Choosing an SSO Strategy: SAML vs OAuth2

By Zach Dennis on 09 May 2013

Chances are you've logged into an application (mobile app or web app) by clicking on a 'Log in with Facebook' button. If you use Spotify, Rdio, or Pinterest, then you know what I'm talking about.

As a user, you likely don't care about how SSO works. You just want to use an application and can be thankful for a smoother experience and that you have to remember fewer logins and passwords.

In order to provide a user with a single sign on experience a developer needs to implement a SSO solution. Over the years there have been many attempts at achieving SSO, but this article is going to focus on a comparison between SAML and OAuth2 - a recent exploration that we took on (thankfully coming out the other end unscathed, but with a lot of information).

Our Need for SSO

We're working on a platform which will have several client applications. Some of these applications will be web-based, others will be native, such as mobile apps.

This platform will roll out being accessible to a few different clients (owned by different organizations). Down the road, additional third-party applications are intended to be built around this platform.

The platform is a front end to a large enterprise system that already has identity information about the people who would be interacting with it. Rather than having each client application maintain their own user database with usernames and passwords, it seems more appropriate to utilize SSO.

Single sign on would allow the enterprise system to securely store and own all of the user credentials. The platform can establish a trust relationship with the enterprise authentication server and client applications can be built to utilize the trusted auth server to authenticate users.

Our goal was to identify an SSO strategy and implementation that could support these needs.

Enter SAML 2.0

We originally looked into SAML 2.0 which is a set of open standards, one of which is specifically designed for SSO.

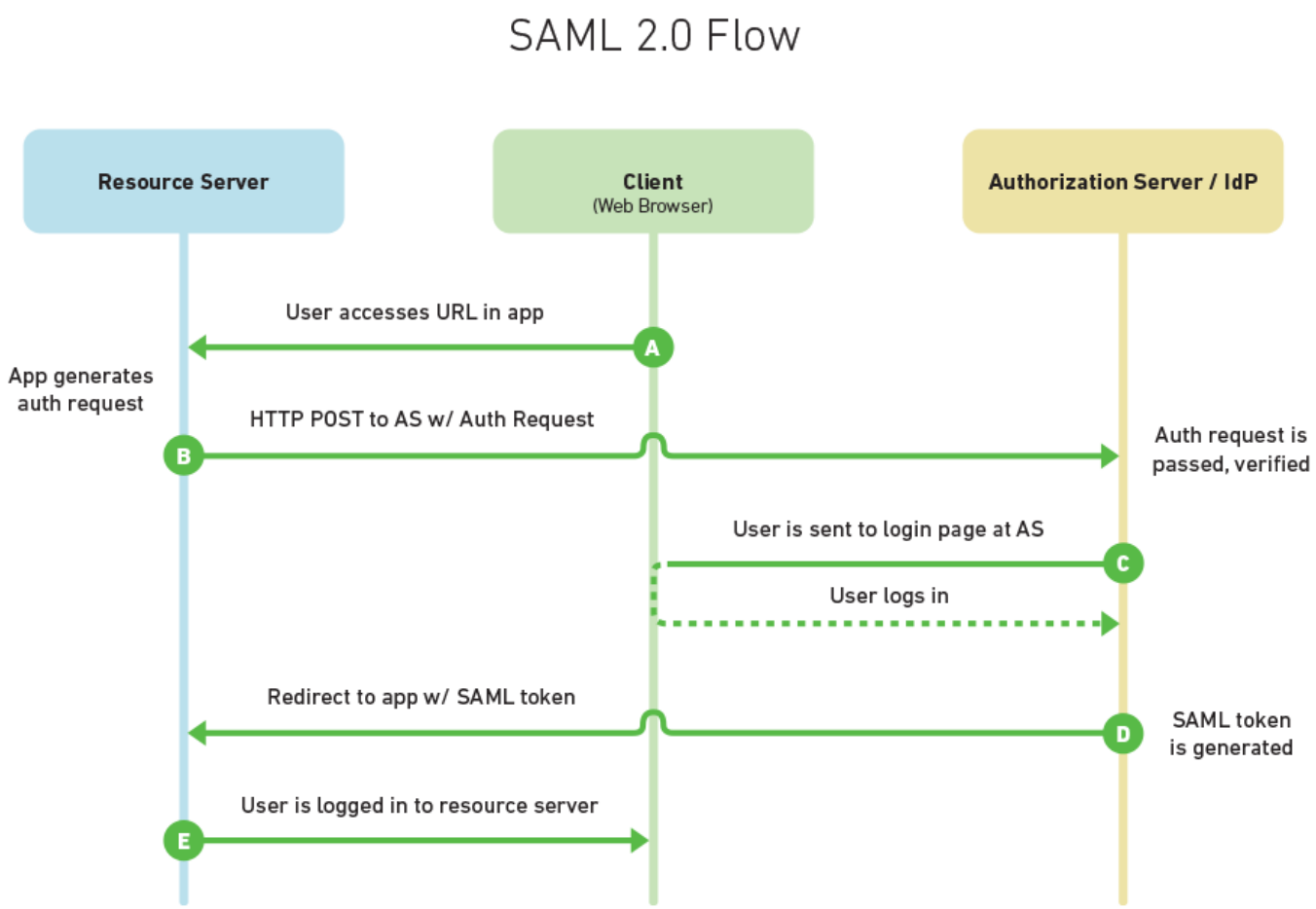
The SAML 2.0 specification (henceforth SAML) provides a Web Browser SSO Profile which describes how single sign on can be achieved for web apps. There are three main players in SAML:

SAML vs. OAuth2 terminology

SAML and OAuth2 use similar terms for similar concepts. For comparison the formal SAML term is listed with the OAuth2 equivalent in parentheses.

- **Service Provider (*Resource Server*)** - this is the web-server you are trying to access information on.
- **Client** - this is how the user is interacting with the Resource Server, like a web app being served through a web browser.
- **Identity Provider (*Authorization Server*)** - this is the server that owns the user identities and credentials. It's who the user actually authenticates with.

The most common SAML flow is shown below:



Here's a fictitious scenario describing the above diagram:

- **A** - a user opens their web-browser and goes to MyPhotos.com which stores all of their photos. MyPhotos.com doesn't handle authentication itself.
- **B** - to authenticate the user MyPhotos.com constructs a SAML Authnrequest, signs it, optionally encrypts it, and encodes it. After which, it redirects the user's web browser to the Identidy Provider (IdP) in order to authenticate. The IdP receives the request, decodes it, decrypts it if necessary, and verifies the signature.
- **C** - With a valid Authnrequest the IdP will present the user with a login form in which they can enter their username and password.
- **D**- Once the user has logged in, the IdP generates a SAML token that includes identity information about the user (such as their username, email, etc). The Id takes the SAML token and redirects the user back to the Service Provider (MyPhotos.com).
- **E** - MyPhotos.com verifies the SAML token, decrypts it if necessary, and extracts out identity information about the user, such as who they are and what their permissions might be. MyPhotos.com now logs the user into its system, presumably with some kind of cookie and session.

At the end of the process the user can interact with MyPhotos.com as a logged in user. The user's credentials never passed through MyPhotos.com, only through the Identity Provider.

There is more detail to the above diagram, but this is high-level of what's going on.

SAML token vs. SAML Assertion

When first being introduced to SAML, the term "SAML token" came up over and over again. It's not actually a term in the SAML spec, but people kept using it, and its meaning was elusive.

As it turns out, the term "SAML token" seems to be a colloquial way to refer to the SAML Assertion, often compressed, encoded, possibly encrypted, and it usually looks like gobbly-gook. And a SAML Assertion is just an XML node with certain elements.

SAML's Native App Limitation

SAML supports the concepts of bindings. These are essentially the means by which the Identity Provider redirects the user back to the Service Provider. For example, in step **D** above, the user gets redirected back to the MyPhotos.com, but how?

The two relevant types of bindings are the HTTP Redirect and the HTTP POST binding defined in the SAML 2.0 spec. The HTTP Redirect binding will use a HTTP Redirect to send the user back to the Service Provider, in the case of our example: MyPhotos.com.

The HTTP Redirect binding is great for short SAML messages, but it is advised against using them for longer messages such as SAML assertions. From wikipedia:

> Longer messages (e.g., those containing signed SAML assertions) should be transmitted via other bindings such as the HTTP POST Binding.

The recommended way of using an HTTP POST has its own oddities. For example, the SAML specification recommends that step **D** above renders an HTML form where the *action* points back to the Service Provider.

You can either have the user click another button to submit that form or you can utilize JavaScript to automate submitting the form. Why is there a form that needs to be submitted? In my opinon, SAML 2.0 is showing it's age (circa 2005), as the form here only exists so an HTTP POST can be used to send the SAML token back to the Service Provider. Which to SAML's defense, in 2005, was likely a necessary decision at the time.

This is a problem when the client is not a web-based application, but a native one, such as a mobile app. For example, let's say we've installed the MyPhotos iPhone app. We open the app, and it wants us to authenticate against the Identity Provider. Once we authenticate, the Identity Provider needs to send the SAML token back to the MyPhotos app.

Most mobile applications can be launched via a custom URI, such as, "my-photos://authenticate", and presumably, the Identity Provider submits the form that includes the SAML token to that URL. Our MyPhotos app launches, but we're not logged in. What gives?

Mobile apps don't have access to the HTTP POST body. They only have access to the URL use to launch the application. This means that we can't read the SAML token.

Launching Mobile Apps via URLs

On Android: launching an application from a url using Intents.

On iOS: launching an application by registering a custom URI scheme.

No SAML token, no authenticated user.

Working Around SAML's HTTP POST Binding

The limitation of the HTTP POST binding for native mobile apps can be worked around. For example, you can use embedded web views, in which you write custom code to watch the entire authentication process. At the very end of the process you scrape the HTML of the page and extract out the SAML token.

A second workaround is to implement a proxy server which can receive the HTTP POST, extract the SAML token, and then make a URL that includes the SAML token (e.g.: "myphotos://authenticate/?SAMLRequest=asdfsdfsdf") The proxy server could then use an HTTP Redirect to cause the device to open the MyPhotos app. And since the SAML token is a part of the URL the MyPhotos app can extract it, and use that to log in.

A third workaround would be to ignore the specification's recommendation against using the HTTP Redirect binding. This is very tempting, but it's hard to shake off the feeling that you're walking into a mine-field, just hoping you don't make one wrong step.

Another approach which avoided workarounds altogether is to not rely on SAML, but look at another approach, like OAuth 2.0.

Enter OAuth 2.0

Unlike SAML, OAuth 2.0 (henceforth OAuth2), is a specification whose ink has barely dried (circa late 2012). It has the benefit of being recent and takes into consideration how the world has changed in the past eight years.

Mobile devices and native applications are prevalent today in ways that SAML could not anticipate in 2005.

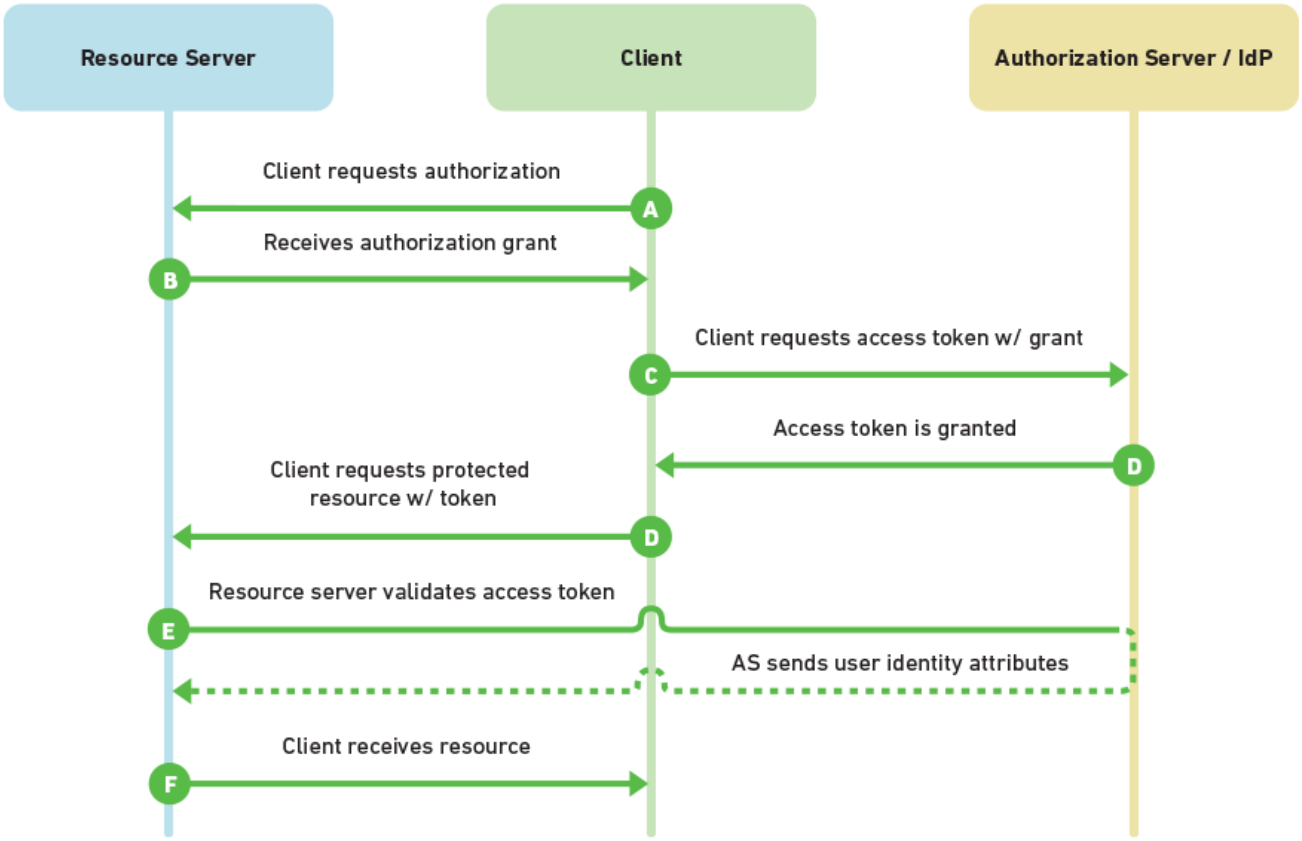
The basic players with OAuth2 are:

SAML vs. OAuth2 terminology

SAML and OAuth2 use similar terms for similar concepts. For comparison the formal OAuth2 term is listed with the SAML equivalent in parentheses.

- **Resource Server (*Service Provider*)** - this is the web-server you are trying to access information on.
- **Client** - this is how the user is interacting with the Resource Server. This could be a browser-based web app, a native mobile app, a desktop app, a server-side app.
- **Authorization Server (*Identity Provider*)** - this is the server that owns the user identities and credentials. It's who the user actually authenticates and authorizes with.

At a high level, the OAuth2 flow is not that different from the earlier SAML flow:



Let's walk through the same scenario we walked through with SAML earlier:

- **A** - a user opens their web-browser and goes to MyPhotos.com which stores all of their photos. MyPhotos.com doesn't handle authentication itself, so the user is redirected to the Authorization Server with a request for authorization. The user is presented with a login form and is asked if they want to approve the Resource Server (MyPhotos.com) to act on their behalf. The user logs in and they are redirected back to MyPhotos.com.
- **B** - the client receives an authorization grant code as a part of the redirect and then passes this along to the client.
- **C** - the Client then uses that authorization grant code to request an access token from the Authorization Server.
- **D** - if the authorization grant code is valid, then the Authorization Server grants an access token. The access token is then used by the client to request resources from the Resource Server (MyPhotos.com).
- **E** - MyPhotos.com receives the request for a resource and it receives the access token. In order to make sure it's a valid access token it sends the token directly to the Authorization Server to validate. If valid, the Authorization Server sends back information about the user.
- **F** - having validated the user's request MyPhotos.com sends the requested resource back to the user.

This is the most common OAuth2 flow: the authorization code flow. OAuth2 provides three other flows (or what they call authorization grants) which work for slightly different scenarios, such as single page javascript apps, native mobile apps, native desktop apps, traditional web apps, and server-side applications where a user isn't directly involved but they've granted you permission to do something on their behalf.

The big advantage with OAuth2 flows are that the communication from the Authorization Server back to the Client and Resource Server is done over HTTP Redirects with the token information provided as query parameters. OAuth2 also doesn't assume the Client is a web-browser whereas the default SAML Web Browser SSO Profile does.

Native mobile applications will just work out of the box. No workarounds necessary.

OAuth2's Favorite Phrase: Out of Scope

The OAuth2 specification doesn't prescribe how the communication between the Resource Server and the Authorization Server works in a lot of situations, such as validating a token. It also doesn't say anything about what information should be returned about the user or in what format.

There are quite a few spots where the OAuth2 specification states that things are "outside the scope of this specification." This has brought criticism to the OAuth2 spec because it leaves a lot of things up to implementation which could lead to incompatible implementations at some point.

OAuth2, is still very young, and it already has widespread adoption with the likes of Google, Facebook, Salesforce, and Twitter to name a few. The true beauty of OAuth2 though is its simplicity. In fact, the OpenID Connect Basic Profile, which builds on OAuth2 fills in some of the areas that the OAuth2 spec itself doesn't define.

OAuth2: Not Requiring Digital Signatures By Default

OAuth2 doesn't require signing messages by default. If you want to add that in, feel free, but out of the box, the spec works without it. It does prescribe that all requests should be made over SSL/TLS.

This has caused commotion in the past:

- OAuth 2.0 (without Signatures) is Bad for the Web
- OAuth 2.0 and the Road to Hell

Having worked with OAuth2 and OAuth1 in the past, I can say that OAuth2 is much simpler than OAuth1 (and more enjoyable to work with). Interoperability and automatic discovery of services may be something useful in the future, but right now, it's not anything we're looking for.

We may be asked to sign messages once the security team of the enterprise does final auditing of the OAuth2 implementation, but for now, OAuth2 fits our current goals in a more standardized manner than SAML. It's also far simpler.

Who's Got Your Keys?

If every application has a secured web-server backing it then signing works great, but when that's not the case the problem becomes more nuanced. How do you securely store your keys in the browser for browser-based JS apps or in native mobile apps?

If you google decompiling iOS and Android apps have your heart will sink. Your keys really aren't that secure if you can't own and secure the device.

OAuth2 is for Authorization, not Authentication

The "auth" in OAuth does stand for "Authorization" and not "Authentication". The pedant in you may be smiling. You've got me!

But – yes, there's always a but! Even though the term OAuth is fairly recent, the fact that "auth" meant authorization seems a tad bit anachronistic. It's already being used to achieve SSO out in the wild (thanks to the likes of Facebook, Twitter, Salesforce, and Google and thousands of sites using them for authenticating and authorizing users).

The biggest complaint I've seen is in lack of prescription and the plentiful "out of scope" usages in the OAuth2 spec. The fact that the OpenID Connect Basic Profile is built directly on top of OAuth2 should be enough to dispel the myth that OAuth2 can't be used for authentication.

What a word meant six years ago is much less important than what it can encompass today.

Summary

SAML has one feature that OAuth2 lacks: the SAML token contains the user identity information (because of signing). With OAuth2, you don't get that out of the box, and instead, the Resource Server needs to make an additional round trip to validate the token with the Authorization Server.

On the other hand, with OAuth2 you can invalidate an access token on the Authorization Server, and disable it from further access to the Resource Server.

Both approaches have nice features and both will work for SSO. We have proved out both concepts in multiple languages and various kinds of applications. At the end of the day OAuth2 seems to be a better fit for our needs (since there isn't an existing SAML infrastructure in place to utilize).

OAuth2 provides a simpler and more standardized solution which covers all of our current needs and avoids the use of workarounds for interoperability with native applications.

As this begins to unfold and we work with various security teams we'll see how far this holds up.

So far, so good.

Zach is a partner at Mutually Human with over ten years of experience in professional software development. He believes strongly in taking a pragmatic approach to the testing and development process, and is a published contributor to The RSpec Book.

@zachdennis
+1 616 475-4225
hello@mutuallyhuman.com
Columbus (Directions) | Grand Rapids (Directions)