CURITY

OAuth Explained

An in-depth explanation of OAuth and OpenID Connect

Abstract

This whitepaper explains the main aspects of OAuth and OpenID Connect that every API provider needs to know. It briefly explains how these fit into an API provider's broader security program, and their place in the "Neo-security Stack," a modern suite of protocols that organizations should be using to deliver safe data access via APIs. It explains the proper and improper uses of OAuth, and provides easy-to-understand examples of key concepts like scopes, tokens, profiles, and method of exchanging tokens. After reading this whitepaper, you will have the requisite knowledge needed to begin protecting APIs with OAuth and OpenID Connect.

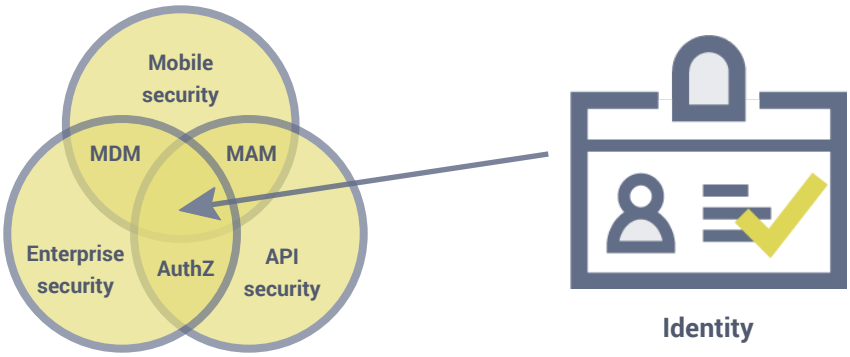
OAuth 2 and OpenID Connect are fundamental to securing your APIs. To protect the data that your services expose, you will need these protocols. They are complicated though, and it is easy to get lost in the hundreds of pages that make up these specifications. To find your way, read on to get a to get a good introduction to these important security standards!

OAuth and OpenID Connect in Context

When considering how to use OAuth and OpenID Connect to secure your APIs, it is important to be aware that they must be a part of a larger effort to secure your organization. To use them in a holistic manner, you need to consider the various fronts that need protecting. Without a comprehensive approach, your API may be incredibly secure, your OAuth server locked down, and your OpenID Connect Provider tucked away in a safe enclave. However, you also need to take measures to protect your servers and the mobiles that run your apps. If you don't, your firewalls, network, cloud infrastructure, or the mobile platform may open you up to attack. Regardless of the industry in which your organization operates, its size, or the type of API you are exposing, the attacks against your services will come on three general fronts. The first is Enterprise Security which relates to the internal servers and back-end services (e.g., mail servers, file servers, etc.). The second front is

Enterprise security **Mobile security** **API security**

mobile security; this has to do with the management of devices connected to your systems, often from external networks. Because these devices are more powerful than ever and because they provide numerous capabilities and vulnerabilities that hackers can exploit, they may serve as a doorway into your organization if you don't take a holistic approach to API security that closes this vector. The final front is API security, of which OAuth and OpenID Connect are of paramount importance but not the exclusive concern. To handle these disparate areas and create a comprehensive approach to securing your organization, you need to find the commonality between the three. While

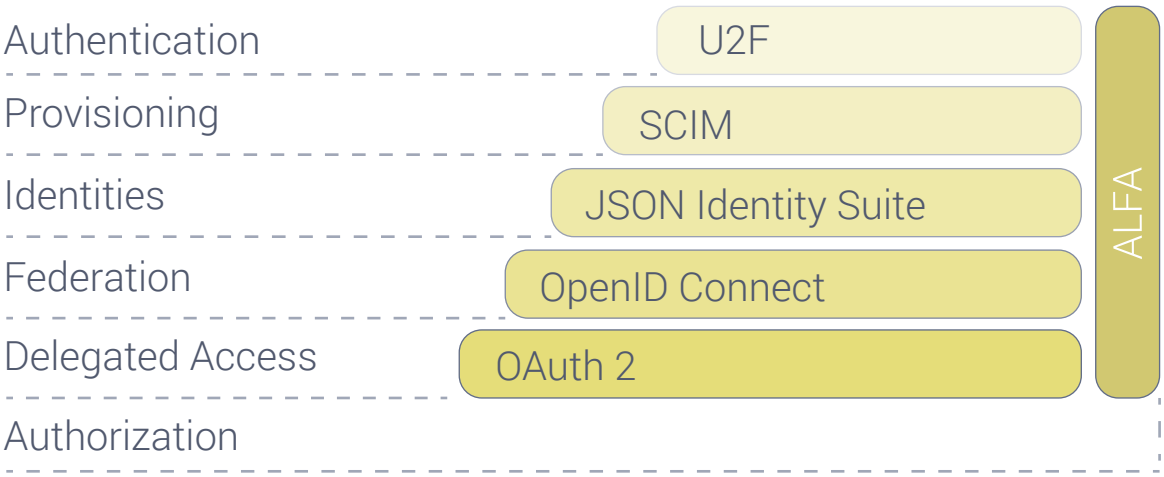


each of these fronts will require their own systems and procedures, the ability to secure them stems from the same point - identity. To properly protect your organization on all three of these axes, you have to know who someone is and what they are allowed to do. To authenticate and authorize someone on your servers, mobile devices, and in your API, you need a complete Identity

Management System. 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 do not forget these as we delve 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, the 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 provides the capabilities needed to build a secure API platform. The base of this -- OAuth and OpenID Connect -- provide a powerful and secure starting point for protecting your data.

Overview of OAuth

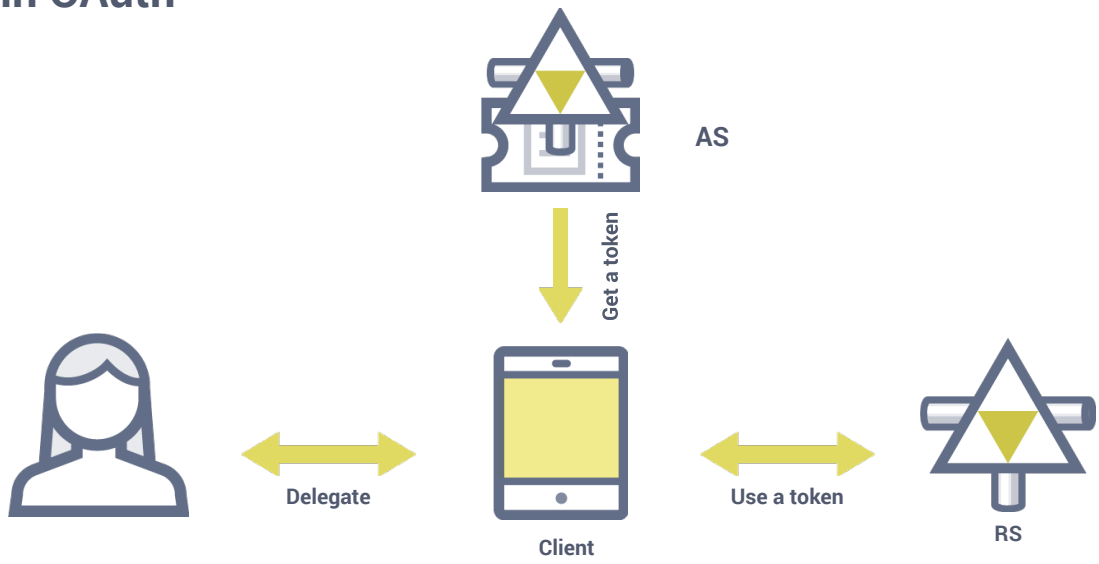
OAuth started to take shape nearly a decade ago when engineers from various social networking sites including Twitter and Google recognized a need for delegating access to APIs. In 2006, groups from various organizations came together to address this universal use case in an interoperable manner. Working together, a relatively informal group defined a unified methodology for doing so which was stipulated in the 1.0 release of OAuth. After this initial version in 2007, the protocol was contributed to the IETF standards body where it was updated to the current version in 2012. This new version obsoletes the older revisions, and is what is recommended for all current and future projects that require OAuth support.

This 2.0 version of the protocol was designed to make it simpler to consume tokens, pushing the difficulties onto organizations who issue them -- since there are far fewer organizations issuing tokens than there are ones that need to consume them. This second version of OAuth is a sort of "protocol of protocols" or "framework." This allows it to be used as the starting point for many other protocols that require delegated access to solve related use cases (e.g., OpenID Connect, NAPS, and UMA). By being a "meta protocol," OAuth has the flexibility necessary to be the basis for numerous complementary cases. This means that API providers can use it to solve scenarios stretching far beyond common ones like:

- 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



OAuth defines four primary “actors” that take part in the OAuth “act” or “dance.” There is a clear delineation between the responsibilities of each of these entities. The four actors are:

- Resource Owner (RO): The entity that is in control of the data exposed by the API, typically an end user
- Client: The mobile app, website, etc. that would like to access data on behalf of the Resource Owner
- Authorization Server (AS): The Security Token Service (STS) or, colloquially, the OAuth server which issues and validates tokens
- Resource Server (RS): The service that delivers and exposes the data, i.e., the API

Basic Real-world Example

To understand what these actors are and how they relate, consider this real-world example. A software company, ZPower, has created a mobile application that allows users to analyze their power consumption, give them tips to reduce their utility bill, and remotely control lights and appliances. For users of the ZPower app to actually perform these activities, they must grant the ZPower app access to data housed at their power company and allow the app to control their lights and other appliances.

This complicated case is made simpler by the use of open standards. ZPower has made arrangements with a number of large power companies, including CoolUtility, who will allow the ZPower app to consume user-specific data through an API, under the condition that the end user consents to ZPower, an authorized third-party, to access their information.

To implement this, CoolUtility deploys an OAuth server (an AS) and an API (an RS). The API exposes power consumption data for users and the capabilities to control appliances and other smart devic-

es. Joe (an RO) is a customer of CoolUtility. Joe downloads the ZPower app (the Client), so he can use it to control his new smart appliances and get suggestions on how to lower his energy bill.

When he starts the app on his smartphone, it asks him to signup and login. He creates a ZPower account and does so. Then, he's shown a list of power companies that the ZPower app can interoperate with. He finds CoolUtility in the list and selects that one. The app opens his phone's system browser. Joe is presented with CoolUtility's login screen. Joe has seen this page many times, and can observe from the address bar that he is indeed communicating directly with CoolUtility. He thus enters his CoolUtility username and password with confidence that ZPower will never see it (avoiding the password anti-pattern).

After logging in, he is presented with a screen asking him to consent to the ZPower app's request to access his energy consumption data, to turn power on and off to his smart appliances, and to switch his smart bulbs on and off. This consent screen is rendered by the CoolUtility OAuth server. Joe grants access to ZPower to perform these actions on his behalf. Automatically, the system browser closes, and the ZPower app reopens.

Upon doing so, the ZPower app makes a call to the CoolUtility OAuth server with a one-time-usage token that was delivered to it when Joe's browser was shut down and the app was restarted. The CoolUtil OAuth server responds to this message with an access token. The ZPower app uses this to make an API call to the CoolUtility API to get Joe's recent power usage data, info about his smart appliances, and data about his smart lights. The app uses this to present him with graphs and UI controls to toggle power to his plugs, to control his appliances, and to turn on and off the power to his smart bulbs.

Joe skims the graphs, and is really impressed with the slick UI. He tries out the app's ability to turn on some lights. Odd. Nothing happens. Oh, well. He's late for work, so he dashes off. Later, he receives a call from his spouse informing him that the house has burned down! Apparently, the new smart oven was some how turned on to a raging temperature, and the kitchen ignited. Joe immediately logs into CoolUtility's Web site and revokes the ZPower app's access to his data and devices. After which, all API calls that the app makes are immediately blocked. This gives Joe peace of mind as he calls up his lawyer about initiating a lawsuit against ZPower and CoolUtility.

A trial ensues. Other homes burn down. Joe's case swells into a giant class action suite. Thankfully, CoolUtility has the OAuth server logs that are proof of Joe's and the other plaintiffs' approval of the third-party access; they also have the API logs, showing that ZPower did not call their API as documented. These are the "smoking gun" that get CoolUtility off clean.

There are many other ways in which the OAuth actors can "dance," but this is a very common one.

Scopes

A central concept in OAuth is something the specification refers to as “scopes.” These are like permissions or delegated rights that the Resource Owner wishes the client to be able to use 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 asking for are often shown in some sort of UI screen. In the example above, access to past energy usage data could be one scope, controlling smart appliances another, and switching smart bulbs a third. In some cases, such a page is not presented to the user. For instance, this page is not needed if the user has already granted the client rights to act on her behalf when she agreed to the EULA of the client app that is making the request or when she signed an employment contract.

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 specification. OpenID Connect does define a few which will be discussed shortly.

Kinds of Tokens

In OAuth, there are two kinds of tokens:

– Access Tokens: These are tokens that are presented to the API

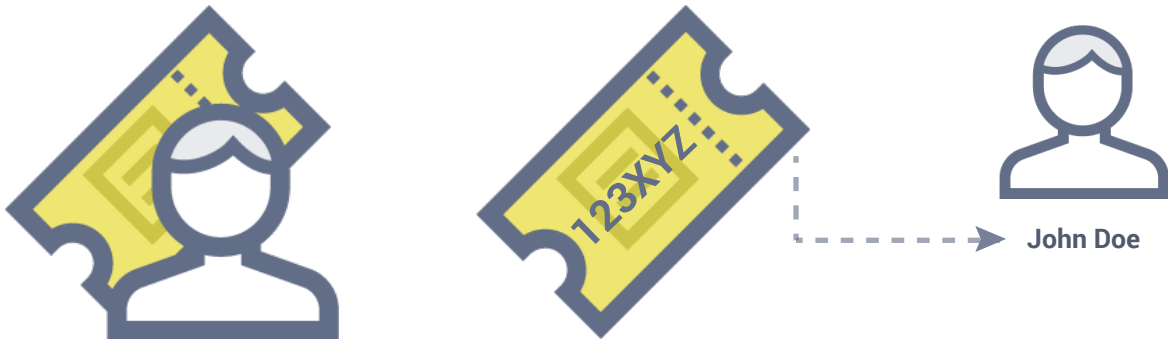
– Refresh Tokens: These are used by the client to get a new access token from the AS

Another kind of token that OpenID Connect defines is the ID token. These are described below.

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 anew 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.

> 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 doubt a client’s ability to keep these privileged tokens safe, don’t issue it one!

Passing Tokens



By Value

By Reference

As you start implementing OAuth, you'll find that you have more tokens than you ever knew what to do with! With so many tokens, how you transmit them is an important consideration that affects the overall level of security. There are two distinct ways in which they can be passed:

- By value
- By reference

These are analogous to the way programming language pass data identified by variables. The runtime 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.

As a rule of thumb, pass tokens by reference when they have to leave your network, and convert them to by value tokens when they enter your domain. This conversion is often done in an API gateway, and will allow you to differentiate and sequester various points of entry and exit in your system.

There is a caveat to passing tokens by reference, however. If you use this technique (and you should!), you need to have the means of dereference the tokens within your API or gateway.

This is typically done by calling a non-standard endpoint that is exposed by your OAuth server.

Profiles of Tokens

Just like there are different kinds of tokens, there are also various profiles of tokens as well. The two that you should be aware of are these:

- Bearer tokens
- 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. He will look at the issuer of the bill, and trust that authority. The salesman doesn't care that you found it on the ground outside the shop. 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 whom the token was issued, HoK tokens should be used.

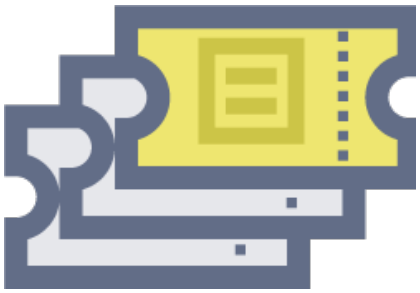
HoK tokens are like a credit card. If you find someone else's credit card on the street and try to use it at a shop, the merchant will (hopefully) ask for some form of ID or a PIN that unlocks the card. This extra credential assures the merchant that the one presenting the credit card is the one to whom it was issued. If your API requires this sort of proof, you will need HoK key tokens. The HoK profile is still a draft, but you should follow it before doing your own thing.

You may have heard of MAC tokens 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 before investing time going down this rabbit trail.

Types of Tokens

When implementing OAuth, you also need to be aware of 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 has turned out to be a very good thing. It gives immense flexibility to solve a wide variety of use cases. This comes with reduced interoperability, but a uniform token type is not an area where interop has been an issue -- quite the contrary in fact! In practice, you'll often find tokens of various types and being able to switch them around enables interop. Example of common token types include:

- WS-Security tokens, especially SAML tokens
- JWT tokens (which you'll use primarily when passing tokens by value)
- Legacy tokens (e.g., those issued by a Web Access Management system)
- Custom tokens (e.g., by ref tokens)



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 OAuth server)
- 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.

JWTs are comparable to SAML tokens. They are less expressive, however, and you cannot use them to 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, another important layer of 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 when implementing both OAuth and OpenID Connect. How exactly is specified in the core OpenID Connect specification and various ancillary OAuth specifications.

OAuth Flows

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 is where the client is (typically) a web server, and that web site wants to access an API on behalf of a user. You have probably used it as a Resource Owner many times, for example, when logging into 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.

Improper and Proper Uses of OAuth

OAuth gets a lot of buzz, and you can see from the preceding explanations that it is quite powerful. For these reasons, it may seem like a logical choice for all your API-related security challenges. Taking this approach, however, neglects the earlies advice of creating a holistic security posture of which OAuth is a part. Which part does it play though? To answer this question and ensure proper usage, you must be aware of these four important truths:

– OAuth is not used for authorization
– OAuth is also not for authentication
– OAuth is also not for federation
– OAuth is for delegation and delegation only!

OAuth is for delegated access 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

Given that OAuth is named "OAuth," how can it not be for authorization? This is a matter of semantics, but an important one. The "authorization" of the client by the Resource Owner is not really authorization, but rather 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 itself, 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. When the assistant attempts to use these newly delegated rights to withdraw some of the company's capital, however, the banker refuses the request. This is because the assistant is not authorized -- certain paperwork hasn't been filed, for example. In this case, your act of delegating rights, or consenting for another to act on your behalf, is completely useless unless you also work within the confines and rules the bank has established. This is the case because the banker always decides if the requested withdrawals should be allowed, not the account holder. In this analogy, the business owner/account holder is the Resource Owner, the assistant is the client, and the banker is the API.

Building OpenID Connect Atop OAuth

By this point, you should have a good introduction to OAuth. With this, you are ready to delve into OpenID Connect. This specification constrains the protocol, turning many of the its “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, an OpenID Connect deployment will inherently includes an OAuth server. Consequently, you will gain both delegated access and federation capabilities. Typically, the implementation of these two protocols are provided by 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. Specifically, 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).

One of the best new features of OpenID Connect is the new kind of token that it defines: 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 a client needs info to customize the user experience, for example. Many times, people errantly use by value access tokens that contain this info, and let the client consumes values from that. Because these access tokens are intended solely for the API and are not meant to be interrogated by the client, you will be stuck when your API needs to change the contents of the access token since doing so would break the client. If your app needs data about the user, issue it an ID token, and avoid this kind of future trouble.

The User Info Endpoint and OpenID Connect Scopes

Another important innovation of OpenID Connect is what's called the "User Info Endpoint." The spec also defines a few specific scopes that the client can pass to the OpenID Connect Provider or OP (which is another name for an AS 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 previously.

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 is important to be aware that OpenID Connect is not backward compatible with OpenID 2 or any previous revisions. OpenID Connect is effectively version 3 of the OpenID specification. As a major update, it is not interoperable with previous versions. Updating from version 2 to Connect will require a bit of work. If you've properly architected your API infrastructure to separate the concerns of federation from token issuance and authentication, this change will not disrupt much.

Conclusion

In this paper, we have discussed the fundamentals of OAuth and OpenID Connect. We have explained that they are the basis of the Neo-security Stack, a suite of protocols that you should use when building a secure API platform. We also explained that API security is only one facet to a comprehensive security stance that you must take to avoid a devastating breach. At the heart of all of these facets to comprehensive security is digital identity. By erecting systems and procedures for API management, enterprise security, and mobiles, you will be able to confidently answer the questions of who someone is and what they are allowed to do with your data and services. These questions will lead to an assurance that only authorized usage occurs.

There is more to learn about OAuth, OpenID Connect, API security and the other aspects of a holistic security program. While there is always more to learn, however, the development team of any API provider should possess at least this basic knowledge. Other non-developers, including product management, ops, and even project management, should also know some of the basics described above.

For Further Information

If you have questions about anything written in this paper or would like to learn more about how to apply them to your situation, contact us by email at **info@curity.io** or by phone at **+46 8-41073770**.

More information can also be found on our Web site **curity.io**.