A single technical pattern for API access control is not sufficient for all types of APIs that organizations must secure. To be successful, security and risk management technical professionals must establish five guiding principles and an identity fabric that’s optimized for API access control.

Overview

Key Findings

- API access control — that is, the authentication and authorization of APIs — is a vital part of API security.

- Organizations are fruitlessly looking for a single “one and done” pattern that can provide encryption, authentication and authorization to a widely divergent variety of API use cases.

- Organizations that are now starting to become successful in their strategies are breaking down API access control into smaller and more use-case-appropriate patterns that are easier to define, promote, enforce and, ultimately, govern.

- Leading organizations are establishing an agile and composable identity fabric for API access control that consists of multiple identity tools and components that enable encryption, authentication and authorization. They are also following five guiding principles that, together, set organizational standards.

Recommendations

Security and risk management (SRM) technical professionals responsible for delivering effective identity and access management (IAM) and API access control capabilities should:
“Divide and conquer” a large and complex API access control problem by breaking it down into smaller, solvable pieces. Establish strict separation between the external applications that access the APIs and the internal services in and behind your enterprise API gateway(s).

Establish an identity fabric for API access control and tailor it for your use cases using full-featured authorization servers, externalized authorization managers and secrets managers that work in concert with mediators such as API gateways and, if necessary, other in-line proxies such as sidecars.

Implement defense in depth, generate optimized access tokens by exchanging tokens and do fine-grained validation in each service. Enable the defense-in-depth strategy to scale by providing developer self-service interfaces.

Be pragmatic and set reasonable expectations in the organization, as even cloud-native and API-first organizations are struggling to get this right. Organizations that have established tooling and strategies for authentication and encryption first are now shifting their focus to more challenging fine-grained authorization where possible.

Improve developer experience by providing implementation guidance, proven libraries and integration strategies for a hybrid and multicloud environment.

Analysis

It’s no longer enough to look for or establish a single technical “one and done” API access control pattern that defines how organizations can secure one API and one single application. Instead, organizations must define strategies that can scale to handle thousands of unique APIs. The strategies are important, as API access control is foundational for the digital transformation. Strategies must also span and support multicloud and hybrid environments and strict guidance has to be put in place.

By 2019, 83% of HTTP traffic already was API requests and only 17% used traditional web applications.¹ That means that the core use case for access control and access management is no longer traditional browser-based access and its associated cookie-based session management. Gartner has seen the number of inquiries regarding API access control increase 300% in just the last year.
In a modern API access control strategy, modern identity protocols are essential and JSON Web Tokens (JWTs) have become the de facto credential used to access APIs.

Modern identity protocols are designed to ensure that security, usability, privacy and scale requirements can be met. JWTs are an example of a modern identity standard that enables those important traits when protecting APIs. That said, Gartner has also observed that any identity standard takes 10 years to be commonly implemented. Once commonly implemented, it then takes 20 years of hard work to replace it with a new and even more modern identity standard. The industry has seen that with passwords, security assertion markup language (SAML), OpenID Connect, System for Cross-Domain Identity Management (SCIM) and now also with the use of JWTs.

This all means that any organizational API access control strategy that is intended to support thousands of APIs must also include legacy applications and protocols in its scope.

One single technical authentication and authorization pattern does not exist for all APIs. Instead, a modern, multifaceted and comprehensive API access control strategy is needed to succeed.

Organizations are now forming API platform teams to create a community of practice for APIs and implement API best practices. That team must involve practitioners such as developers, DevOps teams, cloud, security and IAM to help establish the right guardrails and API access control guidelines.

An API access control strategy must assess challenges across five specific dimensions and then provide five guiding principles to address those challenges. This document will assess those five dimensions and lay out the five guiding principles that form a strategy, as depicted in Figure 1.
The five dimensions of API access control are:

1. Identity functions (such as authentication, authorization, encryption)
2. Applications (apps) and their users
3. Mediators
4. Service maturity
5. Developer enablement

The five guiding principles that help overcome the challenges are the practice of:

1. Divide and conquer.
2. Continuously evolve the identity fabric for API access control.
3. Strike a balance between developer enablement and governance.
4. Use token exchange mechanisms.

5. Implement defense in depth.

These five principles must also function in unison to be effective.

There are no shortcuts to a scalable strategy. A modern API access control strategy requires organizations to understand the requirements of each API use case, make informed risk-based decisions and provide the right guidance. Even though there are no shortcuts, organizations that are looking for quick answers can see the “Guidance” section below for a list of commonly asked questions and their answers.

This document is tailored for mature organizations that are now refining their API access control strategies to ensure that security, usability, privacy and scaling requirements are met. An introduction to modern identity protocols used is provided in Modern Identity and APIs: OpenID Connect, OAuth 2.0 and SCIM 2.0.

API Access Control — an Increasingly Important Part of API Security

Identity, even though it is at times just an afterthought in a larger API strategy, is a crucial component to getting API security right. Historically, API gateway providers and API threat protection vendors have defined the API security market and, to be frank, they have often neglected, or at least underestimated, organizations’ identity and API access control needs. Four out of the top five security risks described in the OWASP API security top 10 list are now identity related.

Organizations must therefore start to redefine API security and split it into three areas, as depicted in Figure 2. Together, the areas of API discovery, API threat protection and API access control (covered in this document) form an API security strategy. They are fundamental to one another, but the tooling, architecture and requirements are different.
Gartner has published extensively on how to form an API threat protection strategy and much of this research is laid out in Solution Path for Forming an API Security Strategy. Guidance on API discovery — an important prerequisite for large organizations — can be found in How to Implement API Discovery and Detection to Improve API Management and Security.

For API access control and API threat protection specifically, the “10,000-foot view” of API security is architecturally depicted in Figure 3.
Figure 3: A 10,000-Foot View on API Security

Anti-distributed denial of service (DDoS), bot mitigation, web application firewalls (WAFs) that are provided individually or as a web application and API protection (WAAP) service, and API gateways are all in line and help protect requests. However, identity requirements have to be more finely defined than is depicted above.

Organizations that are now successful in their API access control strategies define IAM as a function that’s part of a more agile and composable identity fabric — an identity fabric that supports API access control. Its definition is based on API use cases and the organization’s current and future needs. Understanding those defining API use cases requires analysis of them using all five dimensions of API access control, which are defined below.

The 5 Dimensions of API Access Control Challenges

Organizations often have a very large number of APIs and applications with different needs and capabilities. Gartner receives many questions on API access control, such as “Should we use the authorization code flow with proof key for code exchange (PKCE) for everything or not?” This is a symptom of organizations looking for that “one and done” pattern. But the security capabilities of a single-page application are very different from a local service that accesses another service. A SOAP-based API accessing data on a mainframe has very different identity capabilities from a cloud-native microservice running in a service mesh. These are examples of different types of API use cases and they must be treated differently.
A modern API access control strategy is based on an assessment of an organization's use cases. To ensure that proper risk assessments and classifications are made, SRM technical professionals should use the five dimensions laid out in Figure 4. The result will define the organization's API access control requirements.

**Figure 4: The Five Dimensions of API Access Control Challenges**

Let's look at the five dimensions in more detail.

**Dimension No. 1 — Identity Functions**

**Definition:** Identity functions for API use cases are authentication, authorization and/or encryption. All three are usually required but need different tooling and management strategies.
**Key design decisions:** What underlying identity functions are required for the use case? Authentication, authorization and/or encryption? Authenticating a request is easier than authorizing and defining the policies for how to validate the request. It’s also different for how encryption is handled. The document *Modern Identity and APIs: OpenID Connect, OAuth 2.0 and SCIM 2.0* provides further insights into the deployment and use of modern identity protocols in general. But, as stated above, JWTs are very commonly used as a way to authenticate the request. Authorizing requests that use JWTs comes down to managing and pushing policies around that are used to assess what claims to look at in those same JWTs. Encryption, and also in some cases also mutual authentication (mTLS), is done with the help of certificates, which bring questions on how issuance and life cycle management must be done to ensure uptime and appropriate trust.

**Insights gained from the dimension:** The identity fabric, meaning the tools that need to be orchestrated together to solve your identity needs, varies depending on what identity functions are actually in scope for your organization. Encryption using server-side certificates and authentication using JWTs are commonly implemented, but most organizations do not yet have centralized authorization and policy management. That said, organizations often expect there to be out-of-the-box blueprints for how to scale authorization to protect thousands of APIs, and legacy apps without built-in support for modern authorization. Organizations must determine which identity functions are required and in scope when designing their identity fabric and setting organizational expectations.

**Dimension No. 2 — Apps and Their Users**

**Definition:** Mobile native apps, internal or external services, web applications and devices are all examples of applications (apps) that access protected APIs. The apps are typically operated by users such as B2E and B2C users, or machine identities such as service accounts.

**Key design decisions:** Different apps have different capabilities for how they can authenticate themselves and the human users piloting them. In what user directory are the users in scope situated? An app that is acting as a service, and is managed by a known and trusted partner, has a different security composition from a consumer-facing mobile app. Again, they are different and they need to be treated differently. Organizations must assess what users need access to your APIs. Are machines, partners, consumers, employees or all of the above in scope? Where are the users stored and what identity capabilities can be leveraged for each user constituency. What type of application is currently being used?
Insights gained from the dimension: Understanding this dimension helps define what flows, libraries and integrations with third-party identity providers are needed for an API and its calling applications.

For further insights into current best practices for each type of app, see Building Authentication, Authorization and SSO Into API-Driven Apps.

Dimension No. 3 — Mediators
Definitions: APIs are protected by mediators such as enterprise or internal API gateways, as well as more nimble mediators such as sidecars deployed close to the APIs.

Key design decisions: Where will the APIs be deployed and hosted, and what API gateways are mediating the traffic? Clients report that they often leverage multiple API gateways. Are the APIs deployed in a public cloud? If so, then gateways such as Microsoft Azure API Management or Amazon API Gateway are typically used, and it’s important to see how the gateways can be integrated with your existing identities and your policies. If it’s an internal, behind-the-firewall type of service and mediation is done close to the service, then it’s about understanding the identity capabilities of any microgateways. What identity functionality does any sidecar that’s being deployed in line of the request support? As a side note, Gartner has further guidance on how to pick the right mediators (see Decision Point for Mediating API and Microservices Communication). That said, one thing is clear: a hybrid and multicloud environment and any defense-in-depth strategy requires the use of multiple mediators — mediators that need to be assessed for how they can be integrated.

Insights gained from the dimension: Assessing the mediators’ identity capabilities ensures that the strategy includes the right integrations and enforcement in the mediators. It also enables organizations to assess the access management tools support for the mediators used. Typically, the access management vendors provide the integrations, instead of the API gateways. That said, some API gateway vendors now provide documentation and insights on how to integrate third-party authorization servers.

Dimension No. 4 — Service Maturity
Definition: Data exchange protocols and architectural patterns such as REST, SOAP and GraphQL have different levels of service maturity. They have different capabilities and built-in support for modern identity protocols and therefore require different tooling.
**Key design decisions:** How are the APIs architectured and implemented? Are they using new or legacy architectural patterns? Is it possible to go back and reimplement the identity capabilities or are new mediators or credential vaults required to help broker access to a legacy system? It’s important to categorize and classify the internal APIs and services into multiple patterns. An example of a classification can be published versus unpublished APIs. The document *How to Implement API Discovery and Detection to Improve API Management and Security*, and specifically Figure 5 in that document, lays out an API categorization matrix that’s helpful in this assessment.

Each architectural pattern has its own requirements. For example, the requirements of a GraphQL API, a REST-based API, a SOAP-based API, a third-party API and an event-based API all differ.

**Insights gained from the dimension:** Understanding the service maturity dimension enables you to categorize each API and define what identity components are required in your identity fabric. It’s also essential for prioritizations and to help set expectations on what patterns are in scope of any API access control strategy journey. For example, maybe it’s more important for your organization to get identity right for a new, publicly published, externally used, cloud-native service than for an on-premises legacy system. Or vice versa, for that matter.

**Dimension No. 5 — Developer Enablement**

**Definition:** Developer enablement ensures that developers can publish and control their own APIs and control what attributes they need in their APIs. At the same time, the control must be based on a delegated model in which a central team still controls the overarching policies.

**Key design decisions:** Organizations are creating a community of practice for APIs — an API platform team — that promotes API best practices. For reference, the Gartner document (which might require a different Gartner subscription) *Federate, Rebrand and Recharter Your API Center of Excellence to Enable an API Platform Team* defines the API platform team in detail. The main take-away from this document is that the team must provide guidance that enables developers to control their own fate. Without it, it’s not possible to scale your API access control strategy up to thousands of APIs. API access control is about being able to balance the decentralization of administration with centralized governance and control. Ask yourself when and how should developers control the access, their policies and the required identity information they need to enable the API services?
Insights gained from the dimension: Understanding the developer enablement dimension ensures that the strategy includes the right tooling support, such as developer self-service interfaces. It entails understanding the required strategies for how to scale authorization policy management and how to categorize and communicate risk. It also helps in defining the best practices and what organizational structure is required to enforce it. Access management vendors provide self-service interfaces, but Gartner clients report that, due to limited cross-tooling vendor support, they have also been forced to develop their own “single pane of glass” across multiple services.

Note: Only when all five dimensions are assessed and understood, and a proper risk assessment and classification has been made, is it possible to start implementing your guidance.

The 5 Guiding Principles to a Successful API Access Control Strategy

As with any guiding principles, the five principles for API access control are intended to influence the organization, define expected behavior and aid in decision making. Use the principles to continually evolve and improve your API access control strategy, and use them to refine your definition of what good looks like.

Some guiding principles are very technical; others less so. The principles need to be combined, since it's only together that they form a fully functional strategy.

Let’s look at the five principles in detail.

1. Divide and conquer.

2. Continuously evolve the identity fabric for API access control.

3. Strike a balance between developer enablement and governance.

4. Use token exchange mechanisms.

5. Implement defense in depth.

Guiding Principle No. 1 — Divide and Conquer

In computer science, applying a divide and conquer algorithm is the artform of recursively breaking down something large and complex into smaller solvable pieces.
First and foremost, it’s critical to divide the external architecture and the internal architecture.

- The external architecture comprises the applications and services that need to prove to the system who they are and who they are acting on behalf of. The applications typically need to acquire tokens, store tokens and refresh tokens that are later used to access the APIs in a secure way.

- The internal architecture comprises the APIs and the mediators that have to authenticate and authorize the incoming requests and ultimately validate the tokens sent from the external architecture.

The divide and conquer pattern and the external and internal architecture are depicted in Figure 5.

**Figure 5: Divide and Conquer**

*Divide and Conquer*

**External Architecture**
- Web Server
- Mobile Native App
- Single-Page App (with back end/same domain)
- Single-Page App (without back end/cross domain)
- Internal Service
- External Service

**Internal Architecture**
- In Cloud B
- In Cloud A
- On-Premises

Get Token → The Identity Fabric for API Access Control

Use Token → API GW → API A

Legacy

Microservices

DB

API B

Source: Gartner 723547 C
The external and internal architecture are related, but their differences mean guidance is required for each.

The External Architecture

By recursively applying the divide and conquer pattern, organizations can then explicitly look at the recommendations that are required for each type of external application. For example, a single-page web application without any back end has a very different security composition from an internal service. By dividing the applications up, organizations can provide app-specific guidance on how to implement them and still ensure that all security, usability, privacy and scalability requirements are met.

Table 1 provides a snapshot of the current recommendations on how modern identity protocols — OAuth 2.0 and OpenID Connect (OIDC) — should be integrated for each application type and the guidance that must be defined.
Table 1: Grant Types and Their Use Cases
(Enlarged table in Appendix)

<table>
<thead>
<tr>
<th>Grant type</th>
<th>Protocol</th>
<th>Web Servers</th>
<th>Native Apps</th>
<th>SPA (same domain)</th>
<th>SPA (cross domain)</th>
<th>Services</th>
</tr>
</thead>
<tbody>
<tr>
<td>Resource Owners</td>
<td>OAuth 2.0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Password Credentials</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Client Credentials</td>
<td>OAuth 2.0</td>
<td></td>
<td></td>
<td>X</td>
<td></td>
<td>X</td>
</tr>
<tr>
<td>Authorization Code</td>
<td>OAuth 2.0 and OIDC</td>
<td></td>
<td></td>
<td>X</td>
<td></td>
<td>X</td>
</tr>
<tr>
<td>Authorization Code with PKCE</td>
<td>OAuth 2.0 and OIDC</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>X</td>
</tr>
<tr>
<td>Implicit</td>
<td>OAuth 2.0 and OIDC</td>
<td></td>
<td></td>
<td></td>
<td>X (legacy)</td>
<td></td>
</tr>
<tr>
<td>Hybrid</td>
<td>OIDC</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>X</td>
</tr>
<tr>
<td>Device flow</td>
<td>OAuth 2.0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>X</td>
</tr>
<tr>
<td>JWT and SAML grant (RFC7523)</td>
<td>OAuth 2.0</td>
<td></td>
<td></td>
<td>X</td>
<td></td>
<td>X</td>
</tr>
<tr>
<td>OAUTH MTLS (RFC8705)</td>
<td>OAuth 2.0</td>
<td></td>
<td></td>
<td>X</td>
<td></td>
<td>X</td>
</tr>
<tr>
<td>Token Exchange (RFC8693)</td>
<td>OAuth 2.0</td>
<td></td>
<td></td>
<td>X</td>
<td></td>
<td>X</td>
</tr>
</tbody>
</table>

Source: Gartner

The industry recommendations for how to build identity into applications are starting to mature. Good guidance will be detailed. It will define the preferred flows for each application type and its risk level, and include recommendations of development libraries. Access management vendors commonly provide libraries that are continually updated to meet evolving requirements. Testing the libraries, ensuring they meet the security requirements and usability requirements, should be part of your API access control strategy.

Ask the vendor questions such as:
Understanding the external architecture also enables organizations to assess the privacy requirements and what identity information can and should be conveyed through the external applications. Sending personally identifiable information (PII) to an internal and highly trusted service might be reasonable, but it might not be appropriate for a mobile app. This has implications for what types of tokens can and should be used in the external architecture.

Categorize the different types of applications and provide development guidance for each type of application. Providing this detailed guidance on what libraries to use and how flows can be supported for each application type enables developers to move fast.

**The Internal Architecture**

Guidance and tooling support for the internal architecture require additional analysis due to the wide set of APIs that organizations must secure. APIs can be deployed in multiple environments, both on-premises and in multiple clouds, and the different architectural patterns that are used define how API access control can be implemented and enforced. In addition, APIs have different risk profiles depending on whether they are published and/or externally used, or not. See Best Practices for a Successful Security Risk Assessment for further details on risk assessments.

Understanding the dimension of service maturity helps organizations to classify and divide the internal architecture into patterns based on both their architecture and risk assessments. The patterns then make it possible to start providing more tailored guidance for the use cases.

Organizations then need to orchestrate the right components to solve each pattern. This is where a tailored identity fabric is important for API access control (see detailed below).
Table 2 provides examples of commonly used architectures and development patterns that require different handling. In the interest of keeping this document brief, not all patterns are elaborated on in detail further down.

Table 2: Services and Their Commonly Used Authentication and Authorization Architecture
(Enlarged table in Appendix)

<table>
<thead>
<tr>
<th>Authentication</th>
<th>Authorization and Example Policy Management</th>
<th>API Access Control Points</th>
</tr>
</thead>
<tbody>
<tr>
<td>Microservices and Service Mesh</td>
<td>JWTs</td>
<td>JWT claims and policy as code in container platforms</td>
</tr>
<tr>
<td>Serverless</td>
<td>IPaaS provided credentials (commonly JWTs)</td>
<td>IPaaS platform with governance through CIEM and/or federation and integration of existing identities</td>
</tr>
<tr>
<td>REST-based APIs</td>
<td>JWTs</td>
<td>JWT claims, local authorization or EAMS</td>
</tr>
<tr>
<td>SOAP-based APIs</td>
<td>Passwords or X.509 certificates</td>
<td>Seldom implemented</td>
</tr>
<tr>
<td>GraphQL</td>
<td>JWTs</td>
<td>JWT claims</td>
</tr>
<tr>
<td>Events</td>
<td>Passwords or X.509 certificates</td>
<td>Static policies in event brokers</td>
</tr>
<tr>
<td>APIs in COTS and SaaS products</td>
<td>API keys, X.509 certificates, client credentials and headers</td>
<td>Using JWTs, IGA or through secrets managers and injection</td>
</tr>
</tbody>
</table>

Source: Gartner

Microservices and Service Mesh

Microservices are commonly deployed in a container orchestration platform or in an application platform as a service (aPaaS). The service mesh architecture is gaining traction and also provides many best practices for service discovery, mutual TLS (mTLS) for interservice security and traffic management. Depending on where the services are deployed, this pattern might have to be broken up into multiple patterns that use different components in the identity fabric. Authentication, authorization and encryption are typically handled using the following control points and mechanisms:
The microservices and service mesh pattern leverages multiple control points and requires multiple integrations, as discussed above. Assess the capabilities of each platform and extend the identity fabric for this use case accordingly. The Gartner document *Building Identity Into a Microservices Architecture* provides further insights into this specific pattern.

**Serverless**

- **Authentication.** Uses mTLS for interservice security (east-west communication) and JWTs sent in the requests for north-south communication. JWTs are typically first validated in an enterprise API gateway and also closer to the services using a sidecar (such as Envoy). Microservices-based applications often require the use of a secrets manager tool to centrally control access and provide secrets used to authenticate to third-party services such as databases.

- **Authorization.** Authorization is enforced in multiple places: in the access management tool (in the authorization server when issuing the JWTs); in the enterprise API gateway, where coarse-grained authorization is made; and in each service, where fine-grained authorization is made. The authorization policies are controlled through the service mesh and its control plane. This is where technologies such as open policy agent (OPA) and third-party externalized authorization manager-provided mediators push down policies (using policy as code) close to the services. Access to third-party services is also authorized in the above-mentioned secret managers.

- **Encryption.** Encryption through transport layer security is done using server-side TLS and certificates issued by certificate authorities (CAs). Certificates in an enterprise API gateway (and in any TLS terminating gateway in front of the services) require certificate management practices to be in place. Transport layer security is now a requirement in each service due to organizations’ zero-trust and defense-in-depth strategies. Again, secrets management tools are typically orchestrated in concert with private CAs within clusters, external CAs and certificate management tooling to help manage the life cycle of the certificates, which ensures transport layer encryption. For further details, see *Managing Machine Identities, Secrets, Keys and Certificates*. 

The microservices and service mesh pattern leverages multiple control points and requires multiple integrations, as discussed above. Assess the capabilities of each platform and extend the identity fabric for this use case accordingly. The Gartner document *Building Identity Into a Microservices Architecture* provides further insights into this specific pattern.
Cloud-native serverless deployment of APIs in a function platform as a service (fPaaS) is dependent on the IAM capabilities of that fPaaS. The fPaaS is in charge of providing all functions needed. That said, it's not uncommon that organizations want to integrate existing authentication and authorization capabilities for the use case. For example, a serverless Lambda function in Amazon Web Services (AWS) uses Lambda authorizers in Amazon API Gateway to do coarse-grained validation at the edge and then resource-based policies to protect access to the service. Access to other AWS services is controlled through the concept of execution roles, which are applied for each Lambda function. Integration with an organization’s existing (outside of AWS) identity capabilities must be done in the external-facing Amazon API Gateway. To integrate third-party identities, the Lambda authorizers must be used to validate third-party tokens and internal access is brokered in the gateway.

Architecting authentication and authorization for this pattern requires insights into where the service is running and what cloud-native integrations are available. Look for out-of-the-box integrations from the access management tooling into the fPaaS to help apply API access control policies to serverless APIs. For further insights, see Decision Point for Selecting Virtualized Compute: VMs, Containers or Serverless.

REST-Based APIs

As in all above patterns, an enterprise API gateway typically protects REST-based APIs. REST-based APIs are commonly secured using either basic authentication (meaning unencrypted username and passwords), JWTs or mTLS. There’s a wide set of development libraries that enable a defense-in-depth strategy to enable authentication and authorization using things such as JWT validation. However, it’s preferred to externalize the policy creation and management, instead of hardcoding it in the source code. Do so by adding mediators (such as microgateways provided by API gateway vendors or access management vendors) or libraries that enable the usage of an externalized authorization manager. Although not yet broadly adopted, authorization policies can be pushed down to the services from a central externalized authorization manager using technologies such as OPA.

Define this pattern based on application server architectures and provide programming-language-specific guidance. For further details, see A Guidance Framework for Creating Usable REST API Specifications.

SOAP-Based APIs
SOAP-based APIs typically have strict dependencies on the existing systems that call them. Gartner clients often report that SOAP-based APIs are hard to replace and change. SOAP-based APIs often use raw username and passwords or, more preferably, X.509 certificates that enable message security. Support for JWTs is nonexistent, at least outside of closed ecosystems, where one service provides both client libraries and the server-side APIs. There are WS-Federation standards that enable the use of modern identity protocols, but these are not commonly deployed and cross-domain support is very limited. SOAP-based APIs therefore often need to be wrapped in facades using REST-based APIs, where more modern API access control capabilities are enforced.

**GraphQL**

The pure nature of GraphQL — a client-defined interface working as a frontend for a backend — makes any sentences that include both access control and GraphQL sound contradictory. Traditionally, back-end systems have defined all interfaces and their associated access policies. A GraphQL API can instead provide a front-end API for multiple back-end services by leveraging GraphQL servers. This often results in the use of overentitled tokens that can access more back-end services and data than they should. Gartner clients often ask where authentication and authorization should be done in an enterprise API gateway — in the GraphQL server or in the back-end system.

Treating GraphQL as a separate pattern enables organizations to assess the functionality in GraphQL servers, or in the API gateways that increasingly extend their support for GraphQL as well. As API access control support is limited (and often an afterthought) in GraphQL implementations, it's therefore typical that coarse-grained authorization is done at an external-facing enterprise API gateway and fine-grained authorization is done in each back-end API to ensure that the correct APIs are called. See Assessing GraphQL for Modern API Delivery for further details about GraphQL.

**Event-Driven and Streaming APIs**

Event-based (publish-subscribe) protocols such as MQTT, AMQP, RSocket and WebSockets are popular patterns but, as with the adoption of GraphQL identity, are often an afterthought. The challenge with the architectural event pattern is that it's only the services that are authenticated. Authorization defined in servers (for example, the MQTT brokers) enable organizations to lock down what clients can read and write from specific topics, but no message-level authentication and authorization is done.
Organizations often reach out to Gartner looking for ways to pass JWTs in the same requests as the payloads. Modern event-driven protocols, such as gRPC, actually define a channel credential and support call credentials that provide message-level authentication. But for other protocols, it’s more challenging. Protocols such as MQTT and AMQP provide no standards-based approaches for how to authenticate each call. It’s also generally a bad idea to just take an external JWT and bundle it in the payload, since it’s impossible to know when all message subscribers have read and consumed, and therefore validated, a token. This pattern requires extra care and attention. Gartner has observed that clients fall back to REST-based APIs for use cases that require higher assurance levels and only send data over evented or streamed APIs for low-assurance scenarios. Alternatively, payloads and token formats can be specifically defined, but the format must then be understood by all procedures and consumers involved in the interactions. See Decision Point for API and Service Implementation Architecture for further details on events.

Third-Party APIs

As organizations consume third-party APIs in commercial-off-the-shelf (COTS) products, SaaS products or within partner environments, it’s important to establish mechanisms to discover them and ensure their security and health. These are often business-critical APIs, but API access control strategies have to be adopted to the existing APIs more than the other way around.

Third-party APIs require the use of a wide set of IAM tools to ensure authentication and authorization. Identity governance and administration (IGA), access management support for identity federation, certificate management tools, secrets management support for vaulting, and adding access control to legacy credentials are all tools in play here. Assess and categorize the third-party APIs and tailor the identity fabric accordingly. See How to Evaluate SaaS Providers and Solutions by Developing RFP Criteria for insights into how to assess SaaS services identity capabilities.

As APIs and services are commonly deployed in different environments, it’s also important to define each environment as its own pattern. For example, if services are deployed in Google Cloud Platform (GCP), then the GCP API gateway typically needs to be integrated. If it’s in AWS, then it’s the Amazon API Gateway that typically needs to be assessed.

Guiding Principle No. 2 — Continuously Evolve the Identity Fabric for API Access Control

The five dimensions and the divide and conquer practice help to decompose the identity requirements into smaller and more nimble pieces. But now it’s time to assemble the pieces.
The architecture has to be built up as a continually evolving and composable identity fabric that’s optimized for API access control.

An identity fabric is a foundational layer of an emerging architecture called cybersecurity mesh. A cybersecurity mesh is a composable and scalable approach to extending security controls to distributed assets by decoupling policy enforcement from the assets being protected. In Client Question Video: How Can We Architect Our IAM to Be More Adaptive?, Gartner defines the identity fabric and the cybersecurity mesh further. For this document, and for the above-defined APIs and patterns, the identity fabric means orchestrating the right identity components and tools to solve the different API use cases. Some of the more commonly used tools are as follows:

- **Access management (AM) tools**: AM tools have deep support for users, applications and policies, and provide a wide set of integrations to protect both new and old applications. AM tools include federation services that bridge different identity siloes and authorization servers that provide standards-based token endpoints to create, exchange and validate tokens, integrate with multiple directories and databases, and also enforce policies. AM tools include their own policy management to various degrees, but it’s also possible to integrate them with an externalized authorization manager. AM tools are foundational for API access control strategies due to that deep support for evolving standards and integrations with existing user directories and other entitlement repositories. It’s therefore important to assess integrations between API gateways and third-party AM tools. AM tools that are commonly used in API access control scenarios are provided by vendors such as Microsoft (with Azure AD), Curity, ForgeRock, Keycloak (Red Hat Single Sign-On), Okta (including the recently acquired Auth0) and Ping Identity.
Directories: Organizations have multiple user directories and other repositories where existing users, roles and entitlements are stored. These are existing sources of information that need to be integrated and leveraged in a modern API strategy. The information in directories is used for policy decisions, but it is also the source of the claim values passed in tokens to the APIs. Commonly used and integrated directories include Microsoft's Active Directory, ForgeRock's Directory Services, Ping Identity's PingDirectory, cloud directories that are part of offerings such as Azure AD and Okta, virtual directories from vendors such as Oracle, Radiant Logic and Optimal IdM, and a wide set of databases. Gartner clients often report the importance of integrating and leveraging this existing information. Even though front-facing APIs often are new and modern in the backend, they might still call legacy systems that rely on these directories. By sending optimized tokens with claim values from the existing directories, organizations don't have to rearchitect all of their internal services using newfangled concepts such as OAuth 2.0 scopes. Instead, they can rely on the existing mappings. This is often a shift in mindset for many organizations, since OAuth 2.0 scopes are often pushed by tools. The third guiding principle listed below, and also the Guidance section, provide further insights into how to handle attributes, claims and scopes.

Externalized authorization managers (EAM): Central policy creation, management, versioning and integration with directories enables organizations to provide a “single pane of glass” across access. Historically, this capability has often been misunderstood as a panacea, but not all products and scenarios have had support for it out of the box. Look for integrations with enterprise API gateways, access management tools and any mediators used. EAM tools are provided by vendors such as Axiomatics, Ping Identity, PlainID and Styra.

Secrets managers: A secrets manager is a central building piece in modern API development practices, but it’s also a way to integrate with third parties and legacy applications. Secrets management tools are offered by IaaS providers, privileged access management (PAM) tools and by specialized secrets manager vendors. Secrets management is actually a tool that DevOps teams and security teams can come together and agree on. It enables organizations to store and control access to secrets such as keys, OAuth 2.0 credentials, certificates and connection strings. Secrets managers often mentioned by clients include Akeyless, AWS Secrets Manager, Azure Key Vault, BeyondTrust DevOps Secrets Safe, CyberArk Secrets Manager/CyberArk Conjur, GCP Secret Manager and HashiCorp Vault.
Mediators: Enterprise API gateways, microgateways and sidecars broker access to APIs. Some vendors also provide identity capabilities, but they are often subpar in comparison to the deep support provided by AM tools. Given that mediators are typically the only building block that’s actually in line when requests pass from applications to APIs, mediators are a center of gravity for integrations. Given that API gateways are typically not picked by IAM teams, it’s therefore extra important to validate that the AM tools that are used have integrations to the chosen mediators. Commonly used mediators are provided by Google (Apigee), MuleSoft, Microsoft, Kong, Software AG, IBM, Axway, and by the open-source Envoy proxy. Some access management vendors also provide lightweight mediators optimized for API access control.

Also part of the composable identity fabric for API access control are IGA products (see Guidance for Successful Identity Governance and Administration Deployments), public key infrastructure (see Solution Comparison for PKI and Certificate Management Tools), certificate management tools and cloud infrastructure entitlement management tools (see Managing Privileged Access in Cloud Infrastructure). These should be included in your architecture for the API use cases that require them.

As stated above, the tools need to be orchestrated to address your API needs. Figure 6 shows an example of how integrations and tooling are used in a sample identity fabric.
The integrations shown in the figure from left to right are explained as follows:

1. An application authenticates itself, and in some cases an end user is also authenticated, so that the application can request a token.

2. The token is used to access an API.

3. One or more mediators intercepts the communication and validates the request — either using preestablished keys that can be used to validate a self-contained token, or by sending the token off to be validated.

4. The mediator exchanges the external token with an optimized internal token that includes just the right claims for the specific API. (See the token exchange section below for details.)
5. As the token endpoint creates internal tokens, it can also reach out to an EAM to ensure that the calling application, the user and the mediator are authorized to access the internal API.

6. A mediator or API can then also reach out to a secrets manager to acquire additional secrets used to access internal services.

7. The mediator proxies the request and passes the newly created internal token to the API.

8. The API, with the help of adjacent mediators such as sidecar proxies, can then do fine-grained validation of the internal token. Policy can be preestablished (pushed down by EAM tools or as policy as code) and validation is then done locally, or the mediator can reach out to third parties for validation.

9. The internal API might also be in the business of calling third-party APIs. It's therefore not uncommon that the API also has requirements to reach out to the token endpoint to generate its own token used for downstream services.

10. The API can also reach out to a secrets manager to receive secrets used to internal services such as databases and other infrastructure.

The above-depicted capabilities are now becoming bare minimums for a modern API access control architecture that's intended to scale to not just protect one API, but thousands of APIs. This is in addition to any API management tools, threat protection, testing and deployment tools that are also required in an API management strategy. A single vendor can offer multiple components in one product. That said, Gartner has observed that an API access control architecture diagram is more accurately consumed by implementers when the components are visually separated. For example, an access management tool can have support for federation services, token endpoints and built-in EAM tools. The federation service is used to integrate external identity repositories, the token endpoint to create tokens and the EAM to make policy decisions based on information in directories. The components have different integration mechanisms and usage and the logical separation helps to isolate what vendor-provided tool to use for what purpose.

**Guiding Principle No. 3 — Strike a Balance Between Developer Enablement and Governance**
Governing APIs without introducing bottlenecks is yet another artform. The many APIs that have to be protected individually often result in an explosion of policies. There are, however, some strategies that the API platform team can take to make that balancing act easier — still not easy — but at least more scalable.

Provide self-service interfaces and leverage delegated administrative control where needed. The word delegation is key. Developers want to control their own fate. A Gartner client reported that its central API governance team had a two-week service-level agreement for all changes to any API. This did not go down well with developers and the business needed to push out new applications and services instantly. Rather than face a similar situation, provide interfaces for developers to register and control their own applications and help them manage what identity information they need in their specific access tokens. This does not mean that developers can do anything they like. It must be bound to their specific sandbox and those definitions and the overall policies must still be centrally controlled. As stated above, not all vendors have achieved the identity fabric goal of providing self-service, cross-platform tool interfaces. The availability of consolidated dashboards and administrative interfaces is limited. Clients have even reported creating their own management interfaces on top of their hybrid and multicloud deployments. That said, there are access management vendors that offer role-based access control (RBAC) capabilities that lock down a specific developer’s rights to manage their own claims in their own tokens from a controlled subset of attributes available to them.

Enable the use of existing entitlement repositories and integration in your wide set of APIs by leveraging claims in tokens. Traditional authentication and authorization mechanisms have used concepts such as attributes and claims. OAuth 2.0 introduced the concept of scopes, and it’s a commonly misunderstood concept. Scopes are used by the client (apps) to request access. Scopes should not be used as a concept to validate access at the APIs. Instead, use and treat scopes as a convenience method to request a set of claims. Gartner clients have found that sending optimized claims to APIs reduces access policy complexity and improves scalability. A claims-based approach, however, also requires both token exchange and self-service interfaces to be successfully implemented.
Use the claims that were granted the app in access policies protecting the APIs. Claims themselves are commonly constructed by attributes stored in user stores such as directories and databases. Figure 7 provides a graphical overview of how scopes, claims and attributes relate and are being used. Also, note the potential divergence between requested scopes and authorized scopes that the calling application is responsible to adapt to. Documentation and libraries from API gateway vendors often promote the use of scopes. It’s mostly due to vendors focusing on “greenfield” solutions more than on legacy APIs and services.

**Figure 7: Claims and Validation**

![Claims and Validation Diagram](Source: Gartner)

**Guiding Principle No. 4 — Use Token Exchange Mechanisms**

The principle to use token exchange mechanisms is a rather technical one, but it’s a powerful practice when it comes to privacy, security and scale. It’s just good practice to not send external tokens, known by all clients and mediators, in the internal architecture. Token exchange enables organizations to use opaque tokens at the outer edge, or to use tokens with less personally identifiable information (PII) in general. Token exchange can ensure that each internal service still gets exactly what’s required to complete its purpose.

Exchange tokens using standardized specifications such as OAuth 2.0 Token Exchange (rfc8693). The specification supports both delegated and impersonation, and defines a REST interface to pass one token and receive an optimized internal token tagged with the token’s true audience (your internal API in comparison to an API gateway). Access management vendors such as Curity, ForgeRock and Ping Identity support the token exchange specification. There are also proprietary versions such as the on-behalf-of-flow defined in Azure AD. That said, support in libraries and API gateways are still nascent.
Token exchange is just one of five integrations that are important to establish when integrating APIs and mediators with access management tools. Figure 8 details the integrations, including the token exchange mechanisms, that ensure good orchestration of the components in the identity fabric for API access control.

**Figure 8: Five Integrations Between Mediators and Access Management Tools**

Observe that it’s the token service that exchanges the tokens. This is different from the exchange mechanisms that can be provided by the federation service, which enables organizations to bridge multiple access management tools and their tokens.

**Guiding Principle No. 5 — Implement defense in Depth**

Setting reasonable expectations is an important part of an API access control strategy. For example, expectations for a complete zero trust security posture are nearly impossible to meet. Legacy applications, organizational resistance and tooling support for managing granular controls in legacy systems are all barriers to a complete coverage.

In February 2021, Netflix published a very well-written blog post titled *Edge Authentication and Token-Agnostic Identity Propagation* that discussed its current strategies and the challenges of implementing API access control. ² It’s evidence that even organizations with mature and evolved API practices have challenges. Authentication is relatively easy; authorization at the depth of the infrastructure is harder. When in doubt, know that you are not alone in your struggles. We as an industry are still early on this journey together.
That said, a zero trust paradigm — a defense-in-depth strategy — is very reasonable as it helps organizations to continually evolve.

Use the following best practices in your defense-in-depth strategy:

- **Implement TLS**: Use server-side certificates in mediators and in your APIs to ensure that all communication — even internal communication — is encrypted.

- **Implement claims-based authorization**: Pass claim values in tokens down to each service so that the service doesn’t have to rely on local user repositories.

- **Use JWTs**: JWTs have become the de facto standard for all access in the internal network. JWTs ensure that local authorization can be made in individual services. That said, use the right token and separate the external and the internal architecture. It's also not uncommon for organizations to incorrectly rely on OpenID Connect ID tokens instead of bearer tokens — that's not the ID token's intended use. Up-to-date guidance on how JWTs should be used for API access is currently being defined by the Internet Engineering Task Force (IETF). For further details, see [JSON Web Token (JWT) Profile for OAuth 2.0 Access Tokens on the IETF Datatracker site](#).

- **Validate tokens at the edge**: In enterprise API gateways, validate the external tokens. Ensure that the signature and audience is correct. Leave fine-grained authorization to each service.

- **Validate tokens and fine-grained claims at each API**: It’s the developers of the APIs that have the deep knowledge about their services and the identity information that’s required to serve a request. Do fine-grained validation of individual claims close to the services.

- **Delegated administration**: Empower developer and DevOps teams with delegated administration self-service interfaces. A defense-in-depth strategy can only scale if developers can control the access to their services.

**Strengths**

The strengths of establishing an identity fabric for API access control and providing guiding principles are as follows:
An identity fabric and a modern API access control strategy scales and can handle thousands of APIs. API access control is not something that can be solved with just one tool. Instead, it must be solved with a composable fabric of tools and a strategy that guides the organizations to make the right decisions.

It forces organizations away from looking for that faulty one-and-done technical pattern, and instead allows for informed discussions and decisions based on principles.

By combining all five guiding principles in an API access control strategy, it's possible to provide good answers to impatient API access control questions for protecting both the new and the old. See also the Guidance section for a host of quick questions and answers to aid those discussions.

Weaknesses

The weaknesses of establishing an identity fabric for API access control and providing guiding principles are as follows:

API access control relies on an identity industry that still has a long way to go when it comes to education and proven best practices. Technology standards and their implementations are also in need of improvement. At the same time, the variety of use cases that need to be supported and the high stakes if something goes wrong makes API access control mission-critical. Gartner foresees the market providing more identity-aware and cross-infrastructure developer portals, along with crisper developer guidance for internal architecture (service meshes are leading the way of providing good defaults and better guidance). There needs to be more guidance on how centralized policy management can be implemented. Also, better control-plane-like functionality needs to be provided by the market to span hybrid and multicloud environments, as well as new and old APIs, to make it easier for API platform teams to better protect APIs.

API access control is hobbled by misconceptions of how challenges can actually be solved. Attempted use of one-and-done types of solutions is an indication that the complexity of the task is not fully understood, or that all the right teams are not involved. API access control challenges are often underestimated and full-blown API access control strategies with guidance that needs to be adopted by multiple teams are not always well-received. It's therefore important to gather the right stakeholders and insights and set reasonable expectations for the strategies to be successful. This must be an ongoing effort.
Vendor marketing also muddies the waters, especially regarding API security. Breaking up API security into discovery, threat protection and API access control helps organizations to see through vendor comments, such as “Our API security solution is also identity aware.” Here, the vendor means that its tool can correlate two requests that use the same credential, but actually have no knowledge about the identity, authentication or authorization.

**Guidance**

Use the following guidance to establish a modern API access control strategy:

- Implement API access control guidance as a part of your API management strategy.
- Establish strategies that enable you to scale to thousands of APIs instead of just one.
- Assess all five dimensions of an API use case.
- Apply the five guiding principles laid out in this document to your organization by leveraging stakeholders and an API platform team. Use the principles all together to solve your use cases.
- Divide and conquer to break a large and complex API access control problem into more solvable pieces.
- Establish strict separation, with different guidance, between the external applications that access the APIs and the internal infrastructure in and behind your enterprise API gateways.
- Continually evolve your composable identity fabric for API access control.
- Regularly assess the available components against the use case requirements to determine when a new tool needs to be part of the strategy.
- Set expectations in the organizations. A complete zero trust security posture is only realistic to achieve for new greenfield scenarios. It’s the legacy systems that need special assessment, monitoring and nonstandard handling of access control.
- Hats off to the purists, but don’t rearchitect everything. Instead, make a risk-based decision and rearchitect where needed, then assess, live with and adapt to the old.
- Categorize your APIs based on risk, as well as features, and provide good best practices around them.
Distribute implementation guidance, proven libraries and your integration strategies for a hybrid and multicloud environment where your APIs need protection.

Rehearse the following questions and answers:

Understanding the five dimensions and providing good guiding principles provides much more value than providing quick one-and-done answers. That said, given the guiding principles are in place, quick Q&As that describe what good looks like, are at times, still needed.

All one-and-done answers have their own accompanying caveat. The following questions do not constitute an exhaustive list, but they are commonly asked by clients. Their answers, and their associated caveats, are also listed below. Learn them to aid your discussions with developers and other stakeholders.

Q: Is OpenID Connect the protocol of choice for my APIs?

A: No. JWTs are the credential of choice in, and behind, your API gateway. OpenID Connect is a way for user-driven applications to acquire JWTs, especially in cross-domain scenarios; for example, when your authorization server has to reach out to third-party identity providers. Other applications (see Table 1) and user constituencies can rely on other protocols and flows to acquire said token.

Q: Should I use the threat protection mechanisms in the access management tool?

A: Yes. This is where identity-aware analytics can be applied. That said, the access management tool has to be accompanied with API threat protections that also protects your APIs.

Q: Should I use the identity capabilities in the API gateways for all of my identity needs?

A: No. API gateways typically don't have the policies, users and broad support for the 30+ identity specifications that enable you to provide the right security, usability, privacy and scale required for your many APIs. That said, if you don't have a way to exchange external tokens in a centralized access management tool and have no way to replace it, consider using the API gateway for minting your internal JWTs.

Q: Should I send all of the identity information my services need in my tokens?
A: Yes. The less APIs have to rely on third-party directory lookups the better, as it minimizes any latency. That said, it’s important to not send all identity information to all services. Token exchange, developer enablement and a defense-in-depth strategy help you send the right identity information to the right services.

Q: Isn’t it enough to do authorization in my enterprise API gateway?

A: No. That is not sufficient for a defense-in-depth strategy. Authorization takes place in multiple places (see Figure 7). Do coarse-grained authorization at an enterprise API gateway, but fine-grained authorization in each service (or close by using sidecar patterns).

Q: Should I replace my external token?

A: Yes. There are no caveats here. The external token is not intended to be passed around to any internal services. It would bleed information about the internal architecture and it would be a very overentitled token.

Q: Should I really replace my tokens for each service?

A: No, but it should be an important guiding principle and the default integration pattern. That said, if multiple microservices comprise a single application within a single cluster, of course it does not always make sense to exchange the tokens when each service is called, as it increases latency and provides little additional value. But keeping token exchange top of mind when implementing and orchestrating multiple services should be considered a best practice and a must if requests are sent cross domain.

Q: Should I use the ID token when calling APIs?

A: No. The OpenID Connect ID token isn’t intended to be used to call APIs. It’s intended to establish sessions across two different security domains and it’s also scoped to the application, not the API that the application calls. Instead, rely on the access tokens. JWTs are commonly used, but their use differs based on what authorization server issues them. There is current work on standardizing the JWT access token format under the IETF (see JSON Web Token (JWT) Profile for OAuth 2.0 Access Tokens for further details).

Q: Should I use one access management tool for all scenarios and user constituencies?
A: No. Stretching access management tools to support all user constituencies is seldom ideal. Consumers have their own requirements and tooling support and it's typically different from the requirements of your APIs. Instead, ensure that you can broker (federate) between other access management tools and the access management tool that you use to protect your APIs.

Q: Should my APIs accept tokens from multiple token issuers?

A: No. Handle brokering of multiple tokens in the access management tools, not in the APIs. Your internal APIs should just rely on tokens from one authorization server. Leave it to the access management tool to provide policies, handle integrations with third parties and leverage the built-in claims mapping instead of reimplementing it in your APIs.

Q: Should I use mTLS everywhere?

A: Yes, but it requires additional components and it's challenging to retrofit mTLS to existing services. TLS is easy, mTLS is much harder and requires governance, monitoring and automated life cycle management of client certificates and trust anchors. Service mesh architectures have it built in from the start and enable it by integrating good life cycle management capabilities for the client certificates. Other services require automated and, in some cases, manual life cycle management capabilities to be put in place to enable mTLS. Use it for greenfield scenarios, but don't expect to be able to use mTLS for legacy applications and services.

Q: But can't my bearer tokens be stolen?

A: Yes. The core specification that defines bearer tokens only defines how they are sent. There is no proof of possession, holder-of-keys, or sender-constrained token mechanisms built into bearer tokens. If an adversary has access to the token it can be used as a proof that they have the right to access your services. That said, providing API threat protection capabilities such as a web application and API protection (WAAP) service helps you understand the context of when a token was sent. Using TLS everywhere, selecting the right grant type and understanding clients’ capabilities to store tokens make theft less likely. Having short-lived access tokens limits the timespan when a stolen token can be used. It's also possible to choose proof-of-possession flows, such as the mTLS or JWTs signed on the client, as a grant type to actually get the access token. This, in turn, makes a bearer token behave much like a traditional session established using strong authentication.
Evidence


Recommended by the Author

Some documents may not be available as part of your current Gartner subscription.

Modern Identity and APIs: OpenID Connect, OAuth 2.0 and SCIM 2.0
Building Identity Into a Microservices Architecture
Building Authentication, Authorization and SSO Into API-Driven Apps
Architecting Modern Policy-Based Runtime Authorization
Solution Path for Forming an API Security Strategy
How to Implement API Discovery and Detection to Improve API Management and Security
How to Successfully Implement API Management
Decision Point for Mediating API and Microservices Communication
API Security: What You Need to Do to Protect Your APIs
Federate, Rebrand and Recharter Your API Center of Excellence to Enable an API Platform Team
## Table 1: Grant Types and Their Use Cases

<table>
<thead>
<tr>
<th>Grant type</th>
<th>Protocol</th>
<th>Web Servers</th>
<th>Native Apps</th>
<th>SPA (same domain)</th>
<th>SPA (cross domain)</th>
<th>Services</th>
</tr>
</thead>
<tbody>
<tr>
<td>Resource Owners Password Credentials</td>
<td>OAuth 2.0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Client Credentials</td>
<td>OAuth 2.0</td>
<td>X</td>
<td></td>
<td></td>
<td></td>
<td>X</td>
</tr>
<tr>
<td>Authorization Code</td>
<td>OAuth 2.0 and OIDC</td>
<td>X</td>
<td></td>
<td></td>
<td>X</td>
<td></td>
</tr>
<tr>
<td>Authorization Code with PKCE</td>
<td>OAuth 2.0 and OIDC</td>
<td></td>
<td>X</td>
<td></td>
<td>X</td>
<td></td>
</tr>
<tr>
<td>Implicit</td>
<td>OAuth 2.0 and OIDC</td>
<td></td>
<td></td>
<td></td>
<td>X (legacy)</td>
<td></td>
</tr>
<tr>
<td>Hybrid</td>
<td>OIDC</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>X</td>
</tr>
<tr>
<td>Device flow</td>
<td>OAuth 2.0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>X</td>
</tr>
<tr>
<td>JWT and SAML grant (RFC7523)</td>
<td>OAuth 2.0</td>
<td>X</td>
<td></td>
<td></td>
<td></td>
<td>X</td>
</tr>
<tr>
<td>OAuth MTLS (RFC8705)</td>
<td>OAuth 2.0</td>
<td>X</td>
<td></td>
<td></td>
<td></td>
<td>X</td>
</tr>
<tr>
<td>Token Exchange (RFC8693)</td>
<td>OAuth 2.0</td>
<td>X</td>
<td></td>
<td></td>
<td></td>
<td>X</td>
</tr>
</tbody>
</table>
### Table 2: Services and Their Commonly Used Authentication and Authorization Architecture

<table>
<thead>
<tr>
<th></th>
<th>Authentication</th>
<th>Authorization and Example Policy Management</th>
<th>API Access Control Points</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Microservices and Service Mesh</strong></td>
<td>JWTs</td>
<td>JWT claims and policy as code in container platforms</td>
<td>In enterprise API gateways and sidecars such as Envoy</td>
</tr>
<tr>
<td><strong>Serverless</strong></td>
<td>fPaaS provided credentials (commonly JWTs)</td>
<td>fPaaS platform with governance through CIEM and/or federation and integration of existing identities</td>
<td>In platform-provided API gateways</td>
</tr>
<tr>
<td><strong>REST-based APIs</strong></td>
<td>JWTs</td>
<td>JWT claims, local authorization or EAMs</td>
<td>In mediators such as enterprise API gateways or local authorization in code</td>
</tr>
<tr>
<td><strong>SOAP-based APIs</strong></td>
<td>Passwords or X.509 certificates</td>
<td>Seldom implemented</td>
<td>In enterprise API gateways and in code</td>
</tr>
<tr>
<td><strong>GraphQL</strong></td>
<td>JWTs</td>
<td>JWT claims</td>
<td>In enterprise gateways, in GraphQL servers or in downstream APIs</td>
</tr>
<tr>
<td><strong>Events</strong></td>
<td>Passwords or X.509 certificates</td>
<td>Static policies in event brokers</td>
<td>In event brokers</td>
</tr>
<tr>
<td><strong>APIs in COTS and SaaS products</strong></td>
<td>API keys, X.509 certificates, client credentials and headers</td>
<td>Using JWTs, IGA or through secrets managers and injection</td>
<td>In secrets managers or IGAs</td>
</tr>
</tbody>
</table>

Source: Gartner