API Security Best Practices

Table of Contents

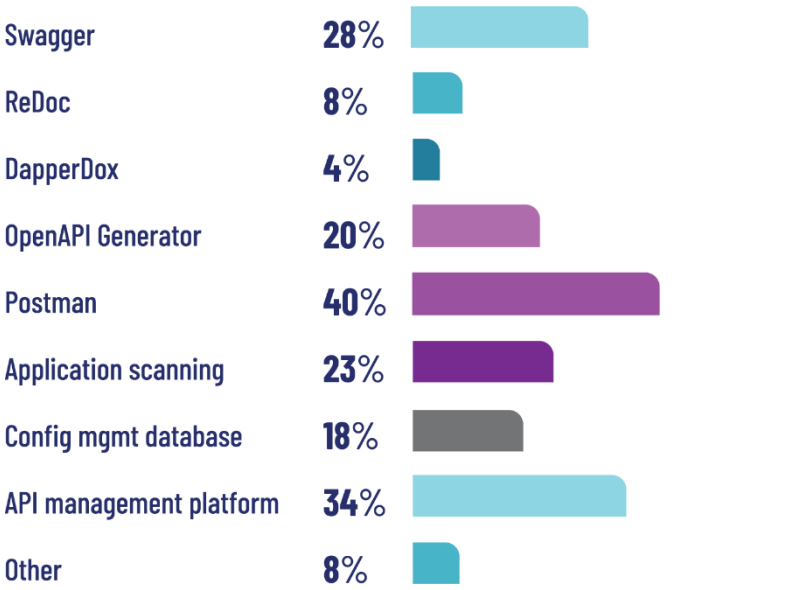
Overview

APIs are the mechanism for data access, service integration, and business functionality. Traditionally, most APIs were internal and private, operating within the confines of an organization's data center and invoked primarily by other internal systems when employees needed some data or functionality. As a result, network access controls including IP deny lists and VPNs were the default security control.

The landscape of APIs has evolved substantially over the years to account for different business cases and consumption models. Internet-facing or external APIs are commonplace to support mobilized workforces and customers accessing services from anywhere. Open APIs, such as we see with open banking, help push forward the financial industry, improve levels of service integration, and provide financial freedom for customers to transact anywhere. Organizations create and integrate partner APIs to power and expand their digital supply chains. Third-party APIs, often consumed as SaaS, can help organizations move faster without re-creating functionality or adding more technical debt. We also still have acquired APIs in applications and systems that are inherited from commercial and open-source software packages. Entire infrastructures are even declared or operated via APIs, such as we see with Kubernetes and cloud providers.

The broad landscape of API design patterns and API consumer types complicates security requirements for organizations. This reality makes arriving at sets of best practices challenging since it must be expansive and inclusive of many traditional security practices yet tailored for API use cases. Salt Security has compiled this list of API security best practices based on field experience and customer feedback. The practices are divided up into multiple security focus areas. Ideally, an organization factors in all of these practices, but your starting point is also likely a factor of scope of responsibility or budget. We see significant variation with how organizations approach any given focus area, including API documentation as highlighted in the most recent State of API Security report and depicted here. APIs are implemented, operated, or interacted with by many roles within an organization including development, API product teams, API operations teams, application security teams, and security operations. Like in DevOps, collaboration is crucial for building and operating secure APIs. You'll need to ensure that your best practices account for the different personas within your organization.

What mechanism(s) do you use to document and inventory your APIs? *(Select all that apply)*



Secure design and development

The top 3 recommendations for secure design and development include:

1. Draft security requirements for building and integrating APIs
2. Include business logic in design reviews
3. Draft secure coding and configuration practices relevant to your technology stacks

Organizations with mature security programs will often approach building applications and APIs with the mindset that security must be "built in." In practice, this strategy often involves design review, risk analysis, and threat modeling activities prior to production delivery. Organizations will also promote secure design by establishing security requirements for engineering teams. In tandem with those requirements, the organization will also establish more detailed secure coding practices that serve as translations of high-level security requirements down to low-level specifics for a given technology stack. As organizations adopt agile methodologies and DevOps practices, secure design activities may be automated partially or fully depending on the organization's risk appetite.

Despite promoting secure design principles as part of a shift-left approach, over 94% of organizations responding in the Q3 2021 State of API Security report experienced an API security incident in the past 12 months.

Best practices for secure design and development include:

1. Don't reinvent the wheel with security requirements: security requirements are essential for all systems engineering work, and APIs are no different. Use well vetted sources such as NIST and OWASP as a start, and expand on them as you grow in your API security maturity. Many organizations struggle with where to start, particularly if they're new to application security. The OWASP Application Security Verification Standard (ASVS) is one such source that is useful for all types of application designs. Section 13 specifically talks to API design elements but all sections can be useful depending on your architecture. Use OWASP ASVS as a foundation and iterate over time.

2. Be sure to include API integration: identify the overlap of security requirements with building APIs and call out the uniqueness for integration scenarios. Organizations typically see this overlap as they expand their partner ecosystems and work within digital supply chains. Overlap also rears its head with low-code platforms and use of integration platforms. Some code and configuration elements may not be in your realm of control, which is why it's important that you identify this uniqueness, assess relative third-party risk, and implement appropriate mitigations such as increased monitoring focus or contractual language to cover the organization in case of security incident.

3. Streamline threat modeling of APIs: secure design review process traditionally includes threat modeling to identify all types of threats to a system and determine appropriate mitigations. Producing accurate threat models requires a great deal of security and non-security subject matter expertise for all API functionality and integrations. Threat modeling difficulty increases exponentially with the use of third-party dependencies (which include their own dependencies) and partner ecosystems. Organizations also run into difficulty scaling threat modeling processes to support agile development methodologies and DevOps practices. Simply stated, the system and all its APIs will change faster than you are able to generate or update threat models. Streamline and automate your threat modeling process as much as possible. Ideally any threat modeling tooling you use should generate threat models that are machine parsable. You will also likely need to confine threat modeling activity to subsets of your most critical or exposed APIs, and likely only done periodically.

4. Don't skimp on the prescriptive guidance for engineering teams: security requirements are useful to security and risk teams for defining "good" or compliant designs, but they are too high level for development and infrastructure teams. Organizations must define secure coding and configuration practices that are translations of those higher-level security requirements for authentication, authorization, input validation, input filtering, output escaping, error handling, encryption, and other requirements. Define secure practices that explicitly detail what functions, libraries, SDKs, infrastructure configurations, and external controls such as Identify and Access Management are mandated or recommended for use when building or integrating APIs. Use vendor technical documentation, language guidance, and OWASP reference as a starting point and iterate. Newer technologies often lack such explicit security guidance, so you'll need to mitigate security risk using other approaches such as runtime protection in those deployments.

5. Include business logic in design reviews: when you perform secure design reviews, ensure that you are evaluating business logic and end to end API flows for susceptibility to abuse. When building or integrating APIs, you must also consider how functionality might be misused or abused. Identifying, triaging, mitigating, and remediating vulnerabilities in your own custom APIs is different from patching vulnerabilities in vendor supplied software. Security issues may also only manifest themselves in a complete system after code has been built and deployed, which is where an organization must stress fast detection and response, not just pre-deployment analysis.

API documentation

The top 3 recommendations for API documentation include:

1. Use machine formats like OpenAPI Specification (OAS)
2. Repurpose API schema as a basic testing approach and protection approach
3. Have a contingency plan for documentation discrepancies and API drift

API documentation serves a range of security and non-security purposes throughout the API lifecycle. Documentation is useful for the application and API teams that are building or integrating APIs. Adequate documentation also provides benefits to a range of activities including design reviews, security testing, operations, and protection. API documentation should be created in machine formats such as OAS for REST APIs. The machine formats allow for auto-generation of documentation as part of design and mocking, and they are also parsable by other testing and protection tooling. Like all forms of documentation though, teams inevitably neglect to document APIs or new functionality as they iterate. This reality of API documentation process leads to a type of environment drift, or API drift, that leaves massive gaps in your API inventory and security posture.

Best practices for API documentation include:

1. Use machine formats for documentation: when generating API documentation, opt for machine formats and schema definitions as opposed to traditional documentation or visual diagrams. Most commonly for REST APIs, these machine formats include Swagger or OAS. Depending on your API design, development or publishing tooling, other formats like RAML or API Blueprint may be present. And if you are exploring GraphQL APIs, then also expect to work with GraphQL schema definitions. Traditional documentation can be useful for reviews by a less technical audience, but such forms of documentation are not easily maintained. The API schema definition formats are designed for quick generation of documentation as part of API design and mocking that is also reusable for testing, integration, publishing, and operations.

2. Use API schema validators but acknowledge limitations: API schema validators can find common issues related to formatting or overlooked parameters, but they are not a panacea. View them as a type of static analysis, much like linters or quality checkers within IDEs. Schema validators are inherent in tools like Postman and API management (APIM) platforms like Apigee, but they can't find all types of API issues, let alone security issues or logic flaws. Organizations may also opt to trigger schema validation as part of secure build pipelines. These validation tools are only as good as what you document as part of API design and development. Inevitably, you may be lacking schema definitions as your API ecosystem expands to include partner integrations, external API dependencies, and third-party SaaS services.

3. Prepare for documentation discrepancies and API drift: organizations of all sizes and across verticals regularly face difficulties with scaling and operationalizing API design, development and publication. API drift is a type of environment drift where the current state of APIs in

production does not match what is expected or documented. It is an inevitability with development turnover, outsourcing, and acquisition. Even if your organization is able to obtain API documentation and schema definitions, they may not be complete. It is possible to publish an API absent any schema definition, and not all API parameters need to be defined. API teams may also document an API fully at the initial launch but then fail to keep up with versions over time due to conflicting work priorities. The only way to address this gap is to monitor your environments and API traffic in runtime or seek tooling that can auto-discover APIs, auto-generate schema definitions, and produce an API inventory.

API discovery and cataloging

The top 3 recommendations for API discovery and cataloging include:

1. Discover APIs in lower environments and not just production
2. Include API dependencies, aka third-party APIs
3. Tag and label APIs and microservices as a DevOps best practice

Automated discovery of API endpoints, parameters and data types is crucial for all organizations since APIs are the primary mechanism for powering business logic and data exchange. API documentation, while a best practice in itself, may not be done consistently. Documentation may be absent entirely or out of date. Adequate API schema definitions may not be available to you as a result of siloed development and security efforts. Or third parties may not make them available to you. Your organization's API catalog is much more than the APIs mediated by API management and API gateways since APIs may be built, acquired, or integrated outside of formal process. An accurate API inventory is critical to many aspects of IT within the organization. Compliance, risk, and privacy teams will require API inventory, particularly as they must answer to regulatory bodies. Security teams also need API inventory so that they can have a realistic view of their attack surface and risk posture to help prioritize the wide range of API security activities that must be accounted for.

Based on results from the Q3 2021 State of API Security report, 85% of organizations surveyed lack confidence that their API inventory is complete. Incomplete API inventory leads to massive security gaps in an organization's API security strategy.

Best practices for API discovery and cataloging include:

1. Discover non-production environments, not just production: it's critically important that you track lower environments including QA, UAT, staging, SIT, and pre-production in addition to your production environments. Attackers know that non-production environments often have fewer relaxed or no security controls, yet APIs in those environments may still allow access to similar sets of functionality and data. Organizations often set up lower environments with minimal hardening to help promote rapid development and integration to meet production

goals and release schedules. Lower environments may also be Internet exposed which further elevates the security risk.

2. Get into the habit of tagging and labeling assets: the practice of labeling and tagging creates a type of virtuous cycle, becoming incredibly useful in DevOps practices and git-based workflows. Tags and labels are useful for a wide range of activities throughout the API lifecycle including:

    - Controlling versions of developed application code and infrastructure-as-code as seen in GitOps workflows
    - Powering DevOps style release patterns such as canary deployments and blue-green deployments
    - Informing access controls, traffic routing rules, and microsegmentation policies for the containers and container clusters that often power APIs and microservices.
    - Routing defect tracking tickets to appropriate owners and speeding up remediation
    - Informing API management policies and monitoring capabilities
    - Informing data security protections

3. Include the API dependencies of your APIs: expand beyond just homegrown APIs to also include APIs from open-source software, acquired application packages, and third-party SaaS services. API security concerns don't begin and end with just your custom-built APIs. Vendor risk attestation and contractual language are useful primarily as reactive measures that provide for legal recourse, and they provide minimal guarantee at the technology layer. Organizations are inherently limited by the configuration options that are within their realm of control for third-party services. This limitation does not absolve the organization from security risk though. Significant gaps often exist between perceived design of an application and its APIs as opposed to the delivered, integrated system. The combination of built, integrated, and acquired APIs defines the digital supply chain that all organizations work within.

4. Repurpose what you have as a start: most organizations have some data sources they can tap into to build a basic inventory. Commonly, these data sources include the catalog of APIs and data sources within API management and integration platforms. Traffic analysis may also be useful, as can dynamic testing tools run as part of application scanning. Using those methods, you will need to ensure you are scanning all of your known network address space to be reasonably sure you are finding all APIs. Stay away from manually updated or static data sources, which includes many asset databases and configuration management databases (CMDB). Repurposing the data you collect for an API inventory will be an extensive and ongoing exercise in data aggregation, correlation, and analytics, but it can be a stopgap solution until you are equipped to procure an API security offering with discovery capabilities.

Security testing

The top 3 recommendations for security testing include:

1. Statically analyze API code automatically as part of version control and CI/CD
2. Check for known vulnerable dependencies in your API code
3. Dynamically analyze and fuzz deployed APIs to identify exploitable code in runtime

Often viewed as the backbone of an application security program, security testing is a significant focus area of many organizations' API security strategies. The emphasis on investing in security testing tooling and integrating it as part of development and release processes has only grown as industry has pushed the ideal of shift-left more heavily. While it is possible to scan for certain types of security issues automatically, particularly known vulnerabilities in published software, this type of scanning is less useful for the world of APIs. Traditional scanning technologies struggle with parsing custom developed code, since design patterns and coding practices vary per developer. As a result, organizations often struggle with high false positive and false negative rates. No scanner is adept at parsing business logic, which also leaves organizations exposed to major forms of API abuse. Use traditional security testing tools to verify certain elements of an API implementation such as well-known misconfigurations or vulnerabilities, but you must operate these tools with awareness of the limitations. Traditional testing tools often fail to identify flaws, or zero-day vulnerabilities, in the application and API code you create.

Shift-left aimed to increase security awareness and reduce friction between security and development teams. But pushing too far left or too much left also creates friction and overloads engineering teams.

Best practices for security testing include:

1. Repurpose vulnerability scanning to identify API infrastructure: most organizations have established vulnerability assessment and vulnerability management (VA/VM) scanning capabilities. These services are helpful for identifying some misconfigurations and well-known vulnerabilities, typically reported as common vulnerabilities and exposures identifiers (CVE IDs), but this information applies only to published software. For custom API development or integration work, vulnerability scanning benefits are limited. These scanning services can still be useful for identifying exposed servers or workloads that may be listening on well-known TCP ports like 80 and 443 for API requests. Bear in mind that API services may also be configured to listen on other TCP ports, so scanning larger port ranges is advisable. Vulnerability scanners should also support ephemeral and containerized environments to adequately assess API hosting infrastructure.

2. Analyze API code automatically where possible: analyze code automatically with static analysis tools like code quality checkers and static application security testing (SAST) upon code commit in version control systems such as git and/or in CI/CD build pipelines. If you are reviewing code manually, the process will quickly hit a wall with the rate of change most

organizations see in their code and APIs. Scan the integrated code base as part of build within CI/CD to obtain the most accurate scan, but some organizations also opt to scan pieces of code as they are committed to version control for speediness. A code quality checker is the least purpose-built, but they are often plentiful in organizations since many design and development tools include native code quality checking capabilities. SAST may be delivered through language-specific linters or a commercial-grade scanning offering. Regardless of the tool you select, prepare for high numbers of findings of potential conditions and false positives, particularly if a codebase has never been scanned. Static analyzers notoriously need tuning to be used effectively. Static analysis will not be able to cover business logic flaws by design.

3. Run fuzzing and dynamic testing against deployed APIs: absent code scanning, the other approach to testing custom APIs is the use of fuzzers and dynamic application security testing (DAST) tools. Fuzzers are difficult to configure properly and require subject matter expertise to run effectively. However, fuzzing typically results in more thorough testing and identifying a wide range of exploitable conditions in code. The time it takes for a fuzzer to run to completion can be unpredictable, and subsequent runs can produce different results due to the number of variables in play. DAST fairs slightly better, since tools, particularly commercial-grade options, are designed to be easier to get started with. When automating the scanning of APIs with DAST, you will need API schema along with recorded traces of an application session or automation scripts like Selenium or Appium to drive the scanner. While DAST scanners can be effective with traditional web application designs, they will often fail to understand how to exercise APIs. It is common to see a DAST scan run for a few minutes and return trivial results because the scanner wasn't configured properly to navigate API functionality in the right sequence.

4. Check for known vulnerable code dependencies: similar to VA/VM where the goal is identifying CVE-IDs, dependency analyzers and software composition analysis (SCA) scanners can identify known vulnerable open-source software packages and third-party libraries in API source code, infrastructure-as-code, and container images that all play a part in the complete systems that run APIs. Quickly identifying these known vulnerable dependencies helps knock out a wide range of potentially exploitable code that inevitably becomes part of your running APIs and serving infrastructure. Run these dependency analysis tools during code commits, in build, in delivery, and continuously. API infrastructure may be mutable depending where your organization is at with DevOps maturity and pursuit of infrastructure automation. New vulnerable dependencies may be inadvertently introduced making it crucial to run these checks continuously.

5. Pentest APIs periodically or as mandated by regulation: penetration testing, specifically application-scoped and API-scoped engagements, involve a mixture of automated and manual testing techniques. It should be handled by those with appropriate subject matter expertise. If a pentesting firm is offering junior level testers or running VA scanners to analyze your most critical APIs, look elsewhere. The interval at which you should or must perform pentests is sometimes outlined by corresponding regulation. Absent compliance or regulatory requirement, it is advisable to coordinate pentest engagements quarterly, semi-annually, or annually for your most critical or exposed APIs.

6. Augment testing further with bug bounties if you want more assurance: some organizations also opt to augment their security testing capacity further with bug bounty programs that are public or private, and possibly coordinated through a crowd-sourced platform. Bug bounty programs can be the subject of debate, and bounty services often provide no guaranteed testing methodology as typically seen with a qualified pentesting firm. Typically, you pay for results, not the engagement and testing activity itself. Still, using the "power of the crowd" continuously with bug bounties can be useful for uncovering API issues that even the most seasoned security experts are unable to find.

Front-end security

The top 3 recommendations for front-end security include:

1. Draft security requirements for front-end code including JavaScript, Android, and iOS
2. Store minimal or no data client-side since it is prone to attack and reverse engineering
3. Explore client-side code protections if you've secured back-end APIs

Organizations often attempt to secure and harden the front-end code that is installed on user devices, but this proposition can be difficult given what is in the realm of control of the organization. For mobilized employee apps, this approach may still be technically feasible for bring your own device (BYOD) and corporate-owned, personally-enabled (COPE) scenarios. However, for mobile apps destined for customers, patients, or citizens, an organization has little control over end user devices where client-side code protections are often circumvented. Securing the front-end application, or the API client, that depends on back-end APIs for functionality and data can be useful as part of a layered security approach, but such an approach still has downsides. Some pitfalls of client-side approaches you should be aware of include:

- Client-side code is readily decompiled or disassembled to uncover and understand API endpoints, including any protections you embed in that code.
- Client-side code controls are not feasible for direct API or machine to machine communication
- Techniques like certificate pinning, while sometimes recommended, can complicate certificate rotations, app updates, and back-end traffic inspection.
- Client-side mechanisms can slow down release cadences for mobile apps and complicate public app store vetting processes.
- Client-side challenges like CAPTCHA are readily bypassed or farmed out to solving services. Client-side behavior analytics and machine tracking inadvertently create privacy concerns.

Best practices for front-end security include:

1. Provide security requirements for front ends: for web channels, front-end code is typically built using some form of JavaScript such as Angular or React. Users also typically interact with APIs using mobile applications. Mobile platforms bring their own uniqueness and security of native mobile binaries should also be considered. Similar to ASVS, OWASP also maintains the mobile ASVS (MASVS) that can be a good starting point for defining security requirements for mobile device platforms. Apple and Google maintain secure coding guidance that can be useful for defining your secure coding practices for mobile apps. You should provide guidance on how to exchange data securely with back-end APIs, how to authenticate users on-device, and how to persist data on-device.

2. Presume client code and client devices are compromised: always operate with the mindset that client-code will be reverse engineered, end user devices are compromised, and data originating from clients lacks integrity. The security risks of these realities are mitigated in varying ways, depending where you want to invest time, energy, and budget. Endpoint protection can quickly become cost prohibitive, and such solutions are not feasible for Internet-facing APIs consumed by the public since you don't "own" consumer devices. For any API, ensure that you are verifying data originating from API clients, filter as appropriate for malicious injections such as SQLi or JSONi, and escape output to avoid certain types of reflected attacks like XSS.

3. Limit the data you store client-side: ideally no sensitive data or intellectual property is stored client-side. Realistically, some pieces of data must be persisted to provide for an adequate front-end user experience. It's common practice to temporarily persist data such as to maintain session state or cache for performance. Attackers know this design practice and will regularly inspect client-side cache and storage for any remnants of sensitive data when reverse engineering apps. Desirable sensitive data includes authentication tokens and session data that can be useful to attackers attempting session hijacking or account takeover (ATO). If you must store data client-side, use hardware-backed cryptographically secure storage to do so. APIs to interface with device-level hardware and encrypt data appropriately are provided by the respective OS vendor and should also be provided to engineering teams.

4. Review client-side protection options after server-side protection: focus first on protecting back-end APIs. We know that client-side code and end-user devices will always be prone to tampering and reverse engineering by attackers. Given enough time, an attacker can circumvent anti-tampering and anti-debugging mechanisms, bypass root or jailbreak checks, potentially defeat app authentication, and parse obfuscated code. Mature organizations are aware of this cat-and-mouse game and bolster back-end security before considering front-end protection options. Some client-side protections can be obtained for low or no cost, such as in the case of Android Studio obfuscation with ProGuard. However, obfuscation by itself will not prevent reverse engineering, it just slows down the process for attackers. System library calls can't be obfuscated since it makes the code unrunnable. As a result, organizations that pursue the path of client-side code protection must also pair obfuscation with anti-debugging and anti-tampering techniques.

Logging and monitoring

The top 3 recommendations for logging and monitoring include:

1. Define all the infrastructure, application, and API elements that must be logged
2. Factor in non-security use cases such as API performance and uptime measures
3. Allocate enough storage for API telemetry, which will lead you to cloud

Logging and monitoring are not specific to just API security, but it is oftentimes an afterthought for even general IT processes. Every interaction that users and machines have with your API tells a story. These story elements include authentication successes and failures, rates of requests, time of day, the location from where requests originate, data stores accessed, and much more. It should be a question of "what should we log?" but rather "how do we extract meaningful signals from logged data?" All of the telemetry you collect ultimately informs detection, incident response, and runtime protection. It is also useful for constructing baselines of what constitutes "normal" so that any outlier events can be quickly identified and resolved. Baselines are useful for analyzing general performance or availability problems but also security issues.

Best practices for logging and monitoring include:

1. Define what elements must be logged: account for the numerous code-level settings and infrastructure configuration as you define what information needs to be logged, at what interval, and how long it must be retained. Ideally, these guidelines are defined as part of the secure code and configuration practices discussed earlier. Realistically, all activity must be logged since attackers may be stealthy in their reconnaissance and attack attempts. You will need to capture full API request and response traffic, and it's not just a matter of alerting on excessive authentication failures.

2. Incorporate non-security logging requirements: to avoid overburdening teams, ensure that your logging requirements also incorporate the needs of API operations or infrastructure teams who are likely more concerned about troubleshooting ability and tracking uptime. There will be some overlap between the needs for non-security and security. Indeed, some indicators of a performance problem, such as error rates, can also be an indicator of compromise in the event that an attacker is probing your APIs. Common logging details and metrics latency, request sizes, response sizes, error rates, and API caller location.

3. Embrace automation for logging configuration: a multitude of "as-code" approaches exist for configuration, infrastructure, and policy you should consider using in your organization to help automate the necessary logging and auditing settings. The "as-code" approaches are also fundamental to most cloud infrastructure and cloud-native designs. Never presume that an acquired software package, cloud-service, or infrastructure component has logging enabled or at a level of detail that is sufficient enough since these features are often left disabled to keep a product more performant.

4. Embrace cloud technology: "cloud" is expansive, and adoption of cloud technology need not be adoption of fully public cloud services. Such is the case for many organizations since it may not appeal to their risk appetite or satisfy regulatory restrictions. "Cloud" also takes the form of cloud-native design patterns and use of cloud-enabling technologies like software defined networking and container platforms. Private cloud, hybrid cloud, and multi-cloud are common strategies in organizations. You need not go "all-in" with one public cloud provider. The volume of data you must capture, retain and analyze to support logging and monitoring leads organizations to elastic storage, cloud scale data analytics and packaged ML to store all the data, analyze the information, and surface meaningful signals. Traditional infrastructure approaches and data storage will simply not scale.

API mediation and architecture

The top 3 recommendations for API mediation and architecture include:

1. Mediate APIs to improve observability and monitoring capabilities
2. Use mediation mechanisms like API gateways to enforce access control
3. Augment your mediation mechanisms with API security tooling that can provide context

While it's possible to directly expose an API via a web or application server, this practice is less common in typical enterprise architectures. API mediation can be achieved through a number of other mechanisms as well, including network load balancers, application delivery controllers, Kubernetes ingress controllers, sidecar proxies, and service mesh ingresses. Design patterns like API facade and front-end for back-ends involve putting a proxying mediation layer "in front of" APIs. Typically, this design pattern is achieved by deploying API gateways that function as reverse proxies, forward proxies, or both. API management suites and integration platforms also make use of API gateways to enable their functionality and enforce policies. Mediation provides a wide range of benefits including improved visibility, accelerated delivery, increased operational flexibility, and improved enforcement capabilities, particularly when it comes to API access control.

Organizations frequently deploy API management and API gateways to mediate their APIs, but these technologies can only analyze API traffic per transaction. Traditional mediation mechanisms fail to provide full context, correlate activity, and identify API abuse.

Best practices for API mediation and architecture include:

1. Mediate APIs to improve observability and monitoring: by virtue of positioning within enterprise architecture, API gateways are deployed in various spots of a network topology and application architecture to mediate inner and outer APIs. Collectively, all these API gateway instances "see" how API callers are consuming your exposed, outer APIs, and how those requests traverse into inner APIs as well as microservices. Rarely is there one gateway unless it

is a monolithic design or enterprise service bus type deployment. Harvest telemetry from your API gateways to improve your monitoring capabilities and create amplifying effects for your non-security and security initiatives.

2. Mediate APIs to enforce access control: API gateways are foundational for providing traffic management, authentication, and authorization mechanisms. Traffic management functions map to well-known network security controls such as rate limits or IP address allow and deny lists. API gateways are also an ideal place to enforce authentication and authorization for APIs, such as OpenID Connect (OIDC) and OAuth2 respectively. Typically, API gateways are paired with external identity and access management (IAM) systems to share the load of storing all types of user or machine identities, authenticating identities, authorizing identities, and maintaining audit trails of all activity. Plan with the notion that machines consuming your APIs (such as in automation use cases or partner integration) can be just as dominant as traditional end user consumption using client front-ends.

3. Adopt API management for non-security use cases: organizations sometimes reach a tipping point where they have too many APIs or too many API gateway deployments that lack standardization and centralization. To bring order to the chaos, organizations will often opt for an APIM offering that brings a broader range of lifecycle capabilities including features to support monetization of APIs, partner enablement, developer self-service, quote management, access control policies, operational workflow, publishing control, and centralized logging. The APIM offering enables and enforces these features via API gateway deployments.

4. Augment API mediation technologies with security-focused controls: organizations historically front-end their mediation layer with web application firewalls (WAF). This approach can provide a level of protection from general web injection attacks, protect partner or developer self-service portals in the API management (APIM) suite, and protect back-end database services used to power the APIM itself. Some APIM offerings also offer lightweight threat protection that are essentially message filters. Much like WAFs, These APIM and API gateway threat protection filters can be useful for blocking some forms of injection, including XML or JSON injection, but rules are typically too static, too generic or not maintained by the vendor. You should look to purpose-built API security offerings that can provide full lifecycle security and API context rather than repurposing traditional controls like WAF.

Network security

The top 3 recommendations for network security include:

1. Enable encrypted transport to protect the data your APIs transmit
2. Use IP address allow and deny lists if you have small numbers of API consumers
3. Look to dynamic rate limiting and rely on static rate limiting as a last resort

Traditional network perimeters were created at the ingress to an organization's datacenters. As organizations move towards an integrated ecosystem of APIs and adopt cloud services those network boundaries erode immensely. Infrastructure is much more ephemeral as well as virtualized and containerized, which makes many network access controls unusable at scale. Network security begins to heavily intersect with identity and access management (IAM) as an organization gets into zero trust architectures. The design goals of zero trust promote that your ability to connect to a given resource depends on what you are doing at a given moment, which is heavily tied to your authenticated context and behaviors within that session. The principles of zero trust and some zero trust focused technologies like microsegmentation or zero trust network access (ZTNA) are sometimes overloaded as "application security." These zero trust technologies are used to control connectivity between workloads or to control connectivity to workloads that power applications and APIs. The level of security protection doesn't go deeper than that.

A primary goal of zero trust architecture is to enforce concepts of least privilege and restrict network access dynamically. However, connectivity must be present in order for APIs to function, and many API attacks still occur in trusted channels and authenticated sessions.

Best practices for network security include:

1. Use encrypted transport to protect the data your APIs transmit: TLS should be enabled for any API endpoints to protect data in transit, ideally version 1.3 but 1.2 at a minimum. All versions of SSL should be disabled due to the number of weaknesses in the protocol or related cipher suites. Legacy infrastructure components sometimes linger within organizations or suppliers, requiring that SSL or older versions of TLS be maintained. Some traffic inspection tooling may also not support more recent encryption protocols, which puts organizations in a bind when they want to maintain visibility over their network traffic. Unfortunately, supporting older protocols and cipher suites exposes the organization to a number of cryptographic and downgrade attacks that can result in encrypted data being viewable by unauthorized parties. Enforce encryption policies through your API mediation layer wherever possible, and ensure legacy protocols and cipher suites are kept disabled. If necessary, refactor or re-architect the supporting infrastructure of your APIs, opting for TLS termination points that allow you to maintain traffic visibility while still mitigating security risk of encryption protocol attacks.

2. Set IP address allow and deny lists for small numbers of API consumers: a common control used to restrict what API callers can even make a network connection to your API, let alone authenticate or transact with it, is the IP address allow and deny list. This network security control is often found within APIM, API gateways, and network infrastructure elements like a load balancer. The lists may also be based on threat intelligence feeds of known malicious IP addresses. IP address allow and deny lists can be useful if your API is interacted with by a limited set of partners or consumers. If your API is public or open though, it is extremely difficult to scale this type of control for the larger Internet. You may opt to block certain blocks of IP addresses allocated to geographical regions, but know that attackers can circumvent IP address deny lists with proxies and VPNs. Attackers will also spin up ephemeral workloads in cloud providers to launch their attacks, which is often allowed address space as organizations adopt cloud technology. Attackers may also use networks of compromised endpoints to perpetuate attacks. In practice, IP address allow and deny lists need to be much more dynamic and paired with behavior analysis and anomaly detection engines to be effective.

3. Use dynamic rate limits and set static rate limits selectively: rate limits can be useful for restricting consumption of APIs for a smaller set of API consumers or partners. Rate limits however often become difficult to scale for larger user bases. The issue is prevalent enough that it appears on the OWASP API Security Top 10 as API4:2019 Lack of Resources & Rate Limiting. You will likely need to relax rate limits, observe traffic consumption to inform a baseline, and tighten limits over time. For organizations where API consumption can experience spikes, such as in retail and with new product launches or seasonable demand, setting effective rate limits can be a lesson in frustration. Realistically, rate limiting mechanisms should be much more dynamic, granular, and based on actual consumption patterns. Setting blanket static rate limits can result in impeded application functionality that directly impacts the organization's business, not to mention attackers will throttle their attempts to evade those limits.

4. Enforce network security via infrastructure, not in code: most network security controls must exist external to the API code. If ever there was evidence that API security is not solely the responsibility of development teams or issues were the result of "poor coding practice," network security techniques are a prime example. Transport protection, rate limits, and IP address allow/deny lists are almost exclusively defined in infrastructure and API mediation mechanisms. One could argue that it might be addressable using infrastructure-as-code, but that form of code is more likely to be owned by network engineering, infrastructure, or API operations teams, not application development.

Data security

The top 3 recommendations for data security include:

1. Use encryption selectively and transport protection suffices for most use cases
2. Avoid sending too much data to clients and relying on the client to filter data
3. Adjust for threats like scraping or data inference where encryption is not a mitigation

Data security approaches aim to provide confidentiality, integrity and authenticity of data. If your organization still includes privacy in the data security bucket, then anonymization and pseudonymization are also in scope. Depending on your data security goals and impacting regulation, appropriate techniques for protecting data include masking, tokenizing, or encrypting. Many data security efforts focus on securing data at rest in a system back-end, such as database encryption or field-level encryption. These approaches protect organizations from attacks where the data storage is targeted directly. If your API is designed to only send encrypted payloads as an additional level of encryption beyond transport protection, attackers will still attempt to extract unencrypted data elsewhere, such as in memory, from client storage, or other positions within network topology. These encryption approaches also do not protect the organization from cases where an attacker obtains a credential or authorized session since the data will be decrypted for them when accessed through an API

> **85% of organizations lack confidence that they know which APIs expose sensitive data based on results from the Q3 2021 State of API Security report. Exposures of sensitive data can lead to significant regulatory penalties, large scale privacy impacts, and brand damage.**

Best practices for data security include:

1. Use encryption selectively or as mandated by regulation: history is riddled with many failed crypto implementations and misconfigurations that were exploited by attackers. Key management is already a complex endeavor, but matters only get worse in the world of automation and API communication where time is of the essence and prompting for key material is a non-starter. As a result, application teams sometimes make the mistake of storing key material in unsecured locations, such as in code, in client-side storage, or in general purpose cloud storage, all of which are frequently harvested by attackers.

2. Transport protection should suffice for most business and security cases: most organizations have a hard enough time implementing TLS. Encrypting message bodies or payloads on top of encrypted transport can be overkill. This added layer of encryption requires a high level of effort to do effectively, not to mention that it can also add latency or can create integration headaches with other systems. More often than not, attackers defeat such mechanisms by harvesting exchanged key material that the client needs in order to encrypt and decrypt data from back-end APIs.

3. Always use well-vetted algorithms and encryption libraries: many implementation details of encryption are important to get "right" to avoid incidents such as salt sizes, rounds of salting, initialization vectors, key sizes, and more. These considerations also vary for symmetric encryption and asymmetric encryption. NIST provides some guidance on encryption, but you will also need to augment with specifics related to your technology stack. Guidance evolves over time as new encryption exploits surface or weaknesses are uncovered in cipher suites. You must also properly maintain encryption tooling and code libraries since flaws can be uncovered over time, such as OpenSSL and the Heartbleed bug. This best practice is not just a developer problem since encryption tooling and libraries are used in many layers of the technology stack.

4. Avoid sending too much data to API clients: back-end APIs are sometimes designed to serve up a great deal of data in responses to API calls, and it becomes the duty of the front-end client code to filter out what should be visible based on the goals of the user experience (UX) or permission levels. This design pattern goes against API security best practice since that data is fully visible by observing API requests and responses. Attackers commonly reverse engineer front-end code and sniff API traffic directly to see what data is actually being transmitted. The issue ranks as one of the OWASP API Security Top 10 as API3:2019 Excessive Data Exposure because it is so commonplace. Don't send too much data, particularly sensitive or private data, to front-end clients and always presume that they are compromised. Filter data appropriately in the back-end and send only the data that is necessary for that particular API consumer.

5. Plan for risks of scraping, data aggregation, and data inference: a few pieces of data may be innocent, but when data is collected and aggregated at scale, the situation becomes much more precarious. The resulting data sets quickly become privacy impacting and brand damaging. No quick fixes exist for these data security and privacy risks. Mitigation requires a combination of many techniques like limiting how much private data you collect in the first place, using rate limiting effectively, and limiting how much data you send to API clients. Attackers will use automation to their advantage to scrape and aggregate data in large volumes. Attackers employ a plethora of tooling including intercepting proxies, debuggers, Python scripting, and command line clients like cURL and HTTPie. Scraped data is also useful in other attack techniques such as brute forcing, credential stuffing, phishing, and social engineering. To detect and stop abnormal API consumption like scraping, you will need to seek API security tooling that continuously analyzes API telemetry, analyzes behaviors, and identifies anomalies.

Authentication and authorization

The top 3 recommendations for authentication and authorization include:

1. Continuously authenticate and authorize API consumers
2. Avoid the use of API keys as a means of authentication
3. Use modern authorization protocols like OAuth2 with security extensions

Authentication and authorization, and by extension identity and access management (IAM) are foundational to all security domains, including API security. As organizations have shifted towards heavily distributed architectures and use of cloud services, the traditional security best practice of locking down a perimeter has become less useful. IAM is now used heavily to control access to functionality and data, and it is also an enabler of zero trust architectures. When considering security best practices for authentication and authorization, remember that you must account for user identities as well as machine identities. While it is possible to challenge a user for additional authentication material in a session, this option is not available for machine communication. Externalize your access controls and identity stores wherever possible, which includes mediation mechanisms like API gateways, user and machine identity stores, IAM solutions, key management services, public key infrastructure, and secrets management. Implementation of these technologies is rarely an application developer responsibility, particularly as you consider the completely integrated system or digital supply chain.

95% of API exploits happen against authenticated APIs based on data from the Salt Security API Protection Platform as detailed in the Q3 2021 State of API Security report. Attackers regularly circumvent access controls and hijack authenticated sessions, spotlighting the fact that API security strategy must focus on more than just authentication and authorization.

Best practices for authentication and authorization include:

1. Authenticate and continuously authorize API consumers: access control has always involved authentication and authorization. Authentication (AuthN), involves identifying the requester of a given function or resource and challenging that entity for authentication material or credentials. Authorization (AuthZ) involves verifying whether that authenticated entity actually has permissions to exercise a function or read, write, update, or delete data. Traditionally, both were handled at the start of a session. In the web world, and by extension APIs, sessions are stateless. The operating environments of back-ends and front-ends are not guaranteed and often ephemeral. Increasingly, environments are also prone to integrity issues or compromise, hence the rise of zero trust architectures. As a result, you must continuously verify whether a user or machine identity should have access to a given resource and always presume the authenticated session might be compromised. This approach requires analyzing behaviors of a given session for an API consumer, and potentially terminating that session, requiring step-up authentication, or blocking access as appropriate.

2. Use modern authentication and authorization protocols: use newer authentication protocols like OpenID Connect and authorization protocols. Using sufficient authentication token lengths and entropy are also critically important to mitigate risk of session guessing or brute forcing. JSON Web Tokens (JWT) are a popular choice as a token format within OAuth2. Two-factor authentication (2FA) should also be in your arsenal for authenticating users that consume APIs. 2FA challenges are delivered through email, SMS, or Time-based One-time Password (TOTP) authenticator apps. Certificate-based authentication is more common for machine-to-machine communication and automation scenarios where it is not technically feasible to prompt for authentication material. Mutual TLS (mTLS) is also prominent for microservice authN and authZ as seen within Kubernetes and service mesh. Never rely on mechanisms like basic authentication or digest authentication. Attacks against these older authentication mechanisms are well documented, and they are trivial for attackers to defeat.

3. Don't rely on API keys as authentication: API keys are commonplace in the world of APIs, and they are seen frequently as a means of connecting partners, connecting client apps to back-end APIs, and enabling machine to machine (or direct API) communication. API keys are easily harvested by attackers through reverse engineering client-side code and sniffing network traffic if keys traverse unprotected networks and the Internet. API keys alone are not a sufficient form of authentication and should be used primarily as a form of version control. If you rely on API keys, ensure that you monitor consumption, generate new API keys, and revoke old API keys or API keys of malicious consumers appropriately. Realistically, API keys should be paired with additional authentication factors such as certificates or other authentication material.

4. Set reasonable idle and max session timeouts: idle session timeout controls how long a given session with the back-end can stay live without receiving requests from the client until a user or machine is required to re-authenticate. Max session timeouts control the total time a session can be live with the back-end regardless of whether the session is active or idle. Idle session timeout recommendations range anywhere from 5 to 30 minutes depending on exposure of the API, business criticality, and data sensitivity. Max session timeouts are usually in the range of a few hours or days. Some organizations opt for shorter session lifetimes, but such an approach requires a trade-off with UX since you will be forcing users to re-authenticate more frequently. You must consider these lifetimes for all session identifiers, authentication tokens, and refresh tokens throughout the technology stack. The intent of controlling session timeouts is to reduce the time window for attackers to steal session identifiers and hijack authenticated sessions. Active session identifiers and authentication tokens are just as valuable to an attacker as an original credential and can easily be used to obtain access to API and data.

5. Weigh the pros and cons of session binding: binding IP addresses of API consumers to session cookies and authentication tokens can provide some security benefit. If a bound session identifier or authentication token is stolen by an attacker, and the attacker attempts to reuse that authenticated session from another machine with a different IP address, the API request will be blocked since the request isn't coming from the original IP address. Session binding has an unintended side effect of limiting mobility. If a given API consumer uses multiple machines or mobile devices normally, they can be forced to authenticate excessively which becomes damaging to UX.

6. Use additional secrets in authorization flows and nonces in requests: adding additional secrets in authentication flows helps reduce the risk of token interception and replay attacks.

OAuth2 provides this type of protection with proof key for code exchange (PKCE). If you are using OAuth for authentication in your mobile app, consider employing PKCE to mitigate the risk of token interception and replay attacks. PKCE is also slated to become mandatory in OAuth 2.1. Using nonces in requests also helps reduce the risk of message replay attacks and cross-site request forgery (CSRF) attacks. You can also use one of the many implementations of anti-CSRF mechanisms within code libraries and frameworks. It's often simply a matter of ensuring you've enabled the mechanism.

Runtime protection

The top 3 recommendations for runtime protection include:

1. Enable threat protection features of your API gateways and APIM if available
2. Ensure that DoS and DDoS mitigation is part of your API protection approach
3. Go beyond traditional runtime controls that are dependent on rules, and make use of AI/ML and behavior analysis engines to detect API attacks

Runtime protection, sometimes referred to as threat protection, is often delivered through network-based proxies like API gateways and WAFs. These mechanisms typically rely on message filters and static signatures, which can catch some types of attacks that follow well-defined patterns but miss most forms of API abuse. Any runtime protection you consider deploying should be much more dynamic and learn continuously. Runtime protections may use signatures for well-known and well-defined attack patterns, such as presence of malicious characters that indicate injection attack attempts. Runtime protections should encompass more than just message inspection and filtering though. Protections should be useful for identifying misconfigurations in API infrastructure as well as behavior anomalies like credential stuffing, brute forcing or scraping attempts by attackers.

Stopping attacks in runtime is the highest priority for the majority of organizations as described in the Q3 2021 State of API Security report. Unfortunately, traditional runtime controls such as WAFs and API gateways provide little confidence in protection, with only 16% of organizations finding these controls to be very effective in preventing API attacks.

Best practices for runtime protection include:

1. Look beyond traditional security controls and attack signatures: most organizations have a presence of traditional runtime security controls including such as intrusion prevention systems (IPS), next-gen firewalls (NGFW), or web application firewalls (WAF). IPS and NGFW try to cover a broad range of protocols and attacks, not just web traffic, which limits their efficacy for APIs. Of these traditional controls, WAF is the most purpose-built for web application and API traffic, but WAFs were designed in a time when web applications were largely static. As design patterns evolved and applications became more dynamic and API-centric, some vendors began including signatures and rules to cover APIs. However, such rules are based on per-transaction analysis and pattern matching, and rules are typically too generic to cover the unique business logic that most organizations build into their APIs. Use IPS, NGFW, and WAF if you must, but expect a low-bar of protection that mainly covers common injection attacks like XSS and SQLi. JSONi and XMLi may be covered, but it is not a given and could fall to your API gateway. You must also ensure that they support the API protocols in use in your organization. If a WAF only supports SOAP APIs, and you're creating REST APIs, it's useless as an API protection. Ultimately, your runtime protection approach should go beyond these traditional technologies and make use of AI/ML and behavior analysis to detect API attacks.

2. Use threat protection features of your API gateways and APIM if available: similar to WAFs since they are both network-based proxies, ensure that you are enabling the threat protection rules and message filters within your API mediation technologies. It's still a low bar for API protection, but it is more API-focused than WAF. Unlike WAFs however, these rules are likely not as well maintained by the vendor. Signature updates are relatively rare. To be effective, they also likely require tuning based on a given API schema to control allowable parameter values within API requests. Because this approach is difficult to scale operationally, organizations will sometimes leave threat protection mechanisms with default settings or disabled entirely.

3. Don't rely on rate limiting and traffic management to stop attacks: organizations that attempt to operationalize rate limits inevitably hit a wall, particularly for customer APIs that are Internet-facing. Proxying all traffic and examining each transaction in isolation, it's impossible for the network-based controls that implement rate limits to understand behaviors and intent of the API consumer and provide context. Attackers regularly throttle their attack requests to evade rate limits, and rate limiting is a relatively low bar to API security. Like "the club" that was one marketed heavily as a theft deterrent device, they might slow down a less-skilled attacker, but such approaches only delay the inevitable. If you are depending on rate limiting, make sure that you are pairing it with traffic analysis and anomaly detection so that rate limits can be set much more dynamically and adjusted per requester.

4. Plan for denial of service (DoS) attacks against exposed APIs: attackers will use DoS and distributed DoS (DDoS) attack techniques to reduce availability in your APIs. Traditionally, protections are sought for volumetric, protocol, or application-layer attacks. DDoS mitigation services and cloud scrubbing services might address volumetric and protocol forms of DoS but can leave application-layer exposed. For vendors that claim to cover application-layer, ensure that they are able to parse API context. API parameters are highly unique per organization, based on the business logic they create and how they integrate other services.

Parameters within API requests vary greatly from one organization's architecture to the next, making application-layer DoS for APIs very nuanced.

5. Explore behavior analysis and anomaly detection: as organizations embrace APIs more rapidly, they quickly realize the need for machine-driven data analytics and behavior analysis to understand normal API consumption and identify attackers abusing APIs. The algorithms must be informed by API metadata as well as API traffic collection, continuously learn, and make decisions dynamically based on the organization's unique business logic. Machine-driven detection and protection should also be built into a larger platform of services and integration so that an appropriate mitigating action, such as setting a dynamic rate limit for an abusive requester, can be implemented temporarily at the appropriate network ingress of the overall architecture. Even mature organizations with development resources and data scientists quickly hit a wall trying to develop such detection and integration. You will inevitably need to explore API security tooling to fill this gap.

Security operations

The top 3 recommendations for security operations include:

1. Account for the non-security and security personas involved in the complete API stack
2. Create API-centric incident response playbooks
3. Spare your SOC from burnout by surfacing actionable API events and not dumping data

Security operations, or SecOps, capabilities in organizations can be a mixed bag. SOC analysts that are full stack are in very short supply, much like the illusive full stack DevOps or DevSecOps engineer. Specializations in technology stacks and understanding of attack pattern specifics are inevitable if not necessary. This reality of the modern SOC increases the need for collaboration within the SOC but also with other teams within the organization. SOC analysts must often depend on application development and API protect teams who best know the application architecture and logic of APIs, which becomes critical in digital forensics and incident response. SOCs may also be outsourced to third parties, as in the case of managed security service providers and virtual SOCs, which can further complicate workflow, data feeds, and integration. You will need to emphasize the people and process aspects of SecOps more than technology, and don't just approach the exercise as "getting a feed into Splunk."

Best practices for security operations include:

1. Account for multiple personas and work streams in the organization: non-security and security teams don't need to see all API event data, which stresses the importance of role-based access control within an API security offering itself. Some data may be privileged or be bound by regulatory restriction. In other cases, providing too much data can lead to information overload and slow down regular work. Provide teams the appropriate API-related data that they need to do their job, and provide it within the tooling they use as part of their daily routine to avoid disruption. As an example, infrastructure teams may only care about event

data related to load balancer misconfigurations, product teams may be concerned with APIM policy misconfigurations, and development teams may only be concerned with code-level vulnerabilities.

2. Create API-centric incident response playbooks: ensure that you document digital forensics and incident response (DFIR) processes for how to respond to the inevitable API attack patterns. If you've already matured your SecOps strategy to include the use of security orchestration, automation and response (SOAR), then also automate some of the workflow items as part of IR. Shutting down an API that is the target of malicious activity is rarely a prudent business decision, not to mention it reduces your ability to gain additional intelligence about an attacker and their techniques. Rather than blocking traffic wholesale, you will likely want to employ more precision such as throttling just the suspicious API caller, challenging them with additional authenticator factors, or monitoring their behavior more heavily. Create IR playbooks for common API attack patterns including application-layer DoS, brute forcing, credential stuffing, enumeration, and scraping.

3. Surface actionable API events, don't just dump data into SIEM: you should consider the funnels of data being ingested into your SIEM from the specialized tooling that is used across the organization. Assign priority levels based on risk-scoring and correlate events to produce useful signals. Realistically, this level of analysis and prioritization requires a Big Data approach, with cloud-scale storage and use of AI/ML to analyze the data at scale. Too often, organizations dump all their log and event data into their SIEM only to find that the SIEM can't keep up or can't provide meaningful, actionable signals. SOC teams quickly get overwhelmed as a result. In some cases, cybersecurity efforts actually focus on reducing the number of feeds into the organization's SIEM so that the SOC can be more effective in their job of triaging and responding to security events.

Summing up the best practices

Enabling API security covers more than then areas of focus, and each is arguably just as critical as the next. You may opt to emphasize sets of best practices where they already have technology investments or manpower. Frequently for organizations, their API security strategy focuses heavily on security testing, API mediation, or network security. You can't do everything at once, so where do you start? Some suggestions on how to scope the problem and prioritize activities include:

- Do security test your APIs, but know that you will also need runtime protection to catch changes that don't go through standard build process and abuses that testing tools aren't designed to find.
- Ensure that you are covering all of your environments and your digital supply chain, which is more than just the APIs mediated by your API gateways or API management suite.
- If you do nothing else, focus on runtime protection as a way to "stop the bleeding," slow down attackers, and buy time for application and API teams.

To avoid being overwhelmed, pick a few best practices areas as a starting point that are most familiar. Expand over time the other sets of best practices since any other approach will leave gaps in your API security strategy. Ideally, you should consider purpose-built API security tooling that addresses the many elements of API security. API security tooling should be able to offer a range of capabilities throughout the lifecycle and provide the necessary context to stop attacks and data exposures for your organization's unique API business logic.

External Resources

Topic Area	Domain	Organization	Link
API security	Secure design	CSA	Security Guidelines for Providing and Consuming APIs | CSA
Credential stuffing	Authentication and authorization	OWASP	OWASP Credential Stuffing Prevention Cheat Sheet
GraphQL	API protocols and data formats	OWASP	OWASP GraphQL Security Cheat Sheet
Incident response	Security Operations	NIST	Computer Security Incident Handling Guide
Injection	Input validation and filtering	OWASP	OWASP Injection Prevention Cheat Sheet OWASP SQL Injection Prevention Cheat Sheet
JSON Web Token (JWT)	Authentication and authorization	OWASP	OWASP JWT Cheat Sheet for Java
Mass assignment	Input validation and filtering	OWASP	OWASP Mass Assignment Cheat Sheet
Microservices and API security	Security architecture	NIST	Security Strategies for Microservices-based Application Systems
Microservice security	Security architecture	OWASP	OWASP Microservices Security Cheat Sheet

Mobile app and API security	Security verification	NIST	Vetting the Security of Mobile Applications
Mobile app security	Security verification	OWASP	OWASP Mobile Security Testing Guide
Redirects and forwarding	Input validation and filtering	OWASP	OWASP Redirects and Forwards Cheat Sheet
REST	API protocols and data formats	OWASP	OWASP REST Assessment Cheat Sheet OWASP REST Security Cheat Sheet
Server-side request forgery (SSRF)	Input validation and filtering	OWASP	OWASP SSRF Prevention Cheat Sheet
Threat modeling	Secure design	OWASP	OWASP Abuse Case Cheat Sheet Threat Modeling Manifesto
Web services	API protocols and data formats	OWASP	OWASP Web Service Security Cheat Sheet
XML	API protocols and data formats	OWASP	OWASP XML Security Cheat Sheet OWASP XML External Entity Injection Prevention Cheat Sheet