David Chappell & Associates

INTRODUCING "GENEVA"

AN OVERVIEW OF THE "GENEVA" SERVER, CARDSPACE "GENEVA", AND THE "GENEVA" FRAMEWORK

DAVID CHAPPELL

OCTOBER 2008

SPONSORED BY MICROSOFT CORPORATION

CONTENTS

Understanding Claims-Based Identity 3

The Problem: Working with Identity in Applications 3

The Solution: Claims-Based Identity 4

Creating Claims 4

Using Claims 6

The Role of "Geneva" 9

Applying Claims-Based Identity and "Geneva" 11

Using Claims inside an Enterprise 11

Using Claims on the Internet 13

Using Claims between Enterprises 16

Using Claims with Delegation 19

A Closer Look at the "Geneva" Technologies 20

The "Geneva" Server 20

CardSpace "Geneva" 22

Information Cards 22

The Self-Issued Identity Provider 25

The "Geneva" Framework 27

Conclusions 27

About the Author 28

UNDERSTANDING CLAIMS-BASED IDENTITY

For people who create software today, working with identity isn't much fun. First, a developer needs to decide which identity technology is right for a particular application. If the application will be accessed in different ways, such as within an organization, across different organizations, and via the public Internet, one identity technology might not be enough—the application might need to support multiple options. Next, the developer needs to figure out how to find and keep track of identity information for each of the application's users. The application will get some of what it needs directly from those users, but it might also need to look up other information in a directory service or someplace else.

This is all more complex than it needs to be. Why not create a single interoperable approach to identity that works in pretty much every situation? And rather than making applications hunt for identity information, why not make sure that this single approach lets users supply each application with the identity information it requires?

Claims-based identity achieves both of these goals. It provides a common way for applications to acquire the identity information they need from users inside their organization, in other organizations, and on the Internet. Along with making the lives of developers significantly simpler, a claims-based approach can also lower the cost of building and managing applications.

Making claims-based identity real requires developers to understand how and why to create claims-based applications. It also requires some infrastructure software that applications can rely on. This overview describes the basics of claims-based identity, then looks at how a group of forthcoming Microsoft technologies—the "Geneva" Server, Windows CardSpace "Geneva", and the "Geneva" Framework—help make this world a reality. All three are still in beta, so be aware that some things might change before their final release. Still, it's not too soon to begin understanding how this future looks and how we're going to get there.

THE PROBLEM: WORKING WITH IDENTITY IN APPLICATIONS

Sometimes, working with identity is simple. Think of a Windows application that doesn't need to know much about its users, for example, and that will be accessed only by users within a single organization. This application can just rely on Kerberos, part of Active Directory Domain Services (AD DS, formerly known as just "Active Directory"), to authenticate its users and convey basic information about them. Or suppose you're creating an application that will be accessed solely by Internet users. Again, the common approach to handling identity is straightforward: require each user to supply a username and password, then maintain a database of this user information.

Yet these simple scenarios quickly break down. What if you need more information about each user than is provided by either Kerberos or a simple username and password? Your application will now need to acquire this information from some other source, such as AD DS, or keep track of the information itself. Or suppose the application must be accessed both by employees inside the organization and by customers via the Internet—what now? Should the application support both Kerberos and username/password-based logins? And what about the case where you'd like to let users from a business partner access this organization without requiring a separate login? This kind of *identity federation* can't be accomplished very well with either Kerberos or username/password logins—more is required.

The right solution is to have one approach to identity that works in all of these scenarios. To be effective, this single approach must be based on widely recognized industry standards that interoperate across both platform and organizational boundaries. But standards alone aren't enough. The solution also needs to be widely implemented in products from multiple vendors and be simple for developers to use. This unified, broadly supported approach is exactly what claims-based identity is meant to provide.

THE SOLUTION: CLAIMS-BASED IDENTITY

Claims-based identity is a straightforward idea, founded on a small number of concepts: claims, tokens, identity providers, and a few more. This section describes the basics of this technology, starting with a look at these fundamental notions.

Before launching into this description, however, there's an important point to make. While this paper focuses on the mechanics, using the technology described here can require more, such as business agreements between different organizations. Addressing the technical challenges is essential, but they're not always the entire story.

Creating Claims

What is an identity? In the real world, the question is hard to answer—the discussion quickly veers into the metaphysical. In the digital world, however, the answer is simple: A digital identity is a set of information about somebody or something. While all kinds of entities can have digital identities, including computers and applications, we're most often concerned with identifying people. Accordingly, this overview will always refer to things with identities as "users".

When a digital identity is transferred across a network, it's just a bunch of bytes. It's common to refer to a set of bytes containing identity information as a *security token* or just a *token*. In a claims-based world, a token contains one or more *claims*, each of which carries some piece of information about the user it identifies. Figure 1 shows how this looks.

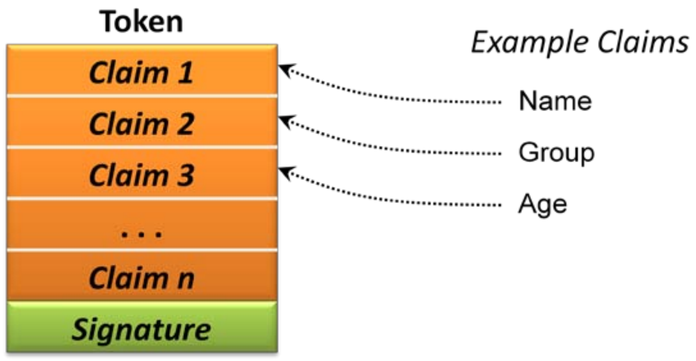


Figure 1: A token contains claims about a user along with a digital signature that can be used to verify its issuer.

Claims can represent pretty much anything about a user. In this example, for instance, the first three claims in the token contain the user's name, an identifier for a group she belongs to, and her age. Other tokens can contain other claims, depending on what's required. To verify its source and to guard against unauthorized changes, a token's issuer digitally signs each token when it's created. As Figure 1 shows, the resulting digital signature is carried with the token.

But who issues tokens? In a claims-based world, tokens are created by software known as a *security token service (STS)*. Figure 2 illustrates the process.

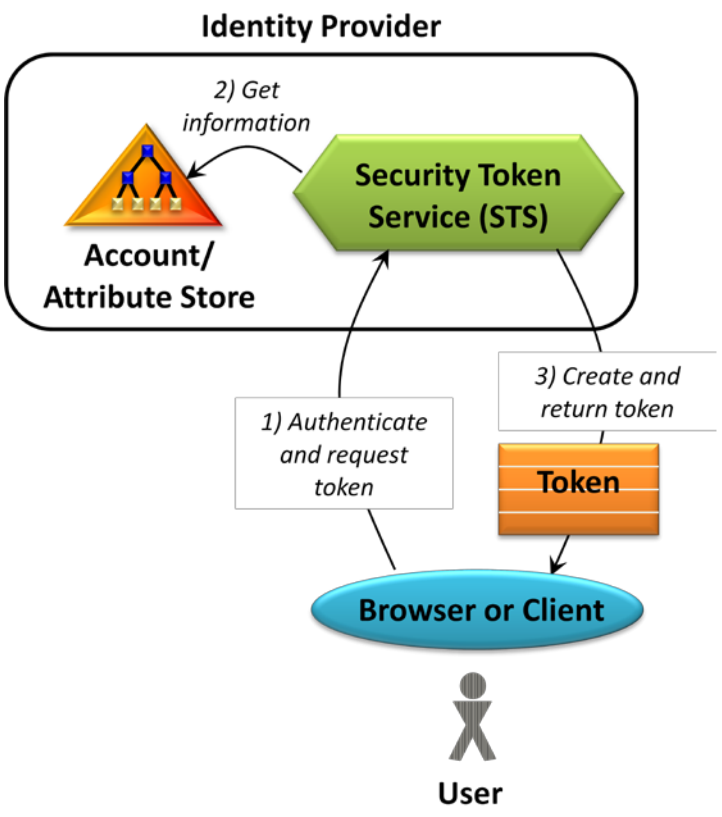

Identity Provider
2) Get information
Security Token Service (STS)
Account/ Attribute Store
3) Create and return token
1) Authenticate and request token
Token
Browser or Client
User

Figure 2: A user acquires a token containing some set of claims from an STS.

In a typical scenario, an application working on behalf of a user, such as a Web browser or another client, asks an STS for a token containing claims for this user (step 1). This request is made using the standard protocol WS-Trust. (In fact, support for WS-Trust is one of the defining characteristics of an STS.) This request is authenticated in some way, such as by providing a Kerberos ticket, a password from the user, or something else. The request typically contains both the name of the user for whom this token should be issued and a URI identifying the application the user wishes to access. The STS then looks up information about the user and the application in a local database (step 2). As the figure shows, this database maintains account information and other attributes about users and applications. Once the STS has found what it needs, it generates the token and returns it to the requester (step 3).

As Figure 2 shows, an STS is owned by some *identity provider* (sometimes called an *issuer*). The identity provider is what stands behind the truth of the claims in the tokens an STS creates. In fact, this is why the contents of a token are called "claims": They're assertions that this identity provider claims are true. The application that receives this token can decide whether it trusts this identity provider and the claims it makes about this user.

Identity providers come in many forms. If you use a token issued by an STS on your company's network, for example, the identity provider is your company. If you use a token issued by the STS provided by Microsoft's Windows Live ID service on the Internet, this Microsoft service is acting as the identity provider. It's even possible to act as your own identity provider, a handy option that's described later.

Whoever the identity provider is, being able to acquire and use a token full of claims is useful. In the pre-claims world (that is, in the world we mostly live in today), an application usually gets only simple identity information from a user, such as her login name. All of the other information it needs about that user must be acquired from somewhere else. The application might need to access a local directory service, for instance, or maintain its own application-specific database. With claims-based identity, however, an application can specify exactly what claims it needs and which identity providers it trusts, then expect each user to present those claims in a token issued by one of those providers. A claims-aware application is still free to create its own user database, of course, but the need to do this shrinks. Instead, each request can potentially contain everything the application needs to know about this user.

Claims can convey a variety of information. As Figure 1 showed, a claim might contain traditional things such as a user's name and group memberships, generally useful information such as her address, or other descriptive data such as her age. A claim might also identify the roles a user can act in, providing more information that the application can use to make an access control decision. Yet another possibility is to use a claim to indicate explicitly the user's right to do something, such as access a file, or to restrict some right, such as setting an employee's purchasing limit. Because an application can count on getting the identity information it needs in a token, claims-based identity makes life simpler for application developers.

This approach also brings one more benefit: It gets developers out of the business of authenticating users. All the application needs to do is determine that the token a user presents was created by an STS this application trusts. How the user proved its identity to this STS—with a password, a digital signature, or something else—isn't the application's problem. This lets the application be deployed unchanged in different contexts, a significant improvement over the usual situation today.

Using Claims

Claims, tokens, identity providers, and STSs are the foundation of claims-based identity. They're all just means to an end, however. The real goal is to help a user present her digital identity to an application, then let the application use this information to decide what she's allowed to do. Figure 3 shows a simple picture of how this happens.

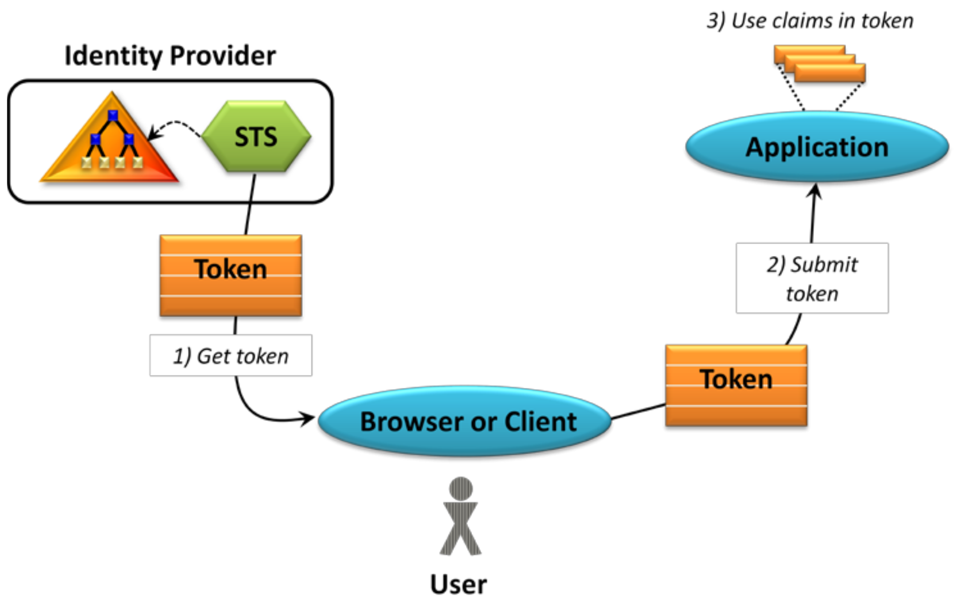


Figure 3: A browser or other client can acquire a token from an STS, then present this token and the claims it contains to an application.

As the figure shows, a Web browser or other client acting on behalf of a user gets a token for a particular application from an STS that's owned by some identity provider (step 1). Once it has this token, the browser or client sends it to the application (step 2), which attempts to verify its signature. If this verification works, the application knows which STS, and thus which identity provider, issued the token. If the application trusts this identity provider, it assumes the claims in the token are correct and uses them to decide what the user is allowed to do (step 3).

If the token contains the user's name, for example, the application can assume that the user really is who she claims to be. Since the user was required to authenticate herself to get this token, as described earlier, the application doesn't need to authenticate her again. In fact, because it relies on the claims in the token, an application is sometimes referred to as a *relying party*.

Although it's not shown in the figure, there's an essential first step before any of this can happen: An administrator must configure the STS to issue the right claims for this user and this application. Without this, there's no way for the STS to create a token containing the claims that the application needs. While doing this might seem like a burden, the reality is that this information must also be configured in the non-claims-based world. The big difference is that now the claims are all in one place, accessible through the STS, rather than spread across different systems.

The implicit assumption in Figure 3 is that the user has just one digital identity that she uses for all applications. The truth, though, is that she probably wishes to send different identity information to different applications. Think about how this works in the real world: You show your passport to a border guard, but give your driver's license to a traffic cop. Neither will accept the identity demanded by the other, because different situations require presenting different information from different sources. Passports are issued by national governments, while driver's licenses might be issued by some more local entity, such as a state government. The analog in the digital world is relying on different identity providers, each offering an STS that issues tokens containing appropriate claims. The claims in these tokens vary, just as the information in your passport is different from what's in your driver's license.

If we're all going to have multiple digital identities—and we are—it would be useful to have a consistent way to select the identity we want to use to access a particular application at a particular time. In other words, we'd like to have an *identity selector*. Figure 4 shows where this component fits.

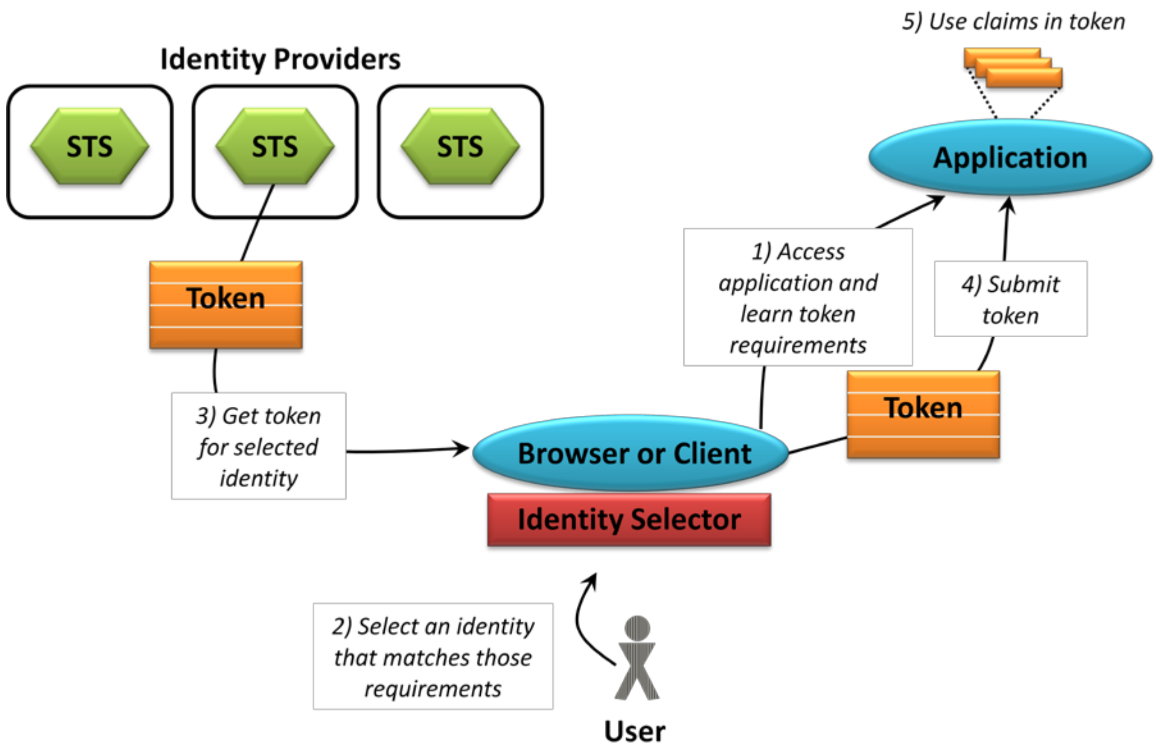

5) Use claims in token
Identity Providers
STS
STS
STS
Application
Token
1) Access application and learn token requirements
4) Submit token
Token
3) Get token for selected identity
Browser or Client
Identity Selector
2) Select an identity that matches those requirements
User

Figure 4: An identity selector provides a consistent way for users to choose which identity they wish to present to an application.

In this more complete illustration, the process begins when the user accesses the application. Whether it's contacted from a browser or some other client, the application can indicate what kind of token it requires, what kind of claims that token should contain, and what identity providers it trusts (step 1). As always in a claims-based world, the application can do this in a vendor-neutral way, using either WS-SecurityPolicy (for requests made via SOAP) or HTML (for requests made via HTTP) to describe these requirements. Once the user's system has this information, its identity selector can present the user with a visual representation of her available identities that meet these requirements. The user selects one of these (step 2), and the identity selector contacts the appropriate identity provider to get a token for this identity (step 3). Once it has the token, the browser or client sends it to the application (step 4), which verifies it, then uses the claims it contains (step 5).

While claims-based identity does specify important aspects of these interactions, such as how tokens are requested from an STS, this approach explicitly omits defining other things. For example, the claims-based approach doesn't mandate any particular format for tokens. It's common today to use tokens defined using the XML-based Security Assertion Markup Language (SAML), but this isn't required. Any token format that an application and an STS agree on can be used.

The Role of "Geneva"

Making claims-based identity real requires several things. STSs must be available, or there will be no place to get tokens. An identity selector would also be nice to have, since without one, users are probably restricted to a single identity. And finally, developers will need to build *claims-aware* applications that know how to receive tokens and use the claims they contain. Rather than making every developer write this code from scratch, it would make sense to provide a standard library that any application could use.

These three things are exactly what the "Geneva" Server, CardSpace "Geneva", and the "Geneva" Framework provide. Figure 5 shows where each of these technologies fits.

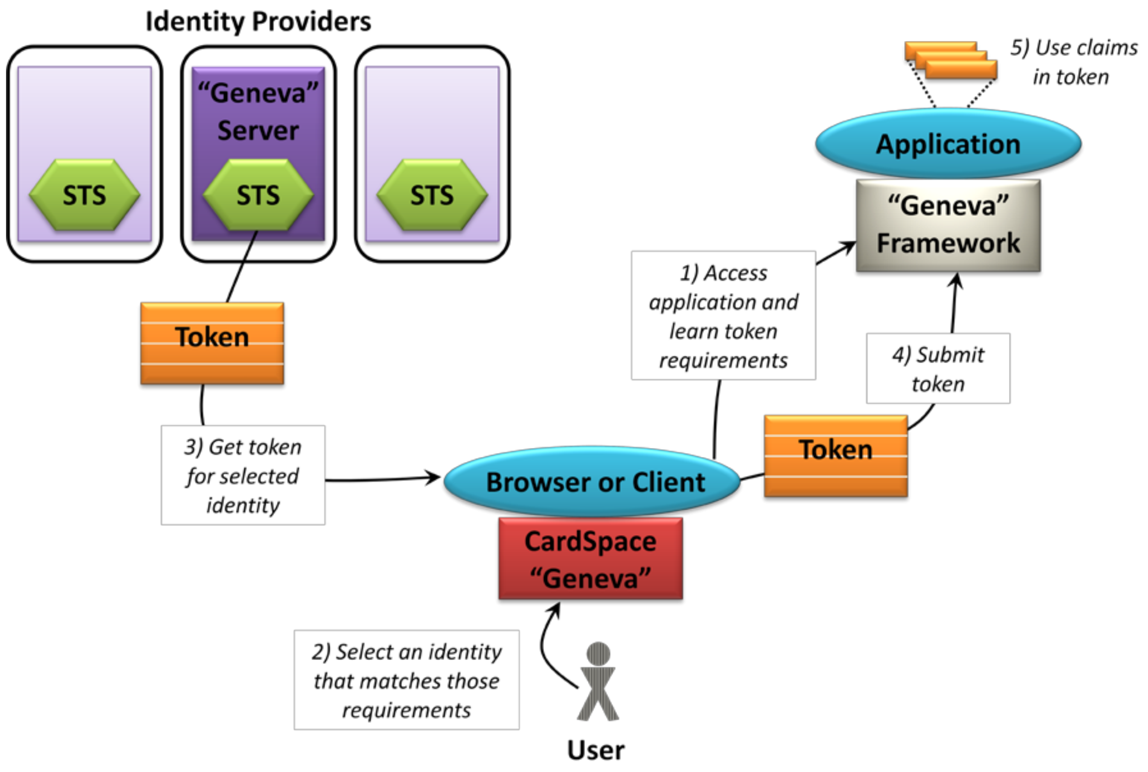


Figure 5: The "Geneva" Server implements a Windows-based STS, CardSpace "Geneva" provides an identity selector for Windows clients, and the "Geneva" Framework is a standard library for creating claims-aware Windows applications.

This figure is a replica of Figure 4; the only difference is that the "Geneva" Server is shown as one of the STSs, CardSpace "Geneva" is shown as the identity selector, and the application is built using the "Geneva" Framework. All three technologies are described in more detail later in this overview, but it's worth looking at the basics of each one here.

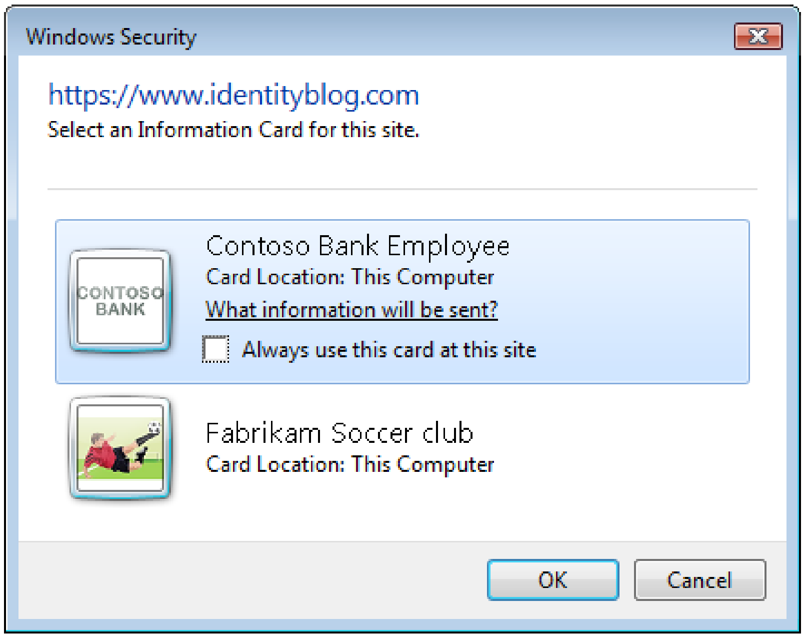
The "Geneva" Server is the next release of Microsoft's Active Directory Federation Services (AD FS). Don't be misled by the word "federation" in the original name of this technology, however. While the "Geneva" Server does support identity federation, it also provides broad support for claims-based identity. For example, unlike its predecessor, the "Geneva" Server implements an STS that generates SAML tokens in response to WS-Trust requests. Also unlike AD FS, which supported only Web browsers, the "Geneva"

Server supports both browsers and other clients, such as those built using Windows Communication Foundation (WCF). (In the jargon of identity, the "Geneva" Server supports both *active* and *passive* clients, while AD FS supported only passive clients.) Another important difference from the original AD FS is that the "Geneva" Server supports both WS-Federation and the SAML 2.0 protocol, letting it work in a broader range of environments.

The "Geneva" Server STS can potentially be used by any identity provider, whether it's inside an organization, exposed on the Internet, or both. Yet it's important to understand that using claims-based identity doesn't require using the "Geneva" Server. As Figure 5 suggests, any STS from any vendor, or even a custom-built STS, can be used. Still, one of Microsoft's primary goals in providing the "Geneva" Server is to make widely available a fully-featured STS built on AD DS. Until STSs are common, the benefits of claims-based identity are unlikely to materialize.

CardSpace "Geneva" is also the successor to an existing Microsoft technology, the original CardSpace. This identity selector can be used both with Web browsers, including Internet Explorer and Firefox, and with other Windows clients, such as WCF applications. And while an STS is fundamental to a claims-based world, using an identity selector isn't required—claims-based identity can still work without one. Yet without CardSpace "Geneva" or a similar technology, users will have no consistent way to select which identity they wish to use. Even though an identity selector isn't strictly required, it's hard to imagine an effective claims-based world without one.

An important aspect of an identity selector is its user interface. Letting users select their identity in the same way for every application and every kind of client can greatly simplify their lives. Toward this end, CardSpace "Geneva" provides the standard screen shown in Figure 6.


Windows Security

https://www.identityblog.com

Select an Information Card for this site.

Contoso Bank Employee

Card Location: This Computer

What information will be sent?

☐ Always use this card at this site

Fabrikam Soccer club

Card Location: This Computer

Figure 6: CardSpace "Geneva" provides a common user interface for selecting identities, representing each identity with a card.

Each identity is represented by a card (hence the name "CardSpace"). Each card is associated with a particular identity at some identity provider. Clicking on a card causes CardSpace "Geneva" to request a token for this identity from the associated identity provider, perhaps prompting the user for a password or something else to authenticate the request. While the software exchanges tokens containing claims, a user sees only this simpler metaphor of cards.

The third component required to make claims-based identity a reality, at least for Windows applications, is the “Geneva” Framework. This library is a set of .NET Framework classes that implement basic functions, such as receiving a token, verifying its signature, accessing the claims it contains, and more. For situations where the “Geneva” Server STS isn’t sufficient, the “Geneva” Framework also provides support for building your own STS. One important example of this already exists: The “Geneva” Server itself is built on the “Geneva” Framework.

It’s important to realize that because all interaction is done in a standard way, none of these technologies specifically requires any of the others. The “Geneva” Server doesn’t require CardSpace “Geneva”, CardSpace “Geneva” doesn’t require the “Geneva” Server, and neither one requires applications to use the “Geneva” Framework. To CardSpace “Geneva”, for example, the “Geneva” Server looks like any other STS, with token requests sent using the standard WS-Trust protocol. Even though all of these technologies are from Microsoft, there are no proprietary links between them—all of the communication is based on industry standards. The goal is to make it easier to use claims-based identity both within the Windows world and across platforms from different vendors.

APPLYING CLAIMS-BASED IDENTITY AND “GENEVA”

Getting your mind around claims-based identity requires understanding the basics of this technology. Still, the best way to get a feel for this approach is to walk through examples of how it can be applied. Accordingly, this section looks at several different ways this technology can be used.

Without claims-based identity, application developers are faced with a diverse set of scenarios, each with its own identity solution. One big contribution of claims-based identity is to collapse all of these down to just one problem: How does an application get information about a user from a trusted source? As this section shows, claims-based identity provides a consistent answer across a range of scenarios.

At the risk of being redundant, it’s important to emphasize that even though these examples show Microsoft technologies, this isn’t required. Products from other vendors, such as IBM’s Tivoli Federated Identity Manager, also provide STSs today. Similarly, other identity selectors are available, including the open source Higgins implementation, as well as other libraries to create claims-aware applications. The key point is that while Microsoft is an important player, the move toward claims-based identity is an industry-wide, multi-vendor endeavor.

USING CLAIMS INSIDE AN ENTERPRISE

Every enterprise acts as an identity provider, and virtually every enterprise application must deal with identity. The “Geneva” Server, CardSpace “Geneva”, and the “Geneva” Framework can provide the foundation for using claims-based identity with applications running inside an organization. Figure 7 shows how this looks.

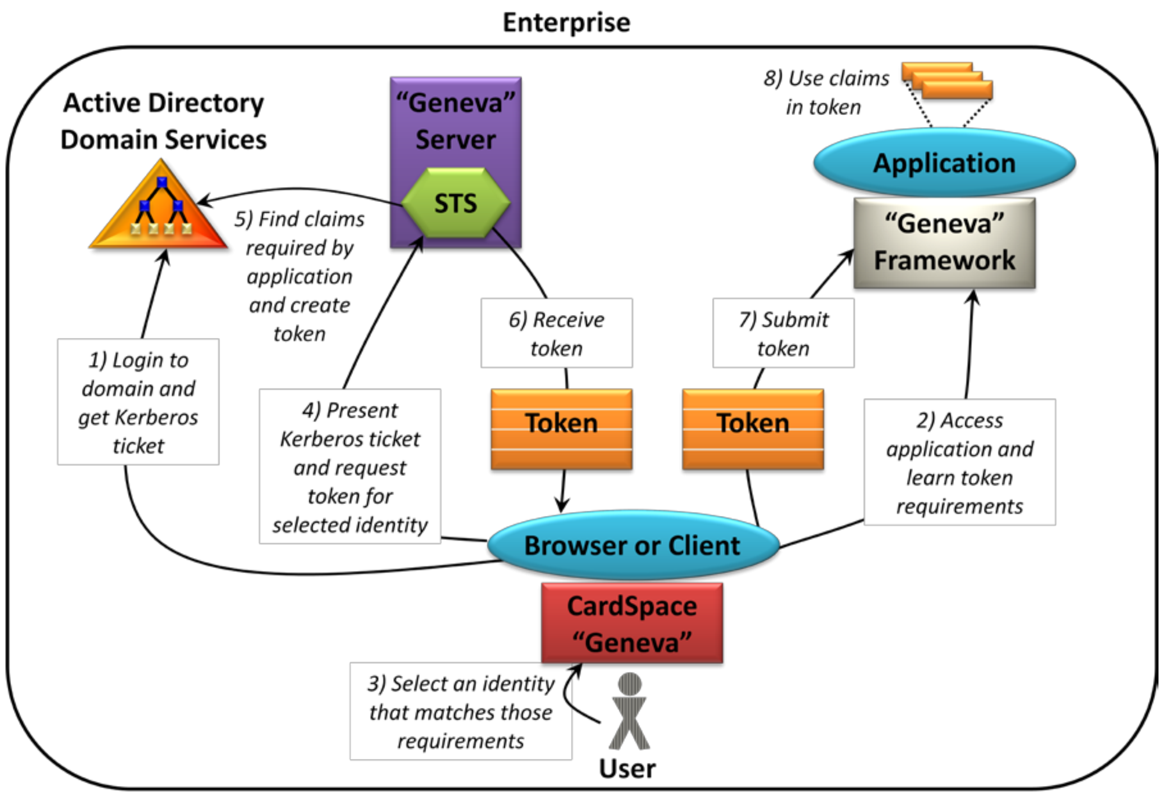

Enterprise
Active Directory Domain Services
"Geneva" Server
STS
8) Use claims in token
Application
"Geneva" Framework
5) Find claims required by application and create token
6) Receive token
7) Submit token
1) Login to domain and get Kerberos ticket
4) Present Kerberos ticket and request token for selected identity
Token
Token
2) Access application and learn token requirements
Browser or Client
CardSpace "Geneva"
3) Select an identity that matches those requirements
User

Figure 7: An enterprise can use Active Directory Domain Services, the "Geneva" Server, CardSpace "Geneva", and the "Geneva" Framework to support claims-based identity for its internal applications.

In this example, a user logs in using AD DS, getting an initial Kerberos ticket (step 1). The user can then access a claims-aware application built using the "Geneva" Framework, learning what kinds of tokens it accepts and what claims those tokens must contain (step 2). If CardSpace "Geneva" is used (it's not required, remember), the user might then see the screen shown earlier in Figure 6 and select an identity by choosing a card (step 3). CardSpace "Geneva" will then request a token for this identity, supplying a Kerberos ticket to authenticate the user (step 4). The "Geneva" Server STS verifies the ticket, then looks in AD DS for the information it needs to create the requested token (step 5). Exactly what claims appear in this token depend on both the user requesting it and the application that user is accessing—each application indicates exactly what claims it needs. Once the token has been created, the "Geneva" Server STS sends it back to the user's system (step 6), which sends it on to the application (step 7). The application uses the "Geneva" Framework to verify the token's signature and make its claims available for use (step 8).

One big plus of a claims-based approach is worth re-emphasizing here: Rather than having to go look for the information it needs about a user, the application can instead get everything handed to it in the token. If the application needs, say, the user's job title, it can specify this in its list of required claims. When the STS creates a token for the application, it finds the user's job title in AD DS and inserts it as a claim that the application can use. Without this, the application developer must write his own code to dig this information out of AD DS. Claims-based identity makes the developer's life significantly easier.

Along with easing the lives of developers, an STS also performs another function: It acts as a *claims transformer*. When the client requests a token from the “Geneva” Server in Figure 7, for instance, it provides a Kerberos ticket. This ticket can be thought of as a token containing a simple set of claims (the user’s name and group memberships). The “Geneva” Server STS uses this token to authenticate the user making the request, then emits another token. This new token is in a different format—it’s a SAML token rather than a Kerberos ticket—and it probably contains a different set of claims, since it can include whatever the target application has specified. In a very real sense, the STS has transformed one set of claims into another.

USING CLAIMS ON THE INTERNET

Now suppose this organization wishes to make the same application accessible to remote employees via the Internet. Rather than modifying the application to accept username/password logins, a traditional solution, the same claims-based approach can be used—the application remains unchanged. Figure 8 illustrates this scenario.

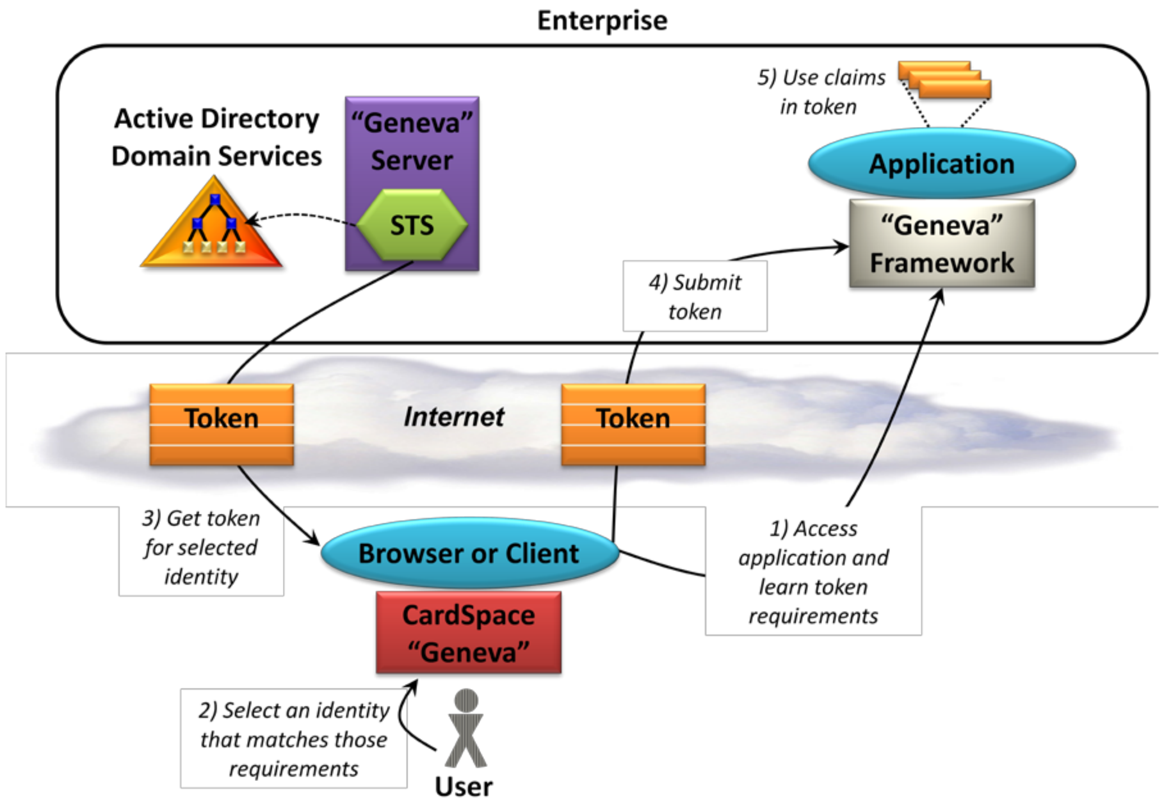

Enterprise
Active Directory Domain Services
“Geneva” Server
STS
5) Use claims in token
Application
“Geneva” Framework
4) Submit token
Token
Internet
Token
3) Get token for selected identity
Browser or Client
1) Access application and learn token requirements
CardSpace “Geneva”
2) Select an identity that matches those requirements
User

Figure 8: An enterprise can use the “Geneva” Server STS to create tokens for users on the Internet.

Here, the user is on another computer outside the enterprise. As before, this user accesses the application and learns what kinds of tokens it will accept (step 1). Using CardSpace “Geneva” (which is useful but not required), the user selects an identity that meets these requirements (step 2). The user’s system then gets a token for this identity from the enterprise’s STS, implemented using the “Geneva” Server (step 3). It can then submit this token to the application (step 4), which relies on the “Geneva” Framework to verify the token and uses the claims it contains (step 5). Rather than requiring a different way of handling identity for Internet access, as is common today, a claims-based approach allows handling this situation just like the inside-the-enterprise case.

Still, some extra complexity creeps in. When the user requests a token in step 3, for example, how does she authenticate herself to the STS? Kerberos tickets work just fine for users inside the enterprise, as shown earlier in Figure 7, but they don't work well for Internet users. Instead, the user might provide a username and password in step 3 to authenticate her request, an option that Microsoft plans to support in the "Geneva" Server. Since the users in this scenario are employees, they already have accounts in AD DS, and so they can log in with no trouble.

Yet what if the users aren't employees? Suppose the application needs to be exposed via the Internet to customers as well. Can this approach still work? The answer, unsurprisingly, is yes. Although it's not an especially common option, information about external users can be mingled with employee accounts in AD DS, letting it be accessed by the "Geneva" Server. Alternatively, external user account and attribute information can be stored in Active Directory Lightweight Directory Services (AD LDS). Formerly known as Active Directory Application Mode (ADAM), this technology provides a simpler directory service that's also an option for the "Geneva" Server.

But wait a minute: If Internet users still need usernames and passwords, how is the claims-based approach making things better? There are a couple of answers. First, recognize that users no longer have a password for each application. Instead, they'll (at most) have one for each STS they use. This frees applications from the need to store sensitive password information, moving that responsibility instead to the much smaller number of STSs. Also, since requests for tokens are made directly from CardSpace "Geneva"—the user never enters a URL for the STS—phishing for these passwords becomes more difficult. There's no way for an attacker to slip in a spurious URL for its own STS. While claims-based identity doesn't necessarily eliminate usernames and passwords, it nonetheless improves the situation.

In the case shown in Figure 8, the organization that provides the application is also acting as the identity provider. While this makes sense in many situations, there are other scenarios in which the identity provider is an external organization. For example, Microsoft today offers Windows Live ID as an Internet-accessible STS, and other identity providers also exist. Rather than implementing its own identity provider (or perhaps along with it), an organization can create an application that accepts tokens from external providers like these. Figure 9 shows how this looks.

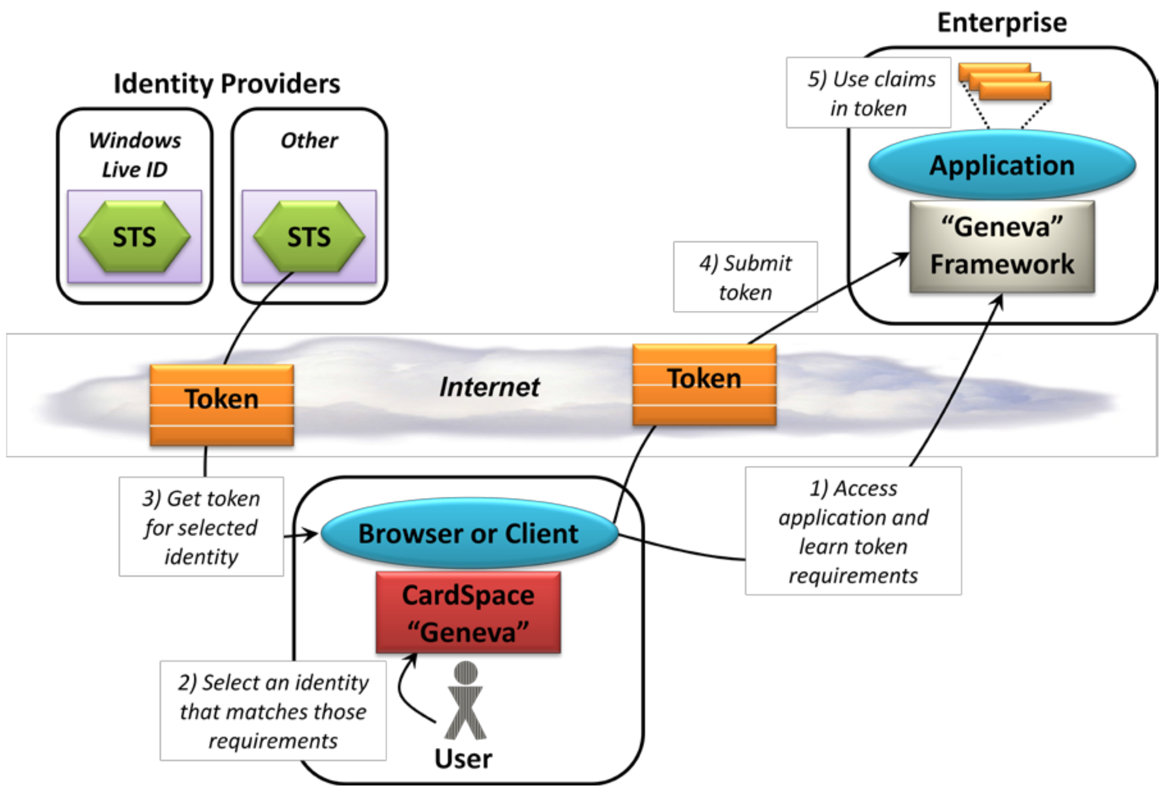


Figure 9: An application can accept tokens issued by identity providers run by external organizations.

As in the example shown in Figure 8, the process begins with the user accessing the application (step 1), then choosing an identity (step 2). This time, however, the token for this identity is provided by an STS run by an outside identity provider (step 3). Once it has the token, the user's system submits it to the application as usual (step 4), which uses the claims it contains (step 5).

Don't be confused: Even though one of the external identity providers in this example is run by Microsoft, neither CardSpace "Geneva" nor claims-based identity in general are bound to a particular provider. Anybody who implements an STS can act as an identity provider, assuming they can convince applications to trust the claims in tokens they issue.

Whether an Internet-accessible application trusts an outside identity provider or only one run by its own organization, a claims-based approach is attractive. It allows handling identity in a consistent way for users inside and outside the firewall. It also gets applications out of the business of maintaining username/password databases for Internet users, making phishing less effective. Along with making developers' lives easier, claims-based identity can also make things simpler for users. For example, since a token can carry whatever claims an application specifies, users can more easily submit the common information requested by Web sites—they need no longer type it in at each site. While the behavior of thousands of application developers and millions of users won't change overnight, the long-term prognosis is positive.

USING CLAIMS BETWEEN ENTERPRISES

Another common identity challenge is letting users in one organization access an application running in some other organization. For example, suppose your company wishes to make an internal SharePoint site accessible to employees at a partner firm. One way to do this is to give each of these external users their own account in your company. While this approach works, it's unappealing. Those users won't like having a separate login, and your firm's administrators won't like having to administer accounts for people outside your company. Doing this also creates security risks—how can your administrators know when an external user has left his company and so should no longer have an account?

A better solution is to let the external users access your application using their own identities. This approach requires no separate logins and no new accounts. What it does require, however, is creating a federation relationship between your firm and its partner. Doing this will likely require some kind of legal agreement between the two organizations, a topic that's beyond the scope of this discussion. It also, of course, requires putting in place the right technology.

AD FS, Microsoft's predecessor to the "Geneva" Server, allowed identity federation for passive clients (that is, for browsers), but not for active clients. The "Geneva" Server still supports the AD FS-style passive option, which relies on a standard called WS-Federation. (If you're interested in how it works, see *Digital Identity for .NET Applications: A Technology Overview*, available at http://msdn.microsoft.com/en-us/library/bb882216.aspx .) With CardSpace "Geneva" and STSs, however, claims-based identity can also be used for federated access by active clients. This lets the same identity technology be used in yet another important scenario, and it also offers users more control than did AD FS over which identity they use.

One approach to providing federated identity in a claims-based world is to configure an application running in one organization to trust an STS in another. Figure 10 shows how this looks.

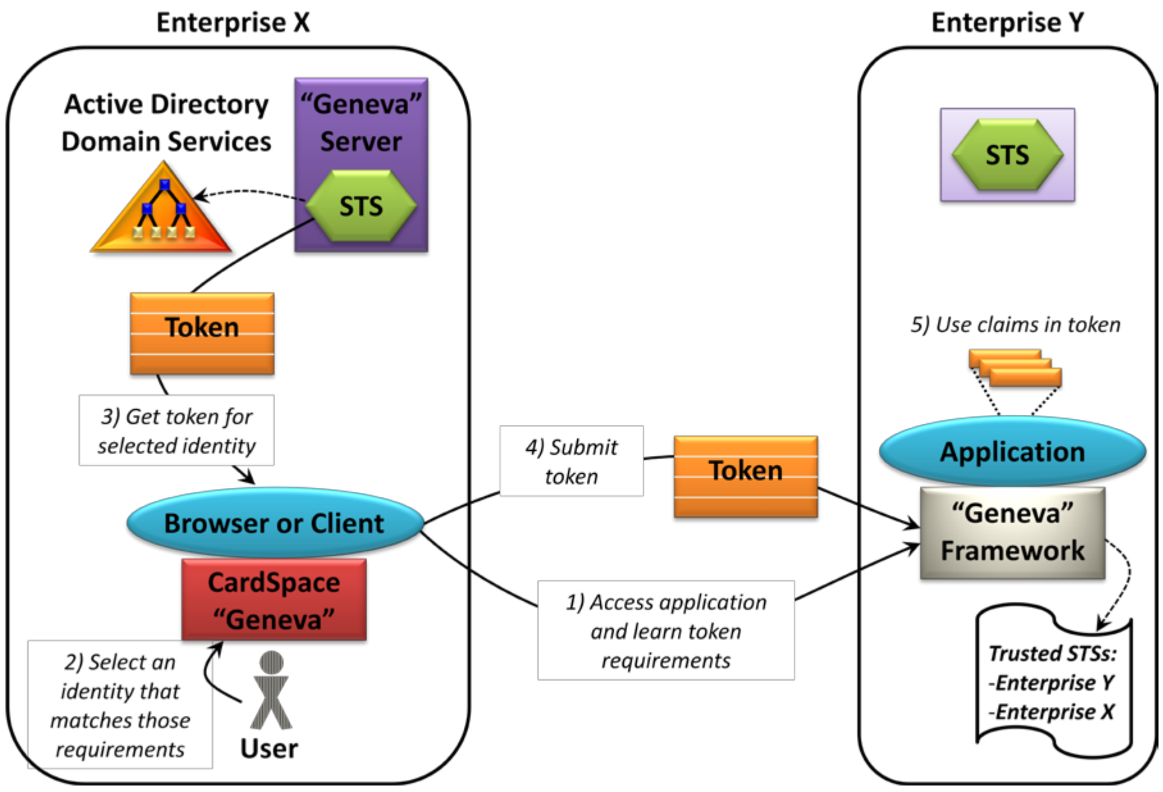


Figure 10: If the application trusts the STS in the client's enterprise, it can accept and use a token issued by that STS.

In this scenario, a user in enterprise X accesses an application in enterprise Y and learns its token requirements (step 1). Here, that application is configured to trust both its own STS, the one in enterprise Y, and the STS in enterprise X. All that's required is for the user in enterprise X to choose an identity that matches this application's requirements (step 2), then get a token for that identity from its own STS (step 3). The browser or client submits this token to the application (step 4), which uses the "Geneva" Framework to verify the token and extract its claims. The application can then use these claims any way it likes (step 5).

This solution is simple, but it's not without problems. Suppose this application has users in several different enterprises, for instance. With the approach shown in Figure 10, the application would need to be configured to trust the STS in each one, an unappealing prospect. A better solution is to let the mechanics of federated identity be handled by the STSs themselves. Doing this means that an application only needs to trust its own STS, making life significantly simpler for the people who build and administer it. Figure 11 shows this more likely situation.

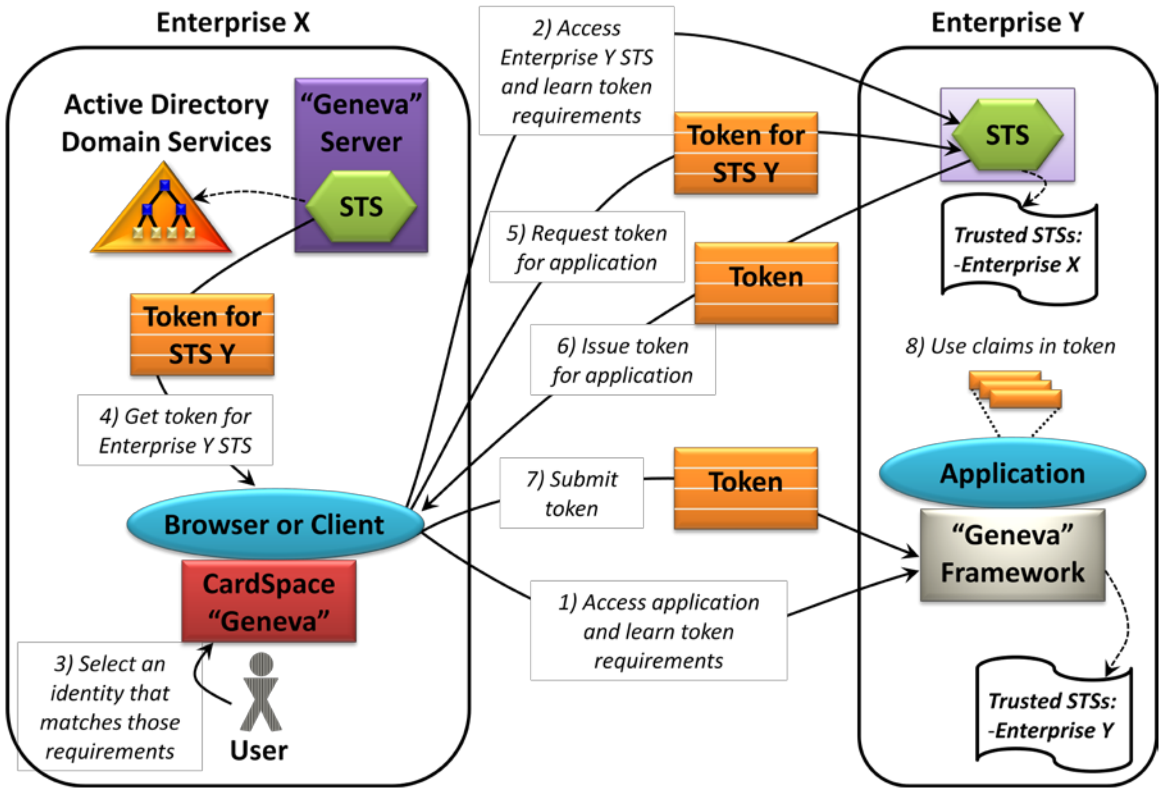

Enterprise X
Active Directory Domain Services
"Geneva" Server
STS
Token for STS Y
4) Get token for Enterprise Y STS
Browser or Client
CardSpace "Geneva"
3) Select an identity that matches those requirements
User
2) Access Enterprise Y STS and learn token requirements
Token for STS Y
5) Request token for application
Token
6) Issue token for application
7) Submit token
Token
1) Access application and learn token requirements
Enterprise Y
STS
Trusted STSs: -Enterprise X
8) Use claims in token
Application
"Geneva" Framework
Trusted STSs: -Enterprise Y

Figure 11: If the application trusts only the STS in its own enterprise, the client must get a token from that STS to access the application.

This scenario starts in the same way: The user in enterprise X accesses an application in enterprise Y and learns its token requirements (step 1). This time, however, that application is configured to trust only its own STS, the one in enterprise Y. Once it determines this, CardSpace "Geneva" on the user's system contacts the STS in enterprise Y to learn its token requirements (step 2). Along with its identity selector role, CardSpace "Geneva" also has built-in intelligence to traverse this kind of federation relationship. CardSpace "Geneva" then prompts the user to select an identity (i.e., a card) that matches those requirements (step 3) and requests a token for this identity from the enterprise X STS (step 4). This token contains claims about the user, but it's not a token that the application will accept—it was issued by an STS that the application doesn't trust. Instead, CardSpace "Geneva" submits this token to the STS in enterprise Y (step 5). This STS is configured to trust the STS in enterprise X, a relationship that must be established explicitly by administrators in the two organizations. (How this trust relationship gets created is described in a bit more detail later.) Because of this trust, the STS in enterprise Y can verify the token it receives from enterprise X, then issue a token that allows this user to access the application (step 6). The user presents this token to the application (step 7), and the application uses the "Geneva" Framework to verify the token and extract its claims. The application can now use these claims as usual (step 8).

It's worth pointing out that even though the "Geneva" Server was shown as one of the STSs in both of these federation scenarios, it's not required. CardSpace "Geneva" can communicate with any STS from any vendor. Note too that the STS in enterprise Y is acting as a claims transformer, accepting a token issued by STS X, then creating its own token. The contents of the token STS Y creates might well be different from those in the token it receives from STS X—it's free to add, remove, or modify the claims.

And finally, think again about how convenient it is for the application to get the information it needs about a user directly in the token. When both user and application are in the same organization, the application might be able to access, say, AD DS directly to get information such as the user's job title. When they're in different organizations, as in the federated case shown here, the application almost certainly won't be allowed to do this. Getting everything it needs handed to it in the user's token is very nice indeed.

USING CLAIMS WITH DELEGATION

Here's yet another identity challenge: Suppose one application needs to access another application on behalf of some user. Solving this problem requires *identity delegation*, where the application that receives a user's identity information is allowed to act as that user when accessing another application. While delegation has long been important, it becomes even more significant in a service-oriented world, where one service depends on another. Once again, claims-based identity can be used. Figure 12 shows how.

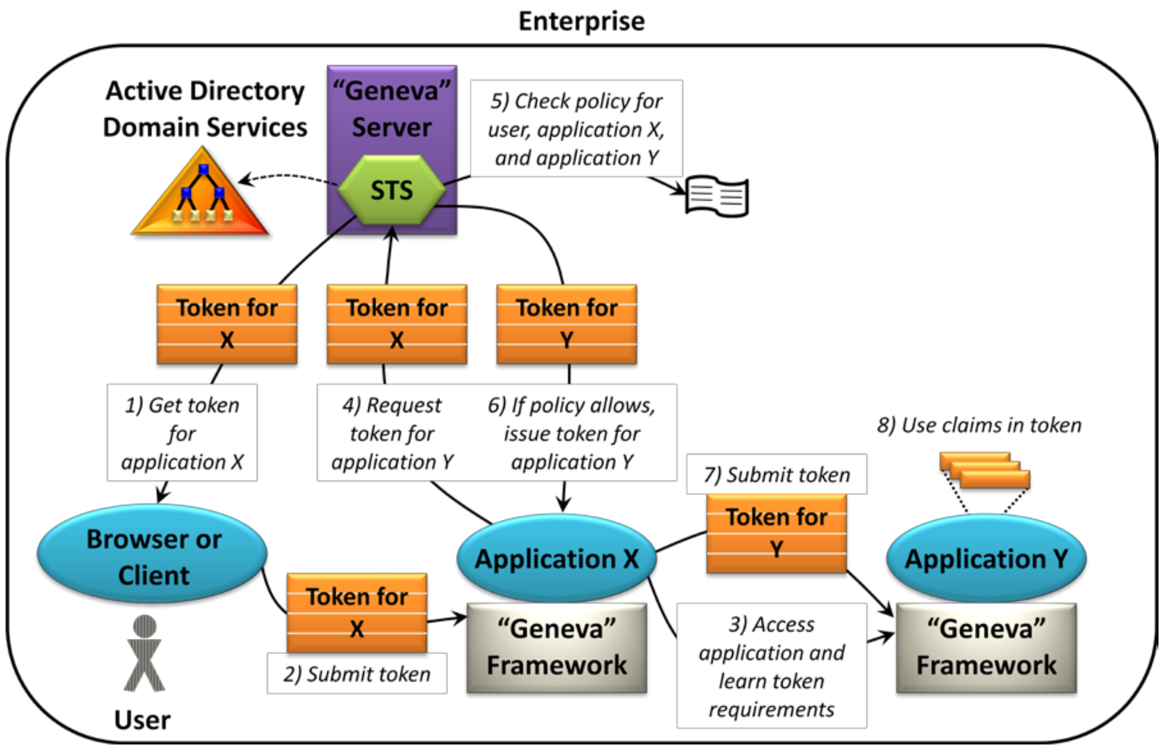

Enterprise
Active Directory Domain Services
"Geneva" Server
STS
5) Check policy for user, application X, and application Y
Token for X
Token for X
Token for Y
1) Get token for application X
4) Request token for application Y
6) If policy allows, issue token for application Y
8) Use claims in token
7) Submit token
Token for Y
Browser or Client
Application X
Application Y
Token for X
"Geneva" Framework
3) Access application and learn token requirements
"Geneva" Framework
2) Submit token
User

Figure 12: Claims-based identity can be used with delegation, where one application invokes another on a user's behalf.

This slightly simplified example begins with the user's browser or other client getting a token for application X (step 1). (Although it's not shown here, assume this relied on the usual steps shown in earlier figures.) This token is then sent to application X (step 2). To carry out the user's request, application X needs to invoke a service provided by application Y. X might be a Web application, for example, invoking a Web service in Y using WCF, or X itself might have been invoked via a Web service. In any case, application X needs a token for application Y that contains claims describing the original user.

To get this token, application X accesses application Y and learns its token requirements (step 3). Among these is an indication of an identity provider (and thus an STS) from which Y will accept a token. In this example, that STS is the same one that issued the token for application X, although this need not be the case. Here, however, application X can request a token for application Y from the STS they both share, indicating that it wishes to act as the original user (step 4). A “Geneva” Server STS can contain policies describing which applications are allowed to access which other applications using delegation. The STS checks this policy (step 5), which in this case allows application X to invoke application Y on behalf of this user. The STS then issues the requested token allowing X to access Y on this user’s behalf (step 6). Application X passes this token to Y when it invokes the service (step 7). Y then verifies the token using the “Geneva” Framework and uses its claims (step 8).

It’s worth comparing this claims-based style to today’s more common approaches to delegation. One alternative is for the user to pass her username and password directly to application X, which then uses these to access application Y as this user. Yet X now has the user’s login information and so might do other things that the user doesn’t approve of. With the claims-based approach, X has only a token for this user to access Y, nothing more. Another way to provide delegation is to make X a trusted subsystem, an application with complete access to other services. While this can work, granting application X this wide latitude requires placing a great deal of faith in the people who build and run it. This approach also makes it impossible for Y to know which users are accessing it, something that might be required for auditing or other reasons. Claims-based delegation lets X access Y on behalf of individual users, preserving Y’s knowledge of those users, with no need to grant broad trust to X.

Once again, notice that the STS is acting here as a claims transformer. It receives a token for X, then emits a token for Y. These two tokens probably contain different claims, since they’re for different applications, and they might even use different formats. (Applications X and Y might have been built at different times by different people, for example.) All of this variation is hidden from the application developer—and from the user—by the STS. This is yet another example of one of the primary goals of claims-based identity: making applications simpler.

A CLOSER LOOK AT THE “GENEVA” TECHNOLOGIES

Understanding how claims-based identity can be applied is important. It’s also useful to understand the software technology that underlies this idea. For Windows, this means understanding the “Geneva” Server, CardSpace “Geneva”, and the “Geneva” Framework. This section takes a closer look at each of these.

THE “GENEVA” SERVER

While the “Geneva” Server adds quite a bit to its predecessor AD FS, including a full-fledged STS, it also supports all of the functions of its earlier incarnation. For example, as mentioned earlier, the “Geneva” Server allows using WS-Federation to provide identity federation for passive clients (i.e., Web browsers). Unlike AD FS, however, the “Geneva” Server also supports using the SAML 2.0 protocol for this purpose, as mentioned earlier. Supporting this alternative protocol, which has been embraced by the Liberty Alliance and others, allows Windows systems with the “Geneva” Server to work with a broader range of identity federation products.

Also like AD FS, the "Geneva" Server allows an administrator to establish trust with other STSs. Figure 13 illustrates the fundamentals of how this works.

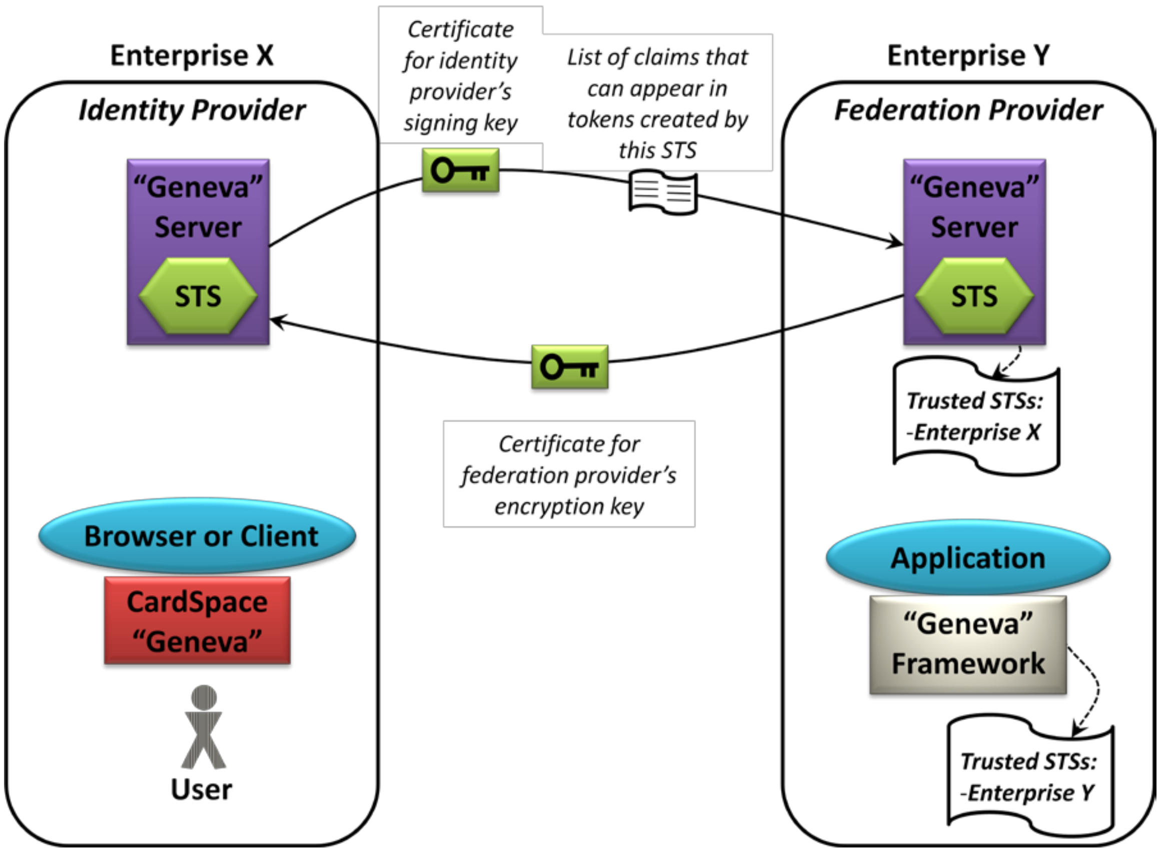

Enterprise X
Identity Provider
"Geneva" Server
STS
Browser or Client
CardSpace "Geneva"
User
Certificate for identity provider's signing key
List of claims that can appear in tokens created by this STS
Certificate for federation provider's encryption key
Enterprise Y
Federation Provider
"Geneva" Server
STS
Trusted STSs: -Enterprise X
Application
"Geneva" Framework
Trusted STSs: -Enterprise Y

Figure 13: Establishing a trust relationship between STSs requires exchanging certificates and more.

The situation shown here is identical to the one shown earlier in Figure 11: The application in enterprise Y only trusts tokens issued by its own STS. This means that a client in enterprise X must first get a token from its own STS, then use this to request a new token from the STS in enterprise Y, as described earlier. Accomplishing this requires addressing several issues.

For example, how does the STS in enterprise Y, called the *federation provider*, know that the token this client sends was actually issued by the STS in enterprise X, referred to as the *identity provider*? The answer is that this token is signed by the identity provider using its private signing key. To allow the federation provider to verify this signature, the identity provider sends it a certificate containing the corresponding public key for this signing key, as Figure 13 shows. Also, the federation provider is able to transform tokens it receives based on a transformation policy defined by its administrator. To define this policy, the federation provider must have a list of every possible claim type that the identity provider might send to it. As Figure 13 shows, the identity provider sends the federation provider a list of the claim types it can expect to receive.

Here's another problem: When the identity provider creates a token that's destined for the federation provider, it can encrypt this token so attackers can't read it. To allow this, the federation provider sends a certificate for its encryption key to the identity provider, as Figure 13 shows. The identity provider uses the public key in this certificate to encrypt all tokens it sends to this federation provider, ensuring that only the federation provider's STS can read them.

AD FS required similar exchanges to establish trust between federated domains. The "Geneva" Server makes this process simpler by automating what were entirely manual processes in AD FS. For example, when a certificate is about to expire, the "Geneva" Server can automatically create a new key pair and certificate, then send the certificate to its partner STS.

Another advance that the "Geneva" Server provides over AD FS is broader support for storing identity information. Unlike its predecessor, the "Geneva" Server views its account store, containing things like usernames and passwords, separately from its attribute store, which holds other information about users. For the account store, the "Geneva" Server supports either AD DS or AD LDS. For the attribute store, Microsoft plans to allow a range of options in the final release, including AD DS, AD LDS, SQL Server, and others (although not all of these are available in the first beta release).

CARDSPACE "GENEVA"

CardSpace "Geneva" is the second version of Microsoft's CardSpace technology. The basics are the same as in the original, with enhancements that reflect what CardSpace's designers have learned since its first release. This section takes a deeper look at how this technology works, including some of the most important changes in CardSpace "Geneva".

Information Cards

To a user, CardSpace "Geneva" represents each available identity as a card, as shown earlier in Figure 6. When a user selects a card, CardSpace "Geneva" requests a token from an STS at the corresponding identity provider. But how is the connection made between the card seen by a user and this STS? The answer is that each association between a card and an identity provided by some STS is represented by an *information card*. Figure 14 shows how this looks.

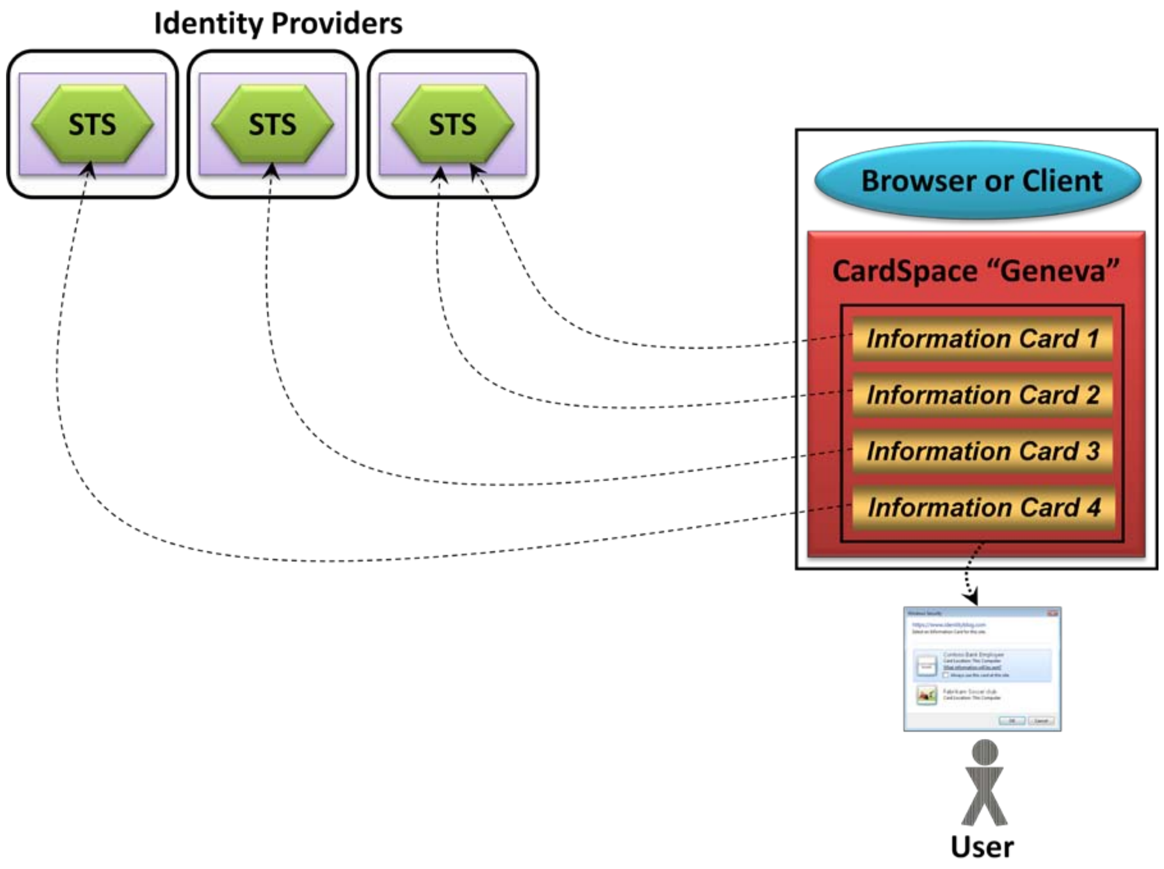


Figure 14: Each information card is associated with a particular digital identity at some identity provider.

An information card is just an XML file, and as the figure shows, each one represents a relationship with an identity provider. This relationship lets the user get tokens from the identity provider for use with applications willing to accept these tokens. The information card contains everything needed to find the right STS at the right identity provider, then request a token for the identity this card represents. The card doesn't contain any claims, however; these are all maintained by the identity provider. The sole purpose of the information card is to store the information needed to find the right STS and request a token for a particular identity.

The terminology can get confusing, so here's a quick recap: A user selects a card (a visual representation) that's associated with an information card (an XML file) that contains all of the information needed to request a token (a signed group of bytes issued by an STS). Don't confuse information cards with tokens—they're not the same thing.

Next question: Since every identity a user has is represented by an information card stored on the user's machine, how do the information cards get there? The answer is that it's up to the STS. Information cards don't contain confidential information—they don't hold a password the user supplies to authenticate token requests to the STS, for instance—so the problem isn't too challenging. The "Geneva" Server can install an information card on a user's machine in various ways, as can STSs from other vendors, such as IBM's Tivoli Federated Identity Manager. Similarly, other identity selectors that support information cards, such as the open source Higgins selector, can accept cards provided by the “Geneva” Server and other STSs.

Another challenge for information card-based identity is supporting roaming users. Many of us use a desktop computer at work, another one at home, and a laptop while we’re traveling, yet we’d like to present the same digital identity from all of them. To allow a user to roam among these different machines, CardSpace provides a *card export* feature. This option allows copying information cards onto an external storage medium, such as a USB key. The cards can then be installed onto other machines, letting a user request security tokens from identity providers in the same way whether he’s using his home computer, his office computer, or his laptop in a hotel room. To guard against attacks, exported information cards are encrypted using a key derived from a user-selected pass-phrase. This ensures that even if the storage medium is lost, only someone who knows the pass-phrase can decrypt the cards it contains. Microsoft is also investigating the possibility of letting a user store information cards on an Internet-accessible server, i.e., in the cloud. On a machine you control, CardSpace “Geneva” could then connect invisibly to these cloud-based information cards, using them to request tokens. This would make it easier to share identities among the small set of machines that each of us use regularly.

Yet another issue that CardSpace “Geneva” must address is revocation. Once an identity provider has issued an information card to a user, how can that card be revoked? In the simplest case, the identity provider itself might wish to stop issuing security tokens based on this card. Perhaps using this identity provider requires a paid subscription, for example, and the user hasn’t kept up his payments. Revocation in this case is simple: the identity provider just stops honoring requests for security tokens made with this card. Every request carries a unique CardSpace reference, so it’s easy for the identity provider to identify requests made with cards it has revoked.

A slightly more complex case is when the user wishes to revoke an information card. Perhaps an attacker has stolen the user’s laptop containing information cards issued by various identity providers. To address this problem, each information card can be assigned a PIN that must be entered each time the card is used. If this is done, an attacker can’t use the stolen cards (and thus the identities they correspond to) unless he knows the PIN for each one. Also, recall that using a card issued by an external identity provider typically requires the user to authenticate himself, such as by entering a password. Since this authentication information isn’t contained in the card itself, an attacker can’t use these stolen cards to impersonate the user.

Making identity selectors successful means making information cards ubiquitous. To help users clearly understand when a Web site accepts information card-based logins, the standard icon shown in Figure 15 has been created. A claims-aware application, whether created with the “Geneva” Framework or in some other way, can display this icon to let users know that they can use card-based logins. There’s also an Information Card Foundation (www.informationcard.net) dedicated to making this technology successful. The foundation’s board members comprise a range of organizations, including Equifax, Google, Microsoft, Novell, Oracle, and PayPal. The Liberty Alliance, another important identity organization, is also a founding member of the foundation.

Figure 15: A Web page can display this icon to indicate that it accepts information card-based logins.

The Self-Issued Identity Provider

In the examples described so far, an identity provider and the STS it provides have always been external to the user's machine. There's no reason why this has to be true, however. Why can't the user act as her own identity provider, requesting tokens from an STS installed on her own machine? The answer is that, by using the *self-issued identity provider* that's part of CardSpace "Geneva", she can. Figure 16 illustrates this idea.

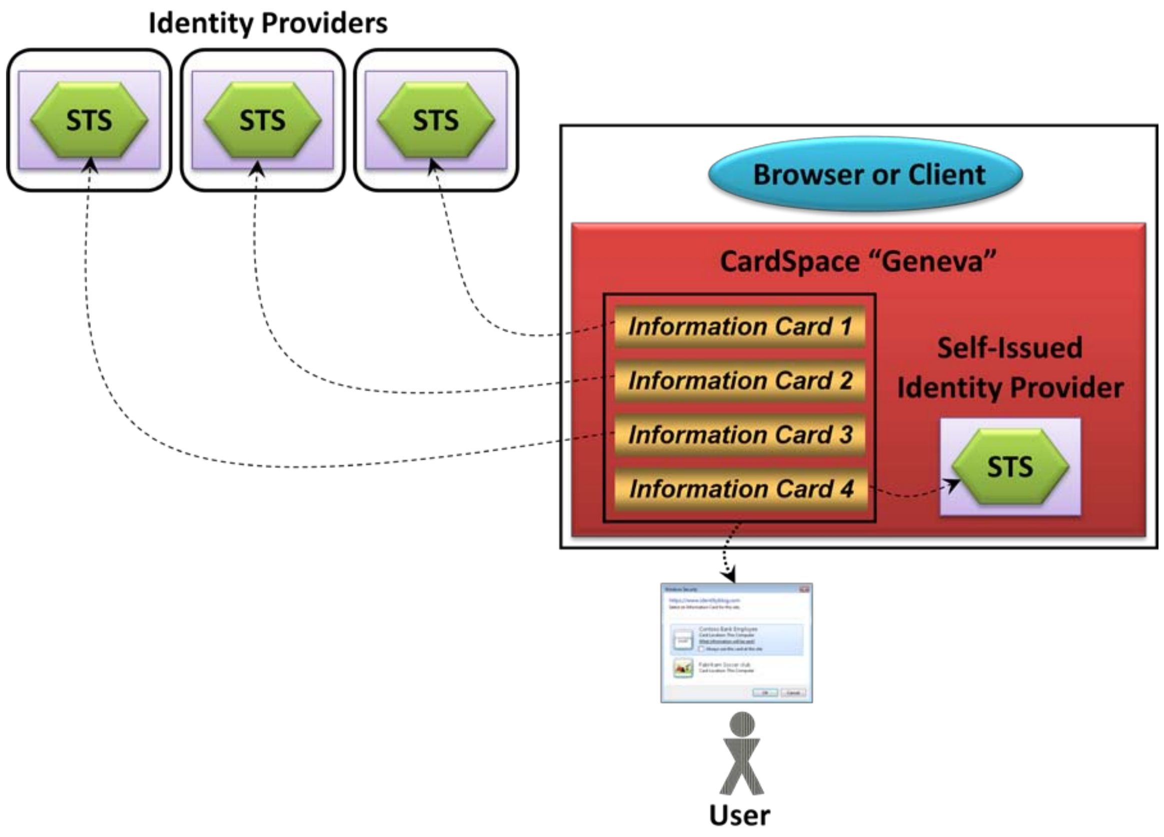


Figure 16: The self-issued identity provider allows a user to act as her own identity provider.

Using the self-issued identity provider changes a number of things, such as how much trust an application can place in the claims this provider issues, but the mechanics remain much the same. As the figure shows, the self-issued identity provider, complete with an STS, runs on the user's computer. As with any other identity provider, the user can install an information card that's associated with an identity this provider maintains. When the user selects the card associated with this identity, the CardSpace "Geneva" identity selector requests a SAML token from the local self-issued identity provider, then sends that token to whatever application the user is accessing. The claims in this token are fixed—users can't add to them—and they include basic information such as the user's name.

Yet there's an obvious question here: What good are these tokens? When an application gets a token issued by an external identity provider, it can have some faith in the claims that token contains (assuming it trusts the identity provider, of course). But how can an application ever trust claims that come from an identity provider owned by the user herself?

To understand when self-issued identities can be useful, think about how usernames and passwords are commonly used on the Internet today. When you access a new site, you're typically asked to create a username and password, which are then stored by that site. The next time you access the site, you provide the same username and password, allowing the site to recognize you. But does the site really know who you are? Of course not. Choosing the username "Angelina Jolie" doesn't mean that you're in fact a famous movie star. The real purpose of a username and password isn't to establish your true identity. It's to allow the site to recognize you as the *same user* each time, proving the continuity of the relationship.

A token created by a self-issued identity provider can be used in the same way. While an application that accepts these tokens can't put much faith in the claims this token contains (just as with a self-chosen username), it can recognize you as the same user over and over. For many applications, including pretty much every Web site that relies on usernames and passwords today, this is all that's needed.

But why bother? How is a token issued by a CardSpace "Geneva" self-issued identity provider better than a simple username and password? Unless there are some advantages over what we do today, there's no reason to switch. These tokens can have significant benefits, however, including the following:

- Unlike passwords, the SAML tokens created by a self-issued identity provider contain cryptographic complexities that make them unusable by anybody except their true owner. Since these tokens can't be stolen and reused, the problem of phishing can be significantly reduced.
- Rather than making users type in basic personal information, this data can be supplied automatically in the token. Forcing users to go through this kind of registration process is one of the major reasons people give up on accessing a site, and so doing this for them can make the site more usable.
- Unlike passwords, which an application must store securely, a token from a self-issued identity provider can be verified using just a public key. Since public keys needn't be kept secret, applications that accept self-issued tokens don't need to worry about them being stolen.

In the lexicon of claims-based identity, an information card issued by a self-issued provider is known as a *personal* card, while one issued by an external identity provider is called a *managed* card. And just as an application can indicate which external identity providers it will accept tokens from, it can also indicate whether it accepts tokens from self-issued providers. (One more point worth making here: The self-issued provider isn't included in the first beta release of CardSpace "Geneva".)

Before closing this discussion of CardSpace, it's worth listing some of the most important changes in CardSpace "Geneva". One of the most important is that CardSpace "Geneva" is now available separately from the .NET Framework, making it smaller and faster. This new release also contains optimizations for applications that users visit repeatedly. For example, a return visit to a Web site can display the card you used to login to this site last time directly in the Web page—the CardSpace "Geneva" screen needn't appear. In some cases, the user can even check a box in the CardSpace "Geneva" selector screen to specify that a specific identity should be used over and over. Also, Microsoft is investigating letting an enterprise administrator push policies to desktops that specify what identities should be used to access specific Web sites. If this is done, those identities will be used without users ever needing to see the CardSpace "Geneva" selector screen.

THE "GENEVA" FRAMEWORK

An STS provides tokens containing identity information, while an identity selector helps users choose which tokens they'd like to use. Yet both are pointless unless applications are modified to accept and use these tokens. The goal of the "Geneva" Framework is to make it easier to do this, helping developers create claims-aware applications.

As described earlier, for example, the "Geneva" Framework provides built-support for verifying a token's signature and extracting its claims. Each claim is extracted into an instance of the "Geneva" Framework-defined Claim class, providing a consistent way for developers to work with a token's information. This class's properties include things such as:

- ClaimType, indicating what kind of claim this is. Does the claim contain a user's name, for example, or a role, or something else? Claim types are identified by strings, which are typically URIs.
- Value, containing the actual content of the claim, such as the user's name.
- Issuer, which specifies the identity provider this claim came from. In other words, this is the entity asserting that this claim is true.

Along with helping developers create claims-aware applications, the "Geneva" Framework also provides support for creating a custom STS. Even though a primary goal of "Geneva" Server is to reduce the need to hand roll your own STS, there are situations where building an STS can make sense. For example, an ISV might use the "Geneva" Framework to create a custom STS for its own purposes.

There's a lot more in the "Geneva" Framework, and creators of claims-aware Windows applications will need to become familiar with this library. For a more detailed look at this technology and how to use it, see Keith Brown's white paper, available at https://connect.microsoft.com/Downloads/DownloadDetails.aspx?SiteID=642&DownloadID=12901.

CONCLUSIONS

Changing how people and applications work with identity is not a small thing. Given this, widespread adoption of claims-based identity is likely to take some time. Still, the foundation is now in place to make this much-improved approach real.

With the advent of “Geneva” Server, CardSpace “Geneva”, and the “Geneva” Framework, all of the pieces required to use claims-based identity on Windows are here. While this style of working with identity is far from a Microsoft-only initiative, making these components widely available for Windows is bound to make it more popular. For anyone who cares about improving the way we use identity in the digital world, this is certainly a step forward.

ABOUT THE AUTHOR

David Chappell is Principal of Chappell & Associates (www.davidchappell.com) in San Francisco, California. Through his speaking, writing, and consulting, he helps people around the world understand, use, and make better decisions about new technology.