA256 signature from the token and then hashes it with the service's copy of the signing key using HMAC with the SHA256 hash algorithm. This signing key was created with ACM.EXE earlier with the TokenPolicy resource. Be sure you update the code of this sample so it has a copy of your service namespace and signing key, otherwise it won't work (see Figure 16, below).

The expiration check serves two purposes. First, if a token is ever stolen (perhaps eavesdropped over an HTTP connection between client and service), this reduces the time window that it can be used. Secondly, it guarantees a certain degree of freshness since clients cannot cache tokens longer than their expiration date, and must eventually return to AC to get a new token with fresh claims.

The issuer check is another test to ensure that this token was issued by AC from our service namespace. From a cryptographic standpoint, it complements the signature check and gives you further confidence that the token was created and signed by AC.

As discussed earlier, the Audience check prevents one form of client cheating by verifying the Audience field which AC copies from the user's token request (recall the applies\_to parameter). This is a critical check as it ensures that the token you are receiving was issued with rules designed for the resource the client is actually accessing. In the StringReverser, the check simply ensures that the value of Audience matches the appliesto value in the scope for this service: http://localhost/ACSGettingStarted.

Once the token is validated, the rest of the code is trivial - it simply reverses the string and returns it to the client. If validation fails, the service does NOT perform the operation, but instead halts immediately, returning a 401 Unauthorized HTTP status code.

public partial class \_Default : System.Web.UI.Page

{

 string serviceNamespace = "replaceWithYourServiceNamespace";

 string trustedTokenPolicyKey = "replaceWithYourTokenPolicyKey";

 // ...

Figure 16: Settings You'll Need to Update in the StringReverser Service

## Key Rollover

Whitfield Diffie once [remarked](http://java.sun.com/developer/technicalArticles/Interviews/diffie_qa.html), "If you're relying on a secret that you can't change readily, then you should think of that as a vulnerability." You should avoid creating [*heavy keys*](http://www.pluralsight.com/community/blogs/keith/archive/2006/06/16/28050.aspx) - cryptographic keys that are difficult to change. Make it easy to change out your keys, and seriously consider automating those changes or institute and enforce policy that ensures they are changed regularly. Why? An obvious reason is that an administrator of your system inevitably has access to those keys, and while you may trust him today, you may not trust him as much after he quits and walks away with all of your keys on a thumb drive. Another reason is that cryptographers that are much smarter than you or me ensure us that the longer a key is used, the weaker it becomes. Remember what can happen if a bad guy were to learn the symmetric key that your service shares with AC. He can mint his own tokens that have any claims in them he wants, sign them just like AC would, and send them to your service. In a claims-based world, your token signing keys become the keys to your kingdom, and must be carefully guarded. So protect your keys[[13]](#footnote-13), update them regularly, and carefully limit who has access to your AC service namespace!

The AC team has clearly thought about this, as they support a feature called *key rollover*. Anywhere AC needs a key to validate something, it allows for two versions of that key: current-key, and previous-key. If validation fails with current-key, it is retried with previous-key. Either key can be used for validation. When you update a validation key, say a key for an issuer resource, current-key is first copied to previous-key before current-key is overwritten with the new key. This ensures that existing clients will continue to function even while you're rolling over keys. This makes it easy to automate key rollover, because if you're regularly updating your keys, only two keys will ever be valid at a given time.

You can see this on the management portal (Figure 17), which shows the current management key and the previous one. And if you regenerate the current-key, you can see how it's first copied to previous-key before being overwritten.



Figure 17: Key Rollover

This is a great feature, but you keep in mind that if a current-key is ever compromised, you'll need to run an update operation twice to completely purge the compromised key out of your service namespace. The first update will copy the key into previous-key, and the second will overwrite previous-key.

Don't forget that your REST service holds a validation key, that is, your copy of the symmetric token signing key that you share with AC and use to validate incoming tokens issued by AC. To avoid interrupting service to your clients who have cached tokens, you'll want to implement a similar scheme. Simply keep two copies of the token signing key: current-key and previous-key, and roll them over like AC does. This will allow you to rotate fresh keys on a monthly, weekly, or even daily basis without interrupting service to clients. You should seriously consider automating the distribution of these keys.

Without automation or strongly enforced policy, your keys will rarely, if ever, change. Guaranteed. And if you've also failed to adequately protect those unchanging keys, you've got a recipe for disaster.

## Implementing Chained Issuers with AC

With its initial release, AC provides three ways to request a token. The first and simplest method was shown in the StringReverser client above. You simply POST a form proving that you know an issuer password, and you're in. This allows you to implement the simplest type of authentication and authorization that is prevalent in the REST world today, user name and password.

But what if you want to do something more interesting, for example to connect AC to a real identity provider (IdP) such as ADFS v2, which issues SAML tokens, or another service which issues SWTs? Or maybe you want to implement your own IdP that issues a SWT that AC can consume? These are actually quite easy to do, given what you already know.

### Exchanging a SWT for a SWT

Remember the simple request format for user name and password? There's a similar format for sending a signed Simple Web Token (SWT) to AC. Instead of sending wrap\_name and wrap\_password, you send a SWT via the wrap\_SWT parameter. The client would then need to be modified to talk to the identity provider of your choice to authenticate and obtain this initial token.

1. Client requests token from the IdP, authenticating the user to the IdP and receiving a SWT.
2. Client sends this token to AC and retrieves a token to be used with the service, just like before.
3. Client sends the token to the service, just like before.

Note that the only difference here is that the client is sending a token to AC in lieu of a user name and password. In order to make this work, you must ask AC to trust the IdP you're using. This is done by simply adding an Issuer and sharing the key with the IdP so it knows how to sign tokens that AC will accept. Hopefully this makes the rationale behind the name of the Issuer resource a bit more clear!

Note that the client can cache both tokens as long as they are valid, in order to reduce round trips to the issuers involved.

And finally, notice that in step 3, the service ends up getting a SWT token from AC in the same way it did in the simple case, before chaining issuers together. Your service doesn't need to change to support this scenario!

### Exchanging a SAML Token for a SWT

If you want to trust ADFS v2 as an identity provider, you must remember that it issues SAML tokens, not SWTs. SAML tokens are signed with a private key, so you'll need the X.509 signing certificate of the federation server, or its Federation Metadata document (which embeds this cert). Once again, just set up an issuer in AC, and upload the cert/metadata document as part of the issuer resource. The issuer name in this case will be the URI of the federation server, which is also part of the Federation Metadata document.

ADFS v2 exposes many different types of endpoints for issuing security tokens, and given that it hasn't been released as of this writing, it is difficult to say whether it will expose a WRAP endpoint that issues Simple Web Tokens. But it certainly will expose a WS-Trust endpoint to support SOAP web services. It's also safe to say that clients running in ADFS v2 domains will likely be built using Microsoft .NET, and therefore will be able to use the Windows Identity Foundation (WIF) to request tokens from ADFS v2 using WS-Trust. Figure 18 shows a snippet of code that does this, it is likely that you'll see examples of this in future versions of the SDK once ADFS v2 ships.

public static string GetSamlToken(SamlVersion samlVersion)

{

 WSTrustChannelFactory trustChannelFactory =

 new WSTrustChannelFactory(new KerberosWSTrustBinding(

 SecurityMode.TransportWithMessageCredential),

 new EndpointAddress(new Uri(Defaults.ADFSUrl)));

 trustChannelFactory.TrustVersion = TrustVersion.WSTrust13;

 try

 {

 RequestSecurityToken rst = new RequestSecurityToken(

 WSTrust13Constants.RequestTypes.Issue,

 WSTrust13Constants.KeyTypes.Bearer);

 rst.AppliesTo = new EndpointAddress(Defaults.ACSUrl);

 switch (samlVersion)

 {

 case SamlVersion.SamlOneDotOne:

 rst.TokenType = Microsoft.IdentityModel.Tokens

 .SecurityTokenTypes.Saml11TokenProfile11;

 break;

 case SamlVersion.SamlTwoDotZero:

 rst.TokenType = Microsoft.IdentityModel.Tokens

 .SecurityTokenTypes.Saml2TokenProfile11;

 break;

 default:

 throw new ArgumentException("Passed unsupported value",

 "samlVersion");

 }

 WSTrustChannel channel = (WSTrustChannel)trustChannelFactory

 .CreateChannel();

 GenericXmlSecurityToken token = channel.Issue(rst)

 as GenericXmlSecurityToken;

 string tokenString = token.TokenXml.OuterXml;

 return tokenString;

 }

 finally

 {

 trustChannelFactory.Close();

 }

}

Figure 18: Using WS-Trust to obtain a SAML token from ADFS v2

Note the highlighted constant that indicates we are requesting a *bearer token*. This is the type of token that AC uses in order to avoid any cryptography requirements in clients other than supporting transport level security (HTTPS). Once you've retrieved this token, you can send it to AC in exchange for a Simple Web Token.

### Delegating with AC

As mentioned earlier, AC has the ability to accept tokens from any issuer that issues Simple Web Tokens. Since AC issues a SWT, you can chain two AC solutions together to delegate permission to another organization to use your service. This is just another example of identity federation, done completely within the confines of AC.

For example, imagine you want to allow another company (Fabrikam, for example) to use the StringReverser service. If Fabrikam uses AC, they can create a service namespace and configure it to trust their issuers. They can then set up a TokenPolicy and share its signing key with you. You can create an Issuer resource in my-service-namespace using the key that you now share with Fabrikam. You have effectively set up a trust relationship between your company and Fabrikam and now your AC WRAP endpoint will accept tokens issued by their AC WRAP endpoint. Note that the service doesn't need to change for this to happen.

## More on Scope Matching, applies\_to, and Audience

When a client requests a token from AC, it is required to indicate the resource to which it plans to send that token. This allows some flexibility in how you set up rules. Simple services like the StringReverser only need one set of rules, but another service that exposes more than one type of resource may eventually require different rules depending on what type of resource is being accessed. AC scopes are designed to handle this (indeed this is why the more abstract term "scope" is used instead of something like "service" or "application"). For example, you may have a service that needs different rules depending on whether the user is accessing https://fabrikam.org/purchasing/products or https://fabrikam.org/purchasing/setup.

The first thing you'd want to do is ensure that clients accessing products pass an applies\_to of https://fabrikam.org/purchasing/products (or even more specifically, products/[product-id]) instead of simply the base address of the service, as the StringReverser sample does. And similarly, a client accessing /setup should specify at least https://fabrikam.org/purchasing/setup for applies\_to. This gives you the flexibility on the back end to set up multiple scopes depending on the needs of your service. Build your clients this way from the beginning and you'll be happy you did.

Imagine at first the claims needed by these resources are exactly the same. You can create a single scope to handle this, with an appliesto URI of https://fabrikam.org/purchasing. AC uses a longest-prefix-match algorithm to match a token request's applies\_to parameter to a scope's appliesto property. So a client asking for https://fabrikam.org/purchasing/setup would match that scope just fine.

You have the flexibility to change this later. For example a year later you decide that you want to authorize users differently depending on whether they are accessing /products or /setup resources in your service. You can then add two scopes that match the more specific URIs, https://fabrikam.org/purchasing/products and https://fabrikam.org/purchasing/setup. The client doesn't need to change at all, because it's already sending very precise URIs for applies\_to, but you now have two distinct rule sets for /products and /setup. You'll only need to update your service if you decide to issue new claim types that the service doesn't already understand.

Because AC always copies the value of applies\_to in a client's token request into the Audience field, the service will always know the full URI that was used to match a scope. And as mentioned earlier, it's critical to check this value in your service to ensure that it matches the request, because *AC cannot control how a client uses a token once it's been issued*. If you receive a request for http://example.org/oranges but the URI in Audience says http://example.org/apples, you'd better think twice before servicing that request.

Note that while scopes can have very specific URI assigned for matching, you may not include a query string in a scope's appliesto setting. Query string parameters are unordered, which would wreak havoc with the longest-prefix matching algorithm.

## A Note on Case Sensitivity

AC compares many strings and URIs, from scope matching to rules execution. In its initial release, all of these string comparisons are performed using ordinal (case sensitive) matching. This conservative approach may be relaxed in the future in certain cases, for example, there might be a rule with a flag that says, "perform a case-insensitive comparison." It would have been far harder to take a liberal approach and then change to a conservative one, possibly breaking lots of existing clients. So for now, be careful with case!

## The Beauty of Claims, Manifested

The beauty of claims-based identity and AC is that *it allows your service to be loosely coupled to identity*. Indeed if you consider all of the ways you can configure AC to allow users to access the StringReverser, from federating with ADFS v2, another AC provider, or some other provider of Simple Web Tokens, *none of these changes require you to redesign, recode, recompile, or even reconfigure the StringReverser service*.

And consider how this system scales under load: *validating tokens is cheap*. The StringReverser doesn't need to do any I/O to validate a token, I/O which can get expensive if you're hosted in the cloud. It does not need to perform user database lookups or complex cryptography - HMACHA256 is quite fast to compute. If StringReverser receives a large set of unauthorized requests, it doesn't cost much to reject them quickly and proceed with processing legitimate requests.

# Summary

Claims are the future of identity, and Windows Azure platform Access Control is a great way to get started. By adopting a claims-based approach to identity, your service will benefit from single sign on, and your development team will no longer have to worry about how they are going to authenticate users, which is a complicated and expensive business to be in. AC can further simplify your service by handling much (if not all) of your authorization logic. And the possibilities will broaden in the future as AC supports more and more identity providers. You can get started today by designing services that are built to accept claims instead of handling their own authentication!

# Additional Resources

Here are some links to additional resources that you might find helpful as you learn more about Windows Azure platform Service Bus and Access Control.

* An Introduction to Windows Azure platform AppFabric for Developers
	+ <http://go.microsoft.com/fwlink/?LinkID=150833>
* A Developer’s Guide to Service Bus in Windows Azure platform AppFabric
	+ <http://go.microsoft.com/fwlink/?LinkID=150834>
* A Developer’s Guide to Access Control in Windows Azure platform AppFabric (*this paper*)
	+ <http://go.microsoft.com/fwlink/?LinkID=150835>
* Windows Azure platform
	+ <http://www.microsoft.com/windowsazure/>
* Service Bus and Access Control portal
	+ <http://netservices.azure.com/>
* WRAP and SWT specifications
	+ <http://groups.google.com/group/oauth-wrap-wg>
* Windows Identity Foundation (WIF) Whitepaper for Developers
	+ <http://download.microsoft.com/download/7/d/0/7d0b5166-6a8a-418a-addd-95ee9b046994/GenevaFrameworkWhitepaperForDevelopers.pdf>
* Justin Smith’s blog
	+ <http://blogs.msdn.com/justinjsmith/>

# About the Author

Keith Brown is a cofounder of Pluralsight, a premier Microsoft .NET training provider offering both in-classroom and online training courses. Keith has authored books on Windows security, and spent eight years as the security columnist for MSJ and MSDN magazines. Keith has spent over a decade developing courses, speaking at conferences, and teaching developers about security. You can reach him at http://www.pluralsight.com/keith.

# Acknowledgements

This paper wouldn’t have been possible without the tremendous help from Justin Smith of Microsoft. His PDC sessions coupled with his personal guidance and sharp eye during editing drove this whitepaper. Cyrus Harvesf, Jenny Lo, Seth Manheim, and Anjan Das also helped review the paper and provided some insightful comments. Thanks guys!

1. For more information on the Service Bus, see the references at the end of this paper. [↑](#footnote-ref-1)
2. [Phishing](http://en.wikipedia.org/wiki/Phishing) is all about convincing a user to divulge sensitive information (such as passwords). This is commonly done by sending an email that masquerades as being from a legitimate company with which the user may have an account. The email includes a link that leads to the attacker’s website, convincingly built to look like the legitimate company’s website. When the user “logs on”, her password is captured by the attacker, along with any other information the user is duped into giving away. [↑](#footnote-ref-2)
3. In the SAML world (Security Assertion Markup Language), the term *security assertion* is used instead of *claim*, but it essentially means the same thing. [↑](#footnote-ref-3)
4. Not to be confused with WS-Policy. [↑](#footnote-ref-4)
5. ACM stands for Access Control Management. [↑](#footnote-ref-5)
6. Epoch time is Jan 1, 1970. [↑](#footnote-ref-6)
7. ACS also accepts federation metadata documents from ADFS v2, which contains the signing certificate and other details about an issuer. For more information on federation metadata, see the [WS-Federation spec](http://msdn.microsoft.com/en-us/library/ms951273.aspx). [↑](#footnote-ref-7)
8. Now that the technological hurdles of identity federation have been conquered, the hard part is negotiating the legal contract that covers agreements like these. But ultimately it's the legal contract that protects the interests of Bob and Fabrikam in this relationship. [↑](#footnote-ref-8)
9. Include the full URI minus any query string. Query strings are not currently supported for applies\_to because of the rules for scope matching, which I'll cover later. [↑](#footnote-ref-9)
10. AC will use the key you've configured for that issuer to validate the token's signature in this case. [↑](#footnote-ref-10)
11. Technically it can also match the previous-key property, which is discussed in a later section called Key Rollover. [↑](#footnote-ref-11)
12. This is based on a longest-prefix match. The rationale behind this approach is explained later in the paper. [↑](#footnote-ref-12)
13. If you're using ASP.NET to build your service, consider storing your key in an encrypted section of your web.config file. To learn more, see [this HOWTO](http://msdn.microsoft.com/en-us/library/ms998283.aspx) from the Patterns & Practices group. [↑](#footnote-ref-13)