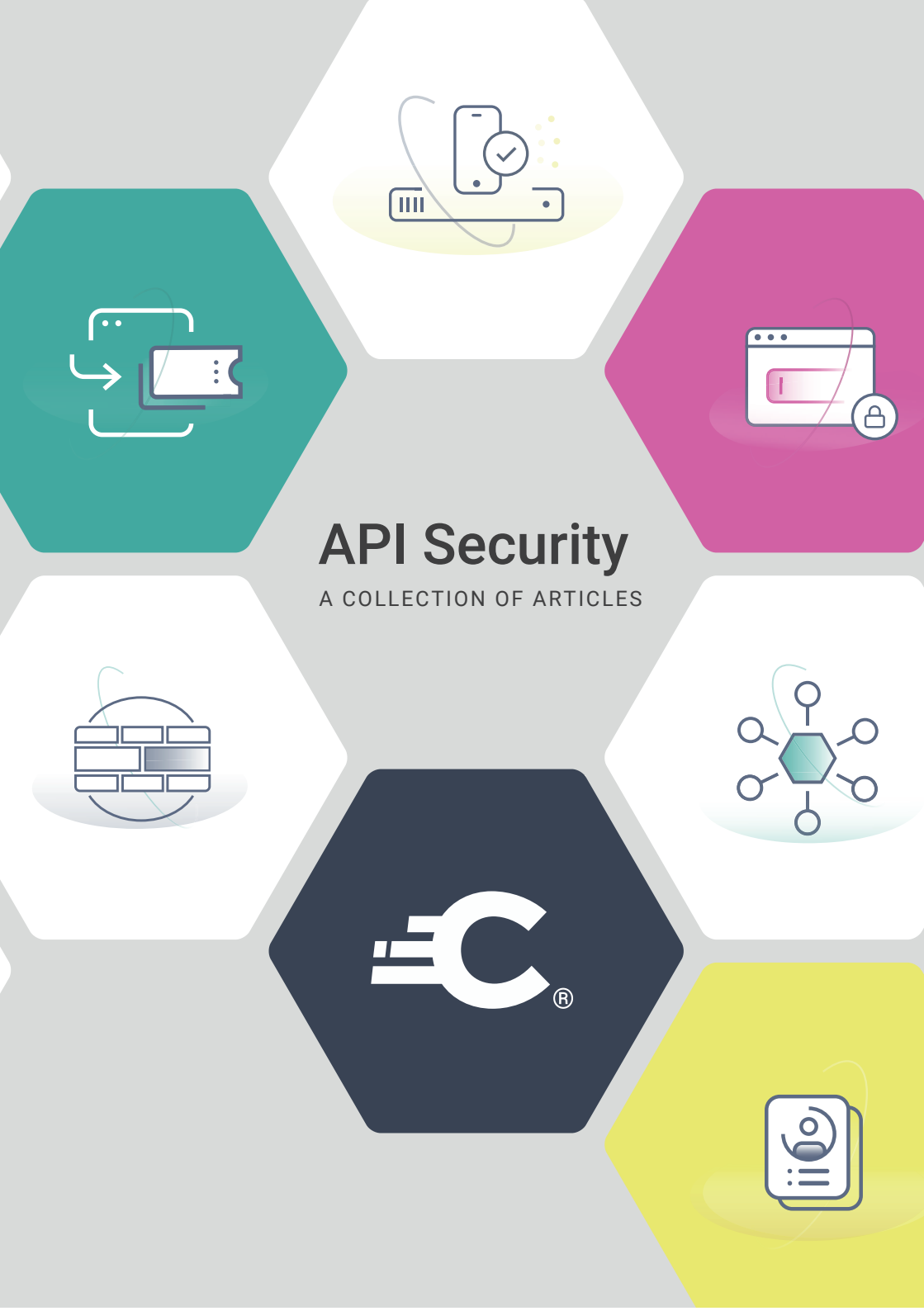

API Security
A COLLECTION OF ARTICLES

API Security for the Modern Enterprise

APIs have in recent years grown to be essential to the digital strategy of a modern organization. To ensure that digital assets are securely distributed, and that privacy is maintained at all times, proper access management needs to be in place. Keeping APIs, and the data provided through them, safe and only available to the intended user is a must. And with users who are used to moving through digital systems friction-free, an efficient identification and authorization process has never been more important. By embedding identity information in tokens, you can simplify the access control decisions that will be made in many different places throughout your architecture.

This booklet gathers a selection of articles that cover the most important aspects of securing APIs and microservices. It gives an introduction to related issues, such as how to utilize well-established standards, like OAuth 2, OpenID Connect and how to connect these to your applications, systems and user identities.

We hope you find this useful, and that it helps you secure your current and upcoming APIs. Happy reading from the Curity team!

CURITY®

Table of Contents

Contact: info@curity.io

The API Security Maturity Model

API security has become a forefront issue for modern enterprises. However, there is a spectrum of API security implementations, and not all of them are effective. Too often, APIs only adopt HTTP Basic Authentication, API keys, or token-based authentication, overlooking a major concern: identity. To prevent vulnerabilities and reap efficiency benefits, a comprehensive identity focus is critical for fully-evolved APIs.

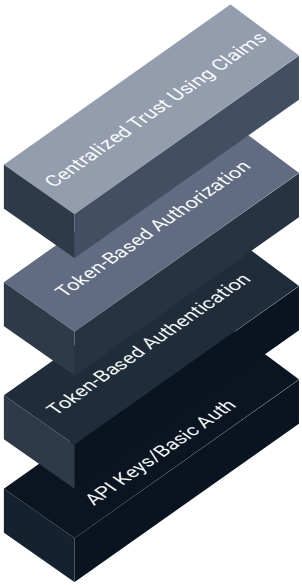
This is why we've created the **API Security Maturity Model.** Inspired by the Richardson Maturity Model, which outlines increasing degrees of web service development maturity, the API Security Maturity Model reframes the model within the context of security. Within this model, security and trust are improved the higher up you go.

The Different Levels of the API Security Model:

- **Level 0:** API Keys and Basic Authentication
- **Level 1:** Token-Based Authentication
- **Level 2:** Token-Based Authorization
- **Level 3**: Centralized Trust Using Claims

The more evolved API security is, the more identity emphasis it tends to have. So, how do we encapsulate identity with APIs and make it useful? APIs that utilize OAuth and OpenID Connect can take advantage of Claims, an advanced form of trust. Tokens such as JWTs utilizing Subject and Context Attributes can delegate platform-wide trust.

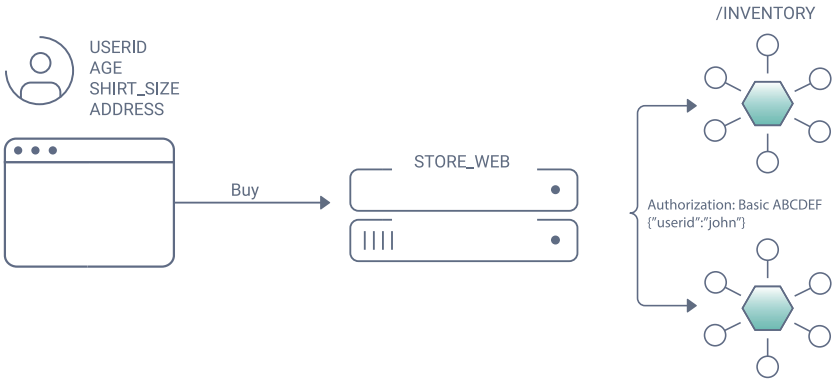
More on the specifics of that below. But first, let's expand on each maturity stage within the model to understand its benefits and drawbacks.



Level 0: API Keys and Basic Authentication

APIs at Level 0 use Basic Authentication or API keys to verify API calls. These are inserted within the header or body of the URL of the API request. This is the level of security that most APIs adopt. Most APIs established this authentication years ago, and unfortunately never evolved from there.

For example, consider an eCommerce store. It makes API calls to a payment API based on user purchases. It sends authentication in the form of an API Key or Basic Authentication in the header to the app and passes it to APIs. The user ID is placed in the Body or URL. In the example below, there are two APIs: BILLING and INVENTORY. Since HTTP Basic Authentication or API keys only authenticate the STORE_WEB, the store must pass on user data.



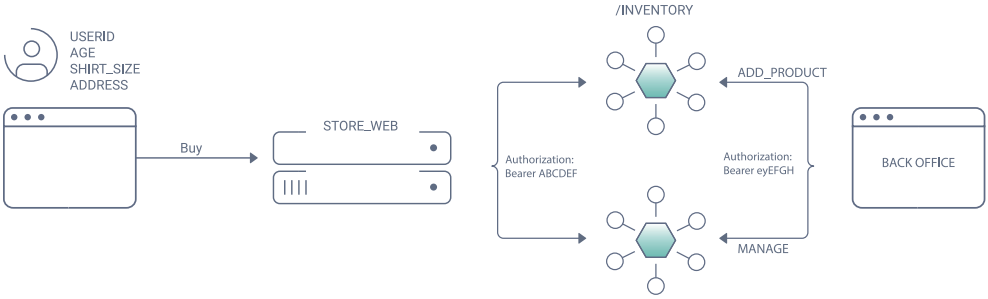
The problems with Level 0:

You may be thinking: aren't API keys sufficient? Well, this method is actually very basic, wrought with vulnerabilities. Not only are keys constantly compromised, but API key verification relies on machine-machine verification, not bound to the identity of the user at all. Lastly, this method only provides authentication, the act of proving an assertion, and does not cover authorization at all.

Level 1: Token-Based Authentication

APIs at Level 1 utilize Access Tokens for authentication within a token-based architecture. Such Access Tokens delineate the type of user (machine, app, user, etc.). As this enables privileged access, it helps in environments where the separation of internal and external users is required. Level 1 provides better auditing since user identity is part of the request.

For example, consider token-based authentication at the eCommerce store. When we introduce a back office, the same problem occurs. Custom logic is needed to know if the request is a back-office request with elevated privileges or if it comes from the store web.



The problems with Level 1:

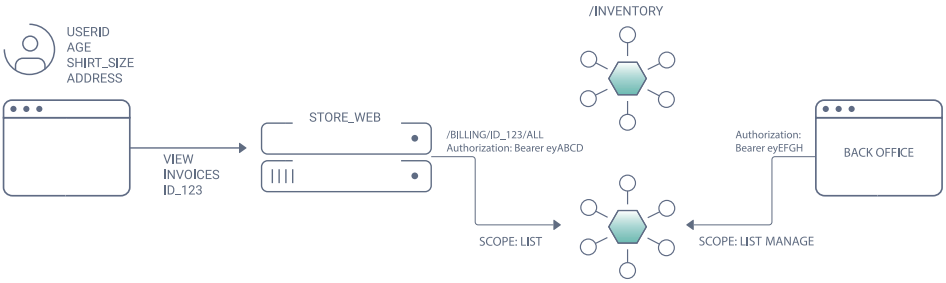
At Level 1, anyone with a token can modify the API, meaning privileged access can be hacked. Furthermore, Level 1 only covers authentication, not authorization. In other words, this strategy doesn't ask what are you allowed to do. When only using tokens for authentication, all authorization becomes custom code. Thus, custom mechanisms like if statements must be coded. This is negated in Levels 2 and 3, where you can utilize token data for authorization, thus generalizing authorization logic.

Level 2: Token-Based Authorization

APIs at Level 2 are more evolved, using token-based architecture for authorization. Authorization delineates privileges for the requesting party, asking what are you allowed to do. APIs in Level 2 adopt OAuth, a widely adopted authorization standard in which client requests require an OAuth server for authorization. Maintained by IETF, OAuth 2.0 defines varying flows to obtain tokens, enabling the ability to grant access to resources without the need for a password.

One great benefit of OAuth is Scopes. Scopes can be utilized as “named permissions” within a token. These scopes can specify user privileges. OpenID Connect defines standard scopes that can be used to generate standard identity arguments. Or, you can create custom scopes for your API. Scopes have more useful data and are better than building if statements into a system.

Let's consider our eCommerce store again. Now, we introduce Scopes, so that the public web store and back-office can have different privileges. However, some operations overlap. The Scope LIST is used to list invoices in the billing API. The ID to list for is in the URL or passed as a request parameter. Thus, it's possible to manipulate the call to list invoices for another user. Thus, the Scope is not sufficient. Scopes also lock down what the client application is allowed to do; they don't help with the particular user since they are only “names” and not “values.” Instead, Claims should be used so that the parameter is baked into the token. Then it's easy to separate back office privileges from store_web privileges.



The problems with Level 2:

One problem in Level 2 is that the system faces the threat of being decompiled. When identity is built the data request is full of errors. You can't always assume the data passed from one API to the next is always correct. These realities cause cascading issues of trust, easily becoming an intertangled mess. We call this a "spaghetti of trust."

Level 3: Centralized Trust Using Claims

The final tier, Level 3, is the most evolved API security platform. This practice involves centralized trust with Claims and possibly signed JSON Web Tokens (JWTs). In doing so, we solve all the problems outlined above.

What are JWTs? Well, to clarify common misconceptions, a JWT is NOT a protocol. It is a signed piece of data. OAuth flows utilize JWTs to verify transactions. JWTs can be used to share Scopes.

And Claims? Claims are essentially assertions. For example, consider a written statement: "Jacob is an identity specialist, says Travis." This claim has a Subject (Jacob), an Attribute (that he is an identity specialist), and an Asserting Party (Travis). If you trust Travis, then you trust the Claim. Many Attributes can make up identity. There are Subject attributes, like name, age, height, weight, etc. For these attributes, the Asserting Party would be the police or tax authorities. There are also Context Attributes, such as the situation, timing, location, weather, etc.

Instead of trusting the attributes themselves, it is far better to trust claims made by common parties. Identity systems use Claims with similar anatomy for verification. If you trust the OAuth Server that issues keys, then you trust the claim being made. To verify a claim (simplified):

- Requesting Party calls the Issuer
- Issuer returns data, signed with a private key
- Requesting Party sends to another party
- Replying Party verifies the signature with a public key

This method solves the issue of trust, by trusting the issuer of tokens rather than the claims themselves.

JWTs Require OAuth & OpenID Connect

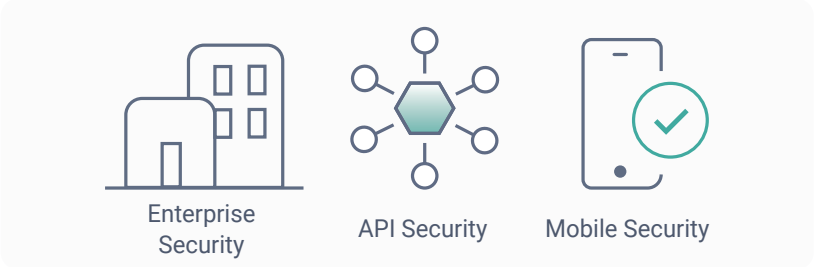
In cybersecurity, it's rarely encouraged to invent your own traffic rules. For centralized trust to function, authorization systems require the use of stable protocols. Just as street traffic follows common protocols, identity systems require their own shared open standards. These protocols are OAuth and OpenID Connect. Utilizing these standards, an app can share secure, asserted data within JWTs for verification.

Deep Dive into OAuth and OpenID Connect

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.

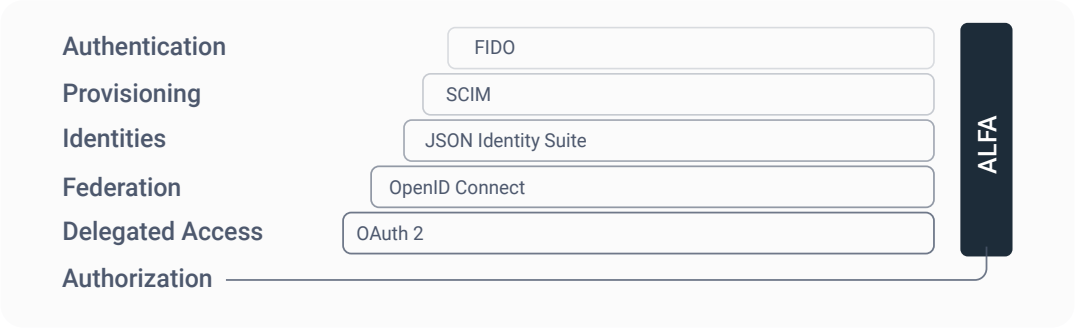
To account for all three of these security concerns, you have to know who someone is and what they are allowed to do. To authenticate and authorize someone on your servers, mobile devices, and in your API, you need a complete Identity Management System. At the head of API security, enterprise security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

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 article.

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, NAPS, and UMA). This is similar to the way WS-Trust was used as the basis for WS-Federation, WS-Secure, 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:

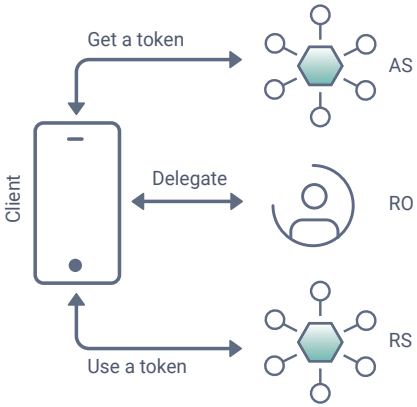
- Delegated access
- Reduction of password sharing between users and third parties (the so called "password anti-pattern")
- Revocation of access

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 (RO)
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 requesting 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 and Token Purpose

In OAuth, there are two kinds of tokens, or put in other words, tokens with different purposes:

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 website. As long as that session is valid, you can continue to interact with the website 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 Authorization Server 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 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! 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.

Profiles of Tokens: Token Types

There are different profiles of tokens as well, in the spec this is loosely referred to as the token type. 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 salesperson 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 whom the token was issued, HoK tokens should be used.

HoK tokens are like a credit card. If you find a 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 tokens

Token Format

We also have different formats of tokens. The OAuth specification doesn't stipulate any particular format 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 format isn't one area where interop has been an issue. Quite the contrary! In practice, you'll often find tokens of various formats 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. When passing by value, 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 user (typically the Resource Owner)
- 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 lightweight 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, 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 and various ancillary specs. In the case of OAuth, this is including the Bearer Token spec.

OAuth Flow

OAuth base specification 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's 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 website 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 log in 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.

Improper and Proper Uses of OAuth

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

- OAuth is not used for authorization. You might think it's from its 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.

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 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. 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 fewer moving parts and reduced complexity.

OpenID Connect is a modern federation specification. It's a passive profile, meaning it's 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. 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 kind of 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 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. The user info 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.

Conclusion

In this article, I dove into the fundamentals of OAuth and OpenID Connect and pointed out their place in the Neo-security Stack. 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) should have this basic knowledge. It is a prerequisite for pretty much everyone on your dev team.

JWT Security Best Practices

JSON Web Tokens (JWTs) are quite common in the OAuth and OpenID Connect world. We're so used to them that we often don't pay much attention to how they're actually used. The general opinion is that they're good for being used as ID tokens or access tokens and that they're secure — as the tokens are usually signed or even encrypted. You have to remember though, that JWT is not a protocol but merely a message format. The RFC just shows you how you can structure a given message and how you can add layers of security, that will protect the integrity and, optionally, the content of the message. JWTs are not secure just because they are JWTs, it's the way in which they're used that determines whether they are secure or not.

This article shows some best practices for using JWTs so that you can maintain a high level of security in your applications. These practices are what we recommend at Curity and are based on community standards written down in RFCs as well as our own experience from working with JWTs.

What is a JWT Token?

A JSON Web Token (JWT, pronounced "jot") is a compact and URL-safe way of passing a JSON message between two parties. It's a standard, defined in RFC 7519. The token is a long string, divided into parts separated by dots. Each part is base64 URL-encoded.

What parts the token has depends on the type of the JWT: whether it's a JWS (a signed token) or a JWE (an encrypted token). If the token is signed it will have three sections: the header, the payload, and the signature. If the token is encrypted it will consist of five parts: the header, the encrypted key, the initialization vector, the ciphertext (payload), and the authentication tag. Probably the most common use case for JWTs is to utilize them as access tokens and ID tokens in OAuth and OpenID Connect flows, but they can serve different purposes as well.

1. JWTs Used as Access Tokens

JWTs are by-value tokens. This means that they contain data. Even if you can't read that data with your own eyes, it's still there and is quite easily available. Whether it's a problem or not depends on the intended audience of the token. An ID token is intended for the client's developers. You expect it to be decoded and its data used by the client. An access token, on the other hand, is intended for API developers. The API should decode and validate the token. If you issue JWT access tokens to your clients you have to remember that client developers will be able to access the data inside that token. And believe us — if they can, they will. This should make you consider a few things:

• Some developers can start using the data from the JWT in their applications. This isn't a problem in itself but can explode the minute you decide to introduce some changes to the structure of the data in your JWT. Suddenly many integrating apps can stop working as they won't be prepared for the new structure (e.g., some fields missing, or a change to the max length of a field).

• As everyone can read what is inside the token, privacy should be taken into account. If you want to put sensitive data about a user in a token, or even Personally Identifiable Information (PII), remember that anyone can decode the token and access the data. If such information can't be removed from the token you should consider switching to the Phantom Token approach or the Split Token approach, where an opaque token is used outside your infrastructure, and JWTs are only available to your APIs, thanks to integration with an API gateway.

• Users' private data is not the only information that can be leaked in a JWT. You should make sure that you don't put any valuable information about your API in the token. Anything that would help attackers to breach your API.

• It's also good to keep in mind, that access tokens are most often used as bearer tokens. That means that you accept the token from whoever presented it to you — it's pretty much like paying with cash in a shop. If you find a $10 bill lying in the street, and pay with it for a coffee, it will be accepted, as long as it's a genuine banknote. The same applies to bearer access tokens. If that could pose problems to your application, you can change the bearer token into a Proof of Possession token (a PoP token) by adding a cnf claim — a confirmation claim. The claim contains information that allows the resource server to verify whether the holder is allowed to use the given token, e.g., a fingerprint of the client's certificate.

2. Avoid JWTs With Sensitive Data on the Front Channel

When it comes to access tokens, they can easily be replaced by opaque tokens (as mentioned in the previous section), but the ID token is always a JWT. This means that you should put extra care into what is available in the JWT so that no sensitive data is unintentionally revealed. It will be much safer for your UI client to call the user info endpoint and get the user's data from there instead of keeping it directly in the ID token.

Once tokens are cleared of sensitive data there will be no incentive for encrypting them. Even though encryption might sound like an excellent solution to keeping data private, the reality is that it is hard to configure and maintain secure encryption mechanisms. What is more, encryption requires much more computational resources to be used, something that might become a burden for high-traffic applications.

3. What Algorithms to Use

Regardless if the token is signed (a JWS) or encrypted (a JWE) it will contain an alg claim in the header. It indicates which algorithm has been used for signing or encryption. When verifying or decrypting the token you should always check the value of this claim with a whitelist of algorithms that your system accepts. This mitigates an attack vector where someone would tamper with the token and make you use a different, probably less secure algorithm to verify the signature or decrypt the token. Whitelisting algorithms is preferred over blacklisting, as it prevents any issues with case sensitivity. There were attacks on APIs that leveraged the fact that the algorithm noNe was interpreted as none (so no validation was performed) but was not discarded by the resource server (even though none was forbidden).

The special case of the none value in the alg claim tells clients and resource servers that the JWS is actually not signed at all. This option is not recommended, and you should be absolutely sure what you're doing if you want to enable unsigned JWTs. This would usually mean that you have strong certainty of the identity of both the issuer of the token and the client that handles the token, and you're absolutely sure that no party could have tampered with the token in transit.

The registry for JSON Web Signatures and Encryption Algorithms lists all available algorithms that can be used to sign or encrypt JWTs. It also tells you which algorithms are recommended to be implemented by clients and servers, given the current state of knowledge on cryptography security. If you want to ensure financial-grade security to your signature then have a look at the recommendations outlined in the Financial-grade API security profile.

When signing is considered, elliptic curve-based algorithms are considered more secure. The option with the best security and performance is EdDSA though ES256 (The Elliptic Curve Digital Signature Algorithm (ECDSA) using P-256 and SHA-256) is also a good choice. The most widely used option, supported by most technology stacks, is RS256 (RSASSA-PKCS1-v1_5 using SHA-256). The former ones are a lot faster than the latter, which is one of the main reasons for the stronger recommendation. The latter has been around much longer and offers better support in different languages and implementations. Still, if your setup enables this, and you're pretty sure that your clients will be able to use it, you should go for the EdDSA or ES256.

If you really need to use symmetric keys, then HS256 (HMAC using SHA-256) should be your choice — though using symmetric keys is not recommended, take a look at When to Use Symmetric Signing to learn why.

4. When to Validate the Token

The rule of thumb is — you should always validate an incoming JWT. You should do it, even if you're working on an internal network — where the authorization server, the client, and the resource server aren't connected through the Internet. You shouldn't rely on your environment settings to be part of

your security scheme. If you move your services to a public domain, the threat model will change, and you will have to remember to update your security measures — experience shows that this is very often overlooked. Moreover, implementing token validation from the start will guard you against situations where someone manages to break into your network, or you would have a malicious actor in your organization.

The one case when you could consider omitting to check the signature of the token is when you first get it in the response from the token endpoint of the authorization server using TLS. You should definitely validate a token if using the implicit flow, and the token is sent back to the client by means of a redirect URI, as in such a case there is a greater risk of someone tampering with the token before you manage to retrieve it.

5. Always Check the Issuer

Another claim that you should always check against a whitelist is the iss claim. When using the JWT you should be sure that it has been issued by someone you expected to issue it. This is especially important if you adhere to another good practice and dynamically download the keys needed to validate or decrypt tokens. If someone should send you a forged JWT, with their issuer in it, and you then download keys from that issuer, then your application would validate the JWTs and accept them as genuine.

This good practice can also be explained in other words: if the token contains the iss claim you should always confirm that any cryptographic keys used to sign or encrypt the token actually belong to the issuer. How to verify this will be different for different implementations. E.g., if you're using OpenID Connect the issuer must be a URL using the HTTPS scheme. This makes it easy to confirm the ownership of the keys or certificates. Thus, it's good practice to always use such URLs as the issuer value. If this is not the case, you should make sure to get to know how to check this ownership.

Also, remember that the value of the iss claim should match exactly the value that you expect it to be. If you expect the issuer to be https://example.com, this is not the same as https://example.com/ secure!

6. Always Check the Audience

The resource server should always check the aud claim and verify that the token was issued to an audience that the server is part of (as the aud claim can contain an array, the resource server should check if the correct value is present in that array). Any request that contains a token intended for different audiences should be rejected. This helps to mitigate attack vectors where one resource

server would obtain a genuine access token intended for it, and then use it to gain access to resources on a different resource server, which would not normally be available to the original server.

An ID token must contain the client ID in the aud claim (though it can also contain other audiences). You expect the token to be decoded by the client, so it can use the data inside it. This token should not be passed to anyone else. Clients should discard ID tokens that do not contain their ID in the audience claim — these tokens are not meant for this client and should not be used by it.

For access tokens, it is a good practice to use the URL of the API that the tokens are intended for.

7. Make Sure Tokens are Used as Intended

JWTs can be used as access tokens or ID tokens, or sometimes for other purposes. It is thus important to differentiate the types of tokens. When validating JWTs, always make sure that they are used as intended. E.g., a resource server should not accept an ID token JWT as an access token. This can be achieved in different ways and will depend on your concrete use case and implementation. Here are some examples:

- You can check the scope of the token. ID tokens don't have scopes, so checking whether an access token has any or a concrete scope will help you differentiate them.
- As noted before, tokens should have different values of the aud claim. If this is the case, you can use that claim's value to check the token type.
- The Curity Identity Server sets a purpose claim on the token, with values of either access_token or id_token.

Some authorization servers might set the not-yet-standardized typ claim in the token header to at+JWT for access tokens. If your server supports that, it can be used to differentiate tokens. Different types of tokens could use different keys for signing. Your services will then reject access tokens signed with keys used for issuing ID tokens. You can use sets of required claims for different types of tokens. Your services can validate whether the received token contains specific claims that you expect from an access token.

8. Don't Trust All the Claims

Claims in a JWT represent pieces of information asserted by the authorization server. The token is usually signed, so its recipient can verify the signature and thus trust the values of the payload's claims. You should be wary, however, when dealing with some claims in the token's header. The JWT's header can contain claims that are used in the process of signature verification. For example: the kid claim can contain the ID of the key that should be used for verification, the jku can contain a URI pointing to the JSON Web Key Set — a set that contains the verification key, the x5c can claim contain the public key certificate corresponding to the signature key, etc. You should make extra care

when using these values straight from the token. If these claims are spoofed they can point your service to forged verification keys that will trick your service into accepting malicious access tokens. As noted before, make sure to verify whether keys contained in such claims, or any URIs, correspond to the token's issuer, or that they contain a value that you expect.

9. Dealing With Time-Based Claims

JWTs are self-contained, by-value tokens and it is very hard to revoke them, once issued and delivered to the recipient. Because of that, you should use as short an expiration time for your tokens as possible — minutes or hours at maximum. You should avoid giving your tokens expiration times in days or months.

Remember that the exp claim, containing the expiration time, is not the only time-based claim that can be used for verification. The nbf claim contains a "not-before" time. The token should be rejected if the current time is before the time in the nbf claim. Another time-based claim is iat — issued at. You can use this claim to reject tokens that you deem too old to be used with your resource server.

When working with time-based claims remember that server times can differ slightly between different machines. You should consider allowing a clock skew when checking the time-based values. This should be values of a few seconds, and we don't recommend using more than 30 seconds for this purpose, as this would rather indicate problems with the server, not a common clock skew.

10. How to Work With the Signature

In the case of a signed JWT — a JWS — you have to remember that the signature is used to sign not only the payload of the token but also the header. Any change in the header or the payload would generate a different signature. This doesn't even have to be a change in the values of claims — adding or removing spaces or line breaks will also create a different token signature.

It's worth noting that in order to mitigate a situation where two tokens would be created with exactly the same signature (so two tokens created in the same second, for the same client and user, with the same scope, etc.) many authorization servers add a random token ID in the jti claim. Thanks to this you can be sure that two different tokens will never have the same signature.

Signatures require keys or certificates to be properly validated. These keys or certificates can be obtained from the authorization server in a few different ways. You can get the keys from the authorization server in an onboarding process, and make sure that all of your resource servers have access to those keys. This however creates a problem when the keys or certificates change. That's why it's good practice to always use an endpoint and dynamically download the keys or certificates from the authorization server (caching responses accordingly to what the server returns in the cache

control headers). This allows for an easy key rotation, and any such rotation will not break your implementation.

Remember that if the keys or certificates are sent in the header of the JWT, you should always check whether they belong to the expected issuer, for example, validate the trust chain in certificates.
As noted before, the alg claim in the header must be checked against an allow list.

11. When to Use Symmetric Signing

The rule of thumb here is to try to avoid using symmetric signing at all. Nowadays, there are probably not many use cases where you would have to use symmetric signing instead of asymmetric. When using symmetric keys then all the parties need to know the shared secret. When the number of involved parties grows it becomes more and more difficult to guard the safety of the secret, and to replace it, in case it is compromised.

Another problem with symmetric signing is the proof of who actually signed the token. When using asymmetric keys you're sure that the JWT was signed by whoever is in possession of the private key. In the case of symmetric signing, any party that has access to the secret can also issue signed tokens.

If, for some reason, you have to use symmetric signing try to use ephemeral secrets, which will help increase security.

12. Pairwise Pseudonymous Identifiers

The OpenID Connect standard introduces Pairwise Pseudonymous Identifiers (PPID) that can be used instead of a plain user ID. A PPID is an obfuscated user ID, unique for a given client. This helps to improve users' privacy. Especially if the user ID is represented by sensitive data e.g., en e-mail or the Social Security Number. Thanks to PPID, the client can still differentiate users, but will not get any excess information.

13. Do Not Use JWTs for Sessions

There is a popular belief among web developers that JWTs have some benefits for use as a session retention mechanism — instead of session cookies and centralized sessions. This should not be considered good practice. JWTs were never considered for use with sessions, and using them in such a way may actually lower the security of your applications.

Conclusion

This article has explored the best practices when using JSON Web Tokens as a way of strengthening API Security in web applications. It's important to remember that JWT safety depends greatly on how tokens are used and validated. Just because a JWT contains a cryptographic signature it doesn't automatically mean that it's safe, or that you should blindly trust the token. Unless good practices are observed your APIs can become vulnerable to cyber-attacks.

The good practices outlined in this article are true at the time of writing and we are making sure to keep them up to date. You should remember, however, that security standards and the security levels of cryptography can change quite rapidly and it's good to keep an eye on what is happening in the industry. You can follow any changes in RFCs that talk about the good practices for JWTs: in RFC 8725 JSON Web Token Best Current Practices and in RFC 7518 JSON Web Algorithms (JWA).

Coarse-Grained Authorization Using Scopes

How does one go about securing APIs, microservices, and websites? One way to do this is by focusing on the authorization - knowing what the caller is allowed to do with your data. Too often, though, providers rely too heavily on user identity, pairing it way too closely with the design of their APIs.

As OAuth doesn't authenticate by itself, the way these flows are structured means that API access often ultimately relies on user social logins, which is an unfavorable dependency that actually decreases API security and scalability. APIs are far better secured with a proxy in-between the API and authentication mechanism, utilizing scopes that delineate the type of access that the API grants.

Authorization needs to be done on different levels. We now focus on the coarse-grained level, where we can decide if we should at all consider the request. The next article will discuss the fine-grained setup where we can make decisions on a per user/request level.

In this article we'll see why APIs and microservices should decouple user identity from their designs, and how to go about this implementation. We'll review some sample flows, and briefly walk through how OAuth scopes can be used to create a more valuable, knowledgeable API. Following these cues, the end result will emboss state-of-the-art Identity and Access Management (IAM) practices within actual API design, in effect utilizing identity data to secure the entire API lifecycle.

AAA

When an API call is made, we must know who made the request, and if they are allowed to read and access the requested data. Identity and Access Management is also described as AAA; an important initialism made up of:

- Authentication: Validation that the user is who they say they are.
- Authorization: Validation of the user, their application, and privileges.
- Auditing: Accounting for user behavior, logging metadata like what is accessed, when it's accessed, with what device, and more.

Caching user logs is great, but it doesn't prevent malformed requests to the server from the onset. We don't want to waste resources, so verifying requests must be processed as early as possible in the code pipeline. So, typically you block this with a proxy. BUT how do you make sure that the proxy knows what to do? How do we instruct the proxy to decipher what user or application is accessing the data, and what data they are allowed to access?

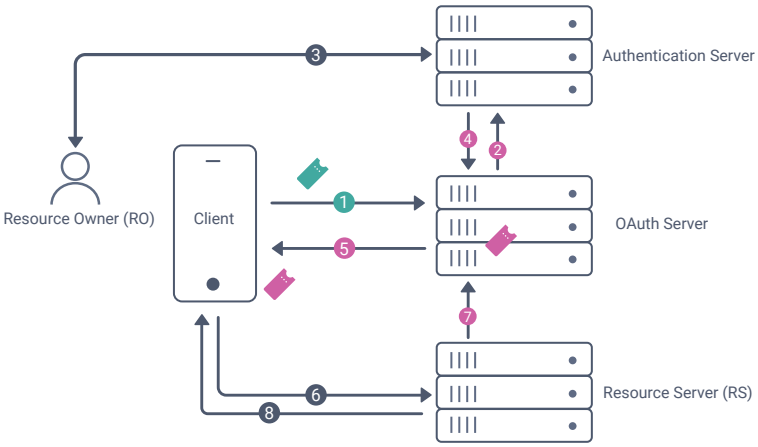
Overview of the OAuth Flow

You may be thinking, just use OAuth, problem solved. Yes, OAuth is a necessary protocol within the security workflow — but OAuth cannot authenticate — a separate server must tell OAuth who the user is. To understand where we're heading, here's a quick overview of a simple OAuth flow. To review, the actors are:

- Resource Owner (RO): The user
- Client: The application, mobile device, server, or website requesting data from the API
- Authentication Server: The login service that authenticates users
- OAuth Server (AS): Also called the Authorization Server
- Resource Server (RS): The API providing data

Let's assume that we have created a mail server with an API that provides information so that a third party app client can sort emails in an improved way. In our walkthrough we'll assume that the app uses Google as an authentication service. There are variants of these flows, but a simple OAuth flow for this scenario would be:

1. The Client first requests accesses the OAuth Server.
2. The OAuth Server next delegates authentication responsibility to a third party Authentication Server.
3. The User enters credentials with the Authentication Server to authenticate.
4. The Authentication Server tells the OAuth Server the authentication was successful.
5. The OAuth Server sends the Token to the Client.
6. The Client uses the Token to access resources from the Resource Server.
7. The Resource Server verifies with the OAuth Server that the Token is valid.
8. The Resource Server (API) then sends the data to the Client app.



Designing an API with Scopes from the Bottom Up

Located within the OAuth token, scope is an interesting data point that you've likely used before. Scope specifies the extent of tokens and are akin to the permissions listed on a consent UI. They are extremely useful, as scopes can be used to delineate API access tiers. Furthermore, OAuth doesn't specify that you have to give the same scopes that you are requesting — if the scope changes you must simply notify the client/user. For access management designers, this grants us a lot of power and flexibility in how we handle scopes and identity. We'll see that building an API with scopes hardwired into the design can be extremely helpful.

Scopes are tied to the client which is where this becomes useful from a coarse-grained perspective. Consider an Invoicing API, the API can create and list invoices. Customers using the company app should only be able to view the invoices, while the internal Finance systems should be able to create invoices. It doesn't matter if a super admin, or a regular customer logs in with the customer app, they will only ever be able to list invoices. In the next article we'll discuss which invoices they will be able to list.

Let's build a sample API to see what we're talking about. We could for example design an Invoice API that taps into a e-commerce platform. The API provides access to customers, who are listing the invoices, as well as to employees, who are writing the invoices.

On the other end, ECommerceApp is an application that consumes the Invoice API. You can think of ECommerceApp as the client in our OAuth flow. As it's a customer client, it will be limited in scope without editing capability.

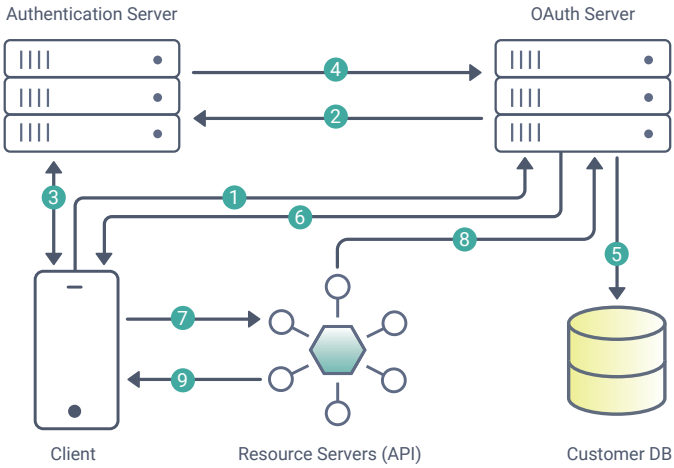
So how do we create permissions? To do so means we define the scopes in the API. The beautiful magic here is that we assign these scopes with different strengths as follows:

SCOPE	STRENGTH	BEHAVIOUR
`No Scope`	Weak	User does not have to be a registered customer
`invoice_read`	Medium	User must be a registered customer
`invoice_write`	Strong	The application must be internal and the user needs to login using internal credentials on the corporate network.

Let OAuth Filter These Scopes

Next we let the OAuth server filter API access based on these scopes. Since we can change our scopes throughout the process, when ECommerceApp sends a request with an `invoice_read` scope, a new flow would look like this:

1. The ECommerceApp Client makes a request to the OAuth Server sending a basic read scope.
2. The OAuth Server next delegates authentication responsibility to a third party Authentication Server.
3. The User enters credentials with Google, the Authentication Server, to authenticate.
4. The Authentication Server tells the OAuth Server the authentication was successful, and sends an OAuth Token. Within the token is information that will affect the scopes, namely the ACR (which in this case is social) and the subject (the username).
5. The OAuth Server checks its Customer Database to see if the username is in fact a customer.
6. In this case it does find the username among the customer files, and thus grants the client an Access Token with the invoice_read scope.
7. The ECommerceApp Client then sends the Access Token to the Invoice API Resource Server.
8. The Resource Server verifies with the OAuth Server that the Token is valid.
9. The Resource Server (API) then sends the data to the Client app.



The Proxy Accepts Only Valid Requests

The last step in decoupling authentication from API design is constructing a proxy to separate our API from the authentication mechanism. The final result is a proxy that only allows access to the Invoice API when the following criteria are met:

- The token exists
- The token is valid
- The token contains one or more scopes. For the Invoice API, that would be one or more of the following:

```
invoice_read
invoice_write
```

Now we have a more secure, strict API front end that only allows access once these three rules are met, blocking unregistered users in the proxy.

Conclusion: Separate the API from Authentication and Client Permissions

In order to build a scalable API infrastructure that is ideal for microservices, you must design your APIs in a way that separates them from authentication. Using scopes to map the permissions, and defining them in your API, can create a robust platform that better protects and informs you as an API provider.

Benefits of this approach also include:

- Overall API security is improved with abstraction;
- API identity control now maps your access tiers, enabling easy enforcement for freemium business models;
- This pattern is simple to grasp and implement;
- Constructing scopes into API design rather than third party authentication means freedom of any authentication method without bothering the APIs with all the details;
- Can help in separating private, public, partner APIs — complementing platform strategy and adding business potential;
- Could be used to inform usage analytics;
- As marketing departments have high demand on smooth customer journeys, this provides a quicker time to market when it comes to authentication.

But perhaps the most critical point is that one and only one pattern is needed for microservices design. This increases the ability to not only build APIs, but easily share identity knowledge across an organization, increasing the service maintainability over time. Authentication is a moving target, whereas APIs may not be.

Fine-Grained Authorization Using Claims

In the previous article we discussed how to use OAuth Scopes to perform coarse-grained authorization. This helps with the separation of access for different applications. The next step is to continue and map the exact access that the user needs. Given the example with an Invoice API, it's obvious that the coarse-grained scope tells us whether or not the application is allowed to perform the listing of invoices, but what the API really needs to know is which invoices the application may list. This is data tied to the user which can be communicated using claims in the OAuth Access Token.

Attributes vs Claims

To understand what claims are, we need to start with attributes. Attributes are properties of a user, such as username, name, age, shoesize etc. They can also be attributes about the session, such as when the user logged in, from what location etc. For that reason, we typically split attributes in two categories: Subject Attributes and Context Attributes.

Subject Attributes are true about the user no matter how the user logged in. Depending on authentication method and orchestration during login, you may receive different Subject Attributes. I.e. if a user logs in using Google, Google will provide certain attributes, and when they login using Active Directory other attributes may be present. It is up to the authentication service to normalize these attributes and make sure the relevant Subject Attributes are always present.

Context Attributes on the other hand tell us something about the circumstances under which the Subject Attributes were established. The time of authentication, the location, what authentication method was used etc. These are relevant when issuing OAuth tokens, because certain properties of a token may be considered more sensitive and should perhaps only be present if we're fairly confident the user is close to the computer. So, if the authentication time is further back in time than say 30 minutes, we can drop properties in the token to weaken its strength.

Now that we understand attributes, we need to look at claims. Attributes are only interesting if we trust the party that issued them or put in other word: if we trust the party that claim them to be true.

A claim has the following form:
Jacob has a Horse, says Travis:
Jacob – is the subject
"has a horse" – is the claim / attribute
Travis – is the asserting party.
If we trust Travis, then we can trust the claim about the horse.
More formally: Subject + Attribute + Issuer = Claim

Claims

Claims are a first-class citizen in OpenID Connect and are easily transferrable to OAuth.
The most common place where you encounter these are in the ID Token which is a JSON Web Token (JWT) but they can be generalized to tokens of any purpose and format:

```
{
   sub: janedoe@example.com
   name: Jane Doe
   iat: 1546300800
   exp: 1893456000
   iss: https://login.curity.io
   subscriber_id: ABC_123
   phone_number: +46 123 123 123
}
```

This JWT contains the following claims: “name”, “iat”, “exp”, “subscriber_id”, “phone_number”.
It also has a subject “sub” and an issuer “iss”. The JWT is normally signed by a key that matches what the issuer has documented or published out of band. A receiver of this token can verify that the “iss” field is a host it trusts and use a key provided by that host to verify that the token has indeed been issued by that party. Once that is done, we can rely on the claims in the token and act on the information.

OpenID Connect describes standard claims for ID Tokens, but the claims infrastructure is not limited to only those tokens. Access tokens can use the same structure, even when using other formats than JSON Web Tokens. This makes claims very powerful when designing systems in need of finer-grained authorization.

Scopes Revisited

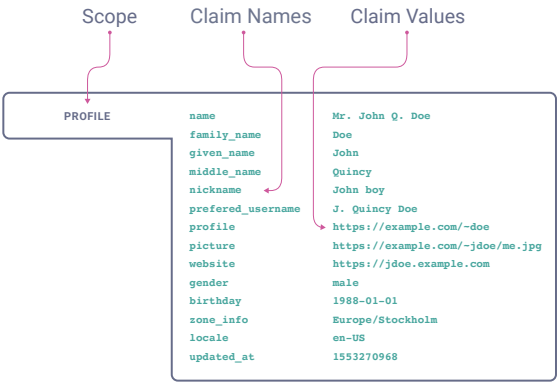
In the previous article we discussed the scope’s role in authorizing access for the client. But scopes play a deeper role in a claims-based system. Without claims, a scope is just a space separated list of strings with Scope Tokens or Scope Names.
Example: scope = “invoice_read invoice_write openid email”

It’s good to think of the scope parameter as Scope of Access. I.e. this lists the things the client needs to access.

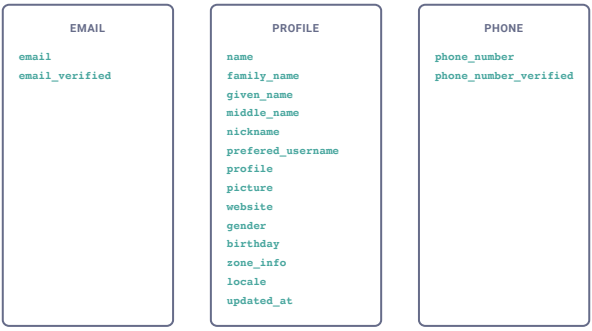
We give each of these meaning in coarse grained authorization, to see if the client should be allowed to query the API. But as discussed in the scope article, even if a scope of invoice_read is present in the token, we don’t know which invoice it should be allowed to read.

Group of Claims

OpenID Connect defines the scope as a group of claims. So, the “email” scope token is mapped to the “email” and “email_verfied” claim. The claims have values associated to them while the scope token is just a name. This means that a scope is simply a bag of claims. Requesting a scope will result in zero or more claims to be issued and present in the tokens.



Each OpenID claim is associated with a scope as shown in the image below.



This can be generalized to arbitrary scopes and claims. When it comes to API access this becomes very useful.

Consider the example of an invoice API. Let's assume we have the following scopes:

```
invoice_list
invoice_read
invoice_write
```

As already discussed, it's not enough to authorize the particular request based on only this information. The API needs to know which invoices that it is allowed to list. Listing the actual invoice IDs in the token is not very efficient, but let's assume we have a claim which is account_id. It is the financial account that is associated with the user.

SCOPE	CLAIM ASSOCIATED WITH SCOPE
`invoice_list`	`account_id, role=customer`
`invoice_read`	`account_id, role=customer`
`invoice_write`	`account_id="*", role=finance`

If any of these scopes are requested, the resulting token will contain the claims needed with an associated value for that user. Now the API can easily know if the requested operation should be authorized or not.

What we have done now is create a contract. The API knows that if the role=customer it needs to check the account_id claim to see for which account the invoices should be provided and if the role=finance, it allows writes.

This mitigates many risks. The API no longer needs to rely on data provided from unreliable sources, but can safely operate on the account, and scope information when performing the task. It makes it virtually impossible for a third party to inject a different account_id in the request and the API doesn't have to look up additional data about the user. The example should be considered illustrative but shows the essence of how to use claims.

Describing Login Information

It's not only the API that will benefit from claims being present in the token. When using OpenID Connect, the client (application) might also be interested in knowing details about the user. In many cases it is interested in knowing details about the context in which the user authenticated.

As mentioned previously, the Context Attributes provide this information. Using the OpenID Connect ID Token, the client can determine not only who logged in, but also when and how. These are part of the standard claims that are associated with the openid scope implicitly.

```
sub: who logged in
auth_time: when the login occurred
acr: Short for Authentication Context Class Reference, this stipulates how the
login happened.
```

It is not unusual for applications to require knowledge if the authentication was fresh or if it occurred with SSO, i.e. the user didn't interact. This can be critical when deciding if sensitive data should be displayed or not.

Knowing how the user logged in can also be important, since it can help in decisions around what actions a user may be allowed to take. A user that logged in with a strong authentication could be allowed to change its account details, while a user that didn't may perhaps only view the same data.

Conclusion

Claims are the contents of a token. They are asserted by the issuer, which in this case is the OAuth or OpenID Connect server. They provide a powerful mechanism to help both the API and the Client make qualified authorization decisions. Claims are based out of the OpenID Connect standard and provide a mechanism to help the API being more fine-grained in the authorization of the requests.

The Phantom Token Approach

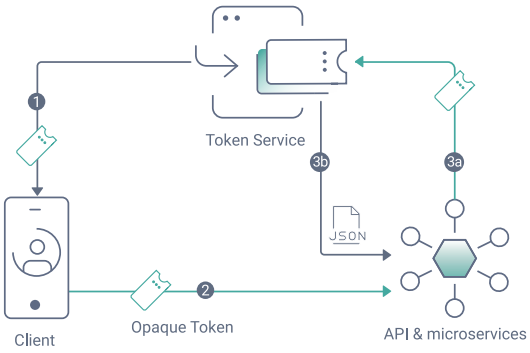
The Phantom Token Approach is a privacy-preserving token usage pattern for microservices. It combines the benefits of opaque and structured tokens. To understand the pattern it is therefore essential to understand the basic differences between these token types.

OAuth 2.0 Token Types

When OAuth 2.0 was defined tokens were intentionally kept abstract and the format was not defined. There is basically no limitation on the format of tokens that an authorization server may issue. In practice you can distinguish two types of tokens:

- Opaque tokens (by reference)
- Structured tokens (by value)

An opaque token is a random string that has no meaning to the resource server thus the token is opaque. However, there is metadata connected to the token such as its validity or the list of approved scopes that may be of relevance or even vital for the authorization decision of the resource server AKA API or microservice. In a system using solely opaque tokens the resource server cannot retrieve this kind of information from the token itself but must call the authorization server by sending a request at the introspection endpoint as illustrated below.



An opaque token can be seen as the reference to the user attributes and token metadata. Thus passing an opaque token can also be referred to as passing a token by reference.

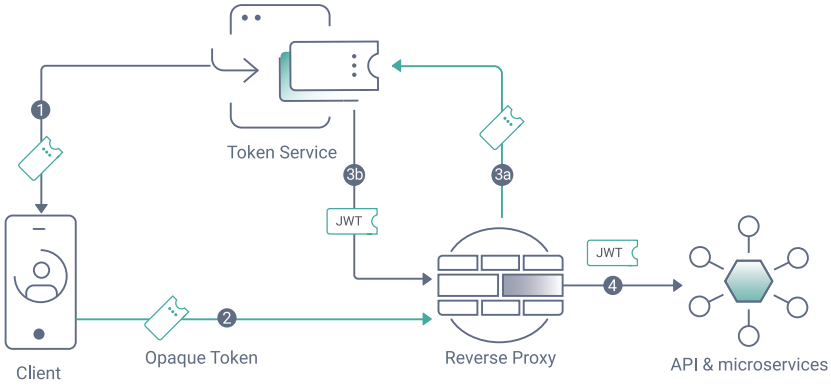
Having to look up each token for validation will inevitably create load on the resource and authorization server as well as infrastructure. Structured token formats such as JSON web tokens (JWT) solve this problem. JWTs are compact and lightweight tokens that are designed to be passed in HTTP headers and query parameters. They are signed to protect the integrity of its data and can even be encrypted for privacy reasons. Since the format is well defined the resource server can decode and verify the token without calling any other system.

Structured tokens are tokens passed by value. The token contains enough data for the resource server to make its authorization decision. Often, it also contains user information. In certain cases such a token may even contain personal identifiable information (PII) or other data protected by law or regulations and the token as well as related systems become a subject to compliance requirements.

The Phantom Token Approach

The Phantom Token Approach is a prescriptive pattern for securing APIs and microservices that combines the security of opaque tokens with the convenience of JWTs. The idea is to have a pair of a by-reference and a by-value token. The by-value token (JWT) can be obtained with the help of a by-reference equivalent (opaque token). The client is not aware of the JWT and therefore we call the token the Phantom Token.

When a client asks for a token the Token Service returns a by-reference token. Instead of having the APIs and microservices call the Token Service for resolving the by-reference token for every request the pattern takes advantage of an API gateway, reverse proxy or any other middleware that is usually placed between the client and the APIs. In that way the APIs and microservices can benefit from the JWT without exposing any data to the client since the client will only retrieve an opaque token.



1. The client retrieves a by-reference token using any OAuth 2.0 flow.
2. The client forwards the token in its requests to the API.
3. The reverse proxy looks up the by-value token by calling the Introspection endpoint of the Token Service.
4. The reverse proxy replaces the by-reference token with the by-value token in the actual request to the microservice.

Benefits of Using Opaque Tokens

The main benefit for opaque tokens is security. Access tokens are intended for the resource server, the API. However, a client may violate this rule and parse a token nevertheless. By using opaque tokens clients are prevented from implementing logic based on the content of access tokens. In addition opaque tokens for clients limit the regulated space and remove the risk of data leakages and compliance violations. It is simply not possible for the client to access or leak any data because it is not given any.

At the same time security is increased performance is optimized. The microservices will use JWT tokens that contain all the data that the service requires for processing. No need for time consuming requests. In addition the pattern can utilize caching mechanisms of the reverse proxy. A by-value token can be cached until it expires. As a result the number of requests needed for token exchange is minimized and the system's performance is optimized.

The Phantom Token Approach is compliant with the OAuth 2.0 standard. Neither the client nor the APIs have to implement any proprietary solution for this pattern. This makes the pattern vendor neutral and applicable for any OAuth 2.0 ecosystem.