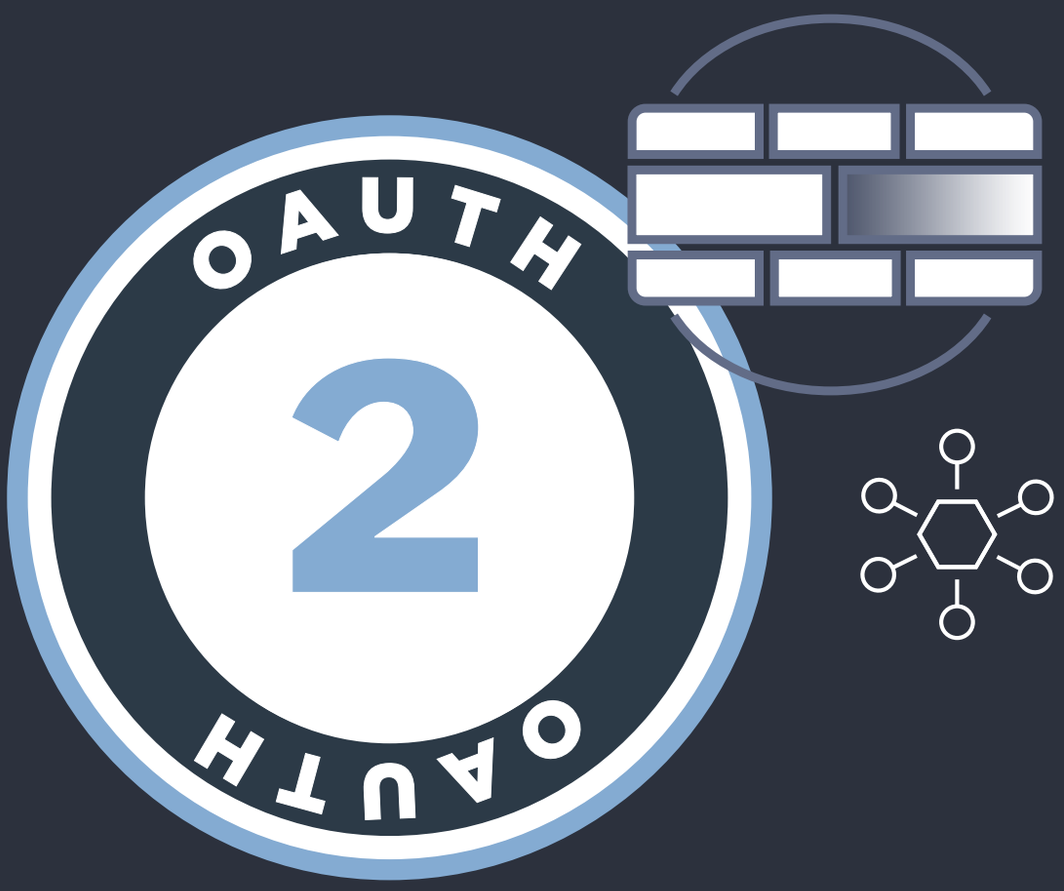
CURITY

OAuth and API Gateways



Conflating Token Issuance and Validation Leads to Brittle API Platform Security

Table of Contents

Introduction

An API gateway or API manager provides a facade to access back-end APIs. This kind of reverse proxy is deployed in most API-based infrastructures to ensure that access to back-end services are authenticated and authorized and to help centralize some common features used across the APIs (like path rewriting). In order to fulfil most of these features, the gateway needs some proof of who the user and/or the caller is. This is commonly done in a token-based manner where the token or ticket is a representation of the end user and includes information about the calling client application that the user is operating. Such tokens can be obtained in different ways, but currently one of the most popular ways to do that is to use OAuth 2 and OpenID Connect standards as the protocol that defines how such tokens should be issued.

Once these tokens are issued and presented to the API gateway, they must be validated. How this is done depends on various implementation details. If the OAuth server is implemented using the same product as for the API gateway, validation is done by using a shared token state that both services have access to. In such deployments, user- and client-application-related information is typically stored in a database where the OAuth token is issued. Later, when it is presented, it is validated by looking up information in the same database. This deployment model tightly couples the two inherently different services. These two capabilities fall into separate subsystems of an API platform. By running an OAuth server within the API gateway, the border between the Identity Management and API Management subsystems are blurred. This leads to fragility, vendor lock in, and brittleness. Because the lifespan of an API platform is at least 5 to 10 years, this unstable foundation will decrease the longevity of the platform and create technical debt that will be difficult if not impossible to pay off. Due to the differences between token issuance and validation, using the same product for both will lead to suboptimal implementations; using the right tool for each job, on the other hand, will ensure greater chances for success over the long run.

Solving Hard Problems Through Decomposition

Computer programs are built by breaking down complex problems into smaller and smaller pieces; the problem is broken down until each individual piece can be comfortably reasoned about. Then, they are assembled into a working whole that solves the difficult problem. Providing secure access to data via APIs is done in the same way. The assembled whole -- the API platform -- is broken down into three primary subsystems:

1. Identity Management System (IMS)
2. API Management System (AMS)
3. Entitlement Management System (EMS)

Each of these systems are decomposed further into major components, but they represent the first breakdown of any API platform.

As these systems are assembled into a working whole, it is important that the interfaces between them are well defined and standardized. Well-defined interfaces ensure that the systems are loosely coupled and the platform is highly cohesive. Known contracts also allow teams and groups to collaborate early and continuously. By using standardized protocols to exchange data between these primary systems, organizations have more implementation options; ready-made products from various vendors can be purchased or in-house implementations can be coded. Knowledge and resources explaining these protocols are available from many different sources, increasing knowledge and allowing new team members to become productive more quickly. Standardized interfaces also allow an organization to swap out products and implementations in each subsystem without affecting other aspects of the API platform. By using protocols that are governed by an international standards body, exchanging one product for another will interoperate with other parts of the platform without disruption to remaining parts.

Unsurprisingly, the API gateway is a part of the AMS while the OAuth server is a part of the IMS. The OAuth server is a specialized Security Token Service or STS that issues and validates tokens according to the OAuth specification. It falls within the boundaries of the IMS because it issues token-based representations of user and client identities. It does so after users are authenticated. Authentication is a complex task that falls into the realm of identity management. Another reason the OAuth server belongs within the IMS is because it is often used in non-API use cases. A common example is when the OAuth server also supports OpenID Connect and is used for Web Single Sign-on (SSO). In such a case, authentication, identity management, and federation are involved, while API access is not. Thus, including the OAuth server in the IMS leads to a more

accurate abstraction of the problems it lends itself to.

Because the OAuth server and API gateway belong in different subsystems, the interface between them should be standards-based. The standard-based communication protocol is naturally OAuth 2* proper (i.e., RFC 6749) as well as auxillary standards, namely RFC 6750 for presenting tokens to the gateway and RFC 7662 for validating and introspecting them. These together form a suite of standard compliant interfaces for which most programming languages and environments already include implementations. Where such alternative implementations are desired, many open source options exist in all commonly used programming languages.

This separation of token issuance and validation concerns means that the OAuth server and API gateway should not be coupled at the database level or share any proprietary state. The tokens issued by the OAuth server should be possible for the gateway to validate using the standards-based interface mentioned in the previous paragraph. This ensures that the two subsystems are decoupled and remain so, regardless of which products or implementations are used. This is a very important priority for API platform architects and product owners who are creating a system that may outlast their tenure if successful.

The Differences Between Token Issuance and Validation

Overview of Token Issuance

A security token is a memento or ticket that represents some user or other entity; a token is asserted by some entity, usually referred to as an STS. To allow applications throughout an eco-system to trust the validity of tokens, the STS will issue tokens in a couple of different ways:

1. Tokens will refer to the identity data of the user (or other entity); or
2. The identity data will be included in the token and the information will be cryptographically signed by the STS.

The first approach is referred to colloquially as "by reference" or "by ref" tokens whereas the latter is called "by value" or "by val" tokens. These common synonyms are taken from the programming paradigms of passing data to functions on the stack (by val) or as a pointer to the data (by ref).

The actual type of the token is different from the way in which it is passed. Tokens can be of various types. Some example include:

- WS-Security tokens, e.g. SAML tokens
- JSON Web Tokens (JWT)
- CBOR Web Tokens (CWT)
- Legacy tokens (e.g., those issued by a Web Access Management system)
- Custom tokens

A token service may issue any of these types of tokens and pass them using either by-val or by-ref methods. Tokens of any kind can be profiled to support various identity data. Common examples include Holder of Key (HoK) and bearer tokens. These provide different levels of security and will be used in various use cases. An STS may also issue different kinds of tokens. Most often the tokens it produces are used as Access Tokens (AT), tokens which provide access to data as the user identified by the token. Another example is the Refresh Token (RT), which is a token that allows a client application to request new tokens; it is sometimes called a Ticket-granting Ticket (TGT). Sometimes a token will include not only information about the user, but about another user that is acting as or on behalf of that user. The former is referred to as an ActAs or delegation token while the latter is called an OnBehalfOf (OBO) or impersonation token. A token service may also limit the number of times a token may be presented. A very common example is a token that can only be used once; this is referred to as a no-more-than-once or nonce token.

All of these token types, kinds, profiles, and semantics of what a token means and can do are exchanged with applications that need tokens. An STS does this using one or more protocols and flows. For modern APIs, the protocol of choice is OAuth 2, but it need not be. Platform providers who have a long-term view must plan from the beginning to be able to issue tokens using alternative protocols in case others become necessary during the lifetime of the platform. Besides the protocol, exchanging tokens requires an agreement on how authenticity of tokens will be ensured. Using by-ref tokens, validity is checked by exchanging them for the identity data to which they refer. For by-value tokens, however, cryptography is used to sign and validate the tokens. The algorithms, key types, and methods for this must be decided and deployed in a flexible way that can be changed later and altered per use case. Even with this brief overview of token issuance, it is hopefully clear that token issuance is a complex procedure. Before it can even take place, however, another even more complex operation has to be performed -- authentication.

Authentication, the Prerequisite to Token Issuance

Some of the aspects of token issuance were touched upon in the previous subsection. There are many others as well that make token issuance a complex process. Before any of the particulars of token issuance can be performed, however, a user must be authenticated. This raises many other issues that are even more intricate. Some of those include:

- How a user obtains a credential and manages that secret.
- How multiple credentials are linked to the same account.
- How the User Experience (UX) is controlled based on contextual information (such as the application the user is operating).
- Should Single Sign On be allowed or disallowed, depending on the conditions of the authentication transaction.
- Which, if any, credential types or authentication providers should be available to the user for any particular transaction.

There are other issues pertaining to authentication as well, but these will be examined briefly to highlight the complexity of authentication which must be performed before the OAuth server may issue a token.

Self-service Credential Management

As business-critical services grow in popularity, it is important that users are able to autonomously create accounts and reset passwords. Users should not have to contact the API provider in order to signup for an account unless that is an explicit requirement. Users who sign up for an

account will usually have to have their email address verified. They will also need to set a password, either before or after their email is verified. After account verification, users will often forget their usernames and passwords. For this reason, it is important to provide a way to recover these without contacting support.

In addition to low-assurance credentials like passwords, many APIs require a higher degree of assurance that the user calling that API really is who they claim to be. This is done using multiple factors or proofs of authentication, commonly referred to as Multi-factor Authentication (MFA) or 2-step Authentication (2FA). In order to support this, users must obtain additional proofs of their identity. How they obtain the first can be a challenging organizational and technical problem. Once users have one kind, however, bootstrapping additional ones can be done using similar self-service capabilities.

Account Linking

It is important to provide users with choices of how they wish to login. This ensures that users complete the login process rather than bounce. It also provides a higher level of security when MFA is used. Choices also allow users to reuse their previous authentication to achieve SSO under conditions that allow for it.

While choices and flexibility can lead to high-levels of security and UX, they also create an integration problem. Applications and APIs need a stable user identifier to associate user-specific application data with. This app data must be connected to an individual regardless of which form of authentication is used. In order to achieve this, disparate authentication methods must be linked together and mapped to an ID that an application can be assured to obtain in all cases. Only then will a user be able to continually work with a service and use the authentication method of their choice. An Authentication Service needs to provide account linking capabilities for these reasons.

SSO and Reduced Sign-on

Another important way of reducing falling off during login is to enable SSO and Reduced Sign-on (RSO). SSO refers to the situation where logging in once is enough for a user to be able to subsequently login without interactively providing a credential. RSO is similar except that it is meant to reduce the frequency for which a user must provide a credential. RSO strikes a balance between security and UX. For example, RSO can be achieved when a user initially logs in with a username/-password. Later, when making an API call that requires a higher level of assurance of the user's identity, the API client may indicate that additional factors are required. In such a case, the SSO is ignored, since it is for a credential that is too weak to satisfy the requirements of the transaction. Once the user logs in with the second factor, the SSO can be enriched with information about both credentials, so that it will subsequently work for both 1- and 2FA. In such cases, it is

common to limit the lifespan for which SSO should be allowed for each type of credential. For instance, the lifetime of the 2FA authentication may be restricted to one hour whereas a username/password authentication may be allowed for 6 months. With such a configuration, SSO would be achieved for either credential type for the first hour after 2FA was used. For the remaining time, SSO is only possible if a username/password credential is satisfactory. During the latter time period, RSO is achieved when low-levels of security are acceptable and stepping up from 1FA to 2FA is only necessary for higher-value transactions.

Different techniques can be deployed to achieve RSO (which is sometimes also called Step-Up Authentication). For example, when using OpenID Connect specification the API can request that the session conforms to a specific Authentication Context Class or an Authentication Method. Information on these are kept in the claims of an ID Token, as, respectively, ACR and AMR. RSO can also be achieved with cookies and keeping the relevant information in SSO session cookies.

Using a Specialized Authentication Service

Authentication is a complex task. Mistakes in this area can have devastating effects on the security of the API. SSO, account linking, and self-service are only a few of the complex requirements that are often needed in medium- and large-scale API platforms. Nowadays, authentication is a complex state machine, in which the user must fulfill different steps to prove their identity. For example two-factor authentication can be required depending on different properties - what is the role of the authenticating user, what country are they logging in from, or whether it is the same country they logged in from previously, what device are they logging in from, etc. As the level of sophistication grows, the chance for errors also increases. Because authentication is a prerequisite to token issuance in the OAuth server, it is very important that these complexities be handled by a dedicated application. This specialized login service must handle all of these difficult matters without exposing them to other components of the IMS, AMS, or consumers of the API platform. For instance, an Authentication Service should obscure integrations with various proprietary and specialized login APIs required for authentication (e.g., Kerberos, LDAP, RADIUS, etc.). Sending of emails and SMS challenges must also be hidden from other actors within the system, or the application of any geo-location filters, and so on. An OAuth server that is attempting to handle all of the complexities of authentication is doing too much. Token issuance is specialized enough that an OAuth server should be relegated to that task alone, leaving authentication to the Authentication Service.

Validation, the Sole Responsibility of the API Gateway

Authentication and token issuance are challenging tasks. Using an Authentication Service for the former and an OAuth server for the latter helps decompose these complex problems into manageable subtasks. By separating these responsibilities into specialized components, each service can do what it does best. Similarly, an API gateway should validate tokens issued by the OAuth

server to ensure that requests are authenticated and authorized. The API gateway is ideally positioned at the perimeter to perform this task. Because an API gateway stands in front of many back-end APIs, it can do so in a reusable manner. This is the duty that an API gateway should perform.

It's worth remembering that the API gateway shouldn't validate authorization against a complicated business logic. The API Gateway should do a coarse-grained validation of the incoming credentials. For example, check whether the incoming token is well-formed, if it hasn't expired, whether the cryptographic signature is valid if it was signed, etc. Any business validation, like whether the user can access the requested data, should be done at the API level.

Regardless the type, profile or kind of the incoming token, the API gateway should perform such coarse-grained validation. If by-value tokens are used, the API Gateway simply validates the received token. If by-reference tokens are used to access the API, the Gateway can use introspection capabilities of the OAuth Server. If both the API Gateway and the OAuth Server are deployed in the same data center, the Phantom Token Approach might be the chosen approach. In this approach, the API Gateway exchanges the by-ref token for a by-value token on each request (caching results as instructed by the OAuth Server) and performs validation on the received token. If your API Gateway is spread among different data centers, especially if it deployed differently than the OAuth Server, the Split Token Approach might be more appropriate. In this approach the Gateway splits the token into the payload and signature parts. It hashes the signature and uses it as a key to cache the payload. The signature part is used by the client as the actual token. Upon receiving the token, the API gateway hashes it and glues together with the part kept in cache. Thanks to this, the API gateway can now validate the whole by-value token, without the need of contacting the OAuth Server on every request.

In either case, the API Gateway eventually forwards a by-value token to the APIs. Because these by-value tokens are self-contained, the internal APIs need not contact the OAuth server for validation. Instead, they can validate tokens using public key cryptography. In such a deployment, the public key that the internal APIs use is that of the OAuth server. This results in a loosely coupled system where trust is concentrated in the IMS and not at the perimeter.

Many API Gateway Products Make Token Issuance Difficult

Using an API gateway solely for token validation is important even when using a commercial gateway product that has some OAuth functionality. If different instances of the product are used as the STS and another as the gateway, the concerns of token issuance and validation are at least separated. This is not common practice, however, due to licensing restrictions of many API gateway products that make this prohibitively expensive. In those cases where licensing is not an

issue, the approach that most gateway products take for defining policies or virtual services makes them ill suited for the task of token issuance. Most gateway products provide administrators with a graphical method to configure their routes. These are often referred to as policies, virtual services, routing procedures, or rulesets. These graphical user interfaces allow administrators to create virtual APIs. This approach can work well for token validation, especially when using the approach previously described. For token issuance, however, it is not a good tactic. The reason is because the atomics or basis by which these policies are defined is too low-level. They require administrators to understand very fundamental aspects of HTTP, cryptography, OAuth, OpenID Connect, and JWT in order to authenticate users and issue tokens. These technologies that are difficult to understand and this graphical approach lead to unmaintainable configuration that is challenging to update and evolve. While many gateway products include "ready made" policies for the basic flows of OAuth, changing them essentially means reconfiguring the policies using the graphical policy editor.

This leads to the same problem of maintainability. It also requires an administrator to not only gain a deep understanding of complex security and Internet protocols, but also to understand the product-specific policy language in great detail.

For these reasons and others, API gateway products should be used solely for token validation and enforcement of authentication. Token issuance and authentication should be left to specialized applications designed to make those tasks easier.

Conclusion

Building an API platform is a large undertaking. To successfully launch and evolve such a system, it must be broken down into manageable parts. All API platforms include an Identity Management System and an API Management System, each containing various components in turn. These parts should concentrate on a single concern. For this reason, an API platform will include not only an API gateway but also an STS that issues, exchanges and validates tokens according to the OAuth 2 protocol as well as an Authentication Service. These subsy tems and their components should be conjoined using standard protocols. This separation of duties and standardized integration leads to a united whole that is evolvable over time. It also gives organizations choices on how and when to implement the various components.

Curity has been recommending this approach to its customers since inception. We have helped architect and deploy numerous API security platforms that follow this guidance in various industries and geographies. During this time, we have worked with customers that have had API gateway products from all of the big vendors as well as open source offerings. Despite this, almost all our customers have adopted this approach to API security, and have separated token validation from issuance and authentication. This approach to API security is so common to us that we refer to it as the Neo-security Architecture.

Also over time, we have found that existing products in the market were unable to fulfill our customers' needs for authentication and login. As our customers' token issuance demands increased, existing OAuth and OpenID Connect implementations were also found wanting. For these reasons, years after we began proscribing this approach and helping customer to follow it, we invented and established an identity server product that meets our customers' needs for authentication and token issuance. This product is called the Curity Identity Server, and it fulfills the roles of an Authentication Service and/or an STS. Each can be used independently, allowing it to fulfill a single function or both. It has been deployed in production at numerous customers' facilities and complies with the relevant standards.

To learn more about the Curity Identity Server and how Curity can help you secure your API, please contact us at info@curity.io.

For Further Information

If you have questions about anything written in this paper or would like to learn more about how to apply them to your situation, contact us by email at info@curity.io. You can also connect with us on Twitter where we are @curityio.

More information can also be found on our website https://curity.io.

CURITY

Address
Curity AB
S:t Göransgatan 66
112 33 Stockholm
Sweden

Website
curity.io

Email
info@curity.io

Twitter
@curityio