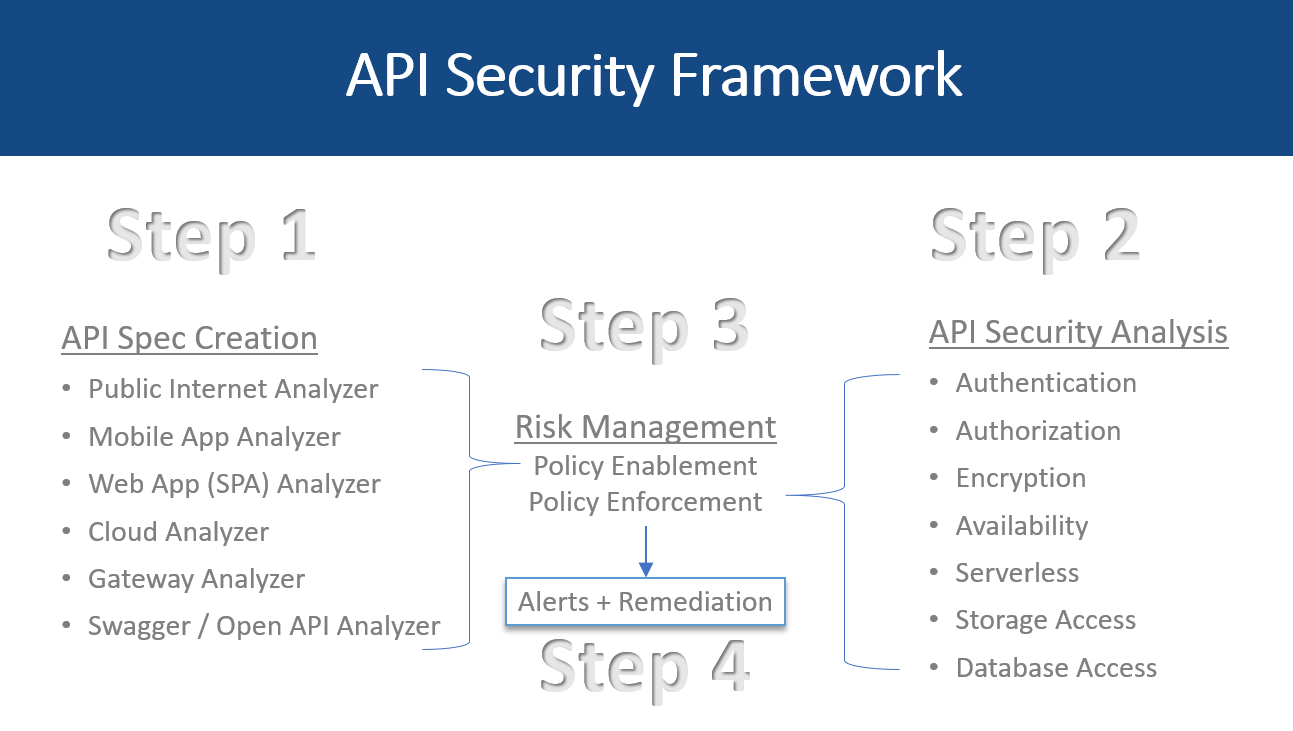
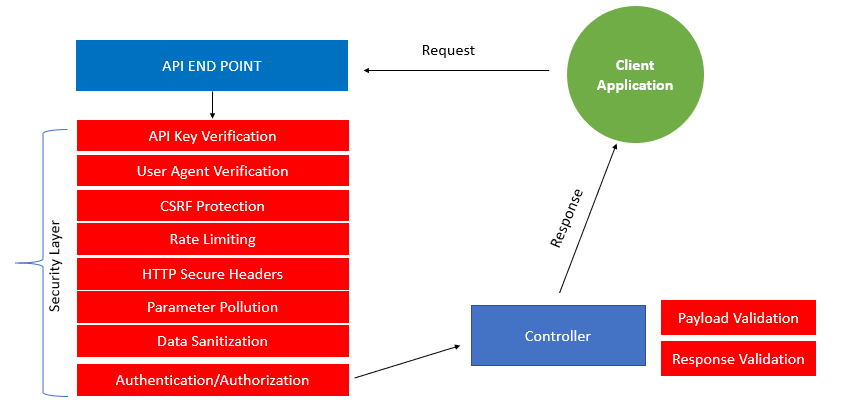
**Web Security Practices**

****



**Introduction to API Security**

API security is increasingly a focus area for both developers and security professionals, given the proliferation of APIs in modern applications. APIs are becoming the number one attack vector in 2022. This Refcard enables developers to understand the elements of API security, common vulnerabilities and attack vectors, and best practices for building secure APIs.

**How Does an API Work?**

Representational State Transfer (REST) is a software architecture developed for hypermedia data on the World Wide Web. This architecture is now the most common method for the implementation of Application Programming Interfaces (APIs), which allow for the transfer of data between a client and a server.

A transaction between client and server comprises a request being made to a given endpoint which consists of a base URL (the domain of the server), a method (the path portion of the URL), and optional parameters. A request specifies the method (often called the verb) to be used; additionally, a body (typically JSON or XML) can be enclosed in the request.

Once the server has validated the request using authentication and authorization schemes, it will process the request and provide a response in return, including an HTTP response code that indicates the status of the request. Additionally, the response may return a data payload in a variety of formats and encoding schemes.

**Why Is API Security Increasingly Important?**

The number of APIs globally is growing rapidly, with [Postman reporting typical growth of 39 percent in the last year](https://www.postman.com/state-of-api/api-global-growth/). [Gartner estimates that by 2022](https://www.techrepublic.com/article/91-of-enterprise-pros-experienced-an-api-security-incident-in-2020/), 90 percent of web applications will have more attack surface exposed in APIs than in the UI, and that API abuses will become the most frequent attack vector. APIs are an attractive attack vector for adversaries since they are frequently publicly exposed with documented endpoints. Excellent attack tools allow for ease of automation of attacks.

The combination of the prevalence of APIs, the relatively easy target they present, and the value of the data exposed means that API security is a top priority for security and risk professionals. Fortunately for developers, there are several well-established patterns and practices that rapidly reduce the risk of API vulnerabilities.

**Foundations of API Security**

The first step toward securing APIs is to have a deeper understanding of foundational elements of APIs such as:

**Transport Layer Security**

The Transport Layer Security (TLS) protocol is the de-facto transport-level cryptographic protocol to ensure secure communications over a TCP/IP network. An encrypted transport layer is essential for APIs to ensure that attackers are unable to eavesdrop data or tokens “on the wire” as they are transmitted and to ensure that the client can validate the integrity of the server (via certificate checking). Certificate management has traditionally presented challenges to organizations; however, with the emergence of providers such as Let’s Encrypt certificate deployment has become relatively trivial. All new API development should secure a communications network as a first line of defense.

**Rate Limiting**

Public APIs serve as the gateway to a business and, as such, must be protected from abuse via denial-of-service (DoS) attacks. An adversary can trivially launch automated attacks against APIs, which can either degrade performance for legitimate users or destroy the API due to resource exhaustion on the server. A secondary attack vector is to attempt to brute force either a sign-up endpoint or a password reset endpoint by rapidly guessing passwords or hashes with a dictionary attack.

Both types of attack may be mitigated using rate-limiting technology, which limits repeated and frequent access from a particular IP address to a given endpoint. Rate limiting is typically implemented in an API gateway or dedicated protection product (such as a WAF or API firewall).

|  |  |  |
| --- | --- | --- |
| **Language** | **Platform** | **Library name** |
| **C#** | ASP.NET | WebApiThrottle, MvcThrottle |
| **PHP** | Laravel | Laravel Karnel Default |
| **JS** | Node/Express JS | express-rate-limit |

**Access Control**

Access control is a key foundation of API security and encompasses two elements — firstly, authentication determines who is accessing an API, and authorization determines what they are allowed to do within the API.

**No Authentication**

Public APIs intended for read-only use may not require authentication at all if they are intended for anonymous access. Typically, these endpoints are used for information and status information.

**Basic Authentication**

Basic authentication is a largely outdated method with origins in early website authentication. The client provides a username and password, which are encoded as a Base64 string, transmitted in the request headers, and verified by the server. Weaknesses of this method included lack of revocation and clear text transmission of credentials. This method is not recommended for new designs.

**Session Cookies**

Another authentication technique with origins in website authentication is the use of session cookies. Once authenticated via a sign-in page, the client can store a session cookie, which is presented to the server on subsequent authentication requests. This, again, is not a recommended solution since session cookies are vulnerable to leakage (via cross-origin script attacks) and lifetime management challenges (expiration and revocation).

**Bearer Authentication**

Bearer authentication seeks to eliminate some of the weaknesses in the basic authentication method by using a hashed value of the user credentials rather than transmitting the credential to the server. The protocol consists of two API operations — the first call returns a 401 Unauthorized code and a header called WWW-Authenticate with a hash value and a nonce. The second issues a call to authenticate with the hash values to generate an authorization header which is stored for subsequent access. Although better than basic authentication, this method still has weaknesses such as vulnerability to machine-in-the-middle attacks.

**API Keys**

API keys are a staple of API authentication due to their simplicity and robustness when managed correctly. The server generates a key (with an optional lifetime) for a given client using a management portal (typically a website allowing account management). The client stores this key and submits it via a header in all API requests, and the server validates this key. Challenges arise with the storage of keys since clients (mobile applications and single-page web applications) typically do not provide secure storage. Additional challenges exist with the lifetime management of keys and subsequent re-issuing of revoked keys. API keys are well suited to service APIs within a microservice architecture and less suited to end-user applications.

**OAuth 2.0**

OAuth 2.0 has become the de-facto standard for authentication of API clients, and the provision of permissions (authorization) on servers. In essence, OAuth is a flow to allow a user to verify an identity to an authorization server (typically via a login page), to then allow the requested access permissions (roles), and to receive a token which is used in subsequent access. The client acts on behalf of the user and does not have access to the user’s credentials and has only specific access permissions.

OAuth provides several flows (grant types) to allow access tokens from the server, depending on the client type. Certain flows have been deprecated, and the two most common in use now are:

* **Authorization Code**: This is the recommended flow for back-end clients capable of securely storing tokens. The client initially authenticates with the authorization server (in a web page usually) and receives an authorization code which is then exchanged for an access token, and optionally a refresh token. The access token can be short-lived to mitigate leakage vulnerabilities, requiring clients to re-authenticate or use the refresh token.
* **Authorization Code with PKCE**: This augments the Authorization Code flow to include a code verifier challenge to ensure that attackers intercepting the code cannot exchange this for a valid token. Typically, this flow is recommended for native applications and single-page applications, which are unable to store client secrets.

**OpenID Connect Discovery**

OpenID Connect (OIDC) is an identity layer built on top of OAuth and is supported by OAuth providers such as Google and Azure Active Directory. As a final stage of the authentication process, the client can receive information (called claims) specific to the user, such as username, email, etc. This information is typically returned in a JWT token, allowing the client to verify the integrity of such information.

**Authorization**

Once a client has identified itself, the server will need to assess whether the client has the necessary permission to access the requested resource. Typically, two types of access control may be applied — firstly at the request level (the endpoint URL) and secondly at the object (or data) level. Certain functions may have public access, while others must require authentication that can be handled using the application middleware (i.e., Spring Boot, ASP.NET Core, etc.).

For more fine-grained permissions, such as object level, the business logic should enforce these controls using typical access control models (RBAC, ABAC). Complex authorization requirements are best handled with dedicated libraries or frameworks such as Open Policy Agent, which provide a single point of enforcement for access policy.

**Tokens (JWTs)**

The final fundamental of APIs is the JSON Web Token (JWT) used to transfer information between client and server in a portable and robust way. JWTs are cryptographically secure, allowing a client to verify the integrity of the message by using public-key cryptography methods to validate the content. Being JSON, they are easily transmitted in APIs as part of the request header or body.

A JWT comprises three parts: the header, the claim, and the signature (or HMAC).

The header provides basic information regarding the JWT and, most importantly, the algorithm type used to produce the JWT.

The claims section contains server-specific data (the claims) in a key-value pair notation. Registered claims include basics such as issuer, expiration time, subject, audience, etc. Although not mandatory, these are strongly recommended. The public claims section contains specific data such as client roles, permissions, identity, etc.

The signature is a hash of the Base64Url versions of the header and the claims, and a server-side secret. By verifying the signature, a client can verify the message body was not changed in transit. If used with a private/public key combination, it can verify the signatory’s identity. In addition to signature validation, JWT claims should be fully validated to ensure the validity of expiration date, source addresses, usernames, etc.

JWTs (specified by RFC 8725) are ubiquitous and applicable to identity authorization solutions; however, care must be exercised when producing and consuming JWTs to avoid several common pitfalls.

**Secure API Lifecycle**

Effective API security is ideally built on the foundation of a well-established DevOps process.

**Design**

Strong API security starts at the design stage to ensure that full consideration is given to the following:

* Authentication and authorization requirements based on the operating environments (i.e., choosing the correct OAuth flows).
* Understand access control models based on user rights.
* Data privacy and protections requirements (i.e., GDPR, PII, PCI-DSS).
* Applicable regulatory and governance requirements.
* Emerging threats and attack methods to minimize attack surfaces.
* A formal threat modeling activity ensures all attack surfaces and weaknesses are understood before implementation.

A well-defined OAS definition is the foundation of strong and secure API design. The [OpenAPI Specification](https://github.com/OAI/OpenAPI-Specification" \t "_blank) prescribes primitives to govern key topics relating to security, including request and response data validation, full endpoint specification, and security, which contain common authentication and authorization patterns.

**Build**

The construction of API back-ends is a critical factor in ensuring API security. Best practices suggest that a design-first approach using an OAS definition as the input enables the audit and review of OAS definitions before coding to form the specifications fully. Code generation tools, such as [Swagger Codegen](https://swagger.io/tools/swagger-codegen/), offer a reliable and automated method of generating much of the API back-end code.

For each of the respective frameworks (i.e., Spring Boot, ASP.NET Core, etc.), developers should consult the specific security recommendations, particularly regarding authentication and authorization middleware.

**Test**

A key benefit of a design-first approach is the ability to perform much of the downstream API testing in an automated manner using tools able to interpret the OAS definition and exercise the API accordingly. Popular tools are available to conduct API functional testing, performance and load testing, and security testing. API conformance testing is essential to ensure that the API produces the expected responses and does not behave in ways not specified in the OAS definition.

**Protect**

Despite the best intent during the previous lifecycle stages, certain API protections are best enforced at a gateway level in line with the API traffic rather than at a code level. A good example is rate-limiting protection — while feasible to perform such limiting in the API code itself, this is best performed upstream before reaching the API. The use of external protections also provides a defense-in-depth approach to security, so there is a reduced likelihood of a single point of failure.

**API Gateways**

Many excellent API gateways are available with all major cloud platforms, offering native solutions. Gateways can typically ingest OAS definitions and allow enforcement of policies governing rate limiting, JWT validation, and transport layer security while also providing observability and monitoring capabilities.

**WAFs and WAAPs**

Web applications have traditionally relied on web application firewalls (WAFs) to protect against attacks. A typical WAF acts as a proxy between the client and the server, inspects traffic for known malicious content, and filters it out according to a ruleset. Modern WAFs can include API-specific protections (so-called Next-Gen WAF and API Protection products), providing protection closer to the application layer using agent-based protections (RASP) and machine learning or artificial intelligence capabilities.

**API Micro-Firewalls**

API micro-firewalls provide dedicated protection for APIs by enforcing the OAS contract using a positive security model. Only methods, endpoints, and data specified in the contract are allowed; anything else is blocked. These micro-firewalls are designed with a small memory footprint in mind and are designed for microservices.

**Monitoring**

The final element of the API lifecycle is the proactive monitoring of APIs — primarily to identify malicious operations, emerging threats and attacks, and service degradation. Telemetry and logging solutions can be embedded into the API back-end code to provide insight into the operation of the APIs, while alerts from gateway and firewall products should be integrated into SIEMs and SOCs to allow security monitoring and isolation of APIs at the gateway level in the event of an attack.

**Governance**

A vital aspect of API security is governance of the entire process from design to operation to ensure that all regulatory, compliance, and privacy requirements are being followed. Failure to provide adequate governance can result in significant fines.

Governance also ensures that organizations maintain an up-to-date inventory of their APIs, reducing the prevalence of shadow APIs (deployed outside of a standard process) and zombie APIs (outdated or deprecated APIs). A well-managed inventory is vital to the reduction of the API attack surface.

Section 4

**Common API Attack Methods (and Defenses)**

By understanding some of the most common API attack methods, developers can better protect their APIs. The following are some of the top attack methods:

**Bot Attacks**

Bots are increasingly becoming a scourge of the security industry — and APIs are particularly vulnerable to attack given their lack of user interface. Bots can be benign when used for process automation (e.g., checking travel status), but bots can also be malicious and used to attack or abuse APIs (e.g., scraping large volumes of data or PII).

Bots will typically have a range of well-known IP addresses for large ISPs and cloud providers and utilize botkits or botnets on the underground web.

**API Discovery and Endpoint Enumeration**

As a first stage, an adversary will attempt to gain knowledge of the APIs and their endpoints. This can range from relatively primitive methods such as the use of nmap to discover open connections, pen testing tools such as OWASP Zap and BurpSuite, and sophisticated tools such as Kiterunner, which are capable of advanced API discovery methods.

**Denial-of-Service Attacks**

The least subtle of API attacks is a simple denial-of-service attack intended to take an API offline by overloading the underlying servers. Botnets can launch massively parallelized attacks against an API and scripts allows for easy automation of this process. While rate limiting is a valuable defense, it can be defeated by rotating source IP addresses or spoofing origin headers.

**Credential Stuffing Attacks**

Any endpoint or authorization mechanism accepting a password as an input is susceptible to credential attacks using common passwords dictionaries. Defenses include rate limiting on such endpoints and multi-factor authentication.

**Account Takeover Attacks**

Related to credential stuffing attacks is the broader topic of account takeover. Techniques here include the exploitation of password reset processes, which are often exposed as an API endpoint. Attackers may be able to rapidly initiate the reset process and attempt to intercept the out-of-band authentication via SIM swap, fake emails, etc. Defenses include rate limiting and robust multi-factor authentication.

**API Scraping and Pagination Attacks**

APIs exist to serve data to end-users and websites. Using bots or scripts, it is possible to scrape APIs to effectively download the entire data store, which may derive the data owner of revenue streams. Typical examples include scraping price, availability or delivery information, or user information from social media or real-estate prices.

APIs should detect excessive requests for data from the same account or IP address and implement some form of throttling. A particularly common attack is a pagination attack where an attacker loads a page in a browser, copies the URL, and uses scripts to traverse through all pages.

**Common Vulnerabilities**

The OWASP organization has categorized common anti-patterns for APIs and publishes a dedicated OWASP API Security Top 10 describing the top 10 most prevalent API vulnerability classes.

**Authentication Vulnerabilities**

Broken or missing authentication is a leading cause of API vulnerability. As described earlier, a wide range of authentication mechanisms exist, and often, developers are overwhelmed when choosing and implementing a method leading to exploits in production.

Common scenarios include:

* Total unprotected endpoints.
* Leakage of tokens and keys (OAuth addresses these issues).
* Exposure of user credentials.
* Abuse of password reset mechanisms.

For every API endpoint, be sure to enforce authentication ideally via built-in middleware. If an API is public, then explicitly mark it as such in the code.

**Object-Level Vulnerabilities**

The top vulnerability by incidence and severity is broken, object-level authorization. In this vulnerability, an endpoint allows a given user access to an object (namely data) that does not belong to that user. The root cause of this vulnerability is a failure in the API back end to validate that the requesting client has access to the specified object. Typically, attackers will gain authentication to an API and then attempt to manipulate object identifiers to probe for poor implementations.

Remember: Always fully validate the permissions of a client to access a given object.

**Function-Level Vulnerabilities**

Broken function-level authorization occurs when a user can successfully access an API endpoint (method or function) to which they are not authorized. For example, an API might include an /admin endpoint intended for administrator-level privilege users. If a user without administrator privileges could access this endpoint, this would constitute broken function-level authorization. The root cause is a lack of user authorization on a given endpoint.

If fine-grained function-level access is required, ensure that the correct authorization checks are made in the relevant back-end code.

**Data Vulnerabilities**

There are two important classes of vulnerabilities associated with data — excessive data exposure and mass assignment. Both are extremely common and carry a significant risk impact on the API and underlying data.

Excessive data exposure occurs when an API returns more information than expected or necessary to meet its goal. For example, if a client attempts to retrieve user details, a poorly implemented API method might return sensitive user information such as PII data. The API should return the bare minimum of data to meet the design requirement and no more.

Mass assignment occurs when an attacker can inject additional request data, which is incorrectly interpreted by the API endpoint and stored in the underlying data record. A typical example is when an attacker guesses a field name (such as isAdmin) and assigns a value. Ideally, the back end would ignore such extraneous fields, but a vulnerability would interpret this value and store it unintentionally. In this way, an attacker can modify the API’s underlying data store.

Data vulnerabilities can be easily tested and detected using conformance scanning of APIs against their definition. Developers should be aware of using the full data object and exposing this on the API — rather, a simplified data object (with sensitive fields removed) should be used for API operations.

**Configuration Vulnerabilities**

Security misconfigurations cover a wide range of operational or runtime misconfigurations, which result in a vulnerability infrastructure for an API. These can include issues relating to misconfigured headers, missing transit encryption, accepting unused HTTP methods, overly verbose error messages, and lack of sanitization.

Improper asset management occurs when an organization exercises inadequate governance over their API development process, leading to the existence of shadow or zombie APIs.

**Implementation Vulnerabilities**

Finally, this catch-all category of vulnerabilities covers various implementation and coding vulnerabilities, including issues such as input injection (SQL, command, LDAP, etc.), lack of data sanitization, use of vulnerable libraries or components, misconfiguration of frameworks, etc.

**Best Practices for Securing APIs**

Fortunately for developers, there are several simple practices that can be employed to improve the security of their APIs, such as:

**Always Use TLS**

Transport Layer Security should no longer be considered a “nice-to-have” and should be enforced as a standard for all API endpoints. Care should be taken to correctly configure security headers and transport security, thus ensuring that a certificate rotation process is in place.

**Use OAuth2 for SSO**

Poor implemented authentication and authorization rank highest on the OWASP API Security Top 10. Developers should resist the temptation to invent their workflows for the issuance and revocation of tokens. This is so that they can leverage the collective wisdom behind the OAuth 2.0 standard. Furthermore, developers are advised to seek a deeper understanding of the various authentication flows by using an OAuth playground and draw on advice from authoritative guides to implement best practices.

Where OAuth is not suitable (such as back-end services), then API keys should be used. Keys should have limited scopes if possible and should be easily revoked and replaced.

**Practice Good Hygiene With API Keys and Tokens**

Once a key or token has been obtained, developers should ensure they are practicing good hygiene, including the following:

* Avoid committing keys and tokens to version control repositories.
* Avoid storing keys and tokens in clear text either on endpoint devices or on CI/CD pipelines.
* Use short-lived keys and tokens where possible.
* Use keys and tokens with the minimum level of access required for the role. Avoid granting overly permissive privileges.

**Use a Policy Agent to Manage Fine-Grained Authorization**

Function-level authorization requires access controls to be implemented in the endpoint code for each function. Implementing such functionality with homegrown libraries or re-suing code is fraught with danger — code can become unmanageable, and bugs in logic can easily be introduced. Best practices prescribe the use of a central authorization endpoint — to centralize decision making — or the use of an authorization framework.

**Eliminate Unused Methods on Endpoints**

A quick win in reducing the attack surface on an API is to remove unused methods (i.e., HTTP methods typically such as DELETE, OPTIONS, PATCH, etc.). Depending on the framework used, invalid methods may be invoked on an endpoint causing unexpected behavior. The recommendation is to explicitly eliminate such methods using route mappings in the framework.

**Apply Quotas and Rate Limiting**

The abuse of APIs can result in denial-of-service attacks or allow large-scale data exfiltration. Endpoints should use both quotas to limit excessive use or rate limiting to reduce abuse cases. Such defenses can be implemented in code, but ideally, they should also be implemented at a firewall or gateway level.

**Use API Firewalls and Gateways**

Low-hanging fruit for API security — all cloud platforms offer API gateway services, and these should be enabled with strong baseline policies applied, e.g., rate limiting and transport security. To address specific API security concerns like data exfiltration, JWT validation, etc., bespoke API micro-firewalls should be utilized.

**Manage Your API Inventory**

API attack surfaces can be dramatically reduced by active management of the organization’s API inventory. Firstly, a governance process should prescribe standards and best practices for new APIs, using guidance from this article and other industry standards. Secondly, actively taking steps to identify the organization’s API landscape enables an analysis of out-of-date or deprecated APIs — and APIs that are outside of the governance process.

**Use Proven Libraries for Standard Functions**

Many of the most challenging aspects of API development pertain to the implementation of common patterns or workflows — think of OAuth2 authorization code flows, JWT token validation, or fine-grained authorization policy. Although tempting for developers, they should resist the temptation to build their implementations for such patterns and rather rely on the wisdom of the crowd. Adopt a standard library, and actively participate in the community to contribute patches and fixes as you find them.