IDENTITY AND APIs

BY
THE NORDIC APIS WRITING TEAM

WITH A FOREWORD BY
TRAVIS SPENCER, CEO, CURITY

CURITY

NORDIC APIS

Identity And APIs

Techniques to Mature Platform Security

Nordic APIs

Contents

Foreword

by **Travis Spencer**

The other day, I rewatched the recording of my first-ever presentation at Nordic APIs' inaugural event in 2013. In that, I discussed digital identity and its bearing on APIs. I pointed out that a new stack of security standards had emerged with the advent of OAuth, OpenID Connect, and SCIM. I ran through some flows and specs that I believed would shape API security for the coming decade, and advised listeners to abandon antiquated predecessors. From my first contribution to this community, I have sought to explain how important it is to know who is on the other end of the wire communicating with our APIs.

The next day, Bill Doerrfeld emailed me a manuscript of this ebook. The serendipitous timing struck me in two ways. First, this central, important intersection of digital identity and APIs continues to be paramount to all practitioners in the space. It is not surprising that 5 out of 10 of the most popular videos on the Nordic APIs YouTube channel are related to API security. This aspect of APIs is *really* challenging, and almost all deployments must overcome it. Secondly, we have come *very* far as a community. From that short 20 minute presentation to this and other ebooks, blog posts, presentations, and workshops flowing out of this community, we've made tremendous progress that has lifted not only this group but, I dare say, IT in general.

On behalf of myself and Curity, we are very pleased to be a part of Nordic APIs and contribute to this critical discussion. I believe that digital identity vis-a-vis APIs will continue to be a central theme in the API space for the coming decade. I think the nuances will change slightly, and that the standards will evolve.

However, what we have today – the info covered in this book – will remain the underpinnings and requisite know-how. Digging into this ebook and the topics it addresses will be well worth the effort and rewarded for years to come.

I hope you enjoy reading the work of Bill and his team, and that this ebook deepens your understanding of this key intersection of identity and APIs!

– Travis Spencer

CEO, Curity
Co-Founder, Nordic APIs

Preface

by **Bill Doerrfeld**

If you're into building with APIs, you're probably introducing new software architecture to bring scalability and efficiency advancements. But are you introducing new vulnerabilities as well?

Developers often don't traditionally build from an external-facing viewpoint. But, in today's cloud-native world of Bring Your Own Device and remote access, security is a more paramount issue than ever, even for internal systems. "If we don't take these concerns early on, they will cause catastrophic problems later on," warned Keith Casey in a recent Nordic APIs webinar.

Continual exploits demonstrate a lack of security maturity across the cloud industry, underscoring the need for more security forethought. To combat these prevalent issues, many cybersecurity experts now turn to **identity**. At Nordic APIs events, we've repeatedly witnessed speakers stress the importance of identity handling to mature API platforms at scale. As digital ecosystems evolve, so must access management strategies.

Authorization and authentication systems have changed significantly for microservices over the past few years. From HTTP Basic Auth, to API Keys, to OAuth 2.0. Now, experts view OAuth Scopes and Claims as the most mature form of API security. Centralized trust using Claims is "the place you want to get to in order to mature your API Security model," says Jacob Ideskog of Curity. With this approach, authorization is uniform and reliable, and attack vectors (or room for honest mistakes) are significantly reduced.

In short, to plug security gaps, **software architects must consider identity alongside an API strategy**. So, to torchlight the identity fires, we've assembled the top relevant Nordic APIs articles on *APIs and Identity*.

We've organized *APIs and Identity*, into three parts:

- **Part One** introduces basic concepts related to API security and identity. Familiarize yourself and see where you sit on the API Security Maturity Model.
- **Part Two** dives deeper into OAuth flows, giving you the tools to see which is best for your scenario. See how other open identity standards like SCIM make this all possible.
- **Part Three** looks at high-risk sectors where API security needs the most focus, arguing why an identity emphasis is so important in these areas.

We've also linked to the original source for each article, so you can review the deeper conversations that many of these articles inspired.

So, please enjoy *APIs and Identity*, and let us know how we can improve. If you haven't yet, consider following Nordic APIs and signing up to our newsletter for bi-monthly blog updates and event announcements. We also accept blog contributions from the community - if interested, please visit our Create With Us page to submit an article.

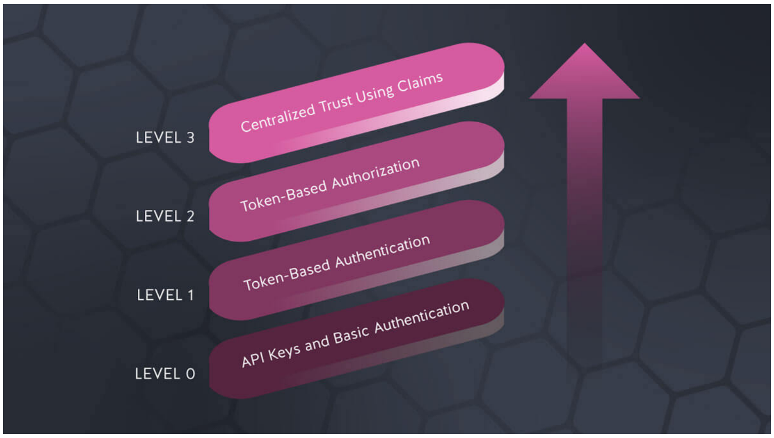
Thank you for reading!

– Bill Doerrfeld, Editor in Chief, Nordic APIs

Part One: Basic Identity Concepts

Introducing The API Security Maturity Model

by **Kristopher Sandoval**
Originally Published Here


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

Identity ranks high atop the API Security Maturity Model

When a user utilizes a service, they must first attest that they are who they say they are. In most use cases, they must then attest that they can do what they're trying to do. For many users, this is an opaque process that happens magically behind the scenes. The reality of implementing this system, however, has resulted in an ever-evolving security landscape that requires specific modes of authentication and authorization.

Thus, the question becomes apparent: how can we encapsulate information about the user, their rights, and their origin, in a useful way? Wouldn't it be great if an API could know who you are, whether to trust you, and whether you can do what you claim to do? That's the idea behind the **API Security Maturity Model**. Today, we're going to dive into this model and look at the fundamental approach to security that if offers.

Review: Richardson Maturity Model

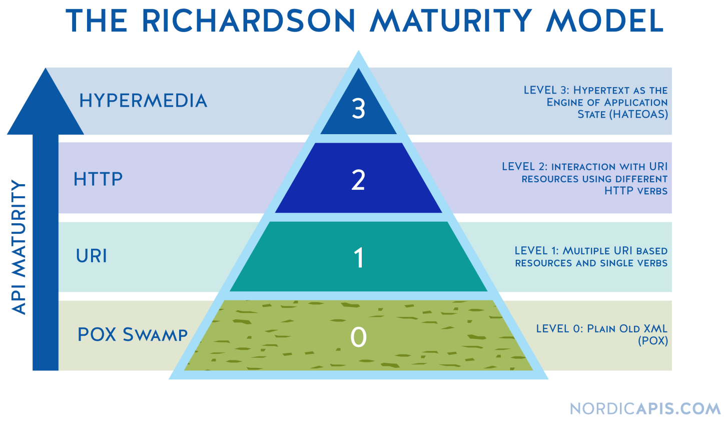
The API Security Maturity Model was invented by Jacob Ideskog, Solution Architect & Identity Specialist at Curity. The idea arose as a corollary to the Richardson Maturity Model. As such, before we dive into the API Security Maturity Model, let's first examine its inspiration.

Leonard Richardson created the Richardson Maturity Model to reflect the reality of the RESTful API space. In essence, it's possible for an API to be REST-like while not being truly RESTful. Accordingly, REST compliance is not a duality, but rather a series of levels of increasing compliance.

- **Level 0**: The Richardson Model has four levels, though the first level is considered "level zero," as it reflects absolutely no REST compliance. This level of maturity represents an API that doesn't make use of hypermedia URIs, often has a single URI and method for calling that URI.
- **Level 1**: As we get more complicated, we enter Level one, where we start to see the use of multiple URIs with simple interactions. Key to remember here is that these multiple URIs still have simple verbiage usage.
- **Level 2**: Level Two is a significant evolution in that it boasts both multiple URIs and multiple ways of interacting with that data. This level sees far more complex APIs than the previous levels due to the nature of the verbiage used – specifically, CRUD (Create, Read, Update, and Delete) is

usable on exposing resources, allowing complex manipulation.

- **Level 3**: The highest level of maturity, APIs in this stage are truly "RESTful" in that they employ Hypermedia as the Engine of Application State (HATEOAS). This results in a highly complex and powerful API, both boasting multiple URIs and the verbiage to interact with them as well as the power behind hypermedia.


THE RICHARDSON MATURITY MODEL
API MATURITY
HYPERMEDIA
HTTP
URI
POX SWAMP
3
2
1
0
LEVEL 3: HYPERTEXT AS THE ENGINE OF APPLICATION STATE (HATEOAS)
LEVEL 2: INTERACTION WITH URI RESOURCES USING DIFFERENT HTTP VERBS
LEVEL 1: MULTIPLE URI BASED RESOURCES AND SINGLE VERBS
LEVEL 0: PLAIN OLD XML (POX)
NORDICAPIS.COM

The Richardson Maturity Model, predecessor to The API Security Maturity Model

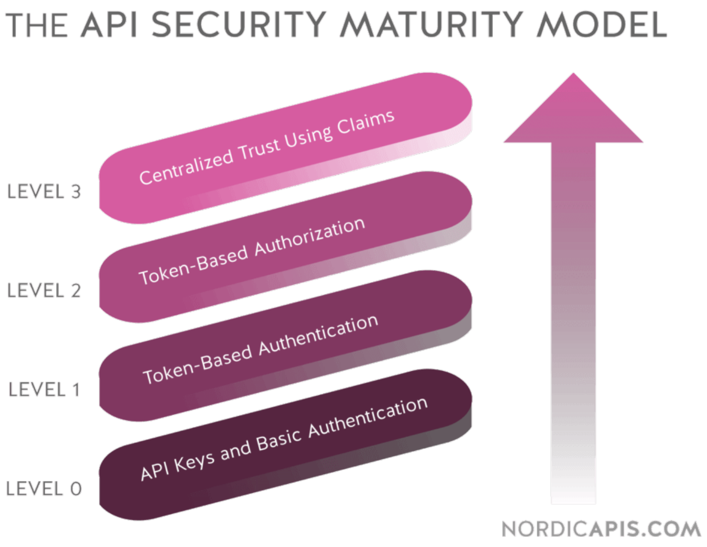
The API Security Maturity Model

The interesting thing about the Richardson Maturity Model is that it's not simply different states of compliance. It represents cumulative upgrades from level to level, with each new step including previous gains and leveraging new additions.

Similarly, the API Security Maturity Model describes API security as ever-increasing levels of security, complexity, and efficiency.

The API Security Maturity Model model, like the Richardson Model, moves from the lowest-maturity to the highest and can be considered akin to a playbook for how to progress into a secure platform deployment.

With this in mind, let's take a look at the specific levels of the API Security Maturity Model, starting with the lowest maturity level and moving towards the highest.


THE API SECURITY MATURITY MODEL
LEVEL 3
Centralized Trust Using Claims
LEVEL 2
Token-Based Authorization
LEVEL 1
Token-Based Authentication
LEVEL 0
API Keys and Basic Authentication
NORDICAPIS.COM

Level 0 - API Keys and Basic Authentication

Level 0 is really just the starting point for most security, and as a result, predictably, it's quite basic in nature – everything in the rest of this model quite literally builds on top of the basic authentication systems and the API keys that interact with them here. Authentication at this level is based upon the notion that whoever has the key must have it because it's their key, and thus, their activity is valid. This "authentication" is then carried forward to other endpoints and APIs in a trusted network, with the key data being carried along that path.

In essence, this type of security is based upon an arguably fundamentally insecure method. All an API key does is confirm that whoever is holding that key can do what that key allows – it does nothing to ensure the person who has the key is meant to have it, is using it in the proper way, or even that they key was legitimately authorized and is still valid. There are also additional concerns in that the user isn't bound to the requested resource – the key could come from almost anywhere as long as it's trusted, and that chain of authentication could, in theory, go anywhere within the network of trust.

There's an even more serious problem with this level of maturity – it only provides authentication. Authentication just says that you are who you say you are (or, at the very least, you have something that says you are who you claim to be). What it does not do, however, is prove that you have the right to make that claim, to access resources that person has rights to access, etc. This is authorization, which is fundamentally different from authentication. In order to have authorization, we need a more complex system.

Level 1 - Token-Based Authentication

Token-Based authentication is a more complex system and represents a different level of security maturity. Tokens are used in this case to establish that whoever holds that token is who they say they are. In the wild, this is often constructed into a sort of quasi-authorization system, as the holding of a token can be seen as an authentication of both who the person is, and what their intent in holding that token is. Tokens can be thought of like an identification card. It may not necessarily say you can do something, but due to the fact that you hold that card, some infer that you are thus trustworthy enough – after all, you have identified yourself through a secure means, so you must be trustworthy!

A good practical way of thinking about this level of maturity is to frame it in terms of a realistic transport workflow. Let's say you're a news publisher. You have an inside organization of writers, editors, etc. who write articles, work on reports, etc. These authors login to their workstations and start pushing their content to an application, which then presents the content forward to the external viewership.

In this case, you have several tokens working in concert. The authors are using their tokens to push content forward and attribute that content to themselves. The readers likewise have their own tokens, which allow them to access the application and leave comments, which are, in turn, attributed to their profile. If they are premium subscribers, they might even have special, different tokens that allow them different access patterns through a quasi-authentication scheme.

This level has its own problems, of course. Authentication Tokens, even though they are often used as a sort of authorization scheme in the wild, are meant only to be used for authentication. Because of that, the quasi-authentication comes from both a supposition of intent (wow, this person has this token, they must be trustworthy enough to do this thing!) and a complex mix of conditional statements and fuzzy logic.

It should also be noted that using authentication tokens as a form of authorization is often highly insecure due to the nature of how tokens get distributed. Machines can get tokens very easily – and if a machine can do it, any malicious actor can do it. When tokens are easy to get, then your authorization scheme depends almost entirely on a system that is spoofable, corruptible, and frankly being used for the exact opposite purpose that was intended.

Level 2 - Token-Based Authorization

Token-Based Authorization is a bit like our previous level, but the focus is shifted to authorization. At this level, we're no longer answering the question of "who are you?" but rather "what can you do?".

One way to think of this is to imagine a great castle with secured gates. In order to enter the castle, you can provide your identity – in other words, you can authenticate that you are who you say you are. While that might get you into the gates, how does anyone know you have a right to sell goods? To sell goods, you might need a seal from the king that says you are allowed to engage in commerce – in other words; you'd need authorization that states you are allowed to do something. Your first token said who you are, and your second token said what you can do.

To take our example of authors and readers to another level, we can look at the ability to consume and the ability to publish. While authentication tokens allowed us to attribute content to a specific user, we'd also need a mechanism to ensure that only authors can publish content. This is where authorization comes in – when authors push their content to the application for consumption, the system needs to be able to ensure that the content came from a trusted source, and has been authored by someone who has the right to upload the content.

This access can be controlled quite granularly using a solution like OAuth – implementing scopes can govern permissions across a token's lifespan and purpose, expiry can be set to ensure that tokens can "age out" of use, etc. In this way, while authentication is very much a "yes or no" proposition (specifically, you either are who you say you are or you're not), authorization can be a much more variable sliding scale of applicability. Authorization isn't just "you either can or you can't," it can be "you can as long as your token is not expired and it is valid for all the sub-steps within this process."

While this might seem like a perfect fix for our security concerns in previous levels, there are a few significant reasons that this is still not enough. First and foremost, we must ask ourselves one question – who do we trust? These systems are designed to be authoritative, and as such, the token systems that come from them must be impervious and trustworthy in order for us to consider their tokens as evidentiary.

Additionally, we must ask ourselves about how data gets handled in transit. These tokens get passed forward, and as they do, they collect more and more data. Accordingly, we must ask what data is being added, and by whom. If we can't know for sure that the data we're handling is, in fact, the same as when it was issued, we lose a significant amount of trust in the data as a core value.

Level 3 - Centralized Trust Using Claims

Claims are a significant missing piece throughout all of our security layers, principally because we are trying to add security at the wrong place. It's one thing for a user to claim to be who they are, or to claim to have certain rights – how do we trust that what they are saying is true? More importantly, how do we trust those who gave the evidence that they are using?

That's the fundamental question here – who do we trust? Do we trust the caller? Do we trust the API Gateway? How about the token issuer? Trust secures us, but it also opens us up to possible attacks. What can we do to fix this?

Claims are the fix because they don't simply tell you something about the subject; they give you context and the ability to verify that information. There are two core types of attributes that a claim can reference – Context Attributes tell us about the situation when a token is issued, and Subject Attributes tell us about the thing that received the token. In order to verify this is true, we trust an Asserting Party. For example, let's say we wanted to get a token, proving that Nordic APIs has published a post. We can look to the attributes:

```
Attribute:
    publisher: Nordic_APIs_Author1
    publish_Date: 12/1/2019
```

In order to instill better security, we can express this information in a claims token as such:

```
Claim:
    Nordic APIs say:
        The publisher is Nordic_APIs_Author1.
```

Using claims, we not only say the information we need to say, but we also specify who is attesting that the information is, in fact, true. In a practical format, the workflow relies quite heavily on signing and verification. When a Requesting Party requests a token from the Issuing Authority, that Authority returns the information requested. This information is signed using a Private Key – when the Requesting Party wants to verify this information, it can simply using the Public Key to ensure that it was signed before it was handed off.

More to the point, encoding and encapsulating data in this way also allows us to add each layer's functionality into a singular source with contextualized information. While the token is granted significant trust due to its signed nature, the meta contextual information (and the attestations from the Issuing Authority) allows us to know who has requested the information, and what they are allowed to see.

Claims also solve the concern of data being added in transit. Because the information encoded is signed and controlled by the

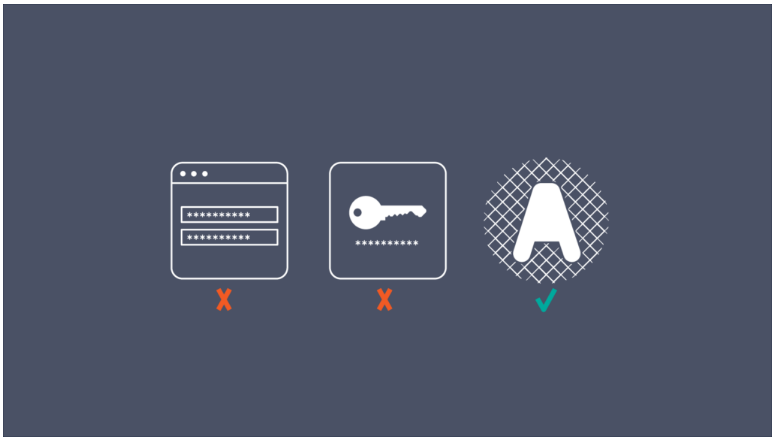
Issuing Authority, nothing is added in transit unless the Issuing Authority is involved – in this way, the source of information can be directly controlled.

Conclusion

Security is not a “one size fits all” equation, but the fundamental requirements of the system are nonetheless quite universal. The need to prove that people are who they say they are, and the need to control access, are fundamental concerns for the modern web and the systems that drive it. Accordingly, choosing the correct approach for your given security flow is paramount to successful communication.

The Difference Between HTTP Auth, API Keys, and OAuth

by **Daniel Lindau**
Originally Published Here



OAuth goes beyond basic authentication measures

When designing systems that enable secure authentication and authorization for API access, you must consider how your applications and users should authenticate themselves. In this article, we'll compare three different ways to achieve this: API Keys, HTTP Basic Auth, and OAuth. We'll also highlight what the benefits and drawbacks are for each method.

API Keys

Using API keys is a way to authenticate an application accessing the API, without referencing an actual user. The app adds the key to each API request, and the API can use the key to identify the application and authorize the request. The key can then be used to perform things like rate limiting, statistics, and similar actions.

How the key is sent differs between APIs. Some APIs use query parameters, some use the Authorize header, some use the body parameters, and so on. For instance, Google Cloud accepts the API key with a query parameter like this:

```
curl -X POST https://language.googleapis.com/v1/documents\
:analyzeEntities?key=API_KEY
```

Cloudflare requires the API key to be sent in a custom header:

```
curl https://api.cloudflare.com/client/v4/zones/cd7d0123e\
301230df9514d \
     -H "Content-Type:application/json" \
     -H "X-Auth-Key:1234567893feefc5f0q5000bfo0c38d90bbeb\
" \
     -H "X-Auth-Email:example@example.com"
```

API Keys Benefits

It’s relatively easy for clients to use API keys. Even though most providers use different methods, adding a key to the API request is quite simple.

API Keys Drawbacks

The API key only identifies the application, not the user of the application.

It's often difficult to keep the key a secret. For server-to-server communication, it's possible to hide the key using TLS and restrict the access to only be used in backend scenarios. However, since many other types of clients will consume the APIs, the keys are likely to leak.

Request URLs can end up in logs. JavaScript applications have more or less everything out in the open. Mobile apps are easy to decompile, and so on. Thus, developers shouldn't rely on API keys for more than identifying the client for statistical purposes. Furthermore, API keys are also not standardized, meaning every API has a unique implementation.

HTTP Basic Auth

HTTP Basic Auth is a simple method that creates a username and password style authentication for HTTP requests. This technique uses a header called Authorization, with a base64 encoded representation of the username and password. Depending on the use case, HTTP Basic Auth can authenticate the user of the application, or the app itself.

A request using basic authentication for the user `daniel` with the password `password` looks like this:

```
GET / HTTP/1.1
Host: example.com
Authorization: Basic ZGFuaWVsOnBhc3N3b3Jk
```

When using basic authentication for an API, this header is usually sent in every request. The credentials become more or less an API key when used as authentication for the application. Even if it represents a username and password, it's still just a static string.

In theory, the password could be changed once in a while, but that's usually not the case. As with the API keys, these credentials could leak to third parties. Granted, since credentials are sent in a header, they are less likely to end up in a log somewhere than using a query or path parameter, as the API key might do.

Using basic authentication for authenticating users is usually not recommended since sending the user credentials for every request would be considered bad practice. If HTTP Basic Auth is only used for a single request, it still requires the application to collect user credentials. The user has no means of knowing what the app will use them for, and the only way to revoke the access is to change the password.

HTTP Basic Auth Benefits

HTTP Basic Auth is a standardized way to send credentials. The header always looks the same, and the components are easy to implement. It's easy to use and might be a decent authentication for applications in server-to-server environments.

HTTP Basic Auth Drawbacks

When a user is authenticated, the application is required to collect the password. From the user perspective, it's not possible to know what the app does with the password. The application will gain full access to the account, and there's no other way for the user to revoke the access than to change the password. Passwords are long-lived tokens, and if an attacker would get a hold of a password, it will likely go unnoticed. When used to authenticate the user, multi-factor authentication is not possible.

Token-based Authentication Using OAuth 2.0

A token-based architecture relies on the fact that all services receive a token as proof that the application is allowed to call the service. The token is issued by a third party that can be trusted by both the application and service. Currently, the most popular protocol for obtaining these tokens is OAuth 2.0, specified in RFC 6749.

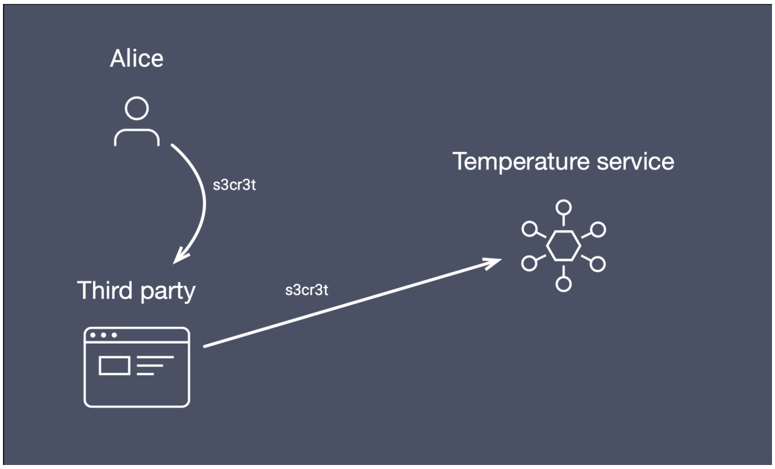
OAuth specifies mechanisms where an application can ask a user for access to services on behalf of the user, and receive a token as proof that the user agreed. To demonstrate how OAuth works, let's consider the following use case.

A user Alice has an account with a service where she can report the current indoor temperature of her home. Alice also wants to give a third-party application access to read the temperature data, to be able to plot the temperatures on a graph, and cross-reference with data from other services.

The temperature service exposes an API with the temperature data, so the third party app should be able to access the data quite easily. But how do we make only Alice's data available to the application?

Collecting the Credentials

Using Basic authentication, the application can collect Alice's username and password for the temperature service and use those to request the service's data. The temperature service can then verify the username and password, and return the requested data.



However, as we noted about, there are a few problems with this approach:

- The user has to trust the application with the credentials. The user has no means of knowing what the credentials are used for.
- The only way for the user to revoke the access is to change the password.
- The application is not authenticated
- The scope of access can not be controlled. The user has given away full access to the account.
- Two-factor authentication cannot be used

Historically, this has created a need for services to develop “application-specific passwords,” i.e., additional passwords for your account to be used by applications. This removes the need to give away the actual password, but it usually means giving away full access to the account. On the service provider side, you could build logic around combining application-specific passwords with API keys, which could limit access as well, but they would be entirely custom implementations.

The OAuth Way

Let's look at how we could solve this problem using an OAuth 2.0 strategy. To enable better authentication, the temperature service must publish an Authorization Server (AS) in charge of issuing the tokens. This AS allows third party applications to register, and receive credentials for their application to be able to request access on behalf of users.
To request access, the application can then point the user's browser to the AS with parameters like:

```
https://as.temperatures.com/authorize?client_id=third_par\
ty_graphs&scope=read_temperatures&…
```

This request will take the user to the AS of the temperature service, where the AS can authenticate Alice with whatever method is available. Since this happens in the browser, multiple-factors are possible, and the only one seeing the data is the temperature service and the owner of the account.
Once Alice has authenticated, the AS can ask if it's ok to allow access for the third party. In this case, the read_temperature scope was asked for, so the AS can prompt a specific question.

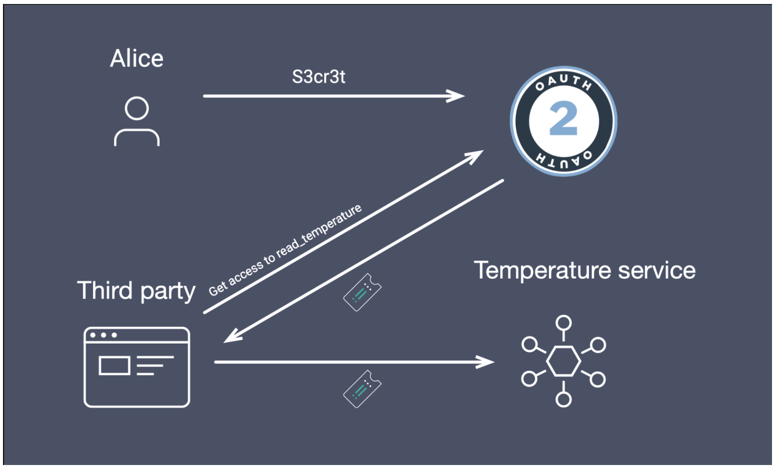
```
<bild consent>
```

When Alice accepts, the client can authenticate itself. A token is issued as proof that Alice accepted the delegated access, and it is sent back to the third party application.

Now, the third party application can call the API using the received token. The token is sent along with the request by adding it to the Authorization header with the Bearer keyword as follows:

```
GET /temperature HTTP/1.1
Host api.temperatures.com
Authorization: Bearer <token>
```

Upon receiving the request, the service can validate the token, and see that Alice allowed the application to read the temperature listings from her account, and return the data to the application.



Token Validation

The issued token can be returned in two ways, either by returning a reference to the token data or returning the value of the token directly. For the reference token, the service will have to send a request to the AS to validate the token and return the data associated with it. This process is called introspection, and a sample response looks like this:

```
{
"active": true,
"sub": “alice”,
"client_id": "third_party_graphs",
"scope": "read_temperatures"
…
}
```

In this response, we can see that the user `alice` has granted the application `third_party_graphs` access to her account, with the scope of `read_temperatures`.

Based on this information, the service can decide if it should allow or deny the request. The `client_id` can also be used for statistics and rate-limiting of the application. Note that we only got the username of the account in the example, but since the AS does the authentication, it can also return additional claims in this response (things like account type, address, shoe-size, etc.) Claims can be anything that can allow the service to make a well informed authorization decision.

For returning the value, a token format like JSON Web Token (JWT) is usually used. This token can be signed or encrypted so that the service can verify the token by simply using the public key of the trusted AS. Tip: Read this resource for recommendations on token types.

We can see a clear difference here:

- Alice only gave her credentials to the trusted site.
- Multi-factor authentication can be used.
- Alice can revoke access for the app, by asking the temperature site to withdraw her consent, without changing her password
- Alice can allow the third-party app to access only certain information from her account.

- Claims about the user can be delivered to the service directly through the request. No additional lookups required.
- The flow is entirely standardized.

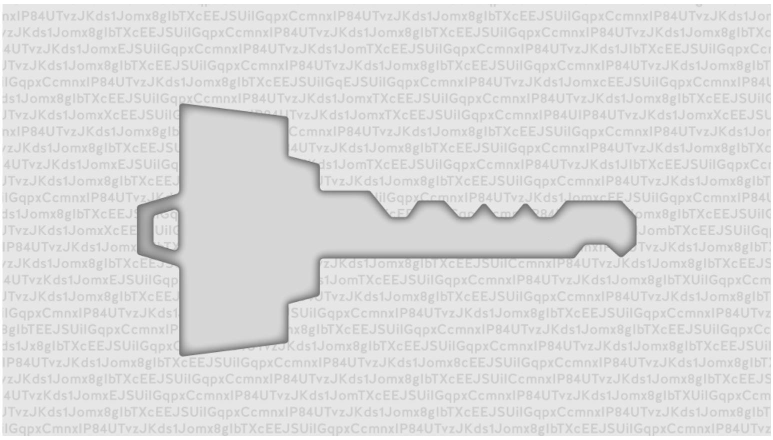
Summary

Since OAuth 2.0 was developed in the time of a growing API market, most of the use cases for API keys and Basic Authentication have already been considered within the protocol. It's safe to say that it beats the competition on all accounts. For small, specific use cases, it might be ok to use API keys or Basic Authentication, but anyone building systems that plan to grow should be looking into a token-based architecture such as the Neo Security Architecture.

In the use case above, I only described the user flow, but OAuth, of course, specifies alternate flows for obtaining tokens in server-to-server environments. You can read more on those in my earlier post that explores eight types of OAuth flows and powers.

API Keys ≠ Security: Why API Keys Are Not Enough

by **Kristopher Sandoval**
Originally Published Here



Who holds the key? APIs need robust identity control and access management.

We're all accustomed to using usernames and passwords for hundreds of online accounts — but if not managed correctly, using passwords can become a significant distraction and a potential security vulnerability. The same is true in the API space. There's nothing inherently wrong with usernames — you need those. But if you use them without also having credentials that allow the service to verify the caller's identity, you are certainly doing it wrong.

Unfortunately, many API providers make a dangerous mistake that exposes a large amount of data and makes an entire ecosystem insecure. In plain English — **if you're only using API keys, you may be doing it wrong!**

What is an API Key?

An **API Key** is a code assigned to a specific program, developer or user that is used whenever that entity calls an API. This Key is typically a long string of generated characters which follow a set of generation rules specified by the authority that creates them:

```
IP84UTvzJKds1Jomx8gIbTXcEEJSUilGqpxCcmnx
```

Upon account creation or app registration, many API providers assign API keys to their developers, allowing them to function in a way similar to an account username and password. API keys are unique. Because of this, many providers have opted to use these keys as a type of security layer, barring entry and further rights to anyone unable to provide the key for the service being requested.

Despite the alluring simplicity and ease of utilizing API Keys in this method, the shifting of security responsibility, lack of granular control, and misunderstanding of the purpose and use amongst most developers make solely relying on API Keys a poor decision. More than just protecting API keys, we need to program robust identity control and access management features to safeguard the entire API platform.

Shifting of Responsibility

In most common implementations of the API Key process, the security of the system as a whole depends entirely on the ability of the developer consumer to protect their API keys and maintain security. However, this isn't always stable. Take Andrew Hoffman's $2375 Amazon EC2 Mistake that involved a fluke API key push to GitHub. As developers rely on cloud-based development tools, the accidental or malicious public exposure of API keys can be a real concern.

From the moment a key is generated, it is passed through the network to the user over a connection with limited encryption and security options. Once the user receives the key, which in many common implementations is provided in plain text, the user must then save the key using a password manager, write it down, or save it to a file on the desktop. Another common method for API Key storage is device storage, which takes the generated key and saves it to the device on which it was requested.

When a key is used, the API provider must rely on the developer to encrypt their traffic, secure their network, and uphold the security bargain's side. There are many vulnerabilities at stake here: applications that contain keys can be decompiled to extract keys, or deobfuscated from on-device storage, plaintext files can be stolen for unapproved use, and password managers are susceptible to security risks as with any application.

Due to its relative simplicity, most common implementations of the API Key method provide a false sense of security. Developers embed the keys in Github pushes, utilize them in third-party API calls, or even share them between various services, each with their own security caveats. In such a vulnerable situation, security is a huge issue, but it's one that isn't really brought up with API Keys because *"they're so simple — and the user will keep them secure!"*

This is a reckless viewpoint. API Keys are only secure when used with SSL, which isn't even a requirement in the basic implementation of the methodology. Other systems, such as OAuth

2, Amazon Auth, and more, require the use of SSL for this very reason. Shifting the responsibility from the service provider to the developer consumer is also a negligent decision from a UX perspective.

Lack of Granular Control

Some people forgive the lack of security. After all, it's on the developer to make sure solutions like SSL are implemented. However, even if you assure security, your issues don't stop there — **API Keys by design lacks granular control**.

Somewhat ironically, before API keys were used with RESTful services, we had WS-Security tokens for SOAP services that let us perform many things with more fine-grained control. While other solutions can be scoped, audienced, controlled, and managed down to the smallest of minutia, API Keys, more often than not, only provide access until revoked. They can't be dynamically controlled.

That's not to say API Keys lack *any* control — relatively useful read/write/readwrite control is definitely possible in an API Key application. However, the needs of the average API developer often warrant more full-fledged options.

This is not a localized issue either. As more and more devices are integrated into the Internet of Things, this control will become more important than ever before, magnifying the choices made in the early stages of development to gargantuan proportions later on in the API Lifecycle.

Square Peg in a Round Hole

All of this comes down to a single fact: **API Keys were never meant to be used as a security feature**. Most developers utilize API Keys as a method of authentication or authorization, but the API Key was only ever meant to serve as identification.

API Keys are best for two things: **identification** and **analytics**. While analytic tracking can make or break a system, other solutions implement this feature in a more feature-rich way. Likewise, while API Keys do a great job identifying a user, other alternatives, such as public key encryption, HoK Tokens, etc. do a much better job of it while providing more security.

The Pros of API Keys

There are definitely some valid reasons for using API Keys. First and foremost, API Keys are **simple**. The use of a single identifier is simple, and for some use cases, the best solution. For instance, if an API is limited specifically in functionality where "read" is the only possible command, an API Key can be an adequate solution. Without the need to edit, modify, or delete, security is a lower concern.

Secondly, API Keys can help reduce the entropy-related issues within an authenticated service. **Entropy** — the amount of energy or potential within a system constantly expended during its use — dictates that there are a limited amount of authentication pairs. Suppose entropy dictates that you can only have 6.5 million unique pairs when limited within a certain character set and style. In that case, you can only have 6.5 million devices, users, or accounts before you run into an issue with naming. Conversely, establishing an API Key with a high number of acceptable variables largely solves this, increasing theoretical entropy to a much higher level.

Finally, **autonomy** within an API Key system is extremely high. Because an API Key is independent of a naming server and master credentials, it can be autonomously created. While this comes with the caveat of possible Denial of Service attacks, the autonomy created is wonderful for systems that are designed to harness it.

When developing an API, a principle of least privilege should be adhered to — **allow only those who require resources to access those specific resources**. This principle hinges on the concept of CIA in system security — Confidentiality, Integrity, and Availability. If your API does not deal with confidential information (for instance, an API that serves stock exchange tickers), does not serve private or mission-critical information (such as a news/RSS API), or does not demand constant availability (in other words, can function intermittently), then API Keys may be sufficient.

Additionally, API Keys are a good choice for developer-specific API uses. When developers are configuring API clients at operation time, and use changing keys for different services, this is acceptable.

Back to Reality

The benefits of using API Keys outlined above are still tenuous in the general use-case scenario. While API keys are *simple*, the limitation of "read-only" is hampering rather than liberating. Even though they provide higher levels of *entropy*, this solution is not limited to API Keys and is inherent in other authentication/authorization solutions. Likewise, *autonomy* can be put in place through innovative server management and modern delegation systems.

Conclusion: API Keys Are Not a Complete Solution

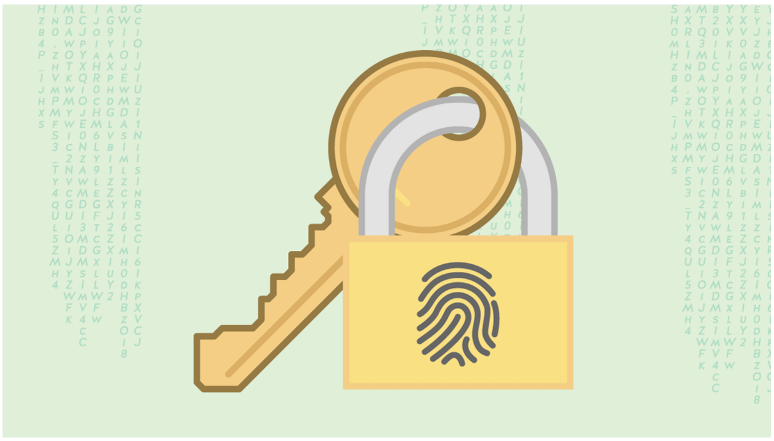
The huge problems with API Keys come when end users, not developers, start making API calls with these Keys, which more often than not expose your API to security and management risks. It comes down to that **API Keys are, by nature, not a complete solution**. While they may be perfectly fine for read-only purposes, they are too weak a solution to match the complexity of a high-use API system. Whenever you start integrating other functionality such as writing, modification, deletion, and more, you necessarily enter the realm of Identification, Authentication, and Authorization.

Basic API Key implementation doesn't support authentication without additional code or services. It doesn't support authentication without a matching third-party system or secondary application. It doesn't support authorization without some serious "hacks" to extend use beyond what they were originally intended for.

While an argument could be made for expanding out the API Keys method to better support these solutions, that argument would advocate re-inventing the wheel. There are already so many improved solutions available that adding functionality to an API Key system doesn't make sense. Even if you did add something like authentication, especially federated authentication, to the system using Shibboleth, OpenID, etc., there are a ton of systems out there that already have support for this.

Why Can't I Just Send JWTs Without OAuth?

by **Kristopher Sandoval**
Originally Published Here



JWTs are only part of the greater API security puzzle

A JSON Web Token or JWT is an extremely powerful standard. It's a signed JSON object; a compact token format often exchanged in HTTP headers to encrypt web communications.

Because of its power, JWTs can be found driving some of the largest modern API implementations. For many, the JWT represents a great solution that balances weight with efficiency, and as such, it's often a very attractive standard to adopt for API security.

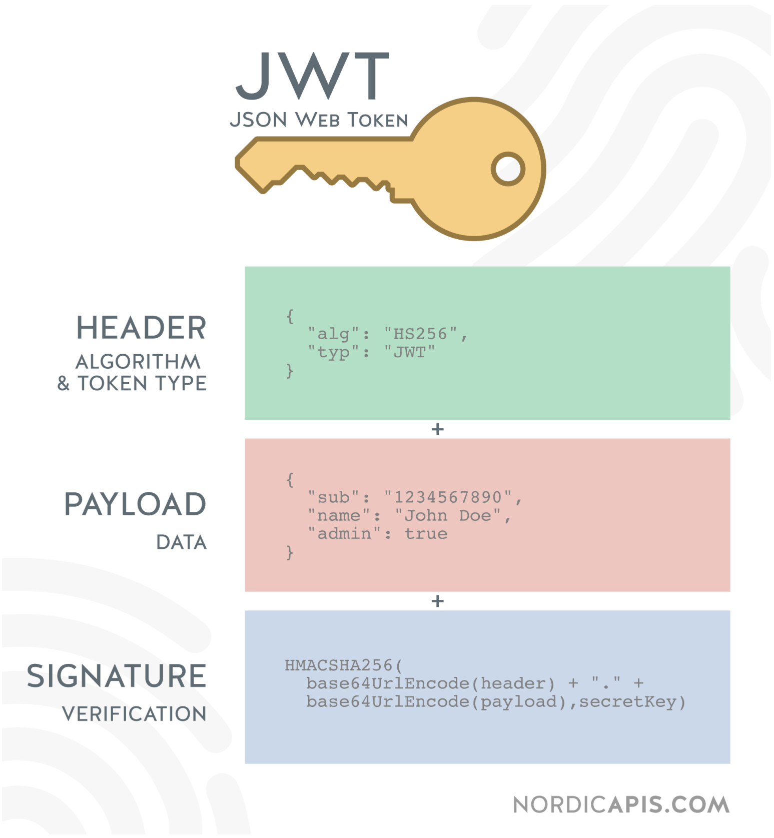
However, a JWT should not be viewed as a complete solution. Unfortunately, it seems that there are some significant misunderstandings as to what a JWT is, and how exactly it functions. In many situations, depending on JWTs alone can be extremely dangerous.

One of the most common questions about using JWTs is: **Why can’t I send JWTs without OAuth?** In this chapter, we answer that very question. We’ll define what a JWT actually is, how it functions, and why adopting it in isolation is dangerous.

What is a JWT?

Before we address why utilizing JWTs alone is insecure, we must define what a JWT actually is. JWTs are often conflated with the additional protocols and systems surrounding them, meaning that the JWT design concept has been bolstered beyond the actual definition of the object itself.

JWT is an open standard defined by RFC 7519. The JWT is considered by its authors to be a “compact and self-contained way for securely transmitting information between parties as a JSON object.” The JWT itself is composed of a Header, a Payload, and a signature that proves the integrity of the message to the receiving server.



```
{
  "alg": "HS256",
  "typ": "JWT"
}
```

```
{
  "sub": "1234567890",
  "name": "John Doe",
  "admin": true
}
```

```
HMACSHA256(
  base64UrlEncode(header) + "." +
  base64UrlEncode(payload),secretKey)
```

Content encoded inside a JWT is digitally signed, either using a secret utilizing the HMAC algorithm or leveraging the Public Key Infrastructure (PKI) model with a private/public RSA configuration. While this does lend a certain amount of integrity protection, it does not specifically guarantee security — we will discuss this at greater length in just a moment, but it should be understood that a JWT is an encoding format and only an encoding format.

JWTs are loved because they are small, and lend themselves to efficient transport as part of a URL, as part of the POST parameter, or even within the HTTP header. More lightweight transport options exist, and further extensions of the concept exist; CWT is a great example, utilizing CBOR, or Concise Binary Object Representation, to even further reduce the size of the package and improve efficiency.

The main benefit (and perhaps the main drawback from a security standpoint) of the JWT standard is that the encoded package is self-contained. The JWT package contains everything the system would need to know about the user, and as such, can be delivered as a singular object.

The Dangers of a Lone Solution

JWTs are powerful — there’s simply no denying that. Unfortunately, many developers seem to think that the JWT is more than an encoding methodology, but a complete and secure implementation. This is often because JWTs are typically paired with a proper protocol and encryption standard in the wild — but this is a conscious choice, not the result of an automatic security due to the structure of the JWT itself.

A JWT is only secure when it’s used in tandem with encryption and transport security methodologies. JWT is a great encoding methodology, but it’s not a holistic security measure. Without additional protocols backing it up, a JWT is nothing more than an admittedly lightweight and slightly more secure API key.

For an example of this insecurity, let’s look at a common use case. A web API serves as the backend to a web application, and when the user generates a JWT, it is stored as an HTML5 data storage element. This is done so as to aid in the utilization of the API over multiple gateways and functions.

In this common situation, the issue is that the JWT is essentially exposed for common use. The JWT is digitally signed, which assures a certain amount of guaranteed integrity. The server itself is also set to reject any JWT with a manipulated Header, Payload, or Signature component, and as such, can reject a modified JWT token. That being said, the token doesn't need to be modified in order to breach security. In theory, an attacker could take that token and use it in a sort of replay attack, getting resources that they do not have the authorization to have.

While this type of attack can be somewhat mitigated through the use of expiration dates, this does nothing for man-in-the-middle attacks. In the MITM attack scheme, the expiration does not matter, as the attack is initiated live as a middleman.

These issues all arise from the simple fact that JWTs are a mechanism for transferring data — not for securing it.

Securing JWTs

JWTs are self-contained solutions containing everything the server needs to know about who the user is, what they need, and what they're authorized to do. Accordingly, they're great for stateless authentication and work well with such methods geared for stateless environments.

While there are a number of third party solutions and implementations of stateless authentication, the fact is that what you'd essentially be creating is a bearer token or, alternately, an access token.

That's ok, and in fact, what we want to do, but this raises a simple question — if we are indeed creating such a bearer token, why not use the built-in functionality of the OAuth schema designed specifically to work with JWTs? There's already a great deal of built-in security functionality in the OAuth specification that's specifically engineered to support the JWT, so using external solutions — often the second question after why can't I just sent JWTs without OAuth — is somewhat nonsensical.

If we utilize the OAuth 2.0 Bearer Token Usage standard under RFC 6750, which incorporates authorization headers, we can essentially create JWTs that would be recognized and specially treated by a wide variety of devices, from HTTP proxies to servers. We would thereby reduce data leakage, unintended storage of requests (as displayed above), and enable transport over something as simple as HTTPS.

Proper JWT Utilization

While it's important to secure your JWTs, it's also important to state what the proper utilization of a JWT within the OAuth schema would look like. While a JWT can serve many functions, let's take a look at a common use case in the form of the access token. Both OAuth 2.0 and OpenID Connect are vague on the type of access_token, allowing for a wide range of functions and formats. That being said, the utilization of a JWT as that token is quite ubiquitous, for the benefits in efficiency and size already noted.

An access token is, in simple terms, is a token that is used by the API to make requests on behalf of the user who requested the token. It is part of the fundamental authorization mechanism within OAuth, and as such, confidentiality and integrity are extremely important. In order to generate an access token, an authorization code is required. All of the elements of this code are also extremely important to keep confidential and secure.

Accordingly, a JWT fits this role almost perfectly. Because of the aforementioned standards that allow for transmission over HTTPS, the JWT can contain all of the information needed to generate the access token. Once the token is generated, it can likewise be kept in JWT form as what is called a self-encoded access token.

The key benefit of handling the encoding of the access token in this way in the OAuth 2.0 schema is that applications don’t have to understand your access token schema — all of the information is encoded within the token itself, meaning that the schema can change fundamentally without requiring the clients to be aware, or even affected, by such changes.

Additionally, the JWT is great for this application because of the wide range of libraries that offer functionality such as expiration. A good example of this would be the Firebase PHP-JWT library, which offers such expiration functionality.

Caveats

Of course, as with any security implementation, there are caveats to consider. In the case of the JWT as a self-encoded authorization solution, replay attacks should be considered. While adopting proper encryption methodologies should negate many of those issues, the fact is that the issue is still fundamental to the concept as a whole, and should probably be addressed as a possibility rather than an impossible threat.

Accordingly, caching the authorization code for the lifetime of the code is the suggested solution from OAuth itself. By doing this, code can be verified against the known cached code for validity and integrity, and once the expiration date is reached, automatically rejected for date reasons.

It should also be noted that, due to the nature of the JWT, once an authorization code is issued, the JWT is self-contained — as such, it cannot technically be invalidated, at least in its most basic configuration. The JWT is designed to not hit the database for every verification, and when using a global secret, the JWT is valid until expiration.

There are a few ways around this, such as adding a counter in the

JWT that increments upon certain events (such as role change, user data change, etc.). This, of course, results in database polling for each request, but the amount of data being checked is minuscule enough to make any processing increase somewhat negligible.

Additionally, at least in theory, you could use sections for specific functions, domains, and scopes, and change that secret when a breach is discovered. While this would affect more users than admins would like, it does have the effect of instituting revocation.

That being said, proper utilization of the JWT should make this largely a non-issue, as the user still has to provide a certain amount of secret information over an encrypted channel, and as such, should already be "vetted" or controlled.

Don't Leave JWT All Alone

The simple fact is that JWTs are a great solution, especially when used in tandem with something like OAuth. Those benefits quickly disappear when used alone, and in many cases, can result in worse overall security.

That being said, adopting the proper solutions can mitigate many of these threats, resulting in a more secure, efficient system. The first step to securing your JWT is to understand what it's not — a JWT is an encoding method, not an encryption or transport security method, and as such, is only part of the puzzle.

How To Control User Identity Within Microservices

by **Bill Doerrfeld**
Originally Published Here

OAuth is necessary to delegate identity throughout a platform

Many developers are well along in their microservices journeys.

Yet, as the number of services increases, so do operational issues. One especially tricky feat is maintaining identity and access management throughout a sea of independent services.

Unlike a traditional monolithic structure with a single security portal, microservices pose many problems. Should each service have its own independent security firewall? How should identity be distributed between microservices and throughout my entire system? What is the most efficient method for the exchange of user data?

Smart techniques leverage standardized technologies to not only authorize but perform Delegation across your entire system. This chapter will identify how to implement OAuth and OpenID Connect flows using JSON Web Tokens. We'll explain how to create a distributed authentication mechanism for microservices — a process of managing identity where everything is self-contained, standardized, secure, and, best of all — easy to replicate.

What Are Microservices, Again?

Microservices architecture

The microservice design pattern is a way to architect web service suites into independent specialized components. These components are made to satisfy a very targeted function and are fully independent, deployed as separate environments. The ability to recompile individual units means that development and scaling are vastly easier within a microservices system.

Microservices architecture is opposed to the traditional monolithic approach that consolidates all web components into a single system. The downside of a monolithic design is that version control cycles are arduous, and scalability is slow. The entire system must be deployed as one unit since it's packaged together.

Monolithic design

The move toward microservices has had dramatic repercussions across the tech industry, allowing SaaS organizations to deploy many small services no longer dependent on extensive system overhauls. Microservices arguably ease development, and on the user-facing side, allow accessible pick-and-choose portals to personalize services to individual needs.

Great, So What's The Problem?

We're faced with the problem that microservices don't lend themselves to the traditional mode of identity control. In a monolithic system, security works simply as follows:

1. Figure out who the caller is
2. Pass on credentials to other components when called
3. Store user information in a data repository

Since components are conjoined within this structure, they may share a single security firewall. They also share the user's state as they receive it and may share access to the same user data repository.

If the same technique were to be applied to individual microservices, it would be grossly inefficient. Having an independent security barrier — or request handler — for each service to authenticate identity is unnecessary. This would involve calling an Authentication Service to populate the object to handle the request and respond in every single instance.

The Solution: OAuth As A Delegation Protocol

There is a method that allows one to combine isolated deployment benefits with the ease of federated identity. Jacob Ideskog of Curity believes that to accomplish this, OAuth should be interpreted not as Authentication, and not as Authorization, but as *Delegation*.

In the real world, Delegation is where you delegate someone to do something for you. In the web realm, the underlying message is there, yet it also means having the ability to offer, accept, or deny data exchange. Treating OAuth as a Delegation protocol can assist in the creation of scalable microservices or APIs.

To understand this process, we'll first layout a standard OAuth flow for a simple use case. Assume we need to access a user's email account for a simple app that organizes a user's email — perhaps to send SMS messages as notifications. OAuth has the following four main actors:

- **Resource Owner** (RO): the user
- **Client**: the web or mobile app
- **Authorization Service** (AS): OAuth 2.0 server
- **Resource Server** (RS): where the actual service is stored

A Simplified Example of an OAuth 2.0 Flow

In our situation, the app (the Client), needs to access the email account (the Resource Server) to collect emails before organizing them to create the notification system. In a simplified OAuth flow, an approval process would be as follows:

1. The Client requests access to the Resource Server by calling the Authorization Server.
2. The Authorization Server redirects to allow the user to authenticate, which is usually performed within a browser. This is essentially signing into an authorization server, not the app.
3. The Authorization Server then validates the user credentials and provides an Access Token to Client, which can be used to call the Resource Server.
4. The Client then sends the Token to the Resource Server.
5. The Resource Server asks the Authorization Server if the Token is valid.
6. The Authorization Server validates the Token, returning relevant information to the Resource Server (i.e., time till token expiration, who the Token belongs too.)
7. The Resource Server then provides data to the Client. In our case, the requested emails are unbarred and delivered to the Client.

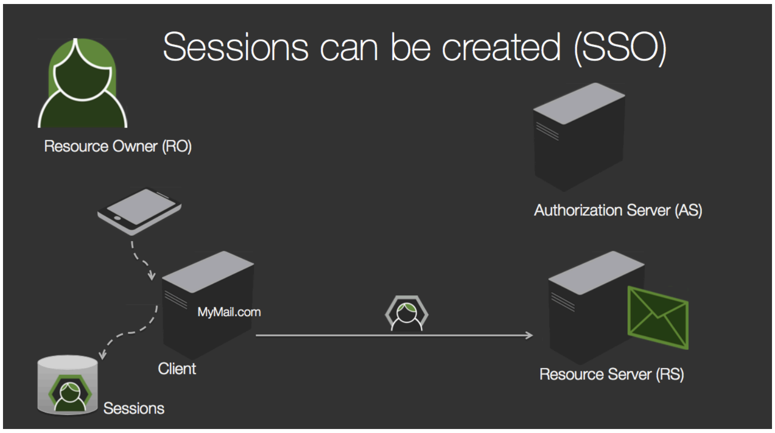
A vital factor to note within this flow is that the Client — our email notification app — knows nothing about the user at this stage. The Token that was sent to the Client was completely opaque — only a string of random characters. Though this is a secure exchange, the token data is itself useless to the Client. The exchange thus supplies access for the Client, but no user information. What if our app needed to customize the User Experience (UX) based on the user type? For example: which membership level the user belonged to, a group they were a member of, where they were located, or their preferred language. Many apps provide this type of experience, but to enable this customization, we will require additional user information.

The OpenID Connect Flow

Let's assume that we're enhancing the email service client so that it not only organizes your emails but also stores them and translates them into another language. In this case, the Client will want to retrieve additional user data and store it in its own user sessions.

To give the Client something other than the opaque Token provided in the OAuth flow, use an alternative flow defined in OpenID Connect. In this process, the Authorization Server, which is also called an OpenID Connect Provider (OP), returns an ID Token along with the Access Token to the Client. The flow is as follows:

1. The Client requests access to the Resource Server by calling the Open ID Connect enabled Authorization Server.
2. The Authorization Server redirects to allow the user to authenticate.
3. The Authorization Server then validates the user credentials and provides an Access Token AND an ID Token to the Client.
4. The Client uses this ID Token to enhance the UX and typically stores the user data in its own session.
5. The Client then sends the Access Token to the Resource Server.
6. The Resource Server responds, delivering the data (the emails) to the Client.



The ID token can contain information about the user, such as authentication details, name, email, or any number of custom user data points. This ID token takes the form of a JSON Web Token (JWT), a coded and signed compilation of JSON documents. The document includes a header, body, and a signature appended to the message. Data + Signature = JWT.

Using a JWT, you can access the public part of a certificate, validate the signature, and understand that this authentication session was issued — verifying that the user has been authenticated. An important facet of this approach is that ID tokens establish trust between the Authorization Server/Open ID Connect Provider and the Client.

Using JWT For OAuth Access Tokens

Even if we don't use OpenID Connect, JWTs can be used for many things. A system can standardize by using JWTs to pass user data among individual services. Let's review the types of OAuth access tokens to see how to implement secure identity control within microservice architecture smartly.

By Reference: Standard Access Token

A Standard Access Token contains no information outside of the network, merely pointing to a space where information is located. This opaque string means nothing to the user, and as it is randomized cannot easily be decrypted. Standard Access Tokens are the standard format — without extraneous content, simply used for a client to gain access to data.

By Value: JSON Web Token

A JSON Web Token (JWT) may contain necessary user information that the Client requires. The data is compiled and inserted into the message as an access token. JWTs are an efficient method because they erase the need to call again for additional information. If exposed over the web, a downside is that this public user information can be read easily read, exposing the data to an unnecessary risk of decryption attempts to crack codes.

The Workaround: External vs. Internal

To limit this risk of exposure, Ideskog recommends splitting the way the tokens are used. What is usually done is as follows:

1. The Reference Token is issued by the Authorization Server. The Client sends back when it's time to call the API.
2. In the middle: The Authorization Server validates the token and responds with a JWT.
3. The JWT is then passed further along in the network.

In the middle, we essentially create a firewall, an Authorization Server that acts as a token translation point for the API.

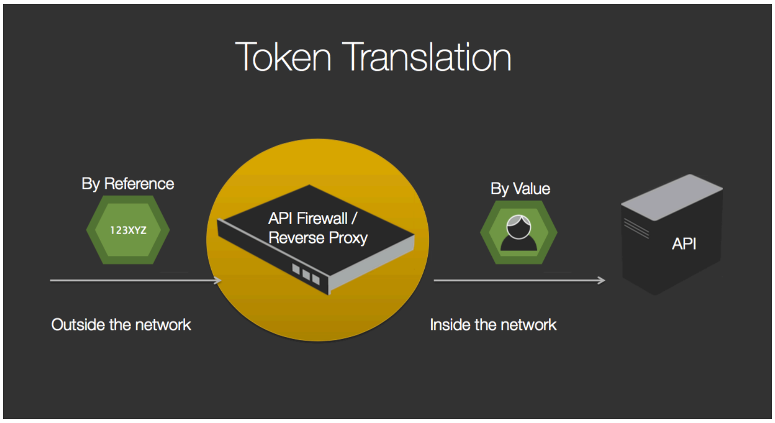
The Authorization Server will translate the Token, either for a simple Reverse Proxy or a full-scale API Firewall. However, the Authorization Server shouldn't be in the "traffic path" — the reverse proxy finds the token and calls the Authorization server to translate it.

Let All Microservices Consume JWT

So, to refresh, with microservice security we have two problems:

- **We need to identify the user multiple times**: We've shown how to leave Authentication to OAuth and the OpenID Connect server so that microservices successfully provide access given someone has the right to use the data.
- **We have to create and store user sessions: JWTs contain the necessary information to help in storing user sessions. If each service can understand a JSON web token, you have distributed your identity mechanism, allowing you to transport identity throughout your system.

In a microservice architecture, an Access Token should not be treated as a request object, but rather as an *identity* object. As the process outlined above requires translation, JWTs should be translated by a front-facing stateless proxy, used to take a reference token, and convert it into a value token to then be distributed throughout the network.



Why Do This?

By using OAuth with OpenID Connect, and by creating a standards-based architecture that universally accepts JWTs, the end result is a distributed identity mechanism that is self-contained and easy to replicate. Constructing a library that understands JWT is a very simple task. In this environment, access, as well as user data, is secured. Creating microservices that communicate well and securely access user information, can greatly increase the agility of an entire system, as well as increase the quality of the end-user experience.

Part Two: OAuth Flows and Deep Dives

8 Types of OAuth Flows And Powers

by **Daniel Lindau**
Originally Published Here

There are multiple OAuth flows catered to various scenarios

The API space requires authorization in order to secure data – this is a given in the modern era. Accordingly, implementing the correct authorization system is vitally important, perhaps even more important than the API it is meant to handle authorization for.

OAuth is a powerful solution for many providers. However, as with any tool, t’s only as powerful as it is understood by the user who chooses it. Understanding what OAuth is and having a general overview of each particular flow is extremely important. In this piece, we’re going to look at OAuth and give a brief rundown of each flow type. We'll look at when each flow is appropriate and what its specific use case is.

What Is OAuth? What Is a Flow?

While it might go without saying, there is some benefit to stating upfront exactly what OAuth is. OAuth is an open standard for delegation and authorization on the internet. The use case for OAuth is usually a client that needs to access some resource on behalf of the user. To accomplish this delegation, an Access Token is issued. The Access Token represents the user's consent, allowing the client to access its data on behalf of the user. The requesting, granting, and life management of this token is often referred to as a **flow**, a term that will be used substantially throughout this article.

While the first version of OAuth was initially created in 2007 as a means to handle authentication on the Twitter API, it has since become extremely popular in a variety of applications with scopes ranging from enterprise-level codebases to home projects. The second version, OAuth 2.0, has become the de facto standard for securely protecting your APIs.

Flows Differ On Use Case

The OAuth specification allows for several ways of obtaining and validating tokens, and not all flows are meant for all types of clients. The OAuth specification talks about public and private clients, which roughly translates into the clients' ability to keep their credentials safely stored. Private clients are typically applications with a backend that can keep a secret to use for authenticating. Public clients have no means of securely keeping a secret, such as a Single Page Application (SPA) that usually doesn't have a backend.

For instance, web applications with a backend are considered private clients, and SPAs are considered public. The backend can securely keep the secret, while the SPA has everything out in the open.

Mobile clients are a bit trickier to classify since they are generally pretty good at keeping a secret, but it's hard to give them one. The way the apps are distributed through app stores makes it harder for clients to authenticate in a way for the OAuth server to trust that it is the correct application. For this reason, they are to be considered public. By using other means of getting credentials, like the Dynamic Client Registration, it can be made into a private client. But more on that later.

Obtaining Tokens

There are four base flows for obtaining tokens in OAuth, and several flows that are defined in sibling specifications. Here, I'll describe the base flows and others that I believe to be important.

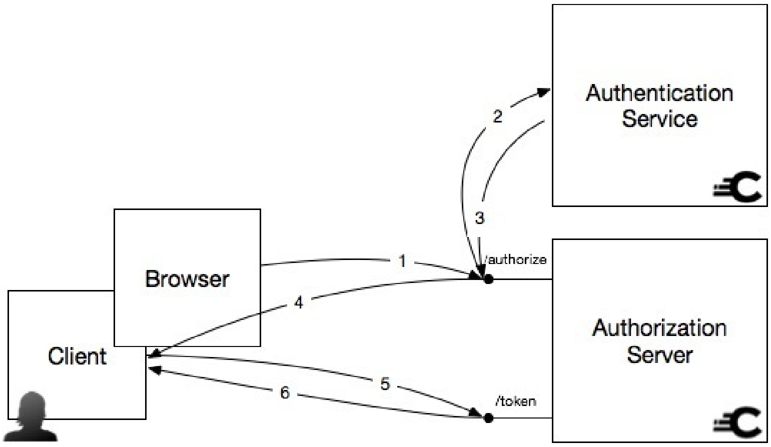
1. Authorization Code Grant

The Authorization Code Grant, or Code Flow, is the most widely used OAuth flow. To obtain a token using code flow, the clients send an authorization request to the OAuth server by simply redirecting the browser to the server. The OAuth server makes sure that the user is authenticated and prompts the user to approve the delegation. When the user approves, a short-lived code is issued to the client. This code can be considered a one time password or a nonce. The client receives this code and can now use it in an authenticated backend call – outside of the browser – and exchange it for the token.

One thing to mention here is that the user only will enter its credentials to the OAuth server. The user won't have to give the credentials to the app; it simply enters them to the server it already knows and trusts. This is one thing that OAuth set out to solve.

The other benefit is that the token owner passes the browser, which makes it harder to steal, and since the call to exchange the token is authenticated, the server can be sure that it delivers the token to the correct client.

Usually, the Code Flow will also allow you to receive a Refresh Token, which allows the client to get new access tokens without involving the user, even after the Access Token is expired. Private clients should only use the Code Flow since the client must authenticate itself when exchanging the code.



Code Flow: The Client consist of two parts, the browser, and the backend

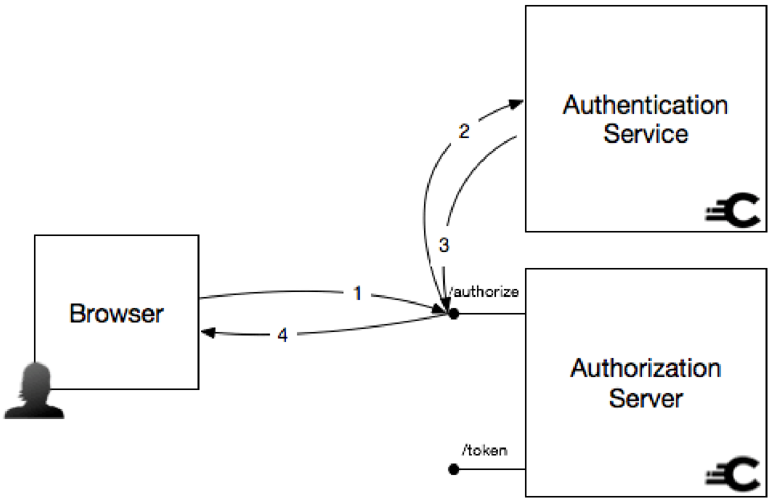
2. Implicit Flow

The Implicit Flow is a less complicated flow than the Code Flow. It starts out in the same way as the Code Flow, with the client making an authorization request to the OAuth server. The user authenticates and approves of the delegation, but instead of issuing a code, the OAuth server responds with an Access Token.

The downside here is that the token is visible in its entirety, and since it is in the browser, the client may handle the token in a way that could make it vulnerable.

The Implicit Flow is designed for public clients that cannot authenticate themselves. So, the trust here instead lies in a parameter called `redirect_uri`. The OAuth server needs to have registered a URL for the client, where the response will be sent. The response will only be sent there, so if a malicious application fools a user into initiating a delegation process, the response will always go back to the real application.

Since this is for public clients, a Refresh Token won't be issued. That means that new Access Tokens can only be received by involving the user.

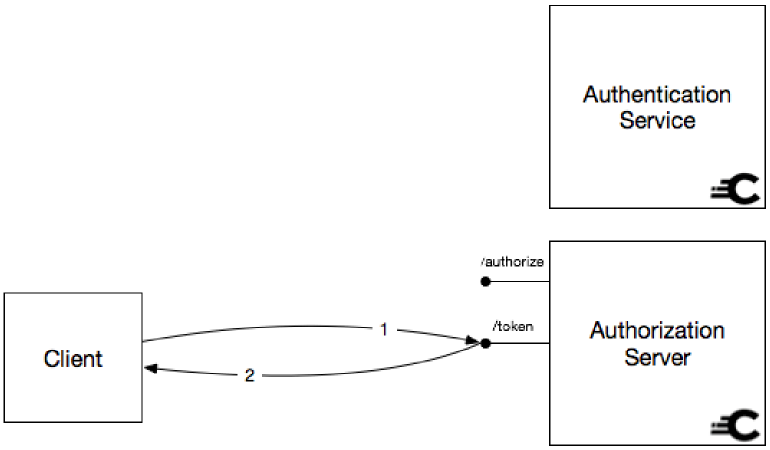


Implicit Flow: The full flow happens in the browser

3. Client Credentials Flow

In the Client Credentials Flow, there is no user. It is a flow that is strictly for server to server communication. In this situation, a server must access an API as itself. Therefore, there is no browser involved, and a private client is needed. To get an Access Token, the client simply passes it's credentials to the OAuth server and receives the token.

No Refresh Token is issued in this flow, since the client can just get a new Access Token using it's credentials anyway.



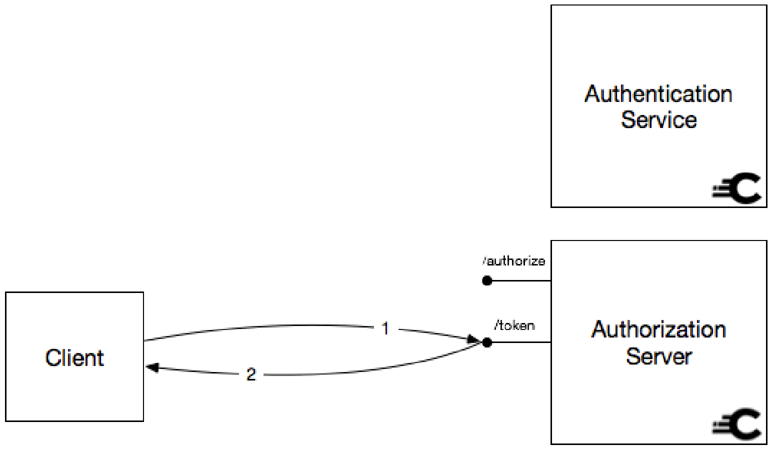
Client Credentials Flow: A server authenticates itself against the token endpoint. No user involved.

4. Resource Owner Password Credentials Flow

The Resource Owner Password Credentials (ROPC) Flow is pretty simple. The client collects the credentials from the user and passes them together with its own client credentials. The server responds with an Access Token and optionally a Refresh Token. Simple right? But there's a "but." And it's a big one.

ROPC is a flow that defeats one of OAuth's purposes; that the user has to give away its credentials to the app and thus has no control over how the client will use it. The flow is not recommended for use if you can use something else. It's only specified in the specification to allow for legacy or migration cases. ROPC should be used with care. An example could be an enterprise desktop application that is not easily updated yet needs to access the API platform.

We don't recommend the use of (ROPC). But if you really need to, ROPC should be used for private clients only, and the client could get a Refresh Token.



ROPC: The client sends users credentials together with its own credentials. Only for legacy use cases.

5. Dynamic Client Registration

While not one of the flows in the core OAuth Spec, Dynamic Client Registration solves an important use case for mobile clients. Since mobile apps are distributed through app stores, it's hard to give them a credential to identify themselves uniquely. Therefore, mobile clients are usually labeled as public.

Dynamic Client Registration tries to redeem that by specifying means for a client to register itself and request a unique credential upon installation. It works by letting the client send a registration token to the OAuth server, which generates a set of credentials and returns them to the client. These credentials can then be used in a Code Flow, and the client can now authenticate itself.

The registration token can be obtained in multiple ways: Either by letting the user authenticate itself in an Implicit Flow or by using the Client Credentials flow with a pre-distributed secret.

Outside of the mobile case, Dynamic Client Registration can be very useful for API management platforms that need to be able to create clients for the OAuth server.

6. Assisted Token Flow

The Assisted Token flow is a draft that is not part of the base flows, but it is worth mentioning. It is a sibling specification to OAuth that tries to make it easier for Single Page Applications to obtain tokens. It can be hard for those types of applications to handle Implicit Flow since it relies heavily on redirects. Instead, Assisted Token Flow defines a similar flow to Implicit, which instead uses iframes and postMessage to communicate.

Token Management

7. Introspection

Introspection is the way to ask the OAuth Server if a token is valid. Access Tokens are usually passed around by reference, meaning that they do not mean anything for anyone but the OAuth server. The introspection clients are usually an API or an

API gateway of sorts. Introspection is a simple authenticated call, where you send in a token, and the response is the data that belongs to the token, such as the expiration time, subject, etc.

8. Revocation

Revocation is one of the powers of OAuth. Without OAuth, a user that gave away its credentials to an application has no means of retracting that consent. The only way is to change the password, which might have bigger side effects than disallowing the app to access the user's account.

With OAuth, the user can decide to recall the consent whenever by revoking the token. In OAuth, you have two options for revocation; you can revoke the Access Token, which could be seen as ending the current session. If there is a Refresh Token, it would still be valid. Revoking the Refresh Token would make the Refresh Token invalid, and any active Access Tokens that came with it.

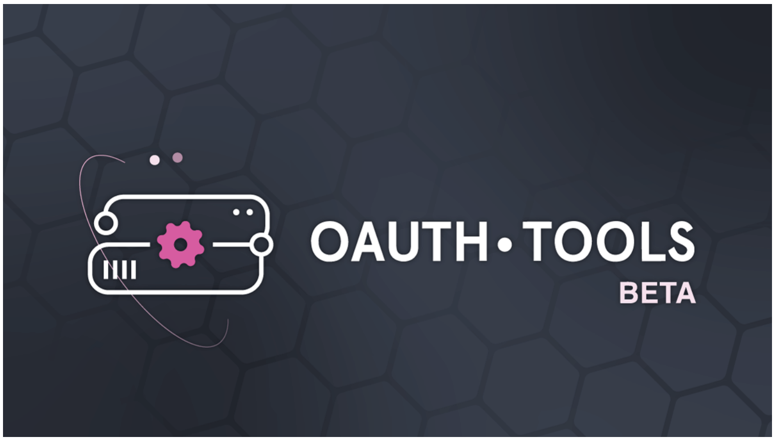
It is the client that performs the actual revocation with an authenticated call. Even though it's authenticated, public clients can be allowed to perform revocation.

Why Distinguishing OAuth Flows Is Important

It can seem like there are a lot of similar flows in OAuth, but each flow has its specific use case. By these essential flows, you should be able to pick the flow(s) that match your application and scenario.

Exploring OAuth.tools, The World’s First OAuth Playground

by **Kristopher Sandoval**
Originally Published Here



OAuth.tools, created by Curity, is a safe, vendor-agnostic place to experiment with a wide variety of OAuth flows.

API security is complex, and the underlying systems that support it are even more so. Getting a grasp on API security requires understanding many underlying components. Accordingly, any tool that can help contextualize these systems is not only an excellent educational tool, but it’s also a good business tool.

OAuth.tools looks poised to be that solution. Developed by Curity, OAuth.tools breaks down complex OAuth related topics, like flows and scopes, into visually understandable components. Each flow is understood in context with other flows by taking on this approach. So, does OAuth.tools succeed in depicting such complex topics? We think so. In this chapter, we review OAuth.tools, and consider why understanding OAuth flows is so valuable for securing your APIs.

What is OAuth.tools?

OAuth.tools is principally an educational resource – a safe, vendor-agnostic place to learn about and experiment with a wide variety of OAuth flows. It's designed to inform, not to proselytize a single vendor solution (or in fact, a single solution at all, as it offers a wide variety of different flows to test). Many flows that can be tested against a live environment, which allows you not only to see what each flow design looks like, but what the expected output and the various restrictions, requirements, and functional components look like from an operational standpoint.

The site currently supports nine flows: Decode JWT, Client Credentials Flow, Code Flow, Hybrid Flow, Implicit Flow, ROPC Flow, Device Flow, Introspection Flow, and Userinfo Flow. This comprehensive coverage allows for a wide variety of implementations to be represented throughout your testing.

When you first enter the site, you are greeted with the JWT token page. This page is a great example of how clarity and brevity are best used to communicate a ton of info at once. The page (and site in general) is well-designed and simple to understand.

The page is broken into three sections, representing a commonsense workflow. On the left, we can see the flows currently in use. From here, we can create, delete, or switch between other flows we've configured (as well as filter the current batch of flows by a stated criteria). In the center of the page, the workspace allows you to paste a JWT and switch between a handful of token types, including Access Token, Refresh Token, ID Token, Client Assertion, User Assertion, OpenID Metadata, and the ambiguously titled "Other."

Before one can test a flow, an environment needs to be created. The environment management system here is well-executed, both intuitive and complete.

One of the big features here is the inclusion of the WebFinger protocol. The protocol, specified by the IETF, is a mechanism by which information and metadata can be discovered automatically through a simple URL. While WebFinger is the specified protocol for OpenID Connect, its inclusion here was a smart move in terms of user experience – it lowers the barrier of entry by reducing the amount of work that must be done when creating a test environment, something that is still a bit of a chore in other solutions.

After the proper URL has been entered, WebFinger can automatically populate the rest of the information in the environments panel, listing the endpoints, the exposed scopes, any published keys, the possible mechanisms for authentication, and the expected response types, not to mention the metadata itself.

Within this system is a client management panel as well. A client will need to be created to act on the environment. This is easily done by simply entering the desired client ID/name, a secret phrase, and then selecting what flows to use. Included flows here are Code Flow, Implicit Flow, Hybrid Flow, Client Credentials Flow, ROPC Flow, Introspect, and Device Flow. Additionally, you can choose whether or not the client can be framed at this point, meaning it can be framed in an HTML element.

Once all of this has been set up, we can start creating flows. Flows in OAuth are methods of authorization that are often starkly different from one another, so the inclusion of a great many flows here is great – the user can familiarize themselves with some of the more esoteric or single-purpose flows while having access to general flows as well.

As stated previously, you can select from Decode JWT, Client Credentials Flow, Code Flow, Hybrid Flow, Implicit Flow, ROPC Flow, Device Flow, Introspection Flow, or Userinfo Flow to start testing a flow.

In this case, we've used a test JWT as provided by jwt.io. Here, we can see the fundamental simplicity of this site and the clarity it gains. To the left, we have all of our flows, and the prompt to add more. In the center, we have our JWT and a wide variety of options as to how we define that JWT to this system. On the left, we have a very clean, efficient way of visualizing what is happening during this decoding.

While this is by far the simplest of these options, it does expose a few things. First and foremost, it exposes how a JWT is composed, and what part of the JWT is responsible for what. This graphical representation of a data-heavy item is extremely useful, and when combined with the greater system of flows, helps to set the base moving forward for how the rest of our flows look and act.

Education vs. Proselytization

As we discuss OAuth.tools, we should recognize the difference between education tools versus the proselytization of specific solutions. When creating test cases, example environments, and so forth, it's very easy to fall into the trap of designing for marketing rather than designing for knowledge. While there's nothing wrong per se with designing forward-facing materials for marketing uses (and in fact, is somewhat expected), these tools must at their core do one thing – educate.

This is something that OAuth.tools does really well. Because this system was designed first and foremost to educate, and given that it's free to use and explore, the value of the tool for educational purposes cannot be overstated. OAuth can, at times, be quite complex – having the ability to play with it safely for free is great. Doing this without also being inundated with upsells is even better, and really suggests a user-experience mindset of teaching, not preaching.

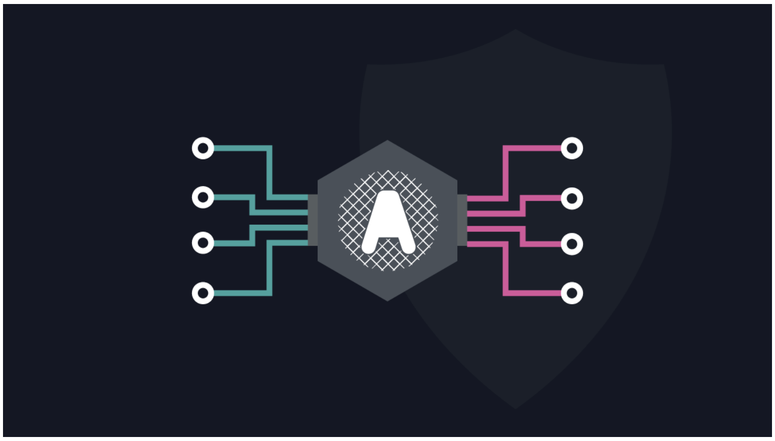
Conclusion

OAuth.tools is a great offering and provides a wonderful environment from which a wide range of testing can be done in a safe, zero-cost environment. In terms of tools that expose complex underlying systems, this solution does so in perhaps one of the more elegant and effective manners.

This tool is great not only for exposing these flows to those trying to test code or approach against the flows themselves, but for exposing the common layperson to what these flows look like, how they work with a variety of pieces of data, and ultimately, how OAuth in general looks, feels, and functions.

Strategies for integrating OAuth with API Gateways

by **Michal Trojanowski**
Originally Published Here



Choose the right OAuth flow for your specific API gateway scenario

Securing your APIs with OAuth proves that you have adopted a mature security model for your service. However, setting up OAuth in front of your API is not a trivial task. There are usually many architectural questions to be answered, more than the usual "which authorization server to choose" or "how to validate a token."

Developers often encounter a problem when their solution consists of a service mesh hidden behind an API Gateway. Should the API Gateway be involved in the process? If so, how can it help?

In this chapter, we'll demonstrate the different approaches of integrating OAuth with an API gateway. We'll also showcase examples of how these approaches can be used with different types of API gateways.

Two Strategies for Sharing Tokens

There are two main strategies for sharing access tokens with the outside world:

- You can share the JSON Web Token generated by the Authorization Server with any external client. In this scenario, the same token is used externally and internally by your APIs.
- You can share externally an opaque token, which is then exchanged for a JWT. This means that an opaque token is used externally, but internally, your APIs work with a JWT.

For the best security, the latter option is typically the recommended choice. In this scenario, you don't share any data embedded in your access token with any external clients. The access token contents are meant for your API, but if you share a JWT with external clients, anyone can decode the token and use its content.

Problems With Externalizing a JWT

Exposing a JWT with external clients can lead to privacy and security issues. Moreover, it can complicate your implementation. When you share a JWT with the external clients, developers may be able to decode the tokens and start using its contents, even though the tokens are intended just for your APIs. This could lead to situations where making any changes in the contents of the token can break existing implementations. The contents of a JWT token become a contract between you and the external developers, and breaking change should be avoided.

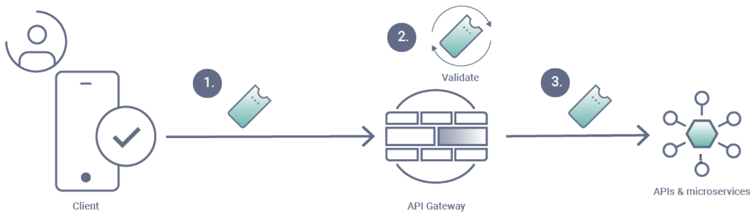
In the first approach, the integration with the API Gateway is not that crucial. The Gateway just forwards the JWT to the API, which can then validate it. Though, you can let the Gateway do some of the validation tasks.

If you decide to implement the recommended approach, proper integration with a gateway becomes more critical. To validate the token and read its contents, a JSON representation must be obtained. The exchange for a JWT can be done by every microservice which uses the token, but it can unnecessarily increase the traffic load and complicate the architecture as each service should be capable of performing the exchange. That's why it's much better to let the API Gateway handle this exchange. Now, let's explore different ways to integrate with API gateways.

Approaches for Integration With API Gateways

A few approaches can be used when integrating with an API gateway. They will depend on the token sharing strategy you choose.

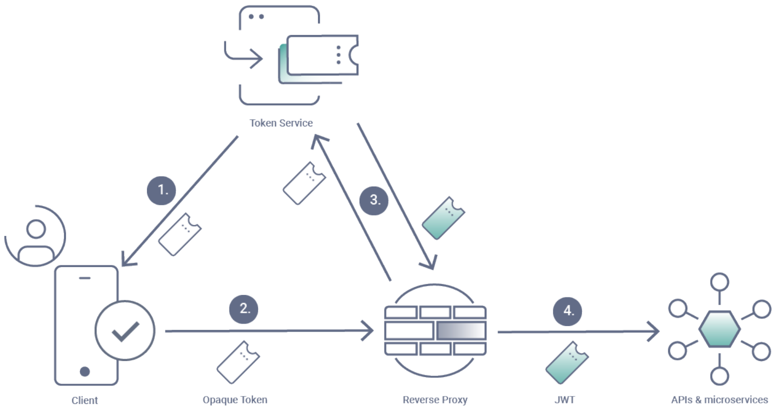
JWT Validation



If you decide to share JWTs with the outside world, it is good to have your API Gateway perform a validation of the incoming token. The Gateway can check the signature and perform some initial validation of the contents of the token. For example, it could assess the validity of the time-based claims, the issuer, the audience, or some required scopes.

Using JWT Validation, an API Gateway can reject requests with invalid tokens, limiting unnecessary traffic that would otherwise reach your internal network.

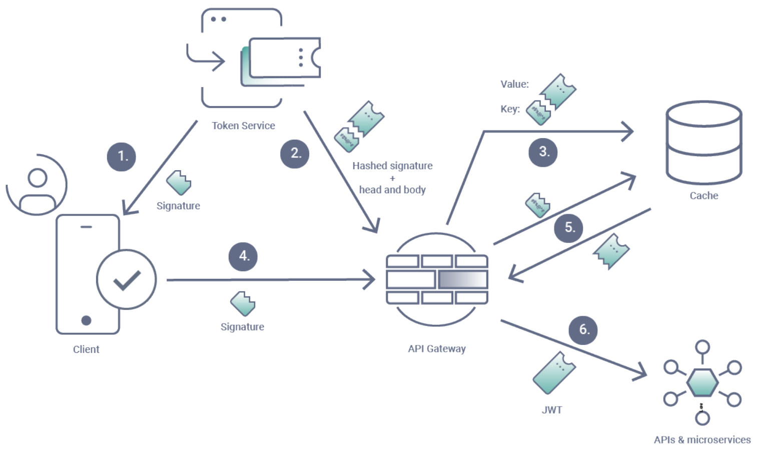
Phantom Token Approach



The Phantom Token Approach is one of the options you can implement if you want to share opaque tokens externally, but use JWTs between your internal services. The opaque token is exchanged for a JWT using the introspection pattern. The API gateway handles this exchange.

With this approach, all the services behind the Gateway don't have to perform the exchange themselves, limiting network traffic. A Phantom Token strategy is easier to maintain than having all services handle introspection on their own, as there is only one point from which the Authorization Server is queried.

Split Token Approach



The Split Token Approach is another option in which an opaque token is exchanged at the API Gateway for a JWT. In this pattern the Authorization Server splits the generated token into two parts:

- The signature of the JWT
- Head and body of the JWT

The signature is used as the opaque token by any external clients. The other part is sent to the API Gateway together with a hashed signature, where it is cached. Upon a request, the API Gateway uses the incoming part of the token to look up the rest of it in its cache and then glues them back together to create a complete JWT. The resulting JWT is added to the request forwarded to any services behind the Gateway.

This approach helps you further limit the network traffic needed to exchange the opaque token for a JWT, as no additional requests to the Authorization Server are required.

Example API Gateway Integrations

Should you decide to integrate your OAuth solution with an API Gateway, the approach you choose depends on the type of the API Gateway that you use. There are many different solutions available on the market, and many companies are even using their own proprietary gateways. Below, we'll examine three commercial API Gateway implementations.

Cloudflare: CDNs as Gateways

Cloudflare CDN provides enough functionality to be used as a distributed API Gateway. It is a Gateway spread across multiple data centers and regions with access to a shared key-value store and capable of making modifications to requests and responses through lambda functions.

If you use Cloudflare or your Gateway shares similar traits, the best approach would be to use the **Split Token Approach**, especially if the Authorization Server is deployed in a substantially lower number of locations. Thanks to the Split Token Approach, the Gateway does not have to query the Authorization Server on every request, which would otherwise considerably slow down the requests, given the servers are in different centers across the world.

When choosing the Split Token Approach, one thing to consider is that the Gateway needs access to a shared cache, one that can be easily populated by the Authorization Server, and accessed by all the distributed Gateway instances.

Nginx: On-Premise Gateways

Nginx is an example of an API Gateway that is installed on-premise. If this is the case for your Gateway solution, you should consider implementing the **Phantom Token Approach**, especially if the Authorization Server that you use is installed in the same data centers as the Gateway.

In such a situation, the additional request needed to introspect the token won't give much of an overhead to the request. What is more, in this scenario, you will not necessarily need a shared cache. Even if you want to cache responses from the Authorization Server, this can be done by each of the Gateway instances on its own.

Google Apigee: A Diverse Solution

The answer to the question of which integration pattern is best in a particular scenario depends on the Gateway features rather than the concrete product or vendor. For example, in the case of Google Apigee, you could either use the cloud version or install your instance on-premise. Thus, choosing the OAuth integration approach will depend on the way Apigee is used. In the cloud environment, it will probably be better to use the **Split Token Approach**, whereas, with the on-premise installation, you would probably go with the **Phantom Token Approach**.

Conclusion

You can use a few different approaches to integrate OAuth with an API Gateway. The solution used should be chosen depending on the features of your API Gateway. When deciding between OAuth integration styles, two key points to consider are whether the Gateway is distributed or centralized and whether the Gateway has reasonable access to a shared cache.

Although sharing JWTs with external clients may seem straightforward to implement, remember that it is not a good practice from a security perspective. You should avoid it, as it lowers your security and privacy, and may lead to problems with your integrators.

Assisted Token Flow: The Answer to OAuth Integration in Single Page Applications

by **Thomas Bush**
Originally Published Here

OAuth is an incredibly popular internet standard for granting apps and web services access to the information available on other websites. Though the implementation is complex, the premise is simple: you tell a website you want to access its data, you log in with the user's details, and off you go — but without some kind of protocol the process would be a whole lot more complicated.

There is one drawback in the current core version of OAuth, and it's increasingly evident with the recent trend towards building **single page applications** (SPAs). The issue is that designing a seamless, secure OAuth solution is difficult without a backend to do the heavy lifting, and, by definition, single page applications don't have one…

Below, we'll follow along with Curity solutions architect Daniel Lindau, as he presents the solution to OAuth usage in single page applications, following a presentation he gave at The Austin API Summit in Texas, 2018.

Implicit Flow: The Status Quo for OAuth in Single Page Applications

The current method of choice for handling **OAuth delegation** within single page applications uses the implicit flow — also known as the **client-side flow**.

It's simple, just redirect the browser to the **authorization server**, where the user directly authenticates and gives the app access, before returning to the application with an **access token** embedded in the URL. Then, the service can parse the URL for the token and immediately start using it.

There's no doubt that this is a messy approach. **Redirects are inherently counter-intuitive** if the goal is to build a *single page* application. What's more, you need to design an architecture whereby the application can seamlessly resume execution after you're done with all the redirects. Then, there's the question of storing your precious access token in a secure way.

Daniel says that Curity typically works with developers who aren't so familiar with OAuth; they don't exactly know the **best practices** for storing and using these tokens. Therefore, it's important to promote a more failproof alternative.

The iFrame: A Coffin for Your OAuth Delegation

An obvious workaround to all the problems caused by a redirect-powered, implicit flow is to hide away your OAuth implementation inside an **iFrame** — an inline frame.

Exactly as it sounds, tucking away your OAuth in an iFrame is a bit like putting it in a coffin. The reason is that an iFrame keeps your OAuth flow and the rest of your application separate, which can make it pretty difficult to communicate between the two.

Another problem with using iFrames, which coffins don't exactly suffer from, is that they can be accessed from **multiple places**. In the interests of secure authorization, you'll probably need to design it such that the frame can only be accessed from the active page.

So what's the solution?

Assisted Token Flow: iFrame Integration Done Right

Curity has built OAuth solutions to their customers' varying needs and constraints in a range of ways. While each case was slightly different, a common denominator was the usage of iFrames — but with precautions taken to avoid the associated problems.

Creating similar OAuth integrations time and time again gave the team a brilliant idea: why not standardize the process of OAuth delegation within an iFrame?

That's how **Assisted Token Flow** was born. It's a draft specification built onto OAuth — which is to say that it uses everything that is today OAuth — but adds new flows and endpoints to facilitate usage on single page applications.

The protocol uses **iFrames** for communication between the parent page and OAuth server, which prevents the pesky redirects we commented on earlier, and JavaScript's postMessage functionality for communication between the frame itself and the parent page.

A much-welcomed addition of new endpoints allowed the developers to remove **unnecessary OAuth parameters**, streamlining the entire delegation process. This is an important feature, as it makes page-iFrame interactions much easier to follow.

The result is a flow whereby only the `client_id` parameter needs to be communicated between the iFrame, parent page, and authorization server — any other parameters are optional.

In the event that other parameters are used, the Assisted Token Flow protocol also offers **parameter validation** within the implementation — on the side of the OAuth server — which cuts down on any excess back-and-forth between the application and authorization server.

Token Grants with Assisted Token Flow

The beauty of using a **standard** for OAuth integration is that every implementation uses the same workflow — in this case, with a simple but effective design.

Let's look at how tokens are granted with **Assisted Token Flow** when the user is already authenticated versus when the user hasn't yet authenticated.

Assisted Token Flow for an Authenticated User

With Assisted Token Flow, the workflow for an already authenticated user is extremely straightforward:

1. The application requires permission at an external resource server.
2. The page opens a hidden iFrame and points it to the resource server with just the `client_id` parameter.
3. The OAuth server serves a near-empty HTML page to the iFrame, including a `postMessage` script with the result of the transaction.
4. The page is loaded in the iFrame and a `postMessage` is performed, sending a success message along with the access token to the parent page.

In comparison to core OAuth, the primary advantage here is that Assisted Token Flow **doesn't mandate the inclusion of a scope parameter** (or any other parameter beyond `client_id`, for the matter); if the user doesn't specify a scope, Assisted Token Flow grants access to all possible scopes.

Assisted Token Flow for an Unknown User

Assisted Token flow for a user who hasn't yet been authenticated is similarly easy, with a few extra steps:

1. The application requires permission at an external resource server.
2. The page opens a hidden iFrame and points it to the resource server with just the `client_id` parameter.
3. The OAuth server sends a more detailed HTML page to the iFrame, including a `postMessage` script that asks the parent page asking for the login details

4. The iFrame is made visible to the user for authentication (e.g. as a username/password dialog) and the user logs in.
5. The application then retrieves data as necessary per the steps for an authenticated user.

Again, Assisted Token Flow shows the benefit of not needing any extra parameters, while it also shows how simple OAuth integration can be made when the iFrame aspect is taken care of.

Security Precautions in Assisted Token Flow

As we mentioned earlier, there are few **security constraints** apparent in using OAuth on single page applications. Two of the more major issues are the security of the iFrame itself, as well as the storage of access tokens.

Here's the precautions that Assisted Token Flow has taken against any such vulnerabilities:

iFrame security

There's a double-barreled approach to keeping the iFrames safe: for starters, the client is registered at the OAuth server to a particular domain (which is enforced with HTTP headers and content security policies), and, secondly, the domain is specified in the `postMessage` to prevent external access to the token.

Token storage

As for token storage, there are only really two options: `localStorage`, as written into JavaScript, or cookie storage. Curity recommends

cookie storage, as it allows the access token to be stored with a domain, path, and expiry time — so all interactions with the endpoint will send an access token for the OAuth server to act on.

Conclusion: Assisted Token Flow Makes OAuth Easy on Single Page Applications

Assisted Token Flow makes OAuth easy, especially for those who've struggled to find a sleek, but secure way to use it within single page applications.

It takes care of iFrames and token storage, creates a new, lightweight endpoint with just one mandatory parameter, and even validates any parameters for you. Best of all, the majority of this is achieved **server side**, so the developer doesn't have to worry about all the basics in their implementation.

The result is an OAuth protocol which is a heck of a lot easier to use, but sacrifices no functionality.

And just how easy is it? Here's an 8-line JQuery implementation where Curity sets up the origin and *client_id*, initiates the library, and off we go:

```
var settings = {
        clientId: "client-assisted-example",
        autoPrepareJqueryAjaxForOrigins: ['https://example.com'],
};

var assistant = curity.token.assistant(settings);

$(document).ready(function () {
        assistant.loginIfRequired();
});
```

Using OAuth Device Flow For UI-Incapable Devices

by **Kristopher Sandoval**
Originally Published Here

As the internet grows and more devices become interconnected, **authorization** is becoming more complex.

Early implementations of online services were easy to authorize against since they were tied to desktops, but modern authorization must consider varying environments, from mobile apps to **IoT** scenarios. Many of our new devices, such as smart TVs and voice-controlled speakers, don't have traditional **UIs** like web browsers.

The growing prevalence of **input-constrained** devices leads to a quagmire concerning how providers should actually authorize these devices. What is a secure way to enter username and password in **UI-incapable** systems?

Today, we're going to look at where this problem comes from and what we can do to fix it. We'll cover three unique **OAuth** flows to see how they stand up to solve the issue at hand: The Resource Owner Password Credentials Flow or **ROPC**, the **OAuth Code Flow**, and the **OAuth Device Flow**. We'll see which is the safest way to incorporate identity into these new environments to ensure that even your living room devices maintain a high-security level.

This chapter follows a presentation given by Jacob Ideskog of Curity at the Nordic APIs Platform Summit. View the slides here.

Identifying the Problem

To get to the root of the issue, let's take a hypothetical case. Let's assume that we want to stream music onto a **TV** from a data source. In this case, from a music streaming provider, we'll call **Musicbox**. Musicbox requires authorization for all streaming sessions, and in our hypothetical situation, this applies to our TV as well.

To get Musicbox to stream on our TV, we have a few options. We can use an app built for this specific purpose — this is commonplace on many modern smart televisions and offers what is essentially a web browser overlay that handles authorization. To do this, we would use a type of flow called the Resource Owner Password Credentials (ROPC) Flow.

ROPC OAuth Flow

In this approach, the ROPC flow has the resource owner issue a `POST` request with a form URL encoded body containing the user credentials to the **OAuth server**. This server is at the streaming service level and utilizes this credential to grant access to the internal systems. This is done by generating an **OAuth token**, which is then handed back to the TV, and passed on to the streaming service for each request. This is a very traditional OAuth flow, but it has some significant problems in our use case.

First, there's a major **security concern**. Most streaming providers are not going to want a TV utilizing proprietary codebases and systems to have the **login credentials** for all their users who choose to utilize the app. The issue of trust is especially relevant for **client apps** built on the streaming service API that are developed by third parties. Even if the application is an official one, this creates a major point of failure that dramatically expands the attack vector on the API itself and all but invites sophisticated token attacks.

More seriously, however, there's the fact that the ROPC flow simply was never designed for this application. While it seems a perfect fit, the ROPC flow is designed for **legacy systems** – utilizing it for smart TVs is an incorrect application and actually works against OAuth. As Jacob states:

> "The resource flow is not really meant for this… It's actually there to just solve **legacy** problems. If we're building a new system, we should never use it. It's why it has built-in **antipatterns**."

OAuth Code Flow

The whole purpose of OAuth is to not give passwords to 3rd parties, which this procedure would do. Consider we ignore ROPC and go for a more regular **OAuth code flow**, where the browser is used to send a `GET` request and an authorization page is used as a prompt.

In this case, we still run into a single fact that we can't avoid – all of this flow was meant for smart devices more capable than our constrained TV. The browser would be terribly slow, and entering a username and password with a remote control is inefficient. As such, even if these were acceptable solutions, they would result in **bad user experiences**.

A Question of User Experience

Even if we could ignore the technical issues inherent in this issue, the fact is that using the solutions often results in frustration because of the limitations on input and **interaction**. Utilizing a tiny remote control to enter in a complicated username/password pair and deal with any additional prompts that might pop up is ultimately quite cumbersome.

Things get worse if there's no controller at all. Imagine that, instead of our smart TV, we're utilizing a speaker such as an Alexa intelligent speaker. In this case, we no longer have a screen or a mode of easy input, and our issue becomes that much more complex.

What is our solution then? Luckily, there's a new OAuth flow being standardized that could help here. Instead of the Resource Flow or the Code Flow, let's turn our attention to the OAuth Device Flow.

OAuth Device Flow

OAuth recognized the issue inherent with authorization using constrained devices and has drafted a standard known as the OAuth Device Flow. The standard, currently under draft as "draft-ietf-oauth-device-flow-06", is specifically designed for **UI-incapable devices**, such as **browserless** and **input-constrained** systems. It therefore should be a good method for devices like our smart TV or voice-controlled system.

The flow looks similar to the traditional OAuth solutions but breaks away quite significantly at a very key stage.

In this solution, we have an API, the OAuth server, and the TV requesting the content. The TV starts by sending a Device Authorization Request, passing with it the scope and the client ID of the requesting device. From here, the OAuth server responds with a device code, a user code, and a verification URI. This also has an expiration timer and an interval that limits the exploitability possible in such communication.

From here on out, the OAuth flow breaks into a new form. The user visits the verification URI, enters in the requested data (typically the user code), and authorizes the device function. During this time, the OAuth server is told to wait for the user, and to expect the user code. A countdown is initiated that will automatically revoke the validity of the code passed if time is surpassed, giving the data an expiration time for security's sake.

During the time the user is entering the code, the device constantly polls the OAuth server on a set interval, and once the OAuth server receives the credentials, this polling is responded to with an authorization token. This token is then handed off from the device to the API to stream content.

How is this Different?

> “[This discussion is] about how we can work with devices that are not as smart as we’re used to… It’s really about getting identity into a new box that we haven’t thought about before.”

This flow is different in some pretty significant ways. First and foremost, there is the obvious fact that authorization of this type occurs **outside** of the device band. The device itself, in this case the TV, is not the flowing credential system that accepts login information – the user uses an **external system**, such as their phone, laptop, computer, etc., to verify the request and gain access.

This also means that access is not restricted to just physically entering in the login information – logins can occur using NFC, Bluetooth, and biometrics, and the code requested can be given using as many solutions. This is limited, of course, to nearby methods – OAuth does not allow this to expand outside of the near field, as allowing access from out of country or out of city could result in a wide attack vector.

This flow also allows for **refresh tokens** to seamlessly request a reauthorization, meaning users make a single request, and then automatically re-apply for this authorization without having to enter their credentials. This makes the entire authorization and user process not only technically more secure and effective, but better in terms of user experience — this, of course, was a major concern for other code flows, and as such represents a significant advancement.

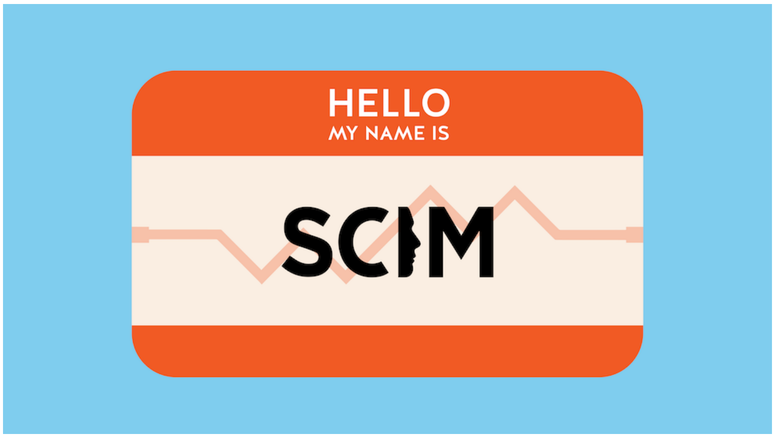
A Solution of Gaps

Setting up authorization for devices with limited user interfaces presents an interesting UX challenge. The problem is not going away. Until every device utilizing a service is either limited by policy or by reality to support advanced interactions and software suites, the issue will only worsen.

We live in a world of smart services, but the devices we use to interact with them are often quite dumb. Ultimately, using the **OAuth Device Flow** is the best current solution we have – the draft solution is effective and offers a wide range of options for authorization. As we saw in Jacob's talk, it's a standardized approach on how to login to a non-UX-friendly device.

SCIM: Building the Identity Layer for the Internet

by **Bill Doerrfeld**
Originally Published Here



In 2014, a working group reached a consensus for v2.0 of SCIM — a simple yet powerful standard that more and more large digital organizations are beginning to adopt for cross-domain identity management. Now, the SCIM specifications are standardized, officially published by the Internet Engineering Task Force as RFC7643 and RFC7644.

In this chapter, we introduce the SCIM protocol, track the progress of the standard, and identify new resource retrieval standards documented in the September 2015 RFC. Guided by IETF working group contributor Erik Wahlstrom, we'll introduce the basics of SCIM and identify lessons learned from the design of the SCIM API that have directed further iteration of the project as a whole.

What is SCIM?

Enterprises are extremely **distributed** — applications and data are sent and stored all over the place, from cloud servers, parter systems, to internal servers. Throughout a scattered environment, it's easy to lose control of where the data is. But as data privacy becomes more and more a heated issue, regaining control of identity is a top priority.

Enter SCIM. SCIM (System for Cross-Domain Identity Management) was developed to standardize how companies create, update, and delete identity data — a standard for the **life cycle management** of online identity by allowing a standard method for exchanging identity to other partners or systems.

SCIM is a lightweight provisioning protocol that specifically defines two things:

- **Scheme**: the identity profile could be a user, group, machine, or other resource entity. SCIM defines what those resources look like and how they are structured.
- **Protocol**: the method of transport; how do we send user data to different systems?

Standardized by the Internet Engineering Task Force (IETF), contributors to the project include companies like Nexus, Oracle, SailPoint, Salesforce, Google, Cisco, Ping Identity, and Microsoft. It seems like the SCIM standard is getting the hype and involvement it deserves, indicating a roadmap to future ubiquity.

Use Cases for SCIM

There are two distinct systems involved in using SCIM: a system that is either **creating** or **reading** user identity data, and the system that **stores** this data. In a world of competing regulations, you often need varying levels of trust between parties. In Germany, for example, in order to send personal information you need user consent every time you do it. SCIM doesn't go down this rabbit hole, not concerning itself with legal obligations. It assumes the right to share information is existent between the two players. Assuming this trust has already been established between the two entities using other security methodologies, SCIM can be used to exchange identity information for a variety of use cases. Here are three examples:

1: Synchronization Across Corporate Systems

What happens when a new employee joins a corporation? An HR manager will likely add a new user profile to their database. As it would be tedious to create redundant profiles in all cloud and internal systems, the company ideally wants to **automatically synchronize** data across all systems. Standardizing identity control with SCIM enables a method for creation, and removal of universal identity data.

2: On-Demand Provisioning

For companies using CRMs like Salesforce, you end up paying a monthly fee per each account. But what happens when employees leave, sales teams change, and thus the number of users altered? SCIM could help companies save money by instigating on-demand provisioning, wherein when a Salesforce account is created, a SCIM account is created. When a user quits, the account can easily be removed, and **operational costs decreased**.

3: Inter-Clouds Transfer

Have you ever had difficulty switching between accounts in Google Apps? Even more troublesome is transferring existing company assets between various cloud platforms. Suppose a company wanted to migrate from Office 365 to Google — there really isn't an easy way to do this. If all providers supported the SCIM standard, however, user information could be moved about more easily.

Schemas & SPML Comparison

Similarly to how web browser single sign-on (SSO) can be achieved by technologies like SAML and/or OpenID Connect, prior to SCIM, there have been attempts at standardizing cross-domain identity control. SPML, developed by OASIS, has been an open protocol since 2003. However, it is heavy, XML based, and doesn't define a **schema** — meaning that every time data is sent, systems on the other end don't really understand what the resource is supposed to look like.

SCIM, on the other hand, allows developers to create their own schemas, and defines two off the bat: **user** and **group**. Standards have also been made to extend this within the core specification with the **enterprise** user schema, to cater to an IT manager with unique privileges. SCIM also has a schema that defines schemas, enabling systems to speak with one another to find out what resources they support, and what they look like. A meta schema helps determine the capabilities of each server: can you create users? Filter users? This is a big difference between SPML and SCIM.

The SCIM API

SCIM is handled via a REST-based API for provisioning, change, and de-provisioning — all of which lie outside the realm of OAuth and SAML. With the rise of web APIs and microservices, SAML, has been deemed by some as too heavy with it's verbose XML. SCIM rather calls for authentication and access tokens in compact JSON, passed via HTTP headers.

The SCIM API can be tested from a command line, is cURL friendly, and firewall-friendly. Wahlstrom notes that REST-based APIs can be proxied through a firewall, and can easily implement standard security using SSL and TLS certifications and data encryption. SCIM standard doesn't necessarily define an authentication method but recommends OAuth2.

The SCIM core schema mentions 5 unique endpoints:

Resource	**Endpoint**	**Operations**	**Description**
User	/Users	GET, POST, PUT, PATCH, DELETE	Retrieve/Add/Modify Users
Group	/Groups	GET, POST, PUT, PATCH, DELETE	Retrieve/Add/Modify Groups
Service Provider Configuration	/ServiceProviderConfigs	GET	Retrieve the Service Provider's Configuration
Schema	/Schemas	GET	Retrieve a Resource's Schema
Bulk	/Bulk	POST	Bulk modify Resources

Reformatted from SCIM API documentation

For example, posting to the `/Users` endpoint can be used to create a user. In this case, a developer would receive an HTTP 201 successful response that includes an ID — a **unique identifier** created for each resource that also obtains its own URL. This acts as a **permalink**, allowing a developer to access the same user information from a `GET` response where the user info is always stored, regardless of future edits. Increasing discovery with the help of schemas is essential for partner-partner communication.

As user storages can be huge, SCIM specifications include features like filtering, paging, and sorting. Next, we'll explore some other features and see why these were standardized by the SCIM working group.

Features

As Wahlstroem describes, they have gone through many iterations of SCIM. After many decisions and voting (which often involved group humming to reach a consensus), the IETF working group reached standards for the following feature sets. Takeaways from these design lessons could definitely apply to other development scenarios, so take heed.

- **Extensibility**: In developing a standard, you can't please everyone — there will inevitably be outliers that request for extended features outside of the project scope. To this end, the SCIM team embraced the 80-20 rule, only specifying the most common 80% of use cases. Focusing on delivering **core cases** is essential for designing standards, as 20% percentile **corner cases** often take the bulk of your time and are far harder to implement.
- **Versioning of API and Schema**: SCIM standards place the versioning of the API in the **URL**. This means that a record of versioning is tracked for each specific resource, providing a historical record for all identity entries. Though the team considered versioning of API in the header, they opted for URL to retain the **permalink** for permanent profile discoverability in a single fixed location. This makes it

easier to understand for implementers and easy to track records with `/v1/Users/username`, `/v2/Users/username`, and so forth.

- **Weak ETags for Versioning of User Data**: ETags are used a lot in the web world, like for caching within your browser, for example. In SCIM, the HTTP function of *weak* ETags is used to track the versioning of specific files. Weak ETags allow systems to hold the same data even across different formatting options. Such may occur if a new developer using a variant JSON parser changes the placement of two different attributes. Weak ETags allow systems to know that data is the same.
- **Error Handling**: Defining error codes can be a tedious process, but it is worth it to increase the satisfaction of the end developer. Users don't like staring blankly at a 404 Error message — error responses need to be machine readable *and* human readable, which is why the group defines robust, detailed error codes in the SCIM specification.
- **HTTP Method Overloading**: Some firewalls and proxies don't like *all* HTTP verbs — often, servers and client servers don't support `DELETE` or `PATCH` calls. So, the SCIM standard solves this by allowing a `POST` call to be made with an `X-HTTP-Method-Override: DELETE` function — an important key in allowing requests to be made to different services with varying verb support.

Conclusion: Progressing a Needed Standard

A huge benefit of SCIM is that customers can own their own data and identities. Simplified Single Sign On (SSO), is an important step for the cloud and increasing interoperability across systems. SCIM is an important step for privacy and, according to Wahlstroem, a vital step in building an identity layer of the internet. According to the latest Request for Comments, “SCIM’s intent is to reduce the cost and complexity of user management operations by providing a common user schema, an extension model, and a service protocol defined by this document.”

Stay tuned for further articles, in which we will dive deeper into using the SCIM standard to create user accounts on a virtual service.

Other Resources

- SCIM Home Page
- SCIM Request for Comments 7644
- Nexus’s SCIM/RFC Announcement
- IndependentID, Phil Hunt
- SCIM Tutorial, Ping Identity
- Introduction to SCIM Presentation Slides, Twobo Technologies
- Managing User Identity across cloud-based application with SCIM
- SCIM Email Discussion Thread, IETF

Part Three: The Role of Identity

OAuth 2.0 – Why It’s Vital to IoT Security

by **Kristopher Sandoval**
Originally Published Here

OAuth 2.0 is vital to IoT security. The internet is fundamentally an unsafe place. For every service, every API, some users would love nothing more than to break through the various layers of security you’ve erected.

This is no small concern, either — in the US alone, security breaches cost companies over $445 billion annually. As the Internet of Things (IoT) grows, this number will only climb.

The problem is our considerations concerning security are for modern web services and APIs — we rarely, if ever, talk about the coming wave of small connected and unconnected IoT devices that will soon make this an even greater concern.

From the connected fridge to the smartwatch, the IoT is encompassing many new web-enabled devices coming to the market. As we're designing new API infrastructures, Jacob Ideskog believes that the "IoT is going to hit us hard if we're not doing anything about it."

Thankfully, there's a great solution by the name of OAuth. OAuth 2.0 is one of the most powerful open authorization solutions available to API developers today. We're going to discuss OAuth 2.0, how it functions, and what makes it so powerful for protecting the vast Internet of Things.

What is OAuth 2.0?

OAuth 2.0 is a token-based authentication and authorization open standard for internet communications. The solution, first proposed in 2007 in draft form by various developers from Twitter and Ma.gnolia, was codified in the OAuth Core 1.0 final draft in December of that year. OAuth was officially published as RFC 5849 in 2010, and since then, all Twitter applications — as well as many applications throughout the web — have required OAuth usage.

OAuth 2.0 is the new framework evolution that was first published as RFC 6749 alongside a Bearer Token Usage definition in RFC 6750.

What Does OAuth Do?

While by definition, OAuth is an open authentication and authorization standard, OAuth by itself does not provide any protocol for authentication. Instead, it simply provides a framework for decisions and mechanisms.

> "OAuth does nothing for authentication. So in order to solve this for the web, we need to add some sort of authentication server into the picture."

That being said, it does function natively as an **authorization protocol**, or to be more precise, as a delegation protocol. Consider OAuth's four actors to understand how it accomplishes this:

- Resource Owner (RO): The Resource Owner is the entity that controls the data being exposed by the API, and is, as the name suggests, the designated owner.
- Authorization Server (AS): The Security Token Service (STS) that issues, controls, and revokes tokens in the OAuth system. Also called the OAuth Server.
- Client: The application, web site, or another system that requests data on behalf of the resource owner.
- Resource Server (RS): The service that exposes and stores/sends the data; the RS is typically the API.

OAuth provides delegated access to resources in the following way. Below is a fundamental flow in OAuth 2.0 known as implicit flow:

- The Client requests access to a resource. It does this by contacting the Authorization Server.
- The Authorization Server responds to this request with a return request for data, namely the username and password.
- The Authorization Server passes this data through to an Authentication solution, which then responds to the Authorization Server with either an approval or denial.
- With approval, the Authorization Server allows the Client to access the Resource Server.

Of note is that OAuth 2.0 supports a variety of token types. WS-Security tokens, JWT tokens, legacy tokens, custom tokens, and more can be configured and implemented across an OAuth 2.0 implementation.

Unique IoT Traits that Affect Security

Now that we understand OAuth 2.0 and the basic workflow, what does it mean for securing the Internet of Things (IoT)? Well, the IoT has a few unique caveats that need to be considered. Ideskog notes that IoT devices are typically:

- **Battery-powered**: IoT devices are often small and serve a particular function, unlike server resources, which have massive calculation-driven platforms and consistent, sanitized power flow.
- **Asynchronous**: They are partially or completely offline, connecting only asynchronously via hub devices or when required for functionality.
- **Lean**: Lastly, IoT devices usually have limited calculation capabilities and depend on central devices and servers for this processing functionality.

Despite all of these caveats, IoT devices are desirable targets to attackers due to their known single-use functions and relatively lax security.

Proof of Possession

Due to all of these caveats, the OAuth workflow is strikingly different — we are, in fact, using a methodology called Proof of Possession. Consider a healthcare scenario in which a doctor must access an EKG IoT device. Since the IoT device cannot perform the same authentication process as a full client device can, we need to do a bit of redirection.

The start is normal. The Client sends an access request to the Authorization Server. From here, the Authorization Server contacts the Authentication Server, which prompts the Client with Authentication Data. When this is provided, the Authentication Server authenticates to the Authorization Server, which issues an authorization code to the Client:

```
authorization_code = XYZ
```

From here, we deviate from the standard OAuth workflow. The Authorization code is a one-time use proof that the user is Authenticated, and this code can be used to further contact that IoT device as an authorized device. The code is not something that can be used to directly access data as other OAuth tokens are, it is simply proof that we are who we say we are and that we've been authenticated and authorized.

The Client then generates a key (though this key can also be generated server-side) to begin the connection process with the IoT device, sending a packet of data that looks somewhat like this:

```
Client_id = device123
Client_secret = supersecret
Scope = read_ekg
Audience = ekg_device_ABC
authorization _code = XYZ
…
Key = a_shortlived_key
```

With the data in hand, the Authorization Server now responds to this packet by providing an `access_token`; a reference to data held in the Authorization Server memory to serve as proof of possession of both authentication and authorization:

```
Access_token = oddfbmd-dnndjv…
```

This is the final step — the client is now fully and truly authenticated. With this access_token, the client can start a session on the IoT device. The IoT device will look at this `access_token`, and pass it to the Authorization Server (if it's a connected device), asking for verification to trust the device. When the Authorization Server accepts the verification, it passes a new key to the IoT device, which then returns it to the Client, establishing a trusted connection.

Disconnected Flow

What happens if a device is unable to ask the Authorization Server for verification due to power or calculation limitations? In this case, we can use something called Disconnected Flow. A key point for Disconnected Flow is unlike other OAuth 2.0 solutions; this eschews TLS (Transport Layer Security) by nature of the Resource Server being a disconnected device with intermittent connectivity and limited communication and processing power.

In this case, we're actually shifting the parties around somewhat. The EKG machine is now the client, and another IoT device, a test tube, is the Resource Server. First, the EKG machine authenticates and authorizes in the same way as before:

```
Client_id = ekg_device_ABC
Client_secret = supersecret
Scope = read_result
Audience = connected_tbie_123
Token = original_token
.....
Key = a_shortlived_key
```

Once the Authorization Server receives this, the server replies not with the access token in the former structure but instead an `access_token` in JWT (or JSON Web Token). This token is a by-value token, meaning it contains the data fed to it and the response. Whereas our first string referenced a memory location in the Authorization Server, the JWT has all of the data in a single key string.

From here, the JWT can be converted into other formats for easier reading by the test tube. By design, the test tube is crafted to trust the Authorization Server in a process called Pre-provisioning. Because of this, when we send the Client token in JWT (or whatever format that's been chosen), the tube implicitly trusts the key as long as it originated from the Authorization Server, and begins a connected session with the Client.

Ideskog notes there would technically be 2 token types involved in the flow above: a signed JWT would contain an encrypted token (JWE), which has a key in it that is later used for the communication channel. The JWS (commonly called JWT) isn't necessarily encrypted and is usually in plain text and signed.

Real-World Authorization Failure

To see exactly why this is all so important, consider some real-world authorization failures. One of the most visible failures is known as "The Snappening," a leak of over 90,000 private photos and 9,000 private videos from the Snapchat application.

Most of the blame for The Snappening came from users using unauthorized third-party applications to save Snaps. These third-party applications did not utilize OAuth solutions, meaning when remote access users attempted to use the undocumented Snapchat URL that the third party application relied on, they could spoof as authorized users and retrieve content without proper token assignment.

This a great example of OAuth 2.0 vs. no implementation, as we have essentially a "control" application (Snapchat secured by OAuth) and a "test" application (the unauthorized applications tying into the undocumented API). With improper authorization integration, the content was allowed to leak through an insecure system with relative ease. Had the third party application properly implemented an authorization scheme, this would never have happened.

This issue is only relevant to non-IoT things, though — it's just a photo-sharing application, right? Wrong. Consider now this same fault of security for something like an IoT button that triggers replacement business items. Attacking this device can result in man-in-the-middle attacks to capture addresses of order processing servers and even spoof orders to the tune of thousands or hundreds of thousands of dollars.

OAuth Embeds Trust into the IoT

Applying OAuth to the IoT makes it truly extensible and customizable. Using OAuth, we can build systems based on trust that use fundamentally secure resources and communications protocols. OAuth is, by design, all about trust.

This trust is key to securing the IoT. For connected devices, Proof of Possession can solve most security issues. For constrained environments by either connectivity or processing calculative power, devices can be secured using pre-provisioning that is independent of TLS and does not require the device to be online at all times.

It should be noted that JWT, JWS, and JWE are all helpful tools, but all work with JSON. For lower processing environments, sibling binary tokens such as CWT, CWS, and CWE can be used as they cater well to building on low power and limited scope devices.

Conclusion

This isn't a game — though having lax security can be convenient for innovation and experimentation when it comes to the IoT, this is a dangerous approach. IoT devices might be underpowered and single-use, but they, as a network, are powerful.

Remember that a network is only ever as secure as the sum of its parts and the weakest point of entry to its ecosystem. Failing to secure one IoT device and adopting a security system based on inherited security can result in a single IoT device comprising every device connected to it.

OAuth 2.0 can go a long way towards solving these issues.

Is OAuth Enough for Financial-Grade API Security?

by **Art Anthony**
Originally Published Here

“If you think about where OAuth [and OpenID Connect] started, it was really about securing comments on blog posts, and now we’re talking about enterprises, so it’s a whole different class of security.”

This is how Travis Spencer, CEO at the identity company Curity, opened his talk at our 2019 Austin API Summit, and it’s an astute summary of the way many products (particularly in the tech scene) are tweaked or re-engineered for new things different from their original purpose.

As Spencer clarified in his talk, “when we say banking grade, we’re not just talking about banks; we’re talking about health-care, government, and high security.” In other words, financial grade security is relevant to any data-centric API.

It’s often true that products that begin their life as one thing struggle to adapt to new tasks because they are not originally designed for. In this post we’ll be looking at whether or not that’s the case here, i.e. how the likes of OAuth have evolved from humble beginnings and if they’re truly capable of being used for financial grade protection.

Can OAuth Make The Grade?

Early in his talk, Spencer provides a summary of some things you can do with your OAuth implementation to provide financial grade security:

- Mutual TLS constrained tokens
- PKCE (Proof Key for Code Exchange)
- Consent
- Signed request/response objects
- Pairwise Pseudonymous Identifiers (PPIDs)
- Phantom tokens
- Strong authentication
- Prefix scopes
- Dynamic clients

There’s no denying that’s a pretty comprehensive list and suggests that OAuth is more than capable of holding its own when it comes to security.

In other words, OAuth shouldn’t be seen as insufficient when it comes to securing data. In fact, Spencer outlines a number of ways in which it works very well for financial grade protection.

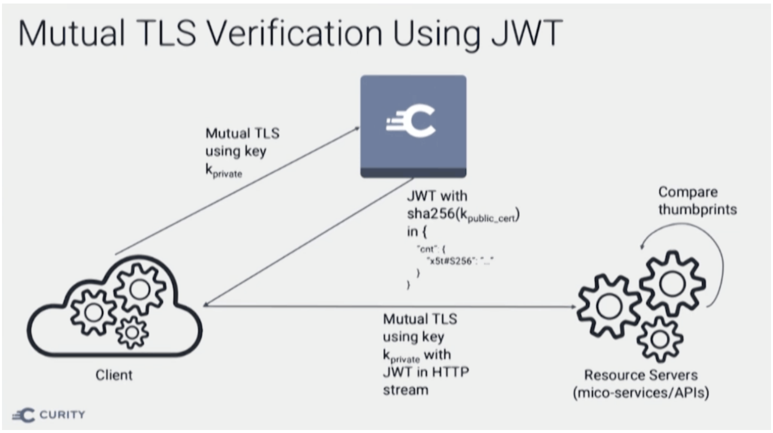
For example, he talks about how PPIDs can be used to prevent collusion between unrelated apps – while allowing interaction between, say, sites and their associated mobile apps – and how phantom tokens (vs., say, a JSON web token) allow for a smaller regulated space while preventing clients from accessing any PII and front-end clients from depending on the content of access tokens.

Some Tokens Are Unbearer-able

The way developers use OAuth isn't, however, always perfect. Its two main vulnerabilities of OAuth, as seen by Spencer? Bearer tokens and the redirect. "Those two things are the primary attack vectors in OAuth."

He likens the former to bearer bonds; "if you bear the bond, then…you can get a million dollars for each piece of paper and have a great life!" That's particularly problematic here because, unlike with bearer bonds, there's no way to lock tokens away in a physical safe.

"That's the Achilles heel because, if I can snatch someone else's token, I can become them." So what's a solution that can get around this? "The opposite of this is a holder of key token or proof of possession token, and that's what mutual TLS constraint tokens are."


Mutual TLS Verification Using JWT
Mutual TLS using key $k_{private}$
JWT with sha256(k_{public_cert}) in { "cnt": { "x5t#S256": "..." } }
Compare thumbprints
Client
Mutual TLS using key $k_{private}$ with JWT in HTTP stream
Resource Servers (mico-services/APIs)
CURITY

In simple terms, the addition of a hash of a certificate to the token exchange process means that works something like using a cipher to decode a message; without the proper credentials, the token won't function properly when used for an API call.

Unfortunately, that isn't the end of the story…

Away With The PKCEs

When talking about redirects, Spencer refers to the vulnerability in callbacks that exists if…

- It's not confidential, i.e., not done over TLS
- There's no authentication done on the token endpoint
- The callback is not unique per client or per client operator

"In cases where a bad actor can intercept the credentials, they may be able to sidestep the use of mutual TLS tokens outlined above entirely." He also highlights that this is something particularly prevalent in mobile devices.

The solution? Proof Keys for Code Exchange (PKCE). “We can add a step 0 into this process…The legitimate application uses a proof key to prove that it is the actual application that started this flow, with the OAuth server refusing to finish it unless that proof is verified.”

You could think of this as a more robust version of Dennis Nedry’s “ah ah ah, you didn’t say the magic word” in Jurassic Park…

Signed, Sealed, Delivered

Spencer highlights the issue that, in most cases, OAuth flows go through the user agent. Malware installed in the browser, for example, can undermine all of the measures taken above on the user side.

> “How do we know that OAuth flows aren’t being tampered with in flight?” Spencer asks. “We sign the request from the client sent to the server and back…We can also sign the response that’s sent back to the client as well so they know nothing was tampered with on the way back.”

All of this signing and hashing is something that’s already proven its worth in recent years: read any article about data leaks by big companies, and you’ll notice, in cases where it’s been done, how eager they are to talk about hashing passwords.

Although hashing passwords, signing requests, etc. isn’t always 100% secure — there are cases, for example, in which weak hashing has been cracked — it is, at this point in time, about the best thing we can do.

What's Next For Financial Grade API Security?

It seems likely, for now at least, that the status quo will continue as the norm in the world of financial and financial grade APIs.

From PayPal and Stripe to Coinbase, and probably Facebook's Libra in the not too distant future, there are plenty of financial services utilizing APIs built around existing frameworks and services like OAuth.

Suppose actual financial services are already relying on these. In that case, there's very little incentive for other API developers to go beyond what already, in effect, constitutes financial grade API security…barring massive data leaks or the exposition of serious vulnerabilities.

In the meantime, we would expect to see OAuth continue to be one of those rare exceptions: a service that was built with something minor, i.e., securing blog comments, in mind that has effectively expanded to something much more robust than that.

The Role of Identity in API Security

by **Bill Doerrfeld**
Originally Published Here

What options do APIs and microservices have when it comes to authentication and authorization? What is the role of identity in API security? In a recent LiveCast, we sought to discover best practices for handling identity within API security. We featured two illuminating lightning talks, one from David Garney of Tyk and another from Travis Spencer of Curity. This event is a nice capstone to the API security and identity themes we've covered in this eBook.

Various Authentication Mechanisms

So, how do we authenticate users for our APIs? Embedded into third-party applications, APIs require a unique type of access control, and there are many options for microservices authentication and authorization.

The Benefits of Microservices

As we’ve seen before, microservices are lightweight, offering individualistic simplicity that monolith applications simply can’t provide. They are also flexible; services can be deployed to different hosts and with varying technical stacks. They work well with containerization, CI, and CD. Furthermore, since these applications are separated, it means the attack exposure is decreased, increasing security overall.

Though microservices are unique, they often work in tandem with one another, meaning we need them to act cohesively. One example is having a standard way of knowing who is consuming them and their access privileges.

So, how do we implement a standard authentication and authorization method across all microservices? In his presentation, Dave Garney of Tyke cites three specific ways to implement this control:

1. Internally within each microservice
2. Externally by using a gateway
3. Or using a combination of external and internal components

Internal Approach

In the 100% internal strategy, microservices handle both authentication and authorization. In this architecture, the user traverses through both. Thus, the service has fine-grained control, but it also means that the service is self-reliant, fitting with the microservices approach. Cons include extra development effort, and likely additional microservices must be being constructed to handle these functions.

Dave argues that by handling authentication and authorization separately for each microservice, we're missing an opportunity to reduce code bloat. The larger the number of microservices involved, the more value in externalizing this approach.

External

With an external approach, a gateway handles both authentication and authorization. Thus, additional microservices do not need to be created. The pros are that by putting responsibility in the gateway, you can remove the burden from microservices to focus on your own needs. The cons could be less fine-grained control if the gateway can't make decisions on information it doesn't have access to. Dave also notes some potential vulnerabilities; relying completely on a gateway means you could attempt to circumnavigate it and access the source directly. Of course, there are ways to mitigate this.

Combination

In Dave's combination approach, authentication takes place in the gateway, as the external approach, thus relieving the burden for each microservice to handle authentication. Authorization takes place in the microservice. Microservices thus become leaner as they don't' have to worry about authentication, yet they can still authorize based on the credentials provided. Cons could be a little bit more development effort.

Goldilocks Approach?

Dave recommends the external approach, if possible, as allowing a gateway to handle both authorization and authentication brings time-saving qualities. He notes that, practically, most will use a Goldilocks approach, a middle ground between combination and external approaches.

> "Try to identify complex authorization requirements and implement them in your microservice."

Dave defines this way of thinking as Macro-authorization vs. Micro-authorization.Macro-authorization as in the gateway authorizing access to all of your microservices and Micro-authorization as each microservice providing further authorization themselves. He recommends complex authorization mechanics catered to the unique scenarios at hand. Database access, for example, is best implemented as micro-authorization.

Dave recommends OpenID Connect for authentication, and for the most part, stresses the importance of gateways for offloading functionality to the gateway:

> "What you should end up with is the right balance of off-loading as much authentication and authorization effort to the gateway as possible while retaining the ability to perform complex authorization at the microservice level."

Moving Forward with OAuth and OpenID Connect

Next, let's dig into OAuth and OpenID Connect a bit more. Travis Spencer, CEO of Curity, is no stranger to strategies for identity control. He describes Curity as an advanced off-the-shelf OAuth server, fit for banking, retail, telecom, and many other use cases.

The 4 Actors of OAuth

Travis defines the basic foundations of OAuth with these four actors. This is what the literature and documentation concerning OAuth will often use. These four actors make up the flows that we discuss on the blog often.

1. Client: The application; can be mobile apps, web apps, server applications, and more.
2. Authorization Server (OAuth Server): Sometimes called identity provider, or in OpenID Connect, it's called the OP.
3. Resource Server: These are the APIs themselves.
4. Resource Owner: Typically, a user, the one that controls the resource. Could also be an organization, but is often a user.

So how do these actors typically interact? Well, for more fine-grained comparisons, reference Part 2 of this eBook for specific OAuth flows!

Nordic APIs Resources

Related API Security and Identity Sessions

- OAuth Assisted Token Flow for Single Page Applications
- OAuth Claims Ontology: Using Claims in OAuth and How They Relate to Scopes
- OAuth for Your Living Room
- Jacob Has a Horse, Says Travis – a Tale of Truths In a Microservice Architecture
- Identity:The Missing Link in API security
- Lessons Learned from the Design of the SCIM API
- LiveCast: The Role Of Identity In API Security

Visit our Youtube Channel for other full videos of high impact API and identity-related talks.

More eBooks by Nordic APIs:

How to Successfully Market an API: The bible for project managers, technical evangelists, or marketing aficionados in the process of promoting an API program.

The API Economy: APIs have given birth to a variety of unprecedented services. Learn how to get the most out of this new economy.

API Driven-DevOps: One important development in recent years has been the emergence of DevOps, a discipline at the crossroads between application development and system administration.

Securing the API Stronghold: The most comprehensive freely available deep dive into the core tenants of modern web API security, identity control, and access management.

Developing The API Mindset: Distinguishes Public, Private, and Partner API business strategies with use cases from Nordic APIs events.

Create With Us

At Nordic APIs, we are striving to inspire API practitioners with thought-provoking content. By sharing compelling stories, we aim to show that everyone can benefit from using APIs.

Write: Our blog is open for submissions from the community. If you have an API story to share, please read our guidelines and pitch a topic here.

Speak: If you would like to speak at a future Nordic APIs event, please visit our call for speakers page.

About Nordic APIs

Facebook | Twitter | Linkedin | YouTube

Blog | Home | Newsletter