NIST Special Publication 800
NIST SP 800-228 ipd

Guidelines for API Protection for Cloud-Native Systems

Initial Public Draft

Ramaswamy Chandramouli
Zack Butcher

This publication is available free of charge from:
https://doi.org/10.6028/NIST.SP.800-228.ipd

NATIONAL INSTITUTE OF
STANDARDS AND TECHNOLOGY
U.S. DEPARTMENT OF COMMERCE

NIST Special Publication 800
NIST SP 800-228 ipd

Guidelines for API Protection for Cloud-Native Systems

Initial Public Draft

Ramaswamy Chandramouli
Computer Security Division
Information Technology Laboratory

Zack Butcher
Tetrate, Inc.

This publication is available free of charge from:
https://doi.org/10.6028/NIST.SP.800-228.ipd

March 2025

U.S. Department of Commerce
Howard Lutnick, Secretary

National Institute of Standards and Technology
Craig Burkhardt, Acting Under Secretary of Commerce for Standards and Technology and Acting NIST Director

Certain commercial equipment, instruments, software, or materials, commercial or non-commercial, are identified in this paper in order to specify the experimental procedure adequately. Such identification does not imply recommendation or endorsement of any product or service by NIST, nor does it imply that the materials or equipment identified are necessarily the best available for the purpose.

There may be references in this publication to other publications currently under development by NIST in accordance with its assigned statutory responsibilities. The information in this publication, including concepts and methodologies, may be used by federal agencies even before the completion of such companion publications. Thus, until each publication is completed, current requirements, guidelines, and procedures, where they exist, remain operative. For planning and transition purposes, federal agencies may wish to closely follow the development of these new publications by NIST.

Organizations are encouraged to review all draft publications during public comment periods and provide feedback to NIST. Many NIST cybersecurity publications, other than the ones noted above, are available at https://csrc.nist.gov/publications.

Authority

This publication has been developed by NIST in accordance with its statutory responsibilities under the Federal Information Security Modernization Act (FISMA) of 2014, 44 U.S.C. § 3551 et seq., Public Law (P.L.) 113-283. NIST is responsible for developing information security standards and guidelines, including minimum requirements for federal information systems, but such standards and guidelines shall not apply to national security systems without the express approval of appropriate federal officials exercising policy authority over such systems. This guideline is consistent with the requirements of the Office of Management and Budget (OMB) Circular A-130.

Nothing in this publication should be taken to contradict the standards and guidelines made mandatory and binding on federal agencies by the Secretary of Commerce under statutory authority. Nor should these guidelines be interpreted as altering or superseding the existing authorities of the Secretary of Commerce, Director of the OMB, or any other federal official. This publication may be used by nongovernmental organizations on a voluntary basis and is not subject to copyright in the United States. Attribution would, however, be appreciated by NIST.

NIST Technical Series Policies

Copyright, Use, and Licensing Statements
NIST Technical Series Publication Identifier Syntax

Publication History

Approved by the NIST Editorial Review Board on YYYY-MM-DD [Will be added to final publication.]

How to Cite this NIST Technical Series Publication:

Chandramouli R, Butcher Z (2025) Guidelines for API Protection for Cloud-Native Systems. (National Institute of Standards and Technology, Gaithersburg, MD), NIST Special Publication (SP) NIST SP 800-228 ipd. https://doi.org/10.6028/NIST.SP.800-228.ipd

Author ORCID iDs

Ramaswamy Chandramouli: 0000-0002-7387-5858

Public Comment Period
March 25, 2025 – May 12, 2025

Submit Comments
sp800-228-comments@nist.gov

National Institute of Standards and Technology
Attn: Computer Security Division, Information Technology Laboratory
100 Bureau Drive (Mail Stop 8930) Gaithersburg, MD 20899-8930

Additional Information
Additional information about this publication is available at https://csrc.nist.gov/pubs/sp/800/228/ipd, including related content, potential updates, and document history.

All comments are subject to release under the Freedom of Information Act (FOIA).

Abstract

Modern enterprise IT systems rely on a family of application programming interfaces (APIs) for integration to support organizational business processes. Hence, a secure deployment of APIs is critical for overall enterprise security. This, in turn, requires the identification of risk factors or vulnerabilities in various phases of the API life cycle and the development of controls or protection measures. This document addresses the following aspects of achieving that goal: (a) the identification and analysis of risk factors or vulnerabilities during various activities of API development and runtime, (b) recommended basic and advanced controls and protection measures during pre-runtime and runtime stages of APIs, and (c) an analysis of the advantages and disadvantages of various implementation options for those controls to enable security practitioners to adopt an incremental, risk-based approach to securing their APIs.

Keywords

API; API endpoint; API gateway; API key; API schema; web application firewall.

Reports on Computer Systems Technology

Call for Patent Claims

This public review includes a call for information on essential patent claims (claims whose use would be required for compliance with the guidance or requirements in this Information Technology Laboratory (ITL) draft publication). Such guidance and/or requirements may be directly stated in this ITL Publication or by reference to another publication. This call also includes disclosure, where known, of the existence of pending U.S. or foreign patent applications relating to this ITL draft publication and of any relevant unexpired U.S. or foreign patents.

ITL may require from the patent holder, or a party authorized to make assurances on its behalf, in written or electronic form, either:

a) assurance in the form of a general disclaimer to the effect that such party does not hold and does not currently intend holding any essential patent claim(s); or

b) assurance that a license to such essential patent claim(s) will be made available to applicants desiring to utilize the license for the purpose of complying with the guidance or requirements in this ITL draft publication either:

    i. under reasonable terms and conditions that are demonstrably free of any unfair discrimination; or

    ii. without compensation and under reasonable terms and conditions that are demonstrably free of any unfair discrimination.

Such assurance shall indicate that the patent holder (or third party authorized to make assurances on its behalf) will include in any documents transferring ownership of patents subject to the assurance, provisions sufficient to ensure that the commitments in the assurance are binding on the transferee, and that the transferee will similarly include appropriate provisions in the event of future transfers with the goal of binding each successor-in-interest.

The assurance shall also indicate that it is intended to be binding on successors-in-interest regardless of whether such provisions are included in the relevant transfer documents.

Such statements should be addressed to: sp800-228-comments@nist.gov

Table of Contents

Acknowledgments

The authors would like to thank Orion Letizi, technical writer at Tetrate for providing continuous, ongoing edits during the development of this document. We would also like to thank Erica Hughberg, an engineer at Tetrate and James Gough, a Distinguished Engineer at Morgan Stanley for their feedback on the initial outline for controls. Their extensive hands-on experience in running API security programs in large enterprises helped us to address the current API security issues and incorporate state of practice API security controls in our recommendations. Last but not the least, the authors would also like to express their thanks to Isabel Van Wyk of NIST for her detailed and extensive editorial review.

Executive Summary

Application programming interfaces (APIs) provide the means to integrate and communicate with the modern enterprise IT application systems that support business processes. However, a lack of due diligence can introduce vulnerabilities and risk factors that exploit the connectivity and accessibility features of APIs. If these vulnerabilities are not identified, analyzed and addressed through control measures, attack vectors could threaten the security posture of the application systems spanned by these APIs. A systematic and effective means of identifying and addressing these vulnerabilities is only possible by treating the development and deployment of APIs as an iterative life cycle using paradigms like DevSecOps.

This document provides guidance and recommendations on controls and protection measures for secure API deployments in the enterprise. In addition, an analysis of the advantages and disadvantages of various implementation options (called patterns) for those controls enable security practitioners to choose the most effective option for their IT ecosystem.

Developing these controls and analyzing their implementation options should be guided by several overarching principles:

The guidance for controls should cover all APIs, regardless of whether they are exposed to customers/partners or used internally within the enterprise.

With the vanishing of perimeters in modern enterprise IT applications, all controls should incorporate the concept of zero trust.

The controls should span the entire API life cycle and be classified into (a) pre-runtime protections and (b) runtime protections that are then subdivided into basic and advanced protections to enable incremental risk-based adoption.

1. Introduction

Application programming interfaces (APIs) represent an abstraction of the underlying implementation of a digital enterprise. Given the spatial (e.g., on-premises, multiple clouds) and logical (e.g., microservices) nature of current enterprise applications, APIs are needed to integrate and establish communication pathways between internal and third-party services and applications. Informally, APIs are the lingua franca of modern IT systems: they describe what actions users are allowed to take. They are also used in every type of application, including server-based monolithic, microservices-based, browser-based client, and IoT.

1.1. Building Blocks and Structures

An Application Programming Interface (API) defines how any two pieces of software communicate – they are ubiquitous in software. An API is a collection of commands or *endpoints* that operate on data or *objects* via some protocol. *Network-based APIs* are APIs built to be consumed by remote applications over the network. Because they're exposed and consumed over the network, they present a unique set of challenges. The growth of (micro-) service-oriented architectures, coupled with Software-as-a-Service (SaaS) becoming commonplace – which are nearly always delivered via APIs – has resulted in an explosion in network-based APIs across organizations. This document focuses on controls for network-based APIs.

Before we can discuss API controls, we need a common understanding and language for the building blocks, and how they relate to each other. The taxonomy is: an API is composed of a set of API Endpoints; API Endpoints are implemented by Services; at runtime, Requests to a specific API Endpoint are served by Service Instances. An API Gateway hosts many APIs and is responsible for mapping each Request to its target API Endpoint, applying policy for that Endpoint (e.g. authentication and rate limiting), then routing that Request to a Service Instance which implements that API Endpoint.

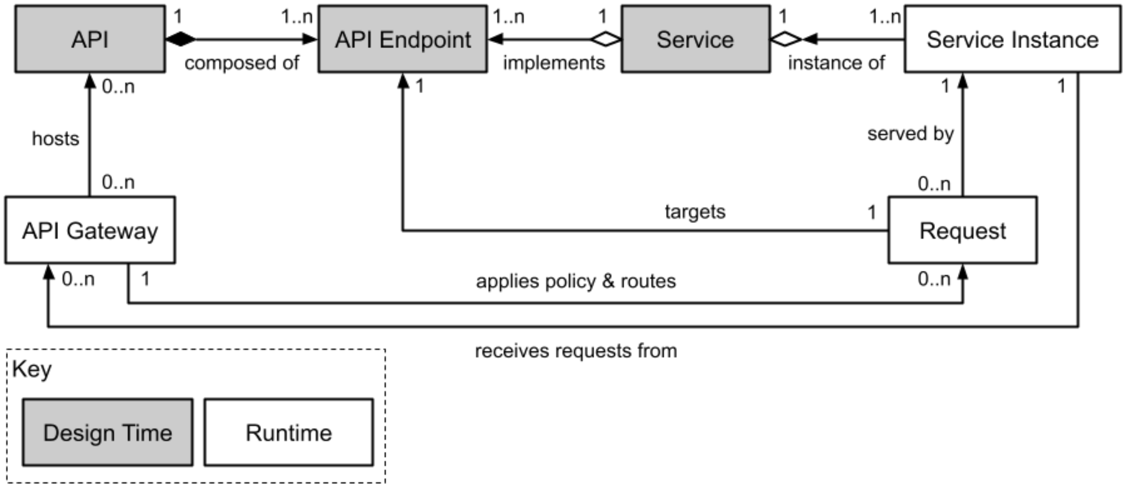

Fig. 1. API, API Endpoint, Service and Service Instance

Traditionally, we think of network-based APIs as being customer-oriented, partner-oriented, or internal – often called “third-party”, “second-party”, and “first-party” APIs, respectively. Second- and third-party APIs are typically exposed to callers outside of the organization via an API gateway. First-party APIs can be exposed to callers inside of the organization on the same API gateway, but they are also often consumed directly by internal callers without traversing a dedicated API serving stack.

An API is a set of API Endpoints, and a Service implements a set of API Endpoints – so every Service implements *some* API. We call these *Service APIs*. Most first-party API integrations happen via the Service API, i.e. they map to a single service. On the other hand, APIs hosted by the API gateway typically have Endpoints that map to many different Services. This is especially common for second- and third-party APIs. We call these *Facade APIs*, because they present a single facade to an outside caller over (potentially many) different Service APIs. Finally, it’s common that multiple Services are grouped together into an *Application* – typically along organizational lines (often an Application maps to a team). Schematic diagrams of a Service API, Façade API and an Application (Monolithic) API are given below:

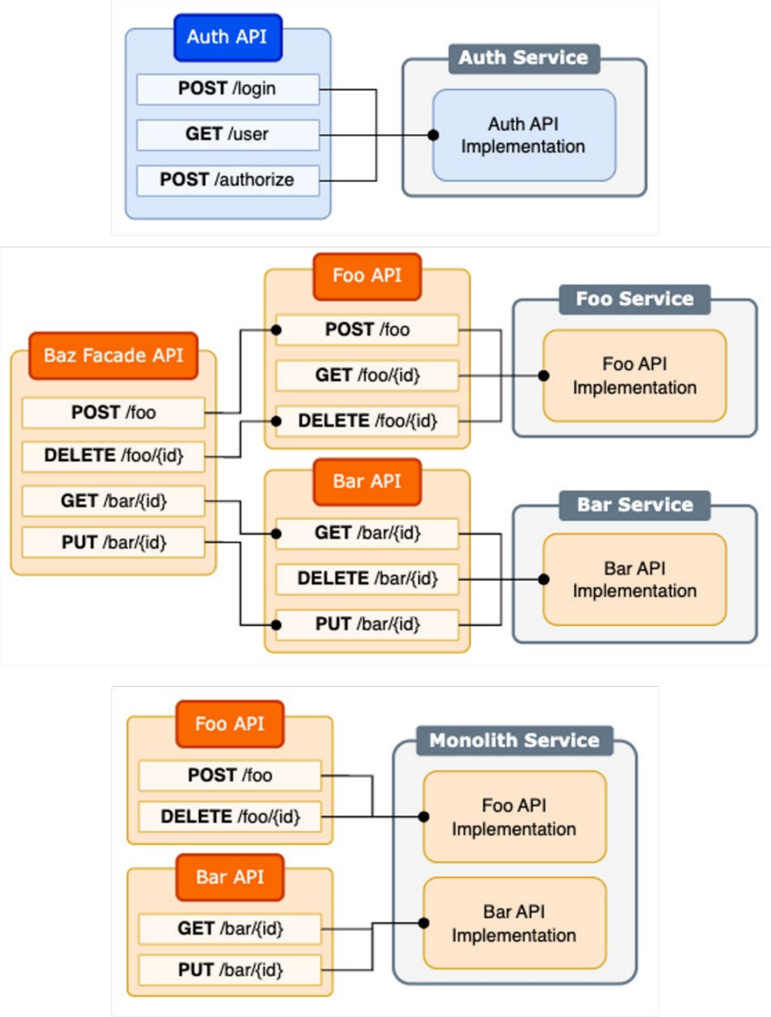

Fig. 2. (Top to Bottom) Service API, Façade API, Service and Application (Monolithic)

Less formally: we can think of the APIs we expose outside the organization as a facade over a set of Services. Those Services implement internal APIs (*Service APIs*). Services in the organization communicate with each other via those internal APIs – sometimes directly, and sometimes via an API gateway. The API Gateway is responsible for some policies, like authentication and rate limiting, as well as being responsible for mapping the facade APIs for external clients to internal APIs. Then, to get a handle on things organizationally, we often group related Services into a bucket called an Application.

While we tend to think of APIs in the context of exposing functionality to clients or partners, APIs don't exist solely at the edge of our infrastructure. Any time systems communicate, there's *some* API involved. Even if that API is something like CSV over FTP. The examples in SP focus primarily on "modern" APIs exposed via mechanisms like HTTP/REST, gRPC, or SOAP, but we believe the principals in this SP are universal and should be applied to *all* APIs.

1.2. Zero Trust and APIs: The Vanishing Perimeter

APIs are built out of services that communicate with each other via APIs, similar to how the internet is a "network of networks." One of the most important implications of zero trust is that there is no meaningful distinction between an "internal" and "external" caller because the perimeter is the service instance itself. Rather, all callers are trusted if they are authorized to be trusted. This contrasts with traditional approaches to API security in which the only "APIs" are those exposed to "external" callers, and API-oriented controls are only enforced at the perimeter, typically via an API gateway.

NIST Special Publication (SP) 800-207A [6] discusses zero trust at runtime and the principle of shrinking the perimeter to the service instance using the five runtime controls of identity-based segmentation:

1. Encryption in transit — To ensure message authenticity and prevent eavesdropping, thus preserving confidentiality
2. Authenticate the calling service — Verify the identity of the software sending requests
3. Authorize the service — Using that authenticated identity, check that the action or communication being performed by the service is allowed
4. Authenticate the end user — Verify the identity of the entity triggering the software to send the request, often a non-person entity (NPE) (e.g., service account, system account)
5. Authorize the end user to resource access — Using the authenticated end-user identity, check that they are allowed to perform the requested action on the target resource

Achieving a zero-trust runtime requires applying these five controls to *all* API communications. This guidance further describes additional controls that are necessary for safe and secure API operations beyond identity-based segmentation. These controls should be enforced on all APIs in a system, including those exposed to the outside world (i.e., public APIs) and those intended only for other applications in a given infrastructure (i.e., internal APIs).

1.3. API Life Cycle

Like all software, APIs grow and change over time as requirements drift and usage patterns change. They also go through a continuous, iterative life cycle, including:

- Plan, Develop, Build, Test, Release — These "pre-runtime" life cycle phases lead to a service that can be deployed in production.
- Deploy, Operate, Monitor, Feedback — These "runtime" life cycle phases involve running and operating a service in production.

DoD Enterprise DevSecOps Fundamentals [1] provides a detailed description of each phase of the software development life cycle. Application of the DevSecOps paradigm in the context of cloud-native applications can be found in [4][5].

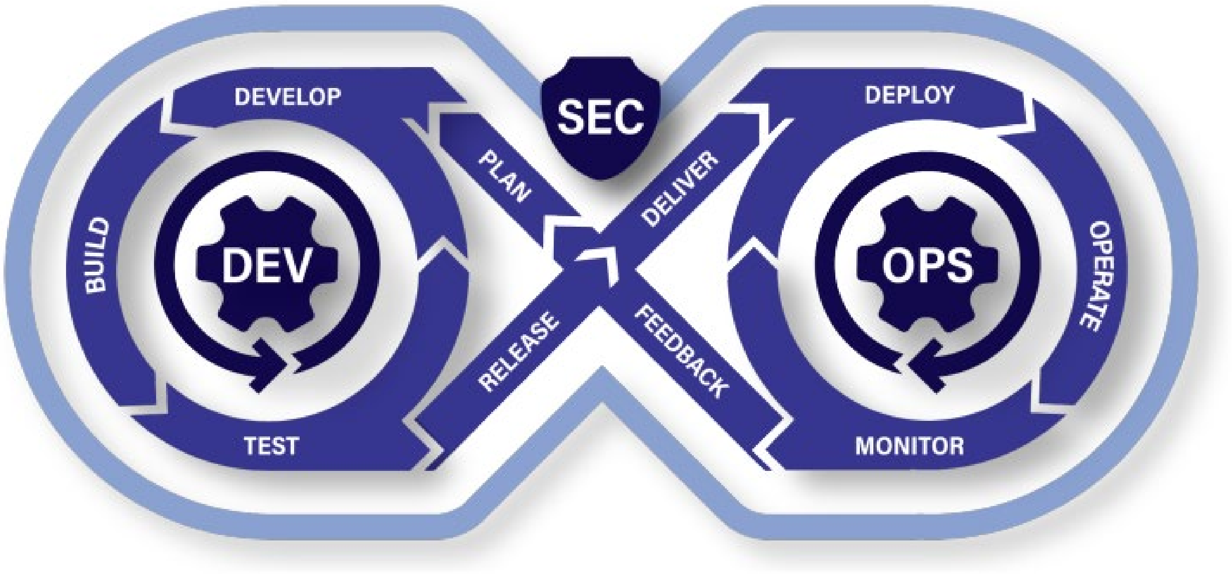

Fig. 3. DevSecOps life cycle phases

1.4. Document Goals

The goal of this document is to recommend guidance or controls for API protection. These controls are classified into two categories:

1. Pre-runtime API protections — These controls need to be applied when designing and building APIs.
2. Runtime API protections — These controls need to be applied to every API request that an infrastructure serves, not just at the perimeter.

Each of these two categories is further divided into two subcategories based on organizational maturity (i.e., basic and advanced), which enables enterprises to adopt them based on an incremental risk-based approach.

A prerequisite for defining any API protection measure or policy irrespective of its category or sub-category is that the protections must be expressed in terms of nouns and verbs that pertain to API components, API endpoint components, API requests, and API responses that in turn contain references to resources/data and operations on those resources. These nouns and verbs form the fundamental surface that is exposed to the consumers of APIs and API endpoints.

1.5. Relationship to Other NIST Documents

Today, most enterprise software development and integration are based on APIs. Section 1.3 articulated the close relationship between software and APIs, demonstrated that API development and deployment follow the same iterative life cycle as the software, and provided NIST guidance on DevSecOps.

Another distinguishing feature of the controls recommended for protecting APIs is the capacity to provide assurance for conforming to the principles of zero trust. This is because there is no distinction between internal and external API requests/calls due to the absence of an identifiable network perimeter and the distributed nature of applications on-premises and multiple clouds. This security assurance can be achieved using authentication and authorization controls using identity-based segmentation [2]. Documents that provide recommendations on the configuration of authentication and authorization controls in the context of cloud-native applications (e.g., [2][3]) are also relevant in the context of configuring controls for API protection.

1.6. Document Structure

This document is organized as follows:

- Section 2 looks at the risk factors and vulnerabilities associated with APIs and the attack vectors that could exploit those vulnerabilities.
- Section 3 recommends controls to protect APIs and classifies them into basic and advanced categories that need to be applied prior to runtime or enforced during runtime.
- Section 4 provides a detailed analysis of implementation options or patterns for the controls described in Sec. 3 and outlines the advantages and disadvantages of each pattern.
- Section 5 provides the summary and conclusions.
- Appendix A provides the classification taxonomy for APIs.

- Appendix B illustrates the API controls related to each DevSecOps phase

2. API Risks — Vulnerabilities and Exploits

This section considers some common risk factors associated with API deployments, including:

- Lack of visibility of APIs in the enterprise inventory [7]
- Missing, incorrect, or insufficient authorization [7]
- Broken authentication [7]
- Unrestricted resource consumption [7]
- Leaking of sensitive information
- Insufficient verification of input data

2.1. Lack of Visibility of APIs in the Enterprise Inventory

Most organizations have gaps in their API inventories, even if they otherwise have mature inventory management capabilities. Without an accurate API inventory, one cannot begin to protect the enterprise estate, and there may be incidents occurring at the API level that the security organization is entirely unaware of. Common reasons for lack of visibility are:

- Organizational silos: APIs are built by many teams across the organization, deployed across cloud and on-premises environments, and inherited in mergers and acquisitions. This results in uneven attention to security concerns and difficulty establishing accurate, up-to-date inventories. This is worsened in organizations that achieve a high degree of developer agility: the faster development happens; the faster inventories grow stale.
- Rogue or shadow APIs: APIs defined for internal use (e.g., debugging, testing, ad hoc solutions to business problems) may not be appropriately documented and often bypass standard security review practices.
- Zombie or deprecated APIs: APIs may have been replaced or superseded by newer systems but have not yet been entirely removed (e.g., because all callers have not yet migrated to the alternative, there no longer exists a team responsible for the system). They risk falling behind the latest security policies and protections.

2.2. Missing, Incorrect, or Insufficient Authorization

Authorization is notoriously difficult to get right. It requires a high-reliability, low-latency system for making decisions about user access to resources at request time, and application developers must integrate their application with the same authorization system to keep it up to date on users, resources, and permissions as the system changes over time (e.g., users create and delete resources, assign new permissions). Even then, developers may enforce access decisions incorrectly in their application code. In the industry-recognized catalogue of API risks three of the top 10 (i.e., 1, 3, and 5) focus on authorization [7].

In line with identity-based segmentation, every service for an API endpoint should perform two levels of authorization: 1) service authorization and 2) end-user-to-resource authorization [6]. However, implementing both levels of authorization can still leave many APIs open to risk. Individual fields of a resource often need to be authorized independently of the resource itself. For example, if additional debug information is embedded in an “internal” field of the API object, that field should not be visible to “external” callers (i.e., callers not authorized to see privileged debug information).

Authorization risks can be categorized in three ways:

1. Missing authorization: There is no fine-grained, resource-level authorization present. For example, a legacy system may be operating under different access models (e.g., in a perimeter-based model, access *is* authorization), or there may be implementation bugs (i.e., an access check that should be enforced is not).
2. Incorrect authorization: The application performs an end-user-to-resource authorization check but fails because it checks any or all of the following: the wrong end user identity, the wrong permission, or the wrong target resource.
3. Insufficient authorization: The application performs a resource-level authorization that is successful, but the resource itself contains information that is “privileged” or not intended for the level of access implied by access to the resource itself. This is often the root cause for the risk of leaking sensitive information (see Sec. 2.5).

2.3. Broken Authentication

Authentication is a prerequisite for authorization, particularly two aspects: the authentication system itself is robust, and the application uses the authenticated identities correctly.

Risks that an authentication system needs to mitigate include [8]:

- Credential stuffing, where an attacker brute forces usernames and passwords without mitigation (e.g., rate limits and Captcha).
- Brute force attacks on a single account without mitigations. This and the previous bullet are closely related to unrestricted resource consumption (see Sec. 2.4).
- Insecure practices like weak passwords, passing sensitive data in public channels (e.g., the URL), missing password validation for changes to sensitive account data, and using weak keys or poor algorithms to encrypt user data in transit and at rest.
- Bad or incorrect token validation, including not validating at all, ignoring expiry, and using insecure signing schemes or weak signing keys.

With a robust and secure authentication system in place, the application must use those credentials correctly. Risks to mitigate include:

- Missing authentication. Tokens can be present but simply not checked. This is often due to a bug or misconfiguration in the application.

- Weak or predictable tokens, default accounts, and default passwords (e.g., a hard-coded bootstrap account with the same username and password on all devices, test accounts with predictable names and weak/guessable passwords).

2.4. Unrestricted Resource Consumption

Services consume resources to serve APIs, many of which can affect external systems or the real world when serving an API call. The effects are an intended part of the business flow, but automation creates avenues for abuse by malicious users. Therefore, usage must be restricted to protect against malicious attackers abusing the system with a denial-of-service attack (DoS) or for its impact on external systems.

2.4.1. Unrestricted Compute Resource Consumption

Broadly, the risks associated with unrestricted compute resource consumption (e.g., memory, CPU, storage) are best mitigated via a combination of rate limiting, timeouts, circuit breaking (i.e., limits on the number of concurrent outstanding requests), bot/abuse detection, and application changes (e.g., reject file uploads over 20MB in size, return at most 10 items in response to a list request). These risks manifest as:

- DoS attacks via bandwidth saturation or resource starvation
- Unreliable performance due to resource utilization for one user or service impacting others
- Cost amplification, where an attacker can spend a small amount of resources (e.g., money, compute, bandwidth) to make requests that trigger a system to spend a much larger amount of resources servicing the request

Even “internal” API consumption poses many of these risks. In most organizations, it is much easier for a developer to accidentally cause a DoS on an internal service than an external attacker causing such an attack maliciously. This is a potential security event that necessitates the need for a zero trust approach.

2.4.2. Unrestricted Physical Resource Consumption

The risks associated with the unrestricted consumption of physical resources are often ignored by software engineers, who tend to be better versed in the threat landscape of the virtual world. Critical business operations can be impacted when an attacker targets software systems that control physical processes (e.g., SCADA systems).

APIs may also result in text messages being sent to users, charges to credit cards, or the consumption of expensive third-party resources. For example, a common challenge seen by organizations that adopt AI is the accidental over-use of expensive AI APIs, resulting in large unplanned expenses for the business.

These risks are best mitigated by a combination of rate limiting, quotas, spending policy controls in third-party software, bot/abuse detection, and application or business flow changes. These risks manifest as:

- Impacts on business operations (e.g., damage to equipment and personnel, the creation of fake orders that require human effort to sort and remove
- Impacts on customer relationships (e.g., scalpers automatically buying inventory to re-list at a higher price elsewhere)
- Infrastructure co-opted for abuse or harassment (e.g., multi-factor authentication fatigue attacks, where an attacker triggers text spam to a user's phone via an SMS 2-factor authentication system [9])
- Unplanned expenses (e.g., consuming far more of a third-party service than planned due to satisfying requests made by a malicious user)

Mitigations for both compute and physical resource consumption are similar. For compute resources, how users interact with a system should be limited. For physical resources, how the user interacts with a system *and* how a system interacts with external systems should be limited and considered early in the design phase. Mitigating these risks can sometimes require business flow changes.

2.5. Leaking Sensitive Information to Unauthorized Callers

Unintentionally leaking business data via APIs is closely related to missing, incorrect, or insufficient authorization (see Sec. 2.2). While correct, robust authorization should mitigate this risk, sensitive data can still be leaked from APIs via side channels. The two most common side channels exploited by attackers are response codes and error information.

Risks include:

- Enumeration of the resources (e.g., users, objects) in a system. This can have secondary impacts on the business, like revealing the customer set, information about product inventory, or the identity of employees in an organization. A common method of enumeration is enabled by services responding with "Not Found" status codes instead of "Permission Denied," allowing an attacker to distinguish between resources that exist (403) and those that do not (404).
- Revealing information about the internal implementation of the infrastructure to attackers. While security through obscurity is no security at all, it is still prudent to make it as hard as possible for attackers to discover an infrastructure's fine-grained specifics, which is often included in error messages (e.g., the exact versions of common software being run, internal names of systems for future pivot attacks).

2.6. Insufficient Verification of Input Data

Trusting unverified input is one of the largest classes of recurring security bugs in software. There are at least two levels of verification that APIs need to be concerned with:

- Validating that the input is syntactically correct
- Ensuring that valid input is not malicious

2.6.1. Input Validation

A service must validate that each request (i.e., input) matches the API's definition, that all expected fields are present and of the correct type, and that no unexpected fields are present. For example, an API definition may say, "The 'name' field is required and must be a non-empty string less than 100 characters long," which must be verified at runtime on every request.

The lack of input validation results in a variety of risks, including:

- Impacting the availability of APIs
  - The "Query of Death" [24] is a DoS attack via specially crafted requests that trigger pathological worst-case behavior in the server.
  - In the worst case, the server itself may crash due to bad input handling, which can be exploited by an attacker to cause DoS on systems.
- Invalid or malicious data being stored in the system, which can cause latent issues (e.g., failure to restart during recovery, crashes when accessing invalid records)
- Unanticipated error handling during request processing, which leaks internal information

2.6.2. Malicious Input Protection

While the input may satisfy "syntactic" validation, it also needs to be verified as non-malicious before it is used. Extending the "name" example above, a caller may send a request that contains a name field with a string less than 100 characters (i.e., valid), but that string may be a SQL injection attack. Common risks include:

- Data leaks or corruption (e.g., a SQL injection attack)
- Unanticipated or unrestricted resource utilization (e.g., an attacker automates account creation and uploads multi-gigabyte "profile pictures" to each account)
- Exposing a surface that attackers can use to pivot within the infrastructure or leverage to mount further attacks on others (e.g., by allowing servers to be used for server-side request forgery [SSRF])
- Cost amplification attacks, like the "billion laughs attack" (XML expansion) [10] or "zip bombs" (zip archive expansion) [11]

2.7. Credential Canonicalization— Preparatory Step for Controls

A common problem at the API gateway is handling the many different credentials that clients use to call APIs: mobile apps use a certificate, clients use an API key and expect HTTPS, internal applications expect an mTLS connection with a SPIFFE identity, and others use HTTPS and a Kerberos ticket. All of them also need to convey the user's credential (e.g., OAuth Bearer token, a custom JWT, some trusted internal header). The combination is immense and challenging for application developers to perform correctly. As a result, organizations may only perform authentication and authorization at the edge via the API gateway. A solution to this problem is to standardize the credentials that an application sees at the API gateway — that is, to *canonicalize* them.

2.7.1. Gateways Straddle Boundaries

A gateway is something in an infrastructure that straddles a boundary and is typically the only way for traffic to cross that boundary. As a result, the gateway is uniquely positioned to enforce policy. One of the most important policies that the API gateway enforces is authentication, ideally of both the user and the calling service.

Identity-based segmentation states that every server should authenticate and authorize both the calling service and the end user of every request and that those policies should be enforced at every hop in the infrastructure [6]. However, changing legacy systems to support new identities is often not possible. The challenge lies in implementing identity-based segmentation and support for both service and user identities without impacting other parts of the infrastructure.

API gateways can be used to draw a boundary around the parts of an infrastructure that perform identity-based segmentation. Within that boundary, all applications expect a standard set of credentials (e.g., user identity via a JWT in a specific header and service identity via a SPIFFE X.509 certificate). Common policy, practices, and tooling can then be used to ensure that all applications perform authentication and authorization correctly. Legacy schemes may continue to be used outside of the boundary. To reach inside, traffic must traverse a gateway that can canonicalize the incoming request's credentials into the expected form.

2.7.2. Requests With a Service Identity but No User Identity

Consider a batch job that runs nightly and touches data for many users. This is a risk because it requires special casing by the applications. For *some* service identities, end user authorization is not required, but for all others, it is required. Any special casing increases the opportunity for incorrect or insufficient authorization.

The solution here is to adopt service accounts that present some system in a user identity domain. That service account can be for an internal system and, therefore, have permission to act on the data of many other users, or it can be for a user's applications with correspondingly fewer permissions. The API gateway can mediate with the user authentication system to exchange the service's runtime identity for a service account credential that represents the service in the user identity domain and attach that service account credential as the end user credential to requests that it forwards into the part of the infrastructure that supports identity-based segmentation.

Applications that perform identity-based segmentation will need to configure policy for that service account user so that it can act on all of the data that the batch job previously used its service identity to access. At the same time, the application can remove any support for special data access without an end user credential. Finally, the existing infrastructure can be leveraged to audit and manage both user and service access to data.

An implication of this is that all applications attempting to implement identity-based segmentation without a user identity should adopt service accounts by changing their application code. This will simplify future migration into the identity segmentation domain and make the system more secure overall.

2.7.3. Requests With a User Identity But No Service Identity

Consider a cloud-provider API gateway that receives user traffic, terminates TLS, performs end-user authentication, and forwards requests to the infrastructure. The gateway enforces authentication, so some user credential is present. However, unless special care has been taken to communicate the service identity (e.g., via an API key or service account JWT), most notions of the calling workload will be lost at the external gateway provider.

Depending on the specifics of the setup, the only option may be to configure service identity-level policy via the external API gateway's controls and then implement fine-grained service-to-service policy for how requests can flow from that external gateway into the infrastructure. In other cases, the external gateway can be configured to pass some notion of the external workload (e.g., forwarding the client's certificate as a header) and then use that to create some canonical workload credential for internal communication (e.g., forwarding the client's certificate and creating a JWT that represents the external service identity from the certificate's common name).

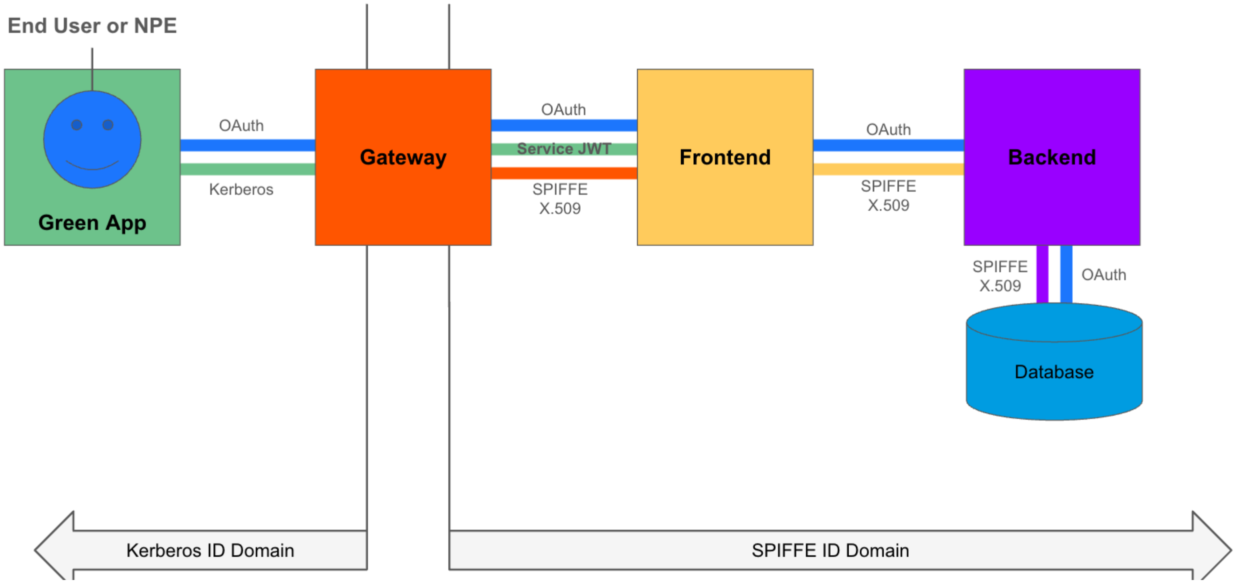

Fig. 4. Handling API Calls with User Identity & No Service Identity

However, the gateway's service identity is already in place between the gateway and the first service performing identity-based segmentation. For that first hop, three identities need to be handled on the request: the gateway's service identity, the service identity of the external service, and the end user's identity. As before, external service authorization can be performed via the gateway and simply drop the external service identity. Services should support validating both the end user and a workload identity via metadata from the request in addition to validating workload identity via the transport (e.g., mTLS certificates).

For example, suppose that an organization A) uses a SPIFFE X.509 identity via mutual TLS for service identity as a service mesh does, B) uses a JWT bearer token for user identity, and C) chooses to represent external service identity as a JWT token attached to the request. The mesh can then enforce that the gateway forward traffic to the service via (A), authenticate the service JWT and authorize the external service (C), and authenticate the end user (B) before forwarding a request to the application. This would fully support authenticating and authorizing all of the communicating parties, and the service in question would not need to be aware of the external service identity or credential. They would simply need to manage a policy of "allowed external service callers" alongside their set of "allowed internal service callers."

2.7.4. Requests With Both User and Service Identities

In the best case, the legacy systems in question are already doing nearly the right thing in that they have both an end user and a service identity attached to requests. However, since they are a legacy system, those credentials likely do not fully conform to the credentials expected by the parts of the system implementing identity-based segmentation. In that case, those other credentials will need to be translated into the canonical form expected by services performing identity-based segmentation in the infrastructure. Essentially, the user's authenticated credential should be exchanged with an identity provider for the canonical form expected by the identity-based segmentation portion of the infrastructure (e.g., a JWT bearer token), and the external service's identity should be represented to the internal system as a token so that the policy can be enforced on all three identities in the first hop.

2.7.5. Reaching Out to Other Systems

A similar problem presents itself in reverse when a service that performs identity-based segmentation needs to reach out to legacy systems that expect legacy credentials. One option is to integrate modern applications with legacy credential systems so that those applications can fetch the legacy credential they need, which can significantly delay the sunsetting of those legacy systems. A better option is to perform a credential exchange on traffic leaving the identity-based segmentation subset of the infrastructure.

Rather than integrating the API gateway with a variety of identity providers to canonicalize inbound credentials, a gateway with a variety of identity providers to fetch outbound credentials can be integrated instead. For example, an external SaaS API may expect a cloud provider service account as credentials. An egress gateway can be deployed to authenticate and authorize credentials used inside of the organization (i.e., identity-based segmentation) and then exchange the internal identities for the external identities needed by the other system. In this way, services that perform modern identity-based segmentation can integrate with legacy systems with little impact and minimize any code dependencies on those legacy systems.

2.7.6. Mitigating the Confused Deputy

One of the biggest risks in any scheme that involves credential exchange is a confused deputy [26], where one caller can trick the "deputy" responsible for handling credentials into using credentials that belong to another caller on its behalf, most often to escalate privileges. Any system that brokers multiple credentials needs more and better authentication and authorization before allowing credentials to be accessed.

An alternative approach is to break down the deputy into separate entities that hold only a single credential and map closely to a single application or service. This is the core idea behind the service mesh's sidecar presenting a service identity on behalf of the application: because the sidecar is one-to-one with a service instance, a service's identity cannot be confused for another at runtime. This same idea can be applied to API and egress gateways. Deploying them granularly — ideally per application — can minimize or eliminate any mixing of credentials, thereby mitigating any risk of a confused deputy. Section 4 discusses API gateway deployment patterns at length.

2.7.7. Identity Canonicalization

Canonicalizing credentials is really canonicalizing the identity domains for which one needs to write policy. Integrating identity providers to standardize credentials at the gateway inherently brings those identities into two identity domains: one for users and one for workloads. This allows for concise and consistent sets of policy that govern access to other services and user access to data. Having both policies in place implements identity-based segmentation and dramatically improves security posture.

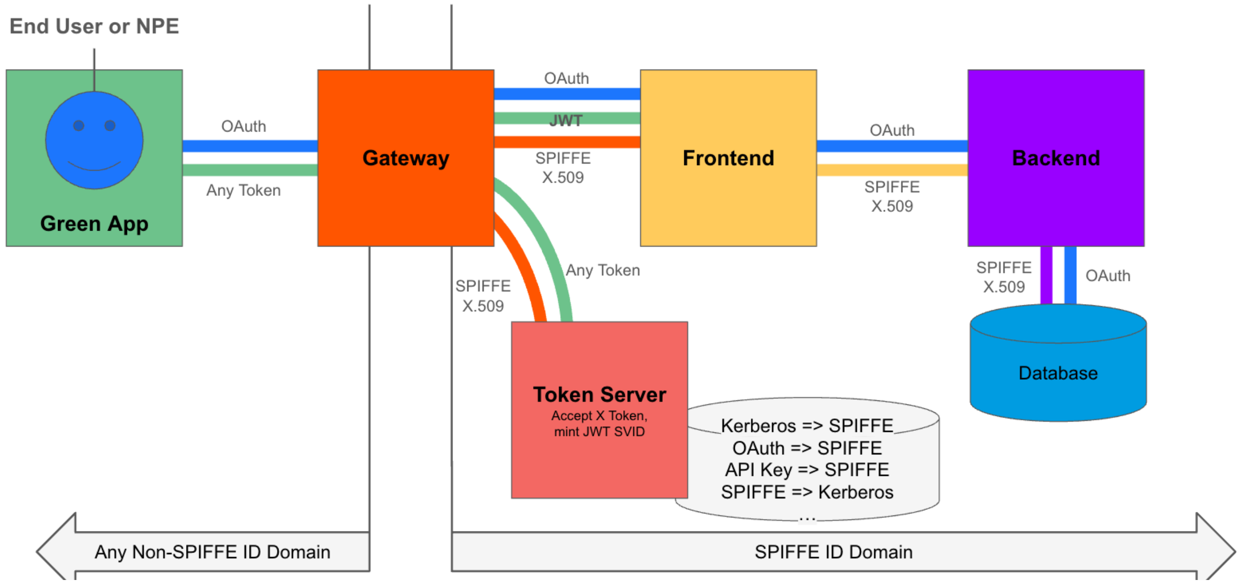

Fig. 5. Identity Canonicalization for Handling API calls

For most organizations, implementing credential canonicalization will require either adopting an identity provider wholesale and standardizing on that throughout (including working out legacy integration so that legacy credentials can be used to get credentials via the new provider) or performing identity exchanges, as described in this section. The API gateway is ideally situated to enforce either choice. Performing identity exchanges also requires a mapping of identities across domains as well as a "token server," which uses that mapping to mint credentials.

3. Recommended Controls for APIs

APIs are the language of the systems we build. They define the “grammar” as well as the “nouns” and “verbs” *any* user of your system works within. As attackers move up the stack, it’s more important than ever to elevate our security posture to think and manage policy in terms of those “nouns” and “verbs” our APIs expose.

In their earliest form, controls for APIs focused primarily on encryption in transit while delegating most other concerns to the application. Over time, a variety of challenges have emerged that necessitate the evolution of controls, including:

- The distributed nature of modern enterprise applications, which span multiple on-premises and cloud environments and communicate over the network using APIs
- The requirement to build robust systems that work around transient failures and handle large volumes of traffic
- An increasingly complicated API surface driven by business needs to integrate more deeply with partners and expose richer functionality to users
- Increasingly sophisticated attackers who have moved up the stack from low-level exploits and DoS attacks to application-level attacks that leverage the APIs that systems use to function

Controls for APIs should cover all of the APIs in the organization, including those exposed to end users, those exposed to partners, and those that are only intended for “internal” consumption. This document’s controls are structured into two primary sections based on the iterative API life cycle discussed in Sec. 1:

1. Pre-runtime protections, which should be applied during design, development, and testing. These include:
    a. Creating a well-defined specification for the API’s contract using some interface definition language (IDL) (e.g., OpenAPI, gRPC, Thrift)
    b. Defining request and response schemas as part of that API specification
    c. Defining valid ranges of values for fields of each request and response
    d. Tagging the semantic type of each field of each request and response
    e. Creating and maintaining an inventory of these API specifications across the organization, including ownership information
2. Runtime protections, which should be applied to each request and response to the API at runtime. These include:
    a. Encryption in transit
    b. End-user authentication and authorization
    c. Service-to-service authentication and authorization

  d. Request and response validation
  e. Resource-consumption mitigations, including rate limiting, timeouts, and circuit breaking
  f. Telemetry (e.g., logging and monitoring) to assess enforcement and detect attacks

Within each section, the controls are grouped into “basic” and “advanced” categories:

- Basic protections should be pursued immediately with the goal of obtaining basic insight into the APIs that exist in an organization (*Identify* in NIST CSF [12]) and can be used to implement essential best practice controls (*Protect*). Generally, basic protections do not require deep introspection of the API’s request and response payloads but operate at the connection or request metadata level (i.e., on HTTP headers rather than the HTTP body).
- Advanced protections perform deeper analysis on requests and responses. Many of these policies require payload inspection, which is CPU- and latency-intensive. The goal is to enhance basic *Protection* and begin to cover the *Detect* and *Respond* functions in NIST CSF [12]. Addressing these concurrently with basic controls is recommended, but the basic protections may provide the most benefit for resource-constrained organizations.

All organizations should move immediately to act on basic controls, while advanced controls should be evaluated by the organization and applied to APIs based on risk profile.

3.1. Pre-Runtime Protections

All API controls must be well-defined and inventoried.

3.1.1. Basic Pre-Runtime Protections

REC-API-1: All APIs must have a specification in the form of a document that describes what endpoints the API exposes (“API spec” for short). To begin, the API spec can be a literal document, a set of internal wiki pages maintained by a team, or something similar. However, it should eventually migrate to a state-of-the-art IDL.

REC-API-2: API specifications should use a well-defined IDL (e.g., OpenAPI for HTTP/REST, gRPC for protobuf, Thrift, SOAP for XML).

- REC-API-2-1: API specs and implementations should conform to industry best practices (e.g., a Create-Read-Update-Delete [CRUD] API exposed as HTTP/REST should map the CRUD endpoints to the HTTP verbs POST, GET, PUT, and DELETE, respectively) for consistency [13].

REC-API-3: Request and response schema for each endpoint should be defined by the API specification, including validation guidelines for the values of each field of the request and response (e.g., “the name field is a string and must be shorter than 100 characters”). Additional information makes integration easier and less error-prone for clients and presents the opportunity for automated enforcement, such as the maximum latency (e.g., “the server will drop requests that take longer than 5 seconds to process”) and rate limits (e.g., “by default, 5 calls per minute are allowed”).

REC-API-4: Organizational API inventory of all internal and external APIs should be maintained. This is in line with the *Identify* directive of the CSF [12]. That inventory should include:

- Each API’s specification, though the inventory does not need to be the *API documentation*
- Ownership information about the API to simplify the translation of runtime problems to organizational response
- Runtime information to enable operations and security teams to understand the impact of each API (e.g., service instances, instance IP addresses, runtime service ID, traffic volume, rate of requests and errors, the status of policy enforcement)

3.1.2. Advanced Pre-Runtime Protections

REC-API-5: Request and response validation in the schema should be included in the API’s specification (e.g., a string field must be non-empty and shorter than 255 characters, or an integer value must be non-negative and less than 2 million). This simplifies documentation and enables runtime tooling to validate request and response schema and syntax.

- Use primitive types in API schemas to reinforce this. For example, if a value is always semantically positive, model it in the schema as an unsigned integer rather than a regular integer (e.g., protobuf’s “uint” rather than “int”). Negative values are then disallowed by construction without any validation needed [14].
- This principle extends to zero or default values as well. Users (malicious or not) will frequently omit fields that the application expects. One approach is to this is annotating fields as “required” or “optional” and rejecting requests with zero values for required fields. However, the application must handle missing optional fields. A second approach adopted by both Golang and protobuf/gRPC is to define “zero values” for each primitive type. The goal is that application code must either handle the zero value for each field or reject the request with a validation error.

REC-API-6: Annotate each field as public or internal for each request and response or with the level of trust or permission required for access. These annotations simplify documentation and enable runtime tooling to remove trusted data for untrusted callers as a cross-cutting policy rather than something that must be built into the business logic of each service. An in-application approach is much harder to implement correctly and to audit in practice.

- REC-API-6.1: Annotate endpoints and fields with permissions required to enable the use of tooling to automate fine-grained per-field authorization checks. Those authorization checks could then be performed by the API serving infrastructure on behalf of the application or via a common library in the application with standard logging and metrics to facilitate easy audit and ensure continuous enforcement. Once the annotations are present, a variety of runtime implementations are possible.

REC-API-7: Annotate each field with its semantic type to indicate fields that contain sensitive information, such as personally identifying information (PII), protected health information (PHI), or payment card information (PCI). This enables runtime systems to track data flow through the system, trigger alerting, and apply cross-cutting policy to ensure data does not leak across inappropriate boundaries.

REC-API-8: Include runtime information in the API inventory with ownership (REC-API-4). This becomes substantially more valuable when annotated with runtime information (e.g., service instances and their IP addresses, runtime identities of the service instances, metrics or health information for the service, runtime metrics for traffic between services). This information can help security identify the blast radius of an event, operations to identify problems and root causes, and application teams to understand their application’s behavior. Correlating this information with the APIs being served makes it simple to link clients to servers as the problem is traced back to its root.

3.2. Runtime Protections

For runtime protections for APIs, apply zero trust principals as a baseline, and augment them with additional policy on requests and their payloads.

3.2.1. Basic Runtime Protections

REC-API-9: All runtime communication must be encrypted, even when the API is “public data” or otherwise unauthenticated. This is necessary to ensure that data has not been tampered with (integrity) and to prevent eavesdropping (confidentiality). Details on encryption in transit can be found in SP 800-53, control SC-8 [15] and SP 800-207A, control ID-SEG-REC-1 [6]. Details on cryptographic algorithms and key lengths can be found in SP 800-57 [16] and FIPS 140-3 [17].

REC-API-10: Perform general request and response validation policies (e.g., WAF, bot detection, DoS mitigation) to mitigate malicious payloads and unrestricted resource consumption. These can and should be executed early in the API serving stack to protect other components (e.g., authentication system) from DoS. Since these protections are general and cross-cutting, there is little risk of unintentionally leaking sensitive information.

REC-API-11: Authenticate the calling user and service, as described in SP 800-207A, controls ID-SEG-REC2 and ID-SEG-REC4 [6].

There are (at minimum) two identities in every API communication: the software calling the API and the end user of that software. For example, it is common to use an API key to identify calling software and an OAuth Bearer token to identify the end user. This is true even if the end-user identity is an NPE (i.e., internal software calling other internal software should use something like a service account to identify the user making the requests). The service identity may contain information (e.g., the device being used to access the system) in addition to a token from the software itself (e.g., an API key).

- REC-API-11-1: Identities must be cryptographically verifiable and should not use weak signing algorithms (e.g., no JWTs with "alg: none," weak algorithms, or short key-lengths). SP 800-57 [16] discusses the strengths of cryptographic algorithms and the necessary key lengths for each.
- REC-API-11-2: Authentication should use standard mechanisms whenever possible. For example, end user authentication should use a mechanism such as OpenID Connect (OIDC), OAuth2, or SAML. Services should use a mechanism like SPIFFE SVIDs, JSON Web Tokens (JWTs), API keys, or similar.
- REC-API-11-3: Tokens must support expiry so that credentials are cycled regularly. Checking for expiry must be an inherent part of token validation. For example, when processing JWTs, the "exp" claim RFC 7519 [18] must be checked. Similarly, when processing an X.509 SVID, check the validity period's "Not Before" and "Not After" [19].
- REC-API-11-4: Return opaque tokens to untrusted systems. It is common for credential tokens to encode information about the internals of the system (e.g., minting a JWT to represent a user in the infrastructure that includes claims that represent the user's capabilities in the system). This is a common scheme to simply and reliably enforce authorization per hop: validate the JWT, and check whether it contains the "claim" that represents the permission for an API endpoint. These claims encode all local operations that can be performed with data from the request and the local application.

  Returning a token with these details to an external user may risk leaking information about the internals of the system. This is where the following issues become critical to the safety of the API: how permissions are modeled, the set of internal permissions/claims that map to a given external API endpoint, and information about the path that the request traverses through the infrastructure.

REC-API-12: Authorize the calling user and service for each identity on the request, including whether the calling software system is allowed to access the API endpoint and whether the end user is authorized to take the action on the resource represented by the endpoint. See SP 800-207A [6], controls ID-SEG-REC2 and ID-SEG-REC4.

Getting these authorization checks correct is one of the most common mistakes in API security [7]. REC-API-6 discusses annotating each request or endpoint with the permission required by the end user to call that endpoint on a resource. With annotations like those in place, runtime tooling can be implemented to ensure that those annotations are transformed into runtime permission checks against the authorization system. Combined with a robust DevOps process to ensure that annotations are present on APIs before they can be deployed, there can be a high degree of assurance that the correct authorization is being performed at the platform level. The idea of using the service mesh to achieve this is discussed in SP 800-204B [3].

REC-API-13: Validate each request and response per the API schema before it is processed by the business logic (e.g., ensure that the request has a "name" field that is a string and no other fields). This ensures that applications only receive well-formed input and minimizes a class of errors and data leaks due to validation inline in the business logic. Additionally, validate that each response from the server conforms to the expected response schema to help prevent a variety of data leaks, abuses, or mistakes.

REC-API-14: Authenticate, authorize, then validate in that order to minimize the risk of leaking data to attackers, since validation messages are at especially high risk of leaking information. For example, rejecting a request with a validation error for using a duplicate user-supplied name as another user may unintentionally leak information to callers regarding the existence of a resource. A likely mitigation may be an underlying per-user segregation of user-provided data, which often requires business logic changes in the application. Generic validations (REC-API-10) are exceptions to this because they are not business logic-aware and do not risk leaking information. They can be safely implemented by the platform ahead of authentication, which is often desirable to help protect the authentication and authorization systems from DoS and other attacks.

REC-API-15: Enforce limits on API and resource usage. API gateway teams should provide reasonable defaults for the organization, and application teams should be able to enforce their own, more fine-grained limits in their application or leveraging the platform. Those limits should include:

- REC-API-15-1: Rate limit all API access for all callers to ensure fair utilization across users, help with capacity planning, and mitigate the risk of unrestricted resource consumption. See REC-API-16 for recommendations on specific rate-limiting implementations.
- REC-API-15-2: Apply timeouts to all requests, including the API gateway. This should be done at the TCP level, where connections are automatically timed out after a modest time (e.g., 5 minutes) rather than the kernel's default of more than one hour per connection. Timeouts should also be configured at the application level. If a required operation should complete in five seconds as part of the API contract, set a 6-second timeout for it. This ensures that the resources in a service do not wait for a response that will never arrive.
- REC-API-15-3: Apply bandwidth and payload limits to enforce maximum request and response sizes. The "correct" limit is highly contextual and based on the organization and application (e.g., a bank will have very different expectations than a video streaming company). This helps avoid a variety of risks related to malicious input and DoS.
- REC-API-15-4: Validate and limit user-supplied query parameters (e.g., amount of processing done, size of their response based on user input), especially in the context of what the system can support and what is typical for users of the system. For example:
  - The number of elements returned per page of a paginated list API. If a typical user has 100 items, cap the maximum number of elements per page to 1000.

  - Time ranges in dynamic queries. If a system is intended for viewing recent events, and the user can provide a time range, limit that range to the last 30 days rather than allowing the user to query "from 1972 onward."
  - GraphQL and similar API facade systems that support query languages over many APIs should have limits on the queries that users can execute (i.e., approved or predefined queries only) and caps on the number of outbound calls allowed in the execution of a single query.

<u>REC-API-16: Rate limiting recommendations</u> are one of the most effective tools to mitigate unrestricted resource consumption and can increase the challenge and discoverability of many attacks with a goal of leaking sensitive information via data exfiltration from API calls (e.g., scraping all chat logs from an organization with a script impersonating a chat client). Most organizations apply some type of rate limit to "external" traffic, but it is equally important to rate-limit internal callers. It is very easy to unintentionally cause a DoS on an internal system with poorly conceived code. It is equally critical to consider the limits placed on internal software that call out to external systems (see Sec. 2.6.2).

The following recommendations on rate-limiting configuration address common pitfalls and misunderstandings:

- <u>REC-API-16-1: Rate limits are not quotas</u>. A quota is a usage limit on an API over an extended duration (e.g., per month) that is associated with a user's payment or billing structure. Many organizations have "API usage tiers" that map prices to higher per-month limits. These quotas need to be strictly enforced and are typically used to generate billing reports that are sent to customers. In contrast, rate limits are intended to protect the system from overuse and help ensure fair usage across separate, concurrent callers. Rate limits do not need to be exact in the way that quotas must be.
- <u>REC-API-16-2: Rate limits for total load provide little benefit</u> and should be dimensioned by user (e.g., 83 requests per 5 minutes per use*r*) using the source IP address or end-user credential as the key. Rate limits without a user dimension (e.g., service can receive 1,000 requests per 5 minutes total) are not particularly effective and allow some users to impact others (e.g., DoS risk). This is true even when total limits are dimensioned by service instance (e.g., a single instance cannot receive more than 100 requests per 5 minutes). Circuit breaking functions must be used to provide protective limits on concurrency for a service instance. More information on circuit breaking and other resiliency and load-shedding techniques can be found in 800-204A, Sec. 2.3 [2].
- <u>REC-API-16-3: Rate limits should be short in duration</u> (e.g., per 60-seconds, per 5-minutes). A rate limit is defined as the number of calls allowed over a time period (e.g., 24,000 requests per 24 hours; 1,000 requests per hour; 16.5 requests per minute). Most systems allow for the configuration of both the number of calls and the amount of time over which they are allowed.

  However, there are two problems with per 24-hour rate limits. First, they cause outages for callers that resolve themselves when the rate limit server resets for the next 24-hour period, even if the rate limit was originally set correctly based on the client's expected usage. A successful API ecosystem will see the increased usage of APIs over time, which results in the increased usage of their dependencies and those APIs. This is the typical organic growth of API usage. Adjusting rate limits before they caused outages is almost never a priority for application teams, so over time, clients may see the API begin to randomly fail with 400 errors. Second, per 24-hour rate limits can result in spiky traffic for the service, where a client consumes the entire 24-hour limit over a very short time and causes a heavy load on the services.

Shorter time limits allow clients to experience a few intermittent failures every minute or five as their traffic grows organically rather than total failure with per 24-hours. Additionally, the system will experience smoother traffic overall because a single client must pace their consumption over a longer duration, resulting in less load from each client at any given time.

<ins>REC-API-17: Fine-grained request and user blocking</ins> allows the API serving stack to block individual users via their end-user credential and/or network address. This is a key capability in enabling an effective response in the face of an ongoing incident (see the *Respond* function in the CSF [12]). The actual enforcement can be handled by separate components (e.g., network-level blocking implemented by a firewall or the load balancer; credential-level blocking implemented by the API gateway, bot/abuse detection systems, or the authorization system). For relevant information on these techniques, refer to SP 800-53, AC-3 [15] and SP 800-204B, Sec. 4.6 [3].

<ins>REC-API-18: API access must be monitored</ins> to ensure that the API serving stack provides sufficient telemetry to assess the availability of APIs and to ensure that policies are being enforced. The traditional triad of logging, metrics, and distributed traces is recommended. All three should be tagged with information about the API being accessed in addition to the runtime service so that service calls can be traced back to APIs.

For the API gateway itself, a range of signals should be produced to enable the identification of:

- Basic communication information, like the information included in the Common Log Format [20] (e.g., who called, what method, from what origin)
- Health (e.g., rate of requests, rate of errors, latency) per API and API Endpoint
- Enforcement results per policy class (e.g., requests allowed or denied due to missing or incorrect authentication or authorization checks, requests blocked due to rate limiting) to assess the aggregate enforcement of each policy
- The health of the services behind the API gateway

General information on audit and logging requirements can be found in SP 800-53 [15], AU-2 Event Logging, AU-3 Content of Audit Records, and AU-12 Audit Record Generation. Information on service mesh telemetry, which can be used for audit and logging, can be found in SP 800-204A [2], SM-DR21 through SM-DR24.

3.2.2. Advanced Runtime Protections

REC-API-18: Field-level validation using API schema annotations can be used to validate the values of requests and responses at runtime. This is beyond the basic syntactic validation of REC-API-13 (e.g., "there is a name field, and it is a string") and more like semantic validation (e.g., "the name field must not be longer than 100 characters," or "the amount field must be positive and less than 2 million"). This can be implemented by the API gateway as part of a cross-cutting policy. An API spec is required (REC-API-2) and should be in a central inventory (REC-API-4). The API gateway team can then enforce the validation of all requests traversing an API gateway. This reduces the risks of insufficient input verification and leaking sensitive information compared to ad hoc, error-prone implementations in each application or standard implementations embedded in the application itself via SDK, which tend to be difficult to update. A timely update is an imperative for infrastructure that enforces security policy.

REC-API-19: Authorization and filtering using API schema annotations enforce access to resources and fields per caller. In this case, the API gateway itself is the policy enforcement point, and it defers to an authorization system to make decisions. The information from the API schema is enough to extract credentials from the request, identify the target endpoint and its associated tags/permissions, and use those to form a call to the authorization service (e.g., "is the request's end user allowed to perform the endpoint's permission on the object targeted by the request?"). The API gateway can then enforce the result of the call at runtime. There are at least three levels of assurance that can be achieved, and each build on the previous one to further mitigate risks at increased runtime or development-time cost:

- REC-API-19.1: Resource-level authorization as a cross-cutting policy should be enforced on all requests using endpoint-level annotations that define the permissions required to call the endpoint (REC-API-6.1). This can be done at the platform level leveraging the API gateway. When combined with a decentralized gateway pattern (Sec. 4.3), this implements ID-SEG-REC-4 [6] at every hop.[1] This also helps prevent and potentially eliminate missing authorization (Sec. 2.2), depending on the organizational guardrails in place. For example, an organization can build an API inventory by mandating an API spec with endpoint-level permission annotations as part of each app's "ticket to the platform" (i.e., the data that an app team needs to submit to run their application on the organization's infrastructure and platform). Combined with standard patterns for authentication (REC-API-11), this can ensure that the correct authentication and some authorization are performed. However, additional organizational controls are required to ensure that the permissions are correct and sufficient in order to fully mitigate risks around authorization (see Sec. 2.2).

  Achieving correct and sufficient authorization at the resource level is likely all that most organizations need to achieve. It mitigates the predominant risks identified by the OWASP API Security Top 10 [7] with respect to authorization. Moving beyond this level

---

[1] Other patterns have a wider perimeter and are susceptible to the API gateway being bypassed. Therefore, they do not satisfy ID-SEG-REC-4.

of assurance into REC-API-19.2 and REC-API-19.3 shifts the focus to mitigating the risk of leaking sensitive information.

- REC-API-19.2: Field-level visibility as a cross-cutting policy can leverage basic "Public" and "Private" annotations on each field. The authorization check effectively asks whether data should be visible to "external" callers.[2] These coarse-grained Public/Private annotations are particularly effective on common types shared across many APIs in the organization. For example, a standard error reporting pattern used by all APIs can leverage field-level annotations to differentiate "user" facing errors versus "developer" facing errors, mitigating the risk of leaking sensitive information via errors. The gRPC Status proto [21] is an example of a consistent error reporting pattern. In the gRPC case, field-level annotations would reside in the message used for the status's "details."

- REC-API-19.3: Field-level authorization as a cross-cutting policy can be leveraged to perform fine-grained field-level authorization (REC-API-6.1). This extends the idea of REC-API-19.1 down to the level of each individual field of the response and allows for the filtering of API objects per-use to implement sophisticated access control schemes.

  While this kind of approach offers a very high level of data security, it causes a sharp increase in the number of policy checks that the authorization system must perform and requires active participation by application developers to keep permissions per field up to date as the application evolves. For example, a resource-level authorization check requires one authorization decision per request. A field-level authorization check requires one authorization decision for the request plus an additional decision for each field of the response. Even an object with a modest number of fields (e.g., 5) results in whole-number multiples more policy decisions made by the authorization system. For developers, the purpose and therefore permission of an endpoint rarely changes, but the fields of the request and response objects for that endpoint regularly evolve over time. This makes upkeep for permissions at the field level more expensive for application developers versus endpoint-level annotations (REC-API-19.1).

  As a result of the cost and load on the authorization system, this level of fine-grained checking is typically only used in the most high-risk situations and only by sophisticated organizations.

SP 800-204B [3] discusses the advantages of using a decentralized API gateway architecture when implementing fine-grained authorization checks. When choosing to implement these authorization policy checks under the centralized and hybrid patterns, care must be taken to ensure that the gateways are not bypassed. For example, a service-level authorization policy could disallow any traffic except from the API gateway as a means of defeating an attempt to bypass gateway checks via pivoting inside the infrastructure.

REC-API-20: Traffic monitoring and policy using semantic field labels can log and monitor the flow of sensitive data in a system. Further, the API Gateway can be used as a policy

---

[2] REC-API-19.1 i focuses on requests, while this control focuses on the data that an application returns to callers in responses. They are complementary controls.

enforcement point to control the flow of that data, potentially blocking traffic flows that transit significant amounts of data. Ultimately, with annotations and enforcement in place, the flow of sensitive data in the organization can be governed by mandatory access control (MAC) policies. A MAC policy is enforced by the authorization system, regardless of the user or resource in question. For example, while not explicitly stated as a hard rule in PCI DSS, a MAC policy followed in implementation of systems handling PCI data is that they should be isolated from systems that do not implement PCI DSS controls to maintain security and prevent potential breaches. Such a MAC policy can be enforced with a combination of understanding PCI-compliant services in the infrastructure and data tags on the semantic types of data that flow through the system.

<u>REC-API-21: Non-signature payload scanning (for generative AI APIs)</u> analyzes request and response data for sensitive information that may not be a literal attack signature. Tools typically analyze (e.g., via regression, AI, simple matching and word filtering) the responses returned by servers to score the risk that they contain sensitive information and take action to block that traffic. Increasingly, AI agents are being deployed to assess the risk of data generated by other agents. At a high level, this technique is like a web application firewall (WAF), but WAFs are fundamentally signature-based, while these analyses are fundamentally content-based.

This is a general category of data egress analysis that is relevant across all APIs, but it has become increasingly important with the growth of generative AI. Generative agents are frequently trained on business-sensitive data or have insight into sensitive business operations and operational data, and they are increasingly exposed to the organization and externally as APIs. From the inception of generative AI agents, a variety of prompt injection attacks [22] have been created to exfiltrate data via these generative models.

Tools for performing non-signature payload inspection should be used whenever an organization is handling data returned by their system, especially when that data is generated on demand (e.g., by AI agents). In most cases outside of dynamically generated output, implementing simple semantic and syntactic validations (REC-API-13, REC-API-18) will typically provide an organization with more risk mitigation for a lower runtime and operational cost.

- <u>REC-API-21.1: Semantic data discovery</u> tools are typically very good for identifying the type of information flowing through a system (e.g., string, email address). Building the inventory of APIs and the developers adopting well-defined API schemas with meaningful annotations takes time. Runtime tools such as these are very helpful for initial discovery, ensuring that rollout is complete across all services, and ensuring that services stay in compliance after the policy is rolled out. When it is reasonable to leverage due to compute and latency constraints, an organization benefits from inspecting traffic for sensitive data flow, even beyond field-level annotations.

<u>REC-API-22: Fine-grained blocking for specific requests</u> can prevent a DoS or service crash. These bad inputs can often trigger a cascading failure [23], but the queries may not be malicious in nature (e.g., users using the system in ways that it was not intended or designed for). In cybersecurity, this is sometimes called the “query of death” (QoD) [24]. These tools help mitigate the risks of unrestricted resource consumption and malicious input validation.

As a system grows in size and complexity, it is necessary to be able to pin-point block these kinds of queries to keep the system stable and available. Depending on the complexity of the query and environment, it may be possible to leverage a WAF or non-signature payload scanning tools to block some types of QoDs. However, application code changes may be required — sometimes even rearchitecting the application itself — to mitigate the impact of these kinds of queries.

The detailed controls in this section fit into broad classes, and their association with the DevSecOps phases is discussed in Appendix B. This emphasizes the observation that APIs should be treated as any other software and go through an iterative, continuous life cycle.

4. Implementation Patterns and Trade-Offs for API Protections

Regardless of the mechanism or architecture of an API and its services, there is a core set of capabilities required to realize the controls outlined in this document:

- Authentication and authorization
- Request and response validation
- Rate limiting
- Circuit breaking
- Error handling
- Logging and Monitoring

In addition to these core capabilities for security, APIs that serve infrastructure typically deal with other common concerns:

- Service discovery
- Routing
- Protocol conversion
- Caching

Three components are needed to provide this functionality to serve an API:

1. A *gateway* to implement the API-oriented policy
2. The *service* itself to implement the API's business logic
3. A method to get traffic to gateway instances (e.g., DNS and a network load balancer) to facilitate service discovery, load balancing, and network reachability to horizontally scaled instances of the gateway itself

For example, if the Gateway functionality is implemented via a Kubernetes ingress routing to a pod (i.e., the service instance), then callers outside of the network to reach the gateway will require the cloud provider or data center network team to provision a network load balancer in front to route network traffic to the Kubernetes load balancer service.

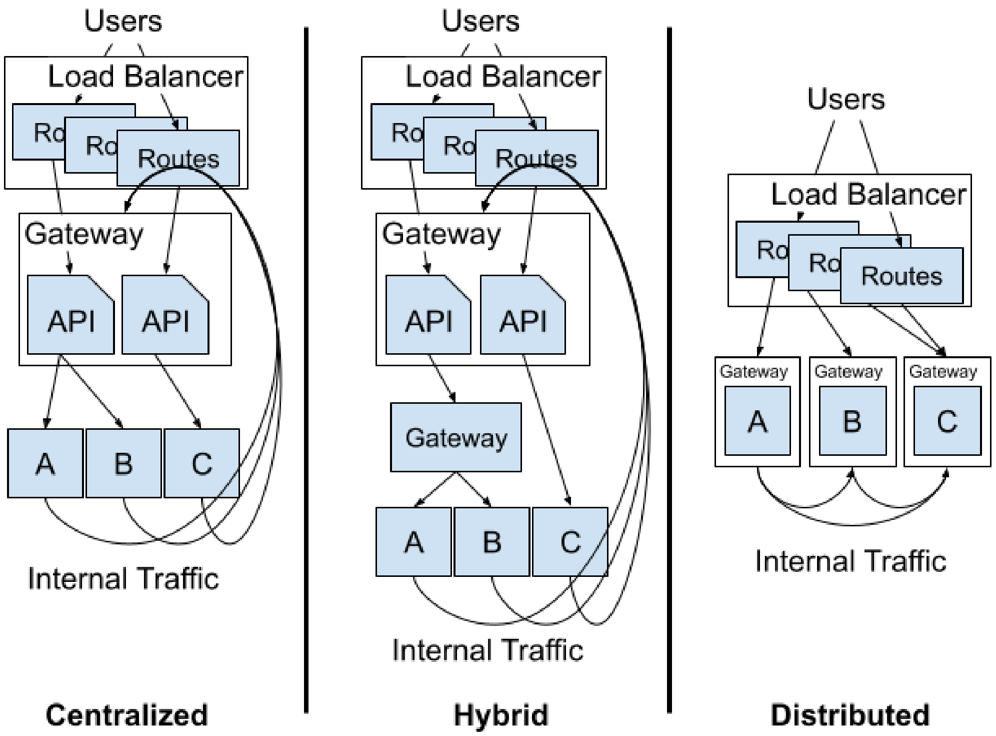

Fig. 6. API gateway patterns

Three patterns have been developed by industry to implement these capabilities:

1. Centralized gateway — Protections for all APIs in the enterprise are implemented by a single shared component: an API gateway.
2. Hybrid deployment — Cross-cutting policies (e.g., authentication) are implemented in the centralized shared gateway, but application-specific policies (e.g., authorization) are implemented in the application itself or by components owned by the application team.
3. Decentralized gateways — All policy checks are performed by gateways dedicated to each application, often deployed beside each service instance.

All three patterns can achieve all of the controls outlined in this document and be used by organizations to operate their APIs safely and confidently. Further, many of these patterns may be in use within a single organization. This section explores the engineering design trade-offs that each pattern provides in terms of risks and operational overhead.

Many API gateway products provide management capabilities, such as API key issuance, discovery documentation (i.e., API definition) hosting, documentation for client developers, and support for quotas and billing tiers. These are all valuable features in the enterprise setting, but all of them can be supported across any implementation pattern and are therefore not addressed in this section.

4.1. Centralized API Gateway

The centralized API gateway pattern implements protections for all APIs with a single component: an “API gateway” that is often deployed close to the perimeter of the system. External traffic enters through the gateway, typically via a load balancer. Internal traffic “hairpins” through the gateway as well, which facilitates service-to-service communication inside the infrastructure. That internal, service-to-service traffic may also have to traverse the load balancer for some service instances. Fig. 5 shows a common configuration for a centralized API gateway pattern.

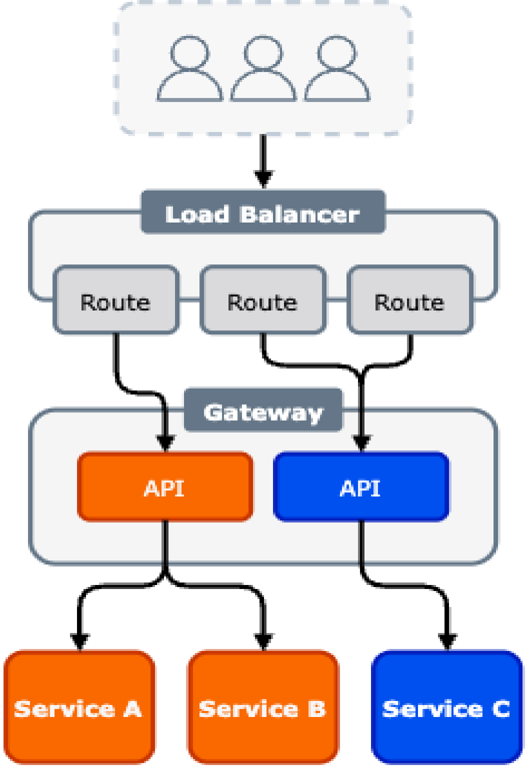

Fig. 7. Centralized API gateway pattern

An API gateway is typically a software application that can be scaled horizontally (i.e., more instances can be deployed side by side). This is one of the reasons why an API gateway often sits behind a load balancer, even for internal service-to-service traffic use cases.

Advantages of this pattern include:

- A single policy enforcement point that is easy to monitor and audit, making it simple to verify that policy is enforced for all traffic that traverses the gateway.
- Implementation matches the organizational structure. Typically, large organizations have a single API team that owns the centralized gateway component. That team is responsible for and able to execute on when an API is available, which API endpoints are

failing, whether policies are being enforced, whether the configuration up to date, and other issues.

- Streamlined setup for application developers who need to "onboard" their API but do not need to deploy or maintain any additional runtime components.

Disadvantages of this pattern include:

- Shared fate outages. Because there is a single component, an outage of that component causes an outage for all APIs, which can be problematic for mission-critical APIs that need to operate continuously.
- Noisy neighbors, where traffic consumes resources for some APIs and increases latency for all APIs. In the worst case, one application team may submit invalid configuration parameters for a service that may crash or cause DoS on the API gateway, triggering a shared fate outage for other APIs.
- Long change lead times due to managing how the changes to an individual team's API configuration impact the shared gateway. This is a frequent side-effect of controls added to mitigate shared fate outages and noisy neighbors.
- Cost attribution. All requests are handled by the central gateways, and resources spent per request per API (e.g., on payload validation) are uneven. Therefore, it can be difficult to attribute API gateway runtime costs to internal application teams. This can be a problem for companies that implement an internal resource economy for planning by assigning cost centers for each application team.
- Caching the results of policy decisions at runtime becomes critical when implementing the policies outlined in this SP due to the sheer number of policy checks required. Caching both increases client-perceived availability and reduces the load on key systems, like authentication and authorization. However, two layers of load balancing (i.e., network load balancer to API gateway and API gateway to service instance) tend to result in poor cache hit rates across policies enforced by the API gateway and for user data in the application layer itself. While some techniques can be used to mitigate this (e.g., distributed caches or streaming connections), they generally add additional development or operational overhead for the application team, API gateway team, or both.
- Because a shared gateway is located at the perimeter, it can be bypassed (e.g., via an attacker pivoting inside the perimeter), which in turn bypasses the policy checks enforced by that gateway. This can be mitigated with techniques like service-to-service access policies that ensure that applications only receive traffic via the centralized

gateway or by attaching proofs (i.e., credentials) to the request that allow an application to authenticate that the request was handled by the gateway.

4.2. Hybrid Deployments

Hybrid gateway deployments split policy enforcement responsibilities between a centralized gateway and the applications themselves. Cross-cutting policies (e.g., authentication, service discovery, routing, rate limiting, caching) are handled by the centralized gateway. Application-specific policies (e.g., authorization, request and response validation, protocol conversion, error handling, logging, monitoring) are handled by the application team. This can manifest in the application itself (e.g., gRPC) or as a separate deployment that handles traffic before the application (e.g., GraphQL or Spring Cloud Gateway). As with the centralized pattern, all internal and external traffic between applications must first go through the centralized gateway and, in some instances, through the load balancer. Fig. 6 shows the schematic diagram of a distributed gateway pattern.

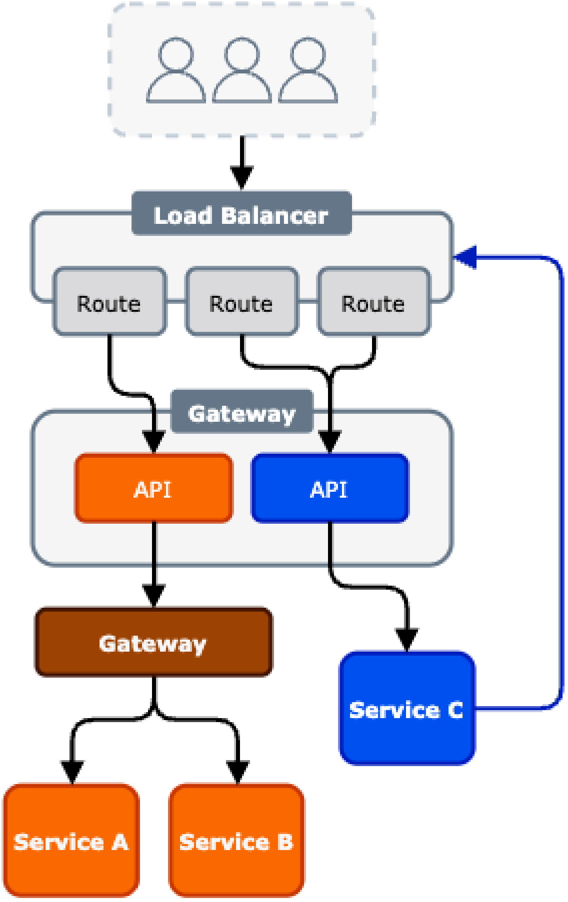

Fig. 8. Distributed gateway pattern (hybrid deployment)

Overall, this pattern behaves similarly to the centralized API gateway pattern, except that some of the most failure-prone parts of the centralized pattern are delegated to the application teams. This streamlines API gateway operations and enables app teams to move at their own pace. However, it also shifts the responsibility for some runtime operational and security concerns from the API gateway team to those application teams.

The exact split of responsibilities between gateway and application (e.g., sidecar in a service mesh architecture) can vary greatly across different organizations based on their risk profiles and past experiences. Typically, the gateway takes responsibility for:

- Authentication
- Rate limiting
- Circuit breaking
- Service discovery
- Routing
- Caching
- Network-level load balancing

The application or dedicated gateway is responsible for:

- Authorization
- Request/response validation
- Protocol conversion
- Error handling
- Application-instance load balancing

Both are responsible for logging and monitoring to enable visibility into the state of the system and to ensure that policies are being enforced at runtime.

There are similar advantages as the centralized gateway pattern that also include:

- Mitigation of most shared-fate outages and noisy neighbors by moving the most error-prone processing like request validation out of the shared gateway and delegating to the application or dedicated gateway.
- Increased iteration speed due to the ability to update configurations with less process overhead and hence quickening the time involved. This is possible due to reduced risk of shared fate outage.

Disadvantages include:

- The enforcement of policies is split across the API gateway and many service instances, which makes it more challenging to ensure that the policy is being enforced consistently and correctly.

- There is increased operational burden on application teams compared to the centralized API gateway pattern, as they are now responsible for ensuring that some policies are enforced in their application.
- Not all classes of shared fate outages and noisy neighbors can be eliminated because the shared central gateway is doing at least some application layer processing.
- Cost attribution is significantly improved compared to the centralized pattern because the most expensive runtime policies are implemented by the application teams. However, the centralized gateway can still be very expensive to operate at high scales and is as difficult to attribute costs as in the centralized pattern.
- Caching hit rates also suffer similarly to the centralized pattern for the same reasons.
- Bypassability/pivot

4.3. Decentralized Gateways

In a decentralized approach, the gateway is directly associated with the application, which is owned by a single team. This ensures that changes are isolated to services owned by that team and that the potential for shared fate outages does not arise. Changes to each gateway are "safe" from the organization's perspective: a bad change will not cause additional problems for other teams, and the team that caused the outage to occur can fix the problem.

External traffic must still enter through a load balancer, which does not enforce any policy and only performs routing. Internal traffic may use the same load balancer but may be routed directly peer-to-peer, removing the central gateway from internal traffic as desired, since enforcement of policy happens at the service instance.

This leaves two key challenges that the implementation must address:

1. Ensuring that the remaining shared configuration (i.e., the load balancer) is safe for each team's changes
2. Ensuring that both cross-cutting and application-specific policies are enforced consistently across the organization

Keeping the load balancer's configuration safe is a universal problem across all three implementations. However, it is most acute in the decentralized pattern because the load balancer must cope with configuration for many applications, while only the API gateway's configuration needs to be present in the other patterns. Regardless of implementation pattern, this is most often handled at the business process level. Organizations decide on a fixed naming scheme that is enforced by and during by the CI/CD process or is otherwise hidden by the organization's platform (e.g., subdomains-per-service, such as foo.api.example.com, bar.api.example.com; paths-per-service, such as api.example.com/v1/foo, api.example.com/v2/bar).

The challenge of cross-cutting policy is unique to this pattern. In recent years, it has been solved robustly in open source via the service mesh, which can provide a single point for policy management and use its proxies to enforce those policies (i.e., API protections) at each service instance. The service's properties [2] and use for security [3][6] have been covered in other NIST guidance documents.

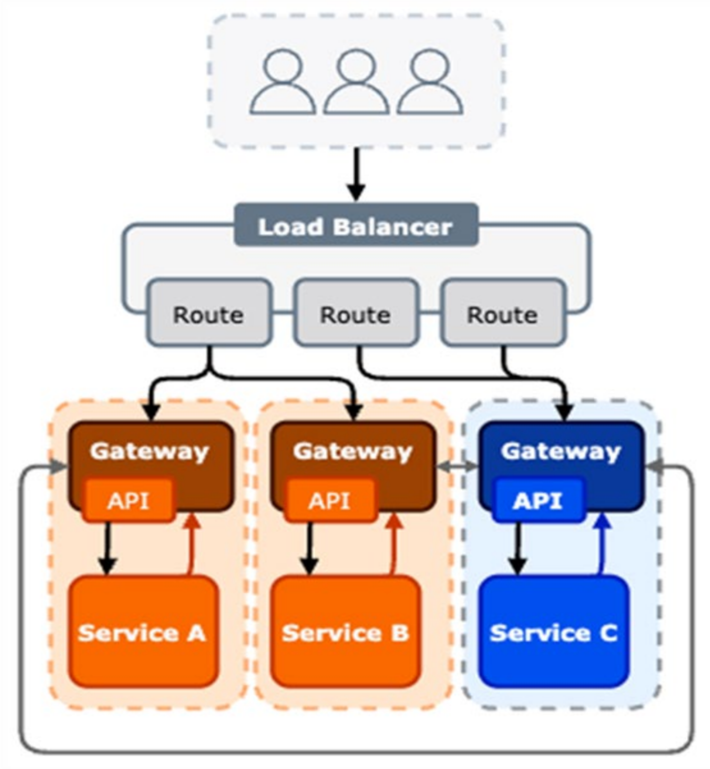

Fig. 9. Decentralized API gateway pattern

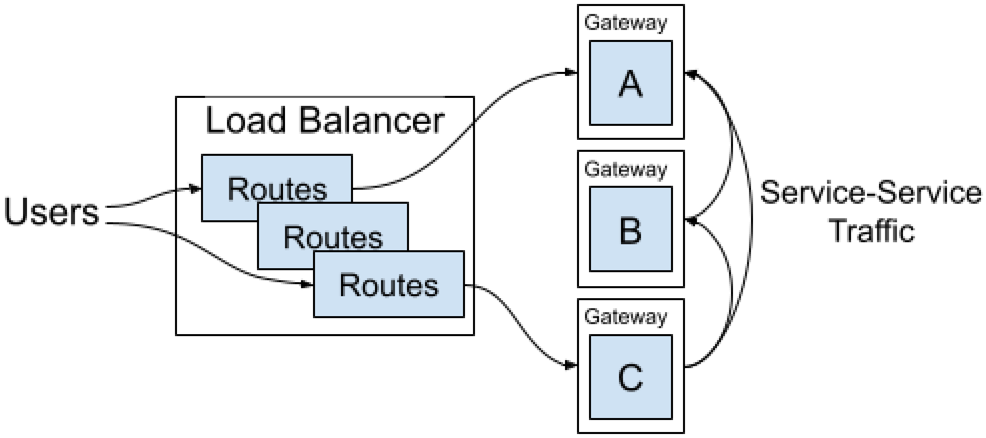

Fig. 10. Service-to-service traffic flows in decentralized API gateway pattern

The advantages of a decentralized gateway pattern include:

- All processing is done per application team (i.e., no noisy neighbors), and the risk of a shared fate outage is only present on the load balancer, which is a risk shared across all implementation patterns.
- It has the highest rate of change for app teams because they have no external dependencies and little chance of causing outages for other teams.
- A cross-cutting policy can be managed by the central API gateway team via the gateway's control plane (e.g., with the service mesh). This pattern can be adopted harmoniously in a mixed environment, where some APIs are implemented via any of the three patterns in a single organization.
- Cost attribution is straightforward and no more or less challenging than attributing any compute resource spent by teams in the organization.
- Cache locality is typically better than in the other patterns because there is only a single layer of load balancing, and the gateway is co-located with the application. This means that gateway policy checks for a given user are cached alongside the application instance caching business logic data for that user. However, if a user's request is load-balanced across multiple service instances, then "duplicate" policy checks have to be performed that would not be required in the other patterns.

Disadvantages include:

- Because the policy is checked and easily cached per application instance, there can be many more policy checks in the system overall. Any time a user's request is load-balanced to a new service instance, it is highly likely that a new policy check has to be performed. This is an inherent problem in any zero-trust system, which pushes enforcement to the application instance and likely necessitates the adoption of a distributed cache managed alongside or as part of the API-serving infrastructure.
- The pattern puts the most burden on application teams. Those teams have to interact with the team managing the load balancer for each API they expose and need to operate at least some of the API-serving infrastructure (e.g., making sure that they have a gateway deployed and routing). Technology like a service mesh can help simplify this, but a burden remains.
- Auditing and verifying policy enforcement can be challenging as enforcement is distributed across all application instances. A robust, distributed gateway implementation (e.g., a service mesh) can help mitigate this via centralized configuration control combined with distributed enforcement and consistent telemetry. If an organization can audit and verify a hybrid gateway pattern, a distributed gateway pattern can be supported with little additional effort.

4.4. Related Technologies

Other technologies fit in and overlap with simplified API gateway patterns and architectures. Notable companion technologies include:

- Web application firewalls (WAFs)
- Bot detection
- DoS/DDoS mitigation
- API endpoint protection
- Web application and API protection (WAAP)

4.4.1. Web Application Firewalls

Web application firewalls (WAFs) mitigate risks related to a request's metadata and payload without needing the application to be involved. In other words, they can be treated as a cross-cutting policy and managed by a central team.

WAFs work at the application level and operate on parsed HTTP requests (i.e., they can implement policy per header and on request bodies). However, WAFs generally do not work at the API level. A WAF can scan a request for a payload that looks like a SQL injection attack, but it cannot assert, for example, that a request has a "name" field that is a string less than 100 characters long. As such, a WAF is an excellent first step for organizations to implement the policies outlined in this document, but it is not a complete solution.

The Open Worldwide Application Security Project (OWASP) publishes research on vulnerabilities based on data from its partners. The OWASP API Security Top 10 [7] was consulted extensively in the preparation of this document. OWASP *also* publishes a generic set of WAF rules — the Core Rule Set (CRS) [25] — that aim to mitigate many common attacks. The CRS should be treated as a starting point for any organization's WAF policy. Deploying a WAF with at least the CRS enabled helps mitigate risks, including malicious input (see Sec. 2.6.2), unrestricted resource consumption (see Sec. 2.4), and leaking sensitive information (see Sec. 2.5).

There are two primary downsides with WAFs:

1. WAFs are relatively expensive to run in terms of both latency and compute. They need to parse every request, perform a variety of scans to identify attack signatures (the number of scans depends on the policy configured), and either block or forward the request. While this overlaps heavily with the functionality of an API gateway, a WAF is typically deployed and operated by a separate team in isolation from the API gateway, often as part of the load balancer. This is convenient because the load balancer is the first place where requests are decrypted in the infrastructure. A secondary consequence is that WAF policies are typically only enforced at the perimeter.
2. WAFs are fundamentally reactive. They operate based on matching requests to known attack signatures. As a result, they are largely ineffective at mitigating novel attacks, and attackers can leverage a variety of obfuscation techniques to hide known attacks behind novel signatures. Care must be taken to ensure that the WAF is running with the latest attack signature configurations, and custom rules must often be written for the organization.

In line with a zero-trust posture, WAF policies should be enforced as close to the application as possible. This helps mitigate a variety of mechanisms that attackers might use to pivot within or otherwise compromise an infrastructure. As a practical matter, it can be cost-prohibitive to run a full suite of WAF mitigations on every internal and external request. This cost can be mitigated in two ways, which can be combined:

1. Incorporate the WAF as part of the overall API-serving infrastructure and deploy the WAF itself in a "hybrid" model (i.e., keep a centralized WAF at the load balancer with a full suite of policies to protect against untrusted traffic). Then enforce a minimum set of app-specific WAF policies near each of the applications (e.g., in the distributed gateway). This minimizes policies run on east-west (i.e., more trusted) traffic while still sanitizing less trusted external traffic and tends to result in a good compromise of risk versus cost.
2. Deploy the WAF as part of the API gateway implementation itself, which can avoid parsing the request multiple times (i.e., reduce the latency and compute costs of WAF policies), regardless of the API-serving implementation pattern chosen. If the API gateway is hybrid or distributed, then this technique can also be incorporated for further performance improvement.

4.4.2. Bot Detection

Bot detection typically involves evaluating risk signals, including origin (e.g., source IP, user credentials) and API usage patterns, over time to determine whether a seemingly legitimate user is likely to be a bot (i.e., an automated script acting maliciously). In response to flagging a high-risk user, bot detection systems will either block traffic or serve some kind of bot-defeating measure (e.g., CAPTCHA) before allowing the system to continue to be used. These tools primarily mitigate the risks of unrestricted resource access (see Sec. 2.4) (e.g., maliciously automating account creation in an email system) and leaking sensitive information (see Sec. 2.5), especially data exfiltration by repeated calls.

Bot detection is frequently deployed in user-facing applications. It can be more challenging with a purely machine-to-machine API because legitimate and malicious traffic patterns are even harder to differentiate. Many APIs are *intended* for use by scripts or non-user-facing applications, so human versus computer checks are irrelevant.

4.4.3. Distributed Denial of Service (DDoS) Mitigation

A distributed denial of service (DDoS) attack is a DoS that originates from many different locations or users. This makes it more challenging to mitigate than a traditional DoS attack, which can often be prevented by blocking a small set of users. While DoS attacks may be targeted and application-level, DDoS attacks are often network-oriented in nature and seek to saturate the server's bandwidth or ability to establish new connections. When a determined attacker is able to build and execute an application-level DDoS attack, it is one of the most challenging attacks to mitigate. Because of the primarily network-oriented nature of DDoS attacks, most DDoS mitigation tools are deployed at the network edge as part of the load balancer or even before the load balancer as part of the CDN and DNS system (often called “Global Traffic Management”). Predictably, DDoS mitigation tools help mitigate unrestricted resource consumption (see Sec. 2.4).

4.4.4. API Endpoint Protection

“API protection” or “API endpoint protection” are nebulous terms for describing a set of capabilities around API inventory, authentication, rate limiting, and data analysis. The exact set of capabilities tends to vary with the implementation. For example, sophisticated implementations can scan requests and responses to tag suspect data on the wire (e.g., to help tag sensitive data and pinpoint possible leaks or exfiltration).

API protection products are typically packaged with the API gateway. API gateway vendors primarily deliver their products in the centralized API gateway pattern, so these controls are often only enforced at the perimeter. Like a WAF, the policies they enforce are typically cross-cutting and do not require an in-depth understanding at the API payload level. As such, the two products are often marketed in a similar niche.

The exact set of risks mitigated by these tools depends on the feature set, but they typically attempt to mitigate lack of API visibility (see Sec. 2.1), broken authentication (see Sec. 2.3), some aspects of unrestricted compute consumption (see Sec. 2.4), and leaking sensitive information (see Sec. 2.5).

There is value in any tool that helps organizations inventory and manage their APIs and traffic. However, the enforcement of any policies should be as close to the individual service instance as possible in order to achieve robust API security assurance. In the use case of data classification, these tools can be especially useful for building an initial inventory. However, as API definitions are rolled out across the organization, data tagging should be implemented as part of the API schema, and the data flow policy should be enforced via explicit policy (e.g., with an authorization system). The runtime discovery of data flow is primarily important in protecting against exfiltration.

4.4.5. Web Application and API Protection (WAAP)

Gartner coined the term “web application and API protection” (WAAP) [27] to describe the trend of packaging the technologies listed here (i.e., WAF, bot detection, DDoS mitigation, and API protection) into a single product. Whether these capabilities are implemented with a single product or a range of technologies, the key is understanding what risk each capability is helping to mitigate and evaluating how it fits into the organization’s existing security posture.

4.5. Summary of Implementation Patterns

Combining the three patterns in API gateway architecture with the companion technologies discussed Sec. 4.4 provides a comprehensive set of enterprise security solutions for API protection. The key point in each pattern is identifying where to enforce each policy. These decisions result in trade-offs in runtime, architecture, and operations for the application teams utilizing the API-serving infrastructure. Many organizations use a mix of all three patterns deployed in production precisely because of those trade-offs. All three patterns can be used to successfully implement all of the controls outlined in this document. That said, the distributed gateway pattern and its companion technologies best align with the principals of zero trust and are strongly recommended for organizations that want to adopt a security-forward approach.

5. Conclusions and Summary

No business-critical enterprise application can be integrated into the digital infrastructure of an enterprise without the use of APIs. With the highly distributed nature of applications (both physically and logically), APIs must be operated under zero trust principles irrespective of whether they are exposed to the outside world or meant to be consumed by other applications within the enterprise infrastructure. Like all software, APIs go through an iterative life cycle whose phases (i.e., Develop, Build, Deploy, Operate) can be broadly classified into pre-runtime and runtime stages.

The sheer proliferation of API deployments, the heterogeneous infrastructures under which they operate, and the access to valuable corporate data that they enable make them targets for exploitation. A detailed analysis of their vulnerabilities and the potential attack vectors that can exploit them is a prerequisite for identifying the appropriate set of protection measures or controls to ensure API security. This document analyzes a spectrum of risk factors that give rise to vulnerabilities, such as the lack of a formal schema, improper inventorying, the lack of robust authentication and authorization support, improper monitoring of resource consumption, and the least leakage of sensitive information.

The recommended controls in this document are classified into pre-runtime and runtime protections. They are further subdivided into basic and advanced protections to enable enterprises to use a risk-based and incremental approach to securing their digital assets. Pre-runtime protections focus on API specification parameters (i.e., syntactic and semantic aspects), while runtime API protections focus on protections during API request and response operations (e.g., encrypted communication channels, proper authentication and authorization).

This document presents a landscape of real-world and state-of-practice implementation options to configure and enforce the recommended controls by describing the advantages and disadvantages of each type of protection deployment or pattern. This will enable practitioners to make an informed decision to realize a robust and cost-effective API security infrastructure for their enterprises.

References

[1] U.S. Department of Defense Chief Information Officer (2024) DoD Enterprise DevSecOps Fundamentals. Version 2.5, October 2024. Available at https://dodcio.defense.gov/Portals/0/Documents/Library/DoD%20Enterprise%20DevSecOps%20Fundamentals%20v2.5.pdf

[2] Chandramouli R, Butcher Z (2020) Building Secure Microservices-based Applications Using Service-Mesh Architecture. (National Institute of Standards and Technology, Gaithersburg, MD), NIST Special Publication (SP) NIST SP 800-204A. https://doi.org/10.6028/NIST.SP.800-204A

[3] Chandramouli R, Butcher Z, Aradhna C (2021) Attribute-based Access Control for Microservices-based Applications using a Service Mesh. (National Institute of Standards and Technology, Gaithersburg, MD), NIST Special Publication (SP) NIST SP 800-204B. https://doi.org/10.6028/NIST.SP.800-204B

[4] Chandramouli R (2022) Implementation of DevSecOps for a Microservices-based Application with Service Mesh. (National Institute of Standards and Technology, Gaithersburg, MD), NIST Special Publication (SP) NIST SP 800-204C. https://doi.org/10.6028/NIST.SP.800-204C

[5] Chandramouli R, Kautz F, Torres-Arias S (2024) Strategies for the Integration of Software Supply Chain Security in DevSecOps CI/CD Pipelines. (National Institute of Standards and Technology, Gaithersburg, MD), NIST Special Publication (SP) NIST SP 800-204D. https://doi.org/10.6028/NIST.SP.800-204D

[6] Chandramouli R, Butcher Z (2023) A Zero Trust Architecture Model for Access Control in Cloud-Native Applications in Multi-Cloud Environments**.** (National Institute of Standards and Technology, Gaithersburg, MD), NIST Special Publication (SP) NIST SP 800-207A. https://doi.org/10.6028/NIST.SP.800-207A

[7] OWASP (2023) OWASP Top 10 API Security Risks. Available at https://owasp.org/API-Security/editions/2023/en/0x11-t10/

[8] OWASP (2023) API2:2023 Broken Authentication. Available at https://owasp.org/API-Security/editions/2023/en/0xa2-broken-authentication/

[9] Wikipedia (2025) Fatigue Attack. Available at https://en.wikipedia.org/wiki/Multi-factor_authentication_fatigue_attack

[10] Wikipedia (2024) Billion laughs attack. Available at https://en.wikipedia.org/wiki/Billion_laughs_attack

[11] Wikipedia (2025) Zip bomb. Available at https://en.wikipedia.org/wiki/Zip_bomb

[12] National Institute of Standards and Technology (2024) The NIST Cybersecurity Framework (CSF) 2.0. (National Institute of Standards and Technology, Gaithersburg, MD), NIST Cybersecurity White Paper (CSWP) NIST CSWP 29. https://doi.org/10.6028/NIST.CSWP.29

[13] Wikipedia (2025) Principle of least astonishment. Available at https://en.wikipedia.org/wiki/Principle_of_least_astonishment

[14] F# for fun and profit (2013) Designing with types: Making illegal types unrepresentable. Available at https://fsharpforfunandprofit.com/posts/designing-with-types-making-illegal-states-unrepresentable/

[15] Joint Task Force (2020) Security and Privacy Controls for Information Systems and Organizations. (National Institute of Standards and Technology, Gaithersburg, MD), NIST Special Publication (SP) NIST SP 800-53r5. Includes updates as of December 10, 2020. https://doi.org/10.6028/NIST.SP.800-53r5

[16] Barker E (2020) Recommendation for Key Management: Part 1 – General. (National Institute of Standards and Technology, Gaithersburg, MD), NIST Special Publication (SP) NIST SP 800-57pt1r5. https://doi.org/10.6028/NIST.SP.800-57pt1r5

[17] National Institute of Standards and Technology (2019) Security Requirements for Cryptographic Modules. (Department of Commerce, Washington, D.C.), Federal Information Processing Standards Publications (FIPS) NIST FIPS 140-3. https://doi.org/10.6028/NIST.FIPS.140-3

[18] rfc7519 (2015) JSON Web Token (JWT). Available at https://datatracker.ietf.org/doc/html/rfc7519

[19] rfc5280 (2008) Internet X.509 Public Key Infrastructure Certificate and Certificate Revocation List (CRL) Profile. Available at https://datatracker.ietf.org/doc/html/rfc5280

[20] Wikipedia (2023) Common Log Format. Available at https://en.wikipedia.org/wiki/Common_Log_Format

[21] Github (2016) grpc. Available at https://github.com/grpc/grpc/blob/master/src/proto/grpc/status/status.proto

[22] Wikipedia (2025) Prompt injection. Available at https://en.wikipedia.org/wiki/Prompt_injection

[23] Wikipedia (2024) Cascading failure. Available at https://en.wikipedia.org/wiki/Cascading_failure

[24] Infoq.com (2020) How to Avoid Cascading Failures in Distributed Systems. Available at https://www.infoq.com/articles/anatomy-cascading-failure/

[25] Coreruleset.org (2025) OWASP CRS PROJECT. Available at https://coreruleset.org

[26] Wikipedia (2025) *Confused deputy problem*. Available at https://en.wikipedia.org/wiki/Confused_deputy_problem

[27] Gartner.com (2025) *Cloud Web Application and API Protection*. Available at https://www.gartner.com/reviews/market/cloud-web-application-and-api-protection

Appendix A. API Classification Taxonomy

A.1. API Classification Based on Degree of Exposure

Since APIs are interfaces that are exposed to relevant stakeholders, they should be classified based on their degree of exposure. Three kinds of APIs are prevalent:

1. Open/Public APIs are exposed to a broader and wider audience (i.e., customers) and used with external partnerships or services. These are also called “facade APIs,” as they may provide limited access to certain functionalities.
2. Private APIs are used to link various systems within an enterprise and are closely guarded, such as a contract between microservices that are internal to an organization. Variations of private APIs are:
    a. Internal APIs (service APIs): Used by enterprises to streamline their internal workflows and create flexible systems that can adapt to changing business needs
    b. Composite APIs: Allow multiple data and service calls to be combined to realize efficiency in system design [10]
3. Partner APIs are used in the context of collaborative ventures between enterprises, as both rely on shared services or data to deliver value to their end users. In terms of exposure, they represent a middle ground between public and private APIs since access is restricted based on collaborative agreements [10].

A.2. API Classification Based on Communication Patterns

There are two fundamental API communication patterns that govern how information flows between the components involved in system interactions:

1. Request-response APIs: A communication pattern in which a client sends a request to a server and awaits a corresponding response. It operates synchronously with stateless, independent requests. This pattern is widely employed in diverse API architectures (such as RESTful APIs, GraphQL, and various web services [11]) and is appropriate for immediate data retrieval or any instant action (e.g., downloading a user’s profile in a social media app). For example, requests are made with verbs that are appropriate for the API architecture (e.g., HTTP method GET in RESTful architecture, a structured query specifying the exact data needed in response in GraphQL architecture).
2. Event-driven APIs: A better choice for receiving real-time updates (e.g., user’s activities in the same app).

A.3. API Classification Based on Architectural Style or Pattern (API Types)

Table 1. API classification based on Architectural Patterns

API Name	Network Protocol	Data Format
REST	HTTP/1.1	Text-based JSON
gRPC	HTTP/2	Binary — Protocol Buffers (Protobuf)
GraphQL	HTTP – POST only	JSON
WebSocket	WebSocket	JSON

Appendix B. DevSecOps Phase and Associated Class of API Controls

The detailed controls in Sec.3 fit into several broad classes, and Table 2 shows their associations with the DevSecOps phases (see Sec. 1).

Table 2. DevSecOps phase and associated class of API controls

DevSecOps Phase	Class of API Controls
Coding	Well-defined API schema definition that calls to routines for annotating schema definitions
Build	Generate routines that validate on-field values in the request and response payloads of API calls and responses, respectively
Test	Ensure that validation routines perform as intended in various runs of API requests and responses
Deployment	Ensure that the deployment package contains all of the runtime policy enforcement routines, API schema definitions, and APIs and is signed off by the right authorities
Observe and Monitor	Ensure that certain security incidents (e.g., data leakage) do not occur due to (a) inherent flaws in API design, (b) the lack of input data validation, or (c) engineered attacks realized through a sequence of requests that each pass all validation tests