International Journal of Computational Research in Science and Technology
E-ISSN: 3108-0871, Volume 2 Issue 5
Sep-Oct 2026, PP 01-9

API Security: A Review of Vulnerabilities, Detection Methods, and Emerging Defences

Prajwal Turankar, Prof. Bhosale S.B, Prof. Bangar A.P, Prof. A.A. Khatri

Dept. of Computer Engineering,

Jaihind collage of Engineering, Pune,India

Abstract. Application Programming Interfaces (APIs) enable communication between applications, services, and distributed systems. As web, mobile, cloud-native, and microservice architectures increasingly depend on APIs, their endpoints also provide opportunities for unauthorized access, data exposure, service disruption, and misuse of security mechanisms. This paper presents a review of four recent peer-reviewed sources on API security (a systematic review of security-API misuse detection, a systematic review of web API vulnerabilities and developer challenges, an overview of API protection across the software development life cycle, and a discussion of emerging technologies) and uses their combined findings to build a case for, and a design of, an integrated two-module security framework: a design-time control module aligned with the SDLC, and a runtime protection and monitoring module. The review covers recurring vulnerabilities (broken object-level authorization, weak authentication, excessive data exposure, injection, misconfiguration, inadequate rate limiting, poor asset management, and insufficient logging), the human and technical causes of insecure implementation, static, dynamic, heuristic, and machine-learning-based detection approaches, and emerging directions including AI/ML, zero trust, and adaptive authentication. The paper then details a proposed framework architecture, data and tooling requirements, an evaluation plan with specific metrics, expected outcomes, and a discussion of limitations and future work. The overall conclusion is that API security should be treated as a continuous lifecycle responsibility, and that emerging technologies should complement, not replace, fundamental controls.

Keywords: API Security, Web APIs, Authentication, Authorization, OWASP, Vulnerability Detection, Secure Software Development, Zero Trust, Machine Learning, Continuous Monitoring.

I. Introduction

1. Background

An Application Programming Interface is a defined mechanism through which software components exchange requests and responses. APIs support mobile applications, websites, cloud services, payment systems, and communication between microservices. They make software integration easier, but they can also expose sensitive data and business functions when security controls are missing or incorrectly implemented. Almjnoony et al. describe web APIs as important attack surfaces and connect recurring vulnerabilities with implementation practices, usability problems, and developer challenges [1].

2. Problem Statement

API security is broader than placing a login screen in front of an endpoint. A secure API must verify the identity of the caller, determine whether that caller is permitted to perform the requested action on the particular resource, validate inputs, limit data returned, protect communication, resist abusive traffic, and produce useful audit records. This creates three structural weaknesses. First, authorization gaps: authentication does not stop a valid user from reaching another user's data when object-level

checks are missing. Second, human factors: confusing documentation, insecure examples, and limited training lead developers to misuse security mechanisms. Third, fragmented coverage: no single tool addresses every layer, and endpoints that the organization does not know about are not monitored or patched.

3. Motivation for a Lifecycle Approach

Mousavi et al. explain that even security-focused APIs can be misused when developers misunderstand their requirements or fail to follow specifications; their systematic review analyzes 69 peer-reviewed studies and classifies security API types, misuses, detection approaches, and evaluation methods [2]. Together with the lifecycle-oriented and future-oriented sources [3,4], this motivates a layered approach in which controls are applied from requirements through maintenance and are supported by continuous monitoring.

4. Scope and Objectives of This Paper

- Describe the main vulnerability patterns affecting APIs.
- Explain why insecure API implementations persist, including human and technical factors.
- Compare static, dynamic, heuristic, and machine-learning-based detection approaches.
- Discuss how security can be integrated across the software development life cycle.
- Propose a two-module framework combining design-time and runtime controls under a shared governance layer.
- Detail a methodology, including requirements, evaluation metrics, and a validation plan, and discuss limitations and future directions.

This is a narrative synthesis and design proposal based on the reviewed sources, not a new systematic review or an independent experiment.

II. Related Work / Literature Review

The four sources cover: (1) systematic research on detecting misuse of security APIs [2]; (2) a PRISMA-guided systematic review of web API vulnerabilities, misuse patterns, and developer challenges [1]; (3) an overview of API threats and security practices across the SDLC, including API gateways and application-performance-monitoring-based visibility [3]; and (4) a discussion of future API security trends, including AI/ML, zero trust, adaptive authentication, and continuous monitoring [4]. They were compared by topic rather than by pooling numerical results, and quantitative claims are not treated as universal performance guarantees.

1. Foundational Concepts in API Security

API security protects the confidentiality, integrity, and availability of data and services exposed through an API. Confidentiality means that sensitive information is disclosed only to authorized parties; integrity means that data and operations cannot be changed improperly; and availability means that legitimate users can access services when needed. Security mechanisms include authentication, authorization, cryptographic protection, validation, traffic controls, monitoring, and secure configuration [2,3].

An important distinction is between vulnerabilities in an API implementation and misuse of a security API by a developer. For example, a web endpoint may fail to check whether a user owns a requested record, while an application developer may misconfigure a TLS library so that it trusts an invalid server identity. Both can undermine security, but they occur at different layers and may require different detection methods [1,2].

2. Common API Security Vulnerabilities

Broken object-level and function-level authorization

Broken object-level authorization (BOLA) occurs when an API does not properly check whether a user may access a particular object, such as an account, file, order, or transaction. An attacker may change an identifier in a request and attempt to retrieve another user's data. Function-level weaknesses occur when a caller can invoke a privileged operation, such as an administrative action, without the required permission. Checks should be enforced on the server for every relevant request and resource; hiding a button in the user interface is not sufficient [1,3].

Authentication and token weaknesses

Weak passwords, insecure token handling, inadequate expiration and revocation, and poorly implemented authentication flows can allow impersonation or account takeover. The reviewed literature recommends established identity protocols, appropriate multi-factor authentication where suitable, careful token lifecycle management, and consistent authentication requirements across endpoints [3,4]. Authentication alone does not establish that the caller is authorized to access a particular record or function.

Excessive data exposure and mass assignment

An API may return more fields than a client needs, exposing personal or business-sensitive information. Server-side response filtering should return only the data required for the task. Mass assignment is a related risk: client-provided fields may be bound directly to internal objects, allowing users to modify properties they should not control. Explicitly allow-listing writable fields and validating request schemas can reduce this risk [3].

Injection and unsafe input handling

Injection vulnerabilities arise when untrusted input is interpreted as part of a database query, command, or other executable expression. SQL, NoSQL, and XML injection are examples discussed in the lifecycle overview. Developers should validate input against expected types, lengths, formats, and business rules, and use parameterized queries rather than unsafe string concatenation. Output encoding and safe error handling are also important [1,3].

Rate limiting and denial of service

APIs that accept unlimited requests may be abused through brute-force attempts, automated scraping, or traffic floods. Rate limiting, throttling, request-size limits, timeouts, quotas, and resource-aware design can reduce the risk of service exhaustion, and should be tested under realistic conditions so that they restrict abuse without blocking legitimate traffic [3].

Security misconfiguration and asset management

Default settings, excessive permissions, exposed debugging information, unnecessary HTTP methods, and incorrect cross-origin policies can create openings for attackers. Organizations also need an inventory of public, internal, partner, legacy, and undocumented endpoints. Unused or deprecated versions should be retired safely, and an API that the organization does not know exists may not receive security updates or monitoring [1,3].

Insufficient logging and monitoring

If authentication failures, authorization denials, unusual request patterns, and system errors are not recorded and reviewed, an attack may continue unnoticed. Logging should support investigation and incident response while avoiding unnecessary storage of secrets, access tokens, passwords, or sensitive personal data, and alerting should be linked to a documented response process [3,4].

3. Why API Security Problems Persist

The reviewed papers identify both technical and human causes. Security mechanisms can be complex, and developers may misunderstand their correct use. Documentation may omit important constraints or provide incomplete or insecure examples,

and developers may copy code from examples or forums without understanding the security consequences. Limited training and time pressure also result in incomplete controls [1,2].

Architecture adds further complexity. Cloud-native and microservice systems can contain many independently deployed services, each with endpoints, credentials, policies, and dependencies, so security settings may become inconsistent across teams and versions. These observations support a developer-aware approach: secure defaults, understandable documentation, safe examples, reusable security components, and checks built into normal workflows [1,3].

4. Detection and Evaluation Approaches

Static analysis

Static application security testing (SAST) examines source code or related artifacts without running the application. It can identify suspicious data flows, unsafe API calls, missing checks, and insecure configuration early in development, and is useful in pull requests and CI pipelines, but it may produce false positives and can miss vulnerabilities that depend on runtime state or business logic [2,3].

Dynamic analysis and security testing

Dynamic application security testing (DAST) evaluates a running API by sending requests and examining responses. Penetration testing can investigate authorization logic and realistic attack paths, fuzz testing supplies unexpected or malformed inputs, and scanners can identify common weaknesses. Results depend on endpoint coverage, test credentials, test data, and test-case quality [3].

Heuristic and machine-learning-based detection

Mousavi et al. classify security API misuse detection approaches into heuristic-based and machine-learning-based categories and report that existing work uses multiple performance measures and benchmarks [2]. Heuristic approaches apply predefined rules and can be interpretable, but may not generalize to unfamiliar patterns. ML-based approaches can learn patterns from examples or traffic, but require suitable data and attention to false positives, false negatives, drift, and adversarial manipulation.

Evaluation quality

Detection systems should not be judged by accuracy alone. Useful measures include precision, recall, F1 score, false-positive rate, detection time, throughput, and resource overhead. Evaluations should also report the benchmark, data source, API types, vulnerability classes, and test conditions. Mousavi et al. identify a need for more standardized evaluation, while Almjnoony et al. highlight limitations in practical evaluation and adoption of security approaches [1,2].

5. Securing APIs Across the Software Development Life Cycle

API security should be continuous rather than postponed until deployment. The lifecycle-oriented paper recommends integrating security controls from requirements and design through development, testing, deployment, and maintenance [3]. API gateways can centralize authentication enforcement, request filtering, rate limiting, and traffic logging, and monitoring platforms can provide visibility into API usage. These are useful layers, but they should not replace authorization checks inside the service, secure application logic, or appropriate testing; a gateway cannot understand every business rule governing access to each record [3].

6. Emerging Directions

AI and machine learning

The future-oriented paper describes AI/ML as promising for analyzing traffic patterns, detecting anomalies, and supporting automated responses [4]; the APM-oriented paper similarly discusses ML-based anomaly detection and context-aware monitoring [3]. However, an anomaly is not necessarily an attack, and a sophisticated attack may resemble normal traffic. These systems should be tested against representative data, with human oversight for high-impact actions.

Zero trust and adaptive authentication

Zero trust means that network location alone does not establish trust: each request is evaluated using identity, permissions, and context. Adaptive authentication can increase verification requirements when risk indicators change. Implementation must account for latency, service-to-service identities, policy consistency, and operational complexity [4].

Continuous discovery and observability

API ecosystems change frequently, so discovery and inventory management must be continuous. Monitoring can connect request patterns with users, services, and endpoint behavior, helping teams detect shadow APIs, misuse, and unexpected data flows. Observability should be paired with governance: every endpoint should have an owner, an intended audience, a documented security policy, and a plan for deprecation [3].

Standardization and developer-centred security

The systematic reviews point to the need for better alignment between security mechanisms and developer needs, and for more consistent evaluation practices [1,2]. Future work should prioritize clear documentation, secure-by-default libraries, reusable authorization patterns, and high-quality benchmarks.

7. Comparative Summary of Reviewed Sources

Table 1 summarises the focus, techniques, and single most relevant finding from each of the four reviewed sources.

Table 1. Comparative summary of techniques and findings across the four reviewed sources.

Source	Primary Focus	Main Techniques Discussed	Key Finding Relevant to This Paper
[1]	Web API vulnerabilities and developer challenges (PRISMA-guided systematic review)	Vulnerability classification, practice and usability analysis	Web APIs are important attack surfaces; technical and human factors both sustain vulnerabilities.
[2]	Security-API misuse detection (69 studies)	Heuristic-based and ML-based detection	Misuse of security APIs is common; evaluation methods are inconsistent and need standardization.
[3]	API protection across the SDLC	Gateways, APM-based visibility, SAST/DAST, ML anomaly detection	Security should be continuous; gateways and monitoring are layers, not replacements for in-service checks.
[4]	Future trends in API security	AI/ML, zero trust, adaptive authentication, continuous monitoring	Emerging tools are promising but must complement fundamental controls and be validated carefully.

8. Synthesis: Literature Gaps

Three consistent gaps emerge. First, authorization and data protection are central yet repeatedly under-implemented: authentication does not prevent a valid user from accessing another user's records if object-level checks are missing. Second, there is a gap between proposed techniques and evidence of their practical effectiveness; AI/ML and zero-trust approaches are promising, but results should be interpreted cautiously when datasets, deployment contexts, or measurement methods differ [1,2,4]. Third, no single tool is sufficient: static analysis, runtime testing, gateway controls, monitoring, and developer education cover different parts of the problem and are strongest when used together [1–4].

III. Proposed Framework

Building on the gaps and precedents identified above, this paper proposes an integrated two-module framework for continuous API security. The framework is deliberately layered, with no single control trusted on its own, and keeps human oversight over high-impact decisions, in response to the false-alarm and trust concerns raised in the literature.

1. System Overview

The system consists of two parallel modules (Module A for design-time controls and Module B for runtime protection and monitoring) that both report into a shared Governance and Integration Layer. This allows each module to be adopted incrementally: Module A relies on established practices such as static analysis and code review, while the AI/ML anomaly-detection component of Module B is treated as an additional, exploratory layer.

Table 2. Side-by-side design of Module A (design-time) and Module B (runtime).

Stage	Module A: Design-Time Controls	Module B: Runtime Protection and Monitoring
Input	Requirements, API specifications, source code, dependencies	Live API traffic, authentication events, logs, performance telemetry
Preprocessing	Threat modelling, schema definition, dependency inventory	Normalisation of logs, removal of secrets and personal data, baselining of normal traffic
Core controls	SAST, dependency scanning, code review, schema validation, allow-listed fields	Gateway authentication and rate limiting, in-service authorization, APM and observability, ML anomaly detection
Output	Findings report, hardened API design, reusable security components	Blocked or throttled requests, security events, anomaly scores with supporting evidence
Action on detection	Block merge or release until the finding is resolved	Alert security team; throttle or block; open incident-response process

2. Module A: Design-Time Controls

Module A applies security from requirements through development. Threat modelling and secure design define authentication, authorization, and data-exposure rules before implementation; static analysis and dependency scanning run in pull requests and CI/CD pipelines; and schema validation and allow-listed request fields address mass assignment and injection risks [2,3]. Developer-facing documentation, safe examples, and reusable authorization components address the human factors identified in the literature [1].

3. Module B: Runtime Protection and Monitoring

Module B protects deployed APIs. An API gateway centralizes authentication enforcement, request filtering, rate limiting, and traffic logging, while object-level and function-level authorization remains enforced inside each service [3]. Monitoring and APM tooling provide visibility, and an optional ML-based anomaly detector can flag unusual request volumes or credential abuse [3,4]. Zero-trust principles and adaptive authentication evaluate each request using identity and context rather than network location [4].

4. Governance and Integration Layer

Both modules report to a shared layer that maintains the API inventory, assigns owners, defines security policy, and presents a dashboard of findings and alerts. Rather than presenting a bare verdict, alerts should show a confidence level alongside the

International Journal of Statistics and Mathematical Research Trends
E-ISSN: 3108-0871, Volume 2 Issue 5
Sep-Oct 2026, PP 01-9

supporting evidence (for example, the request pattern or the code location), reflecting the need for interpretable output. Every endpoint has an owner, an intended audience, a documented policy, and a deprecation plan [3].

IV. METHODOLOGY

1. Data and Tooling Requirements

Module A requires access to API specifications, source repositories, and dependency manifests, along with a set of test cases covering the vulnerability classes in Section 2.2. Module B requires representative API traffic and logs, including authentication failures, authorization denials, and error events; ML-based detection additionally needs labelled or well-baselined traffic, which may need to be collected in a staging environment. Evaluations should record the benchmark, data source, API types, and vulnerability classes covered [2].

2. Preprocessing Pipeline

Design-time artefacts are normalised into a common inventory of endpoints, versions, owners, and data classifications. Runtime logs are cleaned before analysis: secrets, access tokens, passwords, and unnecessary personal data are removed or masked, events are timestamped and correlated by user, service, and endpoint, and a baseline of normal behaviour is established before anomaly detection is enabled [3].

3. Architecture

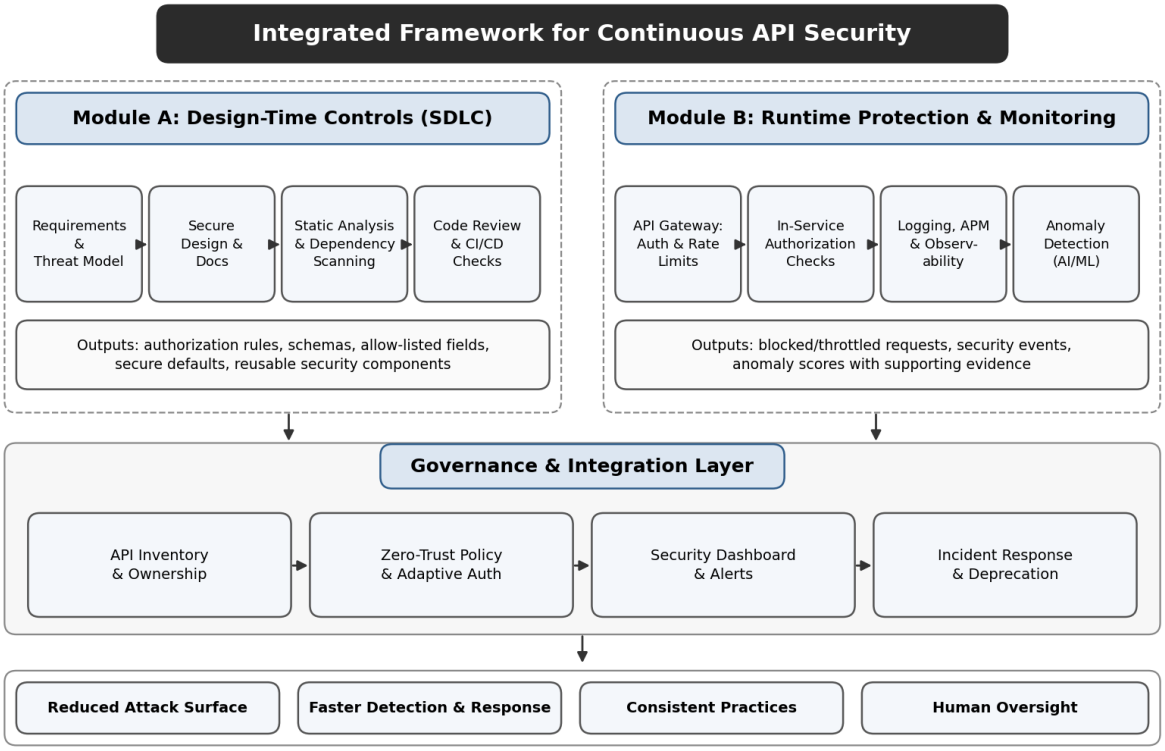


Figure 1. Proposed integrated framework for continuous API security, based on the reviewed literature.

4. Tool Selection and Training

Static and dynamic tools are selected to cover complementary layers: SAST for early code-level issues, DAST, fuzzing, and penetration testing for runtime and authorization-logic issues, and dependency scanning for third-party components [2,3]. Heuristic rules are recommended as an interpretable baseline. If an ML-based anomaly detector is added, it should be trained on representative traffic with a held-out test set and cross-validation to mitigate overfitting, and retrained when traffic patterns drift [2,4].

5. Evaluation Metrics

Detection components would be evaluated using precision, recall, F1 score, false-positive rate, detection time, throughput, and resource overhead, selected according to the purpose of each tool [2]. Gateway controls would additionally be evaluated on whether they restrict abuse without blocking legitimate traffic, and ML anomaly detection on false-alarm rate, since a high false-alarm rate undermines operational trust.

6. Validation Plan

- Retrospective validation: test the controls against existing labelled test cases and historical logs for the vulnerability classes in Section 2.2.
- Staging pilot: deploy both modules on a limited set of APIs in a staging environment, comparing flagged events against manually confirmed findings.
- Broader evaluation: extend to multiple services, versions, and teams to test consistency of policies across a microservice environment.
- Governance review: confirm that each endpoint has an owner, documented policy, and deprecation plan, and that logging avoids storing secrets.

V. Expected Results and Discussion

1. Expected Outcomes

As this is a literature-based design, no numerical performance is claimed. Qualitatively, the layered design is expected to reduce the attack surface through earlier detection of missing checks, improve visibility of unknown or deprecated endpoints, and shorten detection time for abusive traffic. The ML anomaly detector would more realistically be expected to produce a non-trivial false-alarm rate in an initial pilot, improving as baselines and training data mature [3,4].

2. Discussion of Feasibility

Each module reuses practices described in the reviewed sources: SAST and DAST, gateways, monitoring, zero-trust policies, and inventory management. The integration of the two modules under a single governance layer is, however, a proposal rather than an established practice, and operational costs such as latency, policy consistency, and maintenance burden would need to be assessed before adoption [4].

3. Ethical and Operational Considerations

- Evaluation quality: tools should be compared on standardized benchmarks and transparent metrics rather than accuracy alone [1,2].
- Privacy: logs and monitoring data may contain personal data, tokens, or secrets and should be minimised, masked, and protected.
- Interpretability: alerts should show supporting evidence so that analysts can judge whether an anomaly is an actual attack.
- Human oversight: AI/ML detection is an additional layer, with human review for high-impact actions such as blocking users or services.
- Developer experience: controls should be integrated into normal workflows with clear documentation and safe examples [1].

4. Limitations of This Study

This paper is a literature-based design proposal rather than an empirical study; no tools were built or tested. It synthesizes four sources, so conclusions about AI/ML and zero-trust approaches depend on the evaluation conditions reported by the original authors, and the framework would need its own dedicated implementation and validation before any effectiveness claims could be made.

VI. CONCLUSION AND FUTURE WORK

This paper reviewed four recent sources on API security and used them to construct a proposed framework combining design-time controls, runtime protection, and a shared governance layer. The review confirms that authorization and data protection remain central, that API security depends on developers and development processes as much as on tools, and that no single tool is sufficient. Strong authentication and server-side authorization, input and output validation, inventory and monitoring, and layered testing remain the foundation, while AI/ML, zero trust, and adaptive authentication are valuable additions when properly validated. Future work should prioritise: (i) implementing a pilot of the framework in a staging environment; (ii) producing reproducible evaluations with standardized benchmarks; (iii) developer-centred security design, including safer documentation and secure-by-default libraries; and (iv) evidence-based assessment of the cost and real-world effectiveness of emerging tools.

references

1. Almjnoony et al. Systematic review of web API vulnerabilities, misuse patterns, and developer challenges (PRISMA-guided). [Insert full citation: journal, year, volume, pages].
2. Mousavi et al. Systematic review of security-API misuse detection (69 peer-reviewed studies). [Insert full citation: journal, year, volume, pages].
3. [Insert author(s)]. Overview of API threats and protection across the software development life cycle, including API gateways and APM-based visibility. [Insert full citation].
4. [Insert author(s)]. Emerging technologies and future directions in API security (AI/ML, zero trust, adaptive authentication, continuous monitoring). [Insert full citation].