**OpenID Connect**

<http://nordicapis.com/api-security-oauth-openid-connect-depth/>

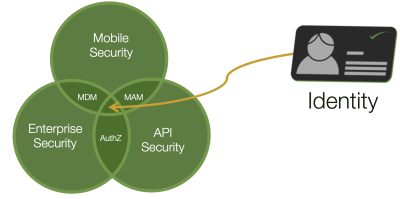
OAuth 2 and OpenID Connect are fundamental to securing your APIs. To protect the data that your services expose, you must use them. They are complicated though, so we wanted to go into some depth about these standards to help you deploy them correctly.

OAuth and OpenID Connect in Context

Always be aware that OAuth and OpenID Connect are part of a larger information security problem. You need to take additional measures to protect your servers and the mobiles that run your apps in addition to the steps taken to secure your API. Without a holistic approach, your API may be incredibly secure, your OAuth server locked down, and your OpenID Connect Provider tucked away in a safe enclave. Your firewalls, network, cloud infrastructure, or the mobile platform may open you up to attack if you don’t also strive to make them as secure as your API.



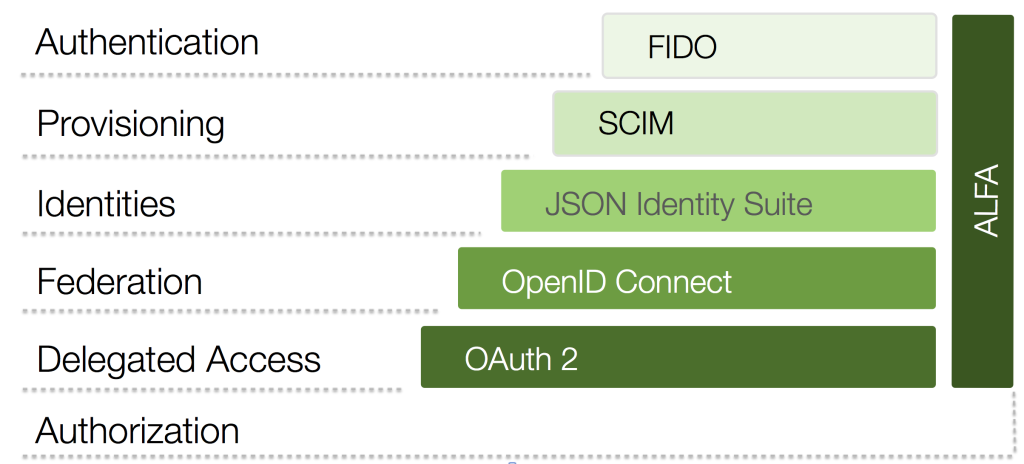
To account for all three of these security concerns, you have to know who someone is and what they are allowed to do. To authenticate and authorize someone on your servers, mobile devices, and in your API, you need a complete Identity Management System. At the head of API security, enterprise security and mobile security is identity!



Only after you know who someone (or something) is can you determine if they should be allowed to access your data. We won’t go into the other two concerns, but don’t forget these as we delve deeper into API security.

Start with a Secure Foundation

To address the need for Identity Management in your API, you have to build on a solid base. You need to establish your API security infrastructure on protocols and standards that have been peer-reviewed and are seeing market adoption. For a long time, lack of such standards has been the main impediment for large organizations wanting to adopt RESTful APIs in earnest. This is no longer the case since the advent of the Neo-security Stack:



This protocol suite gives us all the capabilities we need to build a secure API platform. The base of this, OAuth and OpenID Connect, is what we want to go into in this blog post. If you already have a handle on these, learn more about [how the other protocols of the Neo-security Stack fit together](http://www.twobotechnologies.com/blog/2012/08/xacml-w-openid-connect-saml-oauth-and-scim.html).

Overview of OAuth

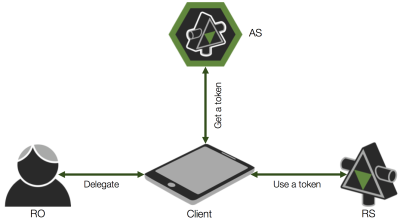
OAuth is a sort of “protocol of protocols” or “meta protocol,” meaning that it provides a useful starting point for other protocols (e.g., [OpenID Connect](http://openid.net/specs/openid-connect-core-1_0.html), [NAPS](http://openid.net/wg/napps/), and [UMA](https://docs.kantarainitiative.org/uma/draft-uma-core.html)). This is similar to the way WS-Trust was used as the basis for WS-Federation, WS-SecureConversation, etc., if you have that frame of reference.

Beginning with OAuth is important because it solves a number of important needs that most API providers have, including:

* Delegated access
* Reduction of password sharing between users and third-parties (the so called “password anti-pattern”)
* Revocation of access

When the password anti-pattern is followed and users share their credentials with a third-party app, the only way to revoke access to that app is for the user to change their password. Consequently, all other delegated access is revoked as well. With OAuth, users can revoke access to specific applications without breaking other apps that should be allowed to continue to act on their behalf.

Actors in OAuth

There are four primary actors in OAuth:

1. **Resource Owner (RO)**: The entity that is in control of the data exposed by the API, typically an end user
2. **Client**: The mobile app, web site, etc. that wants to access data on behalf of the Resource Owner
3. **Authorization Server (AS)**: The Security Token Service (STS) or, colloquially, the OAuth server that issues tokens
4. **Resource Server (RS)**: The service that exposes the data, i.e., the API

Scopes

OAuth defines something called “scopes.” These are like permissions or delegated rights that the Resource Owner wishes the client to be able to do on their behalf. The client may request certain rights, but the user may only grant some of them or allow others that aren’t even requested. The rights that the client is requested are often shown in some sort of UI screen. Such a page may not be presented to the user, however. If the user has already granted the client such rights (e.g., in the EULA, employment contract, etc.), this page will be skipped.

What is in the scopes, how you use them, how they are displayed or not displayed, and pretty much everything else to do with scopes are not defined by the OAuth spec. OpenID Connect does define a few, but we’ll get to that in a bit.

Kinds of Tokens

In OAuth, there are two kinds of tokens:

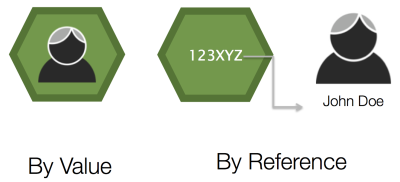
1. **Access Tokens**: These are tokens that are presented to the API
2. **Refresh Tokens**: These are used by the client to get a new access token from the AS

(Another kind of token that OpenID Connect defines is the ID token. We’ll get to that in a bit.)

Think of access tokens like a session that is created for you when you login into a web site. As long as that session is valid, you can continue to interact with the web site without having to login again. Once that session is expired, you can get a new one by logging in again with your password. Refresh tokens are like passwords in this comparison. Also, just like passwords, the client needs to keep refresh tokens safe. It should persist these in a secure credential store. Loss of these tokens will require the revocation of all consents that users have performed.

**NOTE**: The AS may or may not issue a refresh token to a particular client. Issuing such a token is ultimately a trust decision. If you have doubts about a client’s ability to keep these privileged tokens safe, don’t issue it one!

Passing Tokens

As you start implementing OAuth, you’ll find that you have more tokens than you ever knew what to do with! How you pass these around your system will certainly affect your overall security. There are two distinct ways in which they are passed:

1. By value
2. By reference

These are analogous to the way programming language pass data identified by variables. The run-time will either copy the data onto the stack as it invokes the function being called (by value) or it will push a pointer to the data (by reference). In a similar way, tokens will either contain all the identity data in them as they are passed around or they will be a reference to that data.

**TIP**: Pass by reference when tokens have to leave your network, and then convert them to by-value tokens as they enters your space. Do this conversion in your API gateway. [See Jacob Ideskog’s Microservices presentation](https://www.youtube.com/watch?v=BdKmZ7mPNns) for the Platform Summit for details.

If you pass your tokens by reference, keep in mind that you will need a way to dereference the token. This is typically done by the API calling a non-standard endpoint exposed by your OAuth server.

Profiles of Tokens

There are different profiles of tokens as well. The two that you need to be aware of are these:

1. Bearer tokens
2. Holder of Key (HoK) tokens

You can think of bearer tokens like cash. If you find a dollar bill on the ground and present it at a shop, the merchant will happily accept it. She looks at the issuer of the bill, and trusts that authority. The saleswomen doesn’t care that you found it somewhere. Bearer tokens are the same. The API gets the bearer token and accepts the contents of the token because it trusts the issuer (the OAuth server). The API does not know if the client presenting the token really is the one who originally obtained it. This may or may not be a bad thing. Bearer tokens are helpful in some cases, but risky in others. Where some sort of proof that the client is the one to who the token was issued for, HoK tokens should be used.

HoK tokens are like a credit card. If you find my credit card on the street and try to use it at a shop, the merchant will (hopefully) ask for some form of ID or a PIN that unlocks the card. This extra credential assures the merchant that the one presenting the credit card is the one to whom it was issued. If your API requires this sort of proof, you will need HoK key tokens. This [profile is still a draft](https://tools.ietf.org/html/draft-ietf-oauth-proof-of-possession), but you should follow this before doing your own thing.

**NOTE**: You may have heard of [MAC tokens](https://tools.ietf.org/html/draft-ietf-oauth-v2-http-mac) from an early OAuth 2 draft. This proposal was never finalized, and this profile of tokens are never used in practice. Avoid this unless you have a very good reason. Vet that rational on the[OAuth mailing list](http://www.ietf.org/mail-archive/web/oauth/current/maillist.html) before investing time going down this rabbit trail.

Types of Tokens

We also have different types of tokens. The OAuth specification doesn’t stipulate any particular type of tokens. This was originally seen by many as a negative thing. In practice, however, it’s turned out to be a very good thing. It gives immense flexibility. Granted, this comes with reduced interoperability, but a uniform token type isn’t one area where interop has been an issue. Quite the contrary! In practice, you’ll often find tokens of various types and being able to switch them around enables interop. Example types include:

* WS-Security tokens, especially SAML tokens
* JWT tokens (which I’ll get to next)
* Legacy tokens (e.g., those issued by a Web Access Management system)
* Custom tokens

Custom tokens are the most prevalent when passing them around by reference. In this case, they are [randomly generated strings](https://www.owasp.org/index.php/Session_Management_Cheat_Sheet#Session_ID_Properties). When passing by val, you’ll typically be using JWTs.

JSON Web Tokens

JSON Web Tokens or JWTs (pronounced like the English word “jot”) are a type of token that is a JSON data structure that contains information, including:

* The issuer
* The subject or authenticated uses (typically the Resource Owner)
* How the user authenticated and when
* Who the token is intended for (i.e., the audience)

These tokens are very flexible, allowing you to add your own claims (i.e., attributes or name/value pairs) that represent the subject. JWTs were designed to be light-weight and to be snuggly passed around in HTTP headers and query strings. To this end, the JSON is split into different parts (header, body, signature) and base-64 encoded.

If it helps, you can compare JWTs to SAML tokens. They are less expressive, however, and you cannot do everything that you can do with SAML tokens. Also, unlike SAML they do not use XML, XML name spaces, or XML Schema. This is a good thing as JSON imposes a much lower technical barrier on the processors of these types of tokens.

JWTs are part of [the JSON Identity Suite](http://www.slideshare.net/2botech/the-jsonbased-identity-protocol-suite), a critical layer in the Neo-security Stack. Other things in this suite include JWA for expressing algorithms, JWK for representing keys, JWE for encryption, JWS for signatures, etc. These together with JWT are used by both OAuth (typically) and OpenID Connect. How exactly is specified in the [core OpenID Connect spec](http://openid.net/specs/openid-connect-core-1_0.html) and various ancillary specs, in the case of OAuth, including the [Bearer Token spec](https://tools.ietf.org/html/draft-ietf-oauth-jwt-bearer).

OAuth Flow

OAuth defines different “flows” or message exchange patterns. These interaction types include:

* The code flow (or web server flow)
* Client credential flow
* Resource owner credential flow
* Implicit flow

The code flow is by far the most common; it is probably what you are most familiar with if you’ve looked into OAuth much. It’s where the client is (typically) a web server, and that web site wants to access an API on behalf of a user. You’ve probably used it as a Resource Owner many times, for example, when you login to a site using certain social network identities. Even when the social network isn’t using OAuth 2 per se, the user experience is the same. Checkout this YouTube video at time 12:19 to see how this flow goes and what the end user experiences:

We’ll go into the other flows another time. If you have questions on them in the meantime, ask in a comment below.

Improper and Proper Uses of OAuth

After all this, your head may be spinning. Mine was when I first learned these things. It’s normally. To help you you orient yourself, I want to stress one really important high-level point:

* **OAuth is not used for authorization**. You might think it is from it’s name, but it’s not.
* **OAuth is also not for authentication**. If you use it for this, expect a breach if your data is of any value.
* **OAuth is also not for federation**.

So what is it for?

**It’s for delegation, and delegation** **only!**



This is your plumb line. As you architect your OAuth deployment, ask yourself: In this scenario, am I using OAuth for anything other than delegation? If so, go back to the drawing board.

Consent vs. Authorization

How can it *not* be for authorization, you may be wondering. The “authorization” of the client by the Resource Owner is really consent. This consent may be enough for the user, but not enough for the API. The API is the one that’s actually authorizing the request. It probably takes into account the rights granted to the client by the Resource Owner, but that consent, in and of its self, is not authorization.

To see how this nuance makes a very big difference, imagine you’re a business owner. Suppose you hire an assistant to help you manage the finances. You *consent* to this assistant withdrawing money from the business’ bank account. Imagine further that the assistant goes down to the bank to use these newly delegated rights to extract some of the company’s capital. The banker would refuse the transaction because the assistant is not authorized — certain paperwork hasn’t been filed, for example. So, your act of delegating your rights to the assistant doesn’t mean squat. It’s up to the banker to decide if the assistant gets to pull money out or not. In case it’s not clear, in this analogy, the business owner is the Resource Owner, the assistant is the client, and the banker is the API.

Building OpenID Connect Atop OAuth

As I mentioned above, OpenID Connect builds on OAuth. Using everything we just talked about, OpenID Connect constrains the protocol, turning many of the specification’s SHOULDs to MUSTs. This profile also adds new endpoints, flows, kinds of tokens, scopes, and more. OpenID Connect (which is often abbreviated OIDC) was made with mobile in mind. For the new kind of tokens that it defines, the spec says that they must be JWTs, which were also designed for low-bandwidth scenarios. By building on OAuth, you will gain both delegated access and federation capabilities with (typically) one product. This means less moving parts and reduced complexity.

OpenID Connect is a modern federation specification. It is a passive profile, meaning it is bound to a passive user agent that does not take an active part in the message exchange (though the client does). This exchange flows over HTTP, and is analogous to the SAML artifact flow (if that helps). OpenID Connect is a replacement for SAML and WS-Federation. While it is still relatively new, you should prefer it over those unless you have good reason not to (e.g., regulatory constraints).

As I’ve mentioned a few times, OpenID Connect defines a new kind of token: ID tokens. These are intended for the client. Unlike access tokens and refresh tokens that are opaque to the client, ID tokens allow the client to know, among other things:

* How the user authenticated (i.e., what type of credential was used)
* When the user authenticated
* Various properties about the authenticated user (e.g., first name, last name, shoe size, etc.)

This is useful when your client needs a bit of info to customize the user experience. Many times I’ve seen people use by value access tokens that contain this info, and they let the client take the values out of the API’s token. This means they’re stuck if the API needs to change the contents of the access token or switch to using by ref for security reasons. If your client needs data about the user, give it an ID token and avoid the trouble down the road.

The User Info Endpoint and OpenID Connect Scopes

Another important innovation of OpenID Connect is what’s called the “User Info Endpoint.” It’s kinda a mouthful, but it’s an *extremely* useful addition. The spec defines a few specific scopes that the client can pass to the OpenID Connect Provider or OP (which is another name for an Authorization Server that supports OIDC):

* openid (required)
* profile
* email
* address
* phone

You can also (and usually will) define others. The first is required and switches the OAuth server into OpenID Connect mode. The others are used to inform the user about what type of data the OP will release to the client. If the user authorizes the client to access these scopes, the OpenID Connect provider will release the respective data (e.g., email) to the client when the client calls the user info endpoint. This endpoint is protected by the access token that the client obtains using the code flow discussed above.

**NOTE**: An OAuth client that supports OpenID Connect is also called a Relying Party (RP). It gets this name from the fact that it *relies* on the OpenID Connect Provider to assert the user’s identity.

Not Backward Compatible with v. 2

It’s important to be aware that OpenID Connect ***is not*** backward compatible with OpenID 2 (or 1 for that matter). OpenID Connect is effectively version 3 of the OpenID specification. As a major update, it is not interoperable with previous versions. Updating from v. 2 to Connect will require a bit of work. If you’ve properly architected your API infrastructure to separate the concerns of federation with token issuance and authentication, this change will probably not disrupt much. If that’s not the case however, you may need to update *each and every* app that used OpenID 2.

Conclusion

In this post, I dove into the fundamentals of OAuth and OpenID Connect and pointed out [their place in the Neo-security Stack](http://www.twobotechnologies.com/blog/2012/08/cloud-security-standards.html). I said it would be in depth, but honestly I’ve only skimmed the surface. Anyone providing an API that is protected by OAuth 2 (which should be all of them that need secure data access), this basic knowledge is a prerequisite for pretty much everyone on your dev team. Others, including product management, ops, and even project management should know some of the basics described above.

If you have questions beyond what I was able to cover here, checkout this [video recording of a talk I gave on this topic](http://youtu.be/XGmUlyggXVo), this [Slideshare presentation](http://www.slideshare.net/nordicapis/incorporating-oauth-how-to-integrate-oauth-into-your-mobile-app), or drop a comment below.

*[EDITOR’S NOTE: All images used in this post are copyright*[*Twobo Technologies*](http://twobo.com/)*and used by permission. The Mobile/Enterprise/API Security Venn diagram was created by*[*Gunnar Peterson*](http://1raindrop.typepad.com/)*and also used by permission.]*