

Development of Rest Api Using Owasp Top 10 Rules

Omkar G K¹, Nitesh M², Sandeep Reddy³, Dr. Prabakar M⁴

^{1,2,3}Student, Department of Computer Science & Engineering Dayananda Sagar University Bengaluru, India

⁴Professor, Department of Computer Science & Engineering Dayananda Sagar University Bengaluru, India

Abstract

Application Programming Interfaces (APIs) play an important role in digital transformation but still stand among the most vulnerable entities in modern IT architectures. An extensive literature review, including such titles as Secure APIs, API Security in Action, Microservices Security in Action, and OWASP API Security Top 10, highlights common vulnerabilities associated with APIs, which include, among others, broken authentication, over-exposure of data, and misconfiguration of endpoints. Complementing these works are such sources as The Web Application Hacker's Handbook and REST API Design Rulebook, which emphasize the importance of secure API design, while Cloud Native Security and Practical API Security provide similar recommendations for cloud-native implementations. Combining the ideas from above, this research paper presents a coherent framework for building secure APIs based on the OWASP Top 10 guidelines for the whole lifecycle of application development. Using a prototype, the author demonstrates how early control measures can prevent potential exploits, ensure resilience in case of an automated attack, and increase regulatory compliance.

Keywords: OWASP API Security Top 10, broken object level authorization (bola), broken authentication, excessive data exposure, rate limiting, injection attacks, server-side request forgery (ssrf), security misconfiguration, jwt (json web token), oauth 2.0, openid connect, api gateway, zero trust architecture, penetration testing, microservices security, rest api, graphql security, api lifecycle, threat modeling, input validation, least privilege principle, transport layer security (tls), api versioning, role-based access control (rbac), api throttling, sensitive data exposure, mass assignment, improper asset management, function level authorization, secure sdlc

INTRODUCTION

Application Programming Interfaces (APIs) play a key role in modern digital ecosystems as tools that enable interactions between applications, services, and users. They are widely used in various domains including banking, healthcare, e-commerce, governance, social networks, etc. Indeed, as the number of APIs increases, they account for over 80% of all internet requests nowadays. However, the growing number of APIs leads to their becoming one of the primary targets of malicious attacks on cybersecurity. Recent data breaches and cyberattacks, including the Facebook and Cambridge Analytica scandal, the Twitter API security issue that took place in 2022, and the T-Mobile data breach serve as illustrative examples showing the potential negative outcomes associated with API exploitation.

In order to address the challenges associated with cyber threats to APIs, several projects have been implemented and continue being actively developed. For instance, the Open Web Application Security Project (OWASP) published a document outlining 10 primary risks associated with API security. They include broken object level authorization, broken authentication, excessive data exposure, injections, server-side request forgery, security misconfigurations, and improper asset management [1]. Although these issues are clearly stated, there still exist many cases when APIs are deployed without appropriate safety mechanisms in place due to the lack of proper knowledge among practitioners and organizations. Thus, the present project is designed to provide an insight into the development of safe APIs and present the Secure API Development Lifecycle Model as an effective solution.

The study will employ both scholarly and practical perspectives in its analysis. Specifically, the research will be focused on identifying the major principles of creating secure APIs. The emphasis will be placed not only on theoretical insights related to secure API development but also on practical issues, including architectural patterns, example code, configuration suggestions, and tests. Furthermore, the study will pay special attention to the latest trends in APIs' security such as microservices, GraphQL, API gateways, zero trust approach, etc. Furthermore, it highlights future research directions in threat modeling, automated detection of business logic flaws, and continuous security validation, aiming to advance the state of API security in an era of rapid technological change.

LITERATURE REVIEW

The scholarly and practitioner discussion of API security has grown tremendously in the last 10 years, as the dependence on APIs as the glue of the new software systems has increased. Foundational literature, like *The Web Application Hacker's Handbook* by Dafydd Stuttard and Marcus Pinto (2011) [1], introduced the core web application vulnerabilities of parameters tampering, session hijacking and injection attacks, which subsequently became the API specific threat categories listed in the OWASP API Security Top 10. In the same way, *REST API Design Rulebook* (2011) by Mark Masse [2] defined a set of principles that must be followed when constructing a REST API, focusing on resource names, semantics of the HTTP protocol and status codes, and each of them has security connotations. *Designing Web APIs* (2018), by Brenda Jin, Saurabh Sahni, and Amir Shevat [3], took this view a step further by associating usability and predictability with more positive security results.

The history of API vulnerabilities is closely related to changes in architectural paradigms. XML injection, WSDL enumeration, and poor WS Security were the bane of early SOAP implementations. As REST became popular, new security risks such as insecure direct object references (since formalized as Broken Object Level Authorization, BOLA) and JSON injection were introduced. Microservices adoption, which Prabath Siriwardena and Nuwan Dias examine in *Microservices Security in Action* (2020) [4], brought about such issues as distributed identity management, service to service trust boundaries, and polyglot persistence. More recently, GraphQL APIs have also introduced their own distinct attack vectors of insider abuse (introspection misuse), query depth abuse (batching), and abuse of query depth (batching) as described by Alira Voxel in *API Security for Beginners* (2021) [5]. Cloud native environments cumulated additional complexity with the risks of container orchestration, misconfigurations of service meshes, and insecure cloud provider APIs, as detailed by Chris Hughes and Brent Laster in *Cloud Native Security* (2022) [6].

Some of the main reference works overlap with some common themes. *API Security in Action* by Neil Madden (2020) [7] provides a serious approach to authentication and authorization, specifically OAuth

2.0 flows, token validation, and cryptographic practices of JWTs. Secure APIs by Jose Haro Peralta (2025) has a lifecycle oriented approach, which argues in favor of security by design, defense in depth and shift left practices, which are consistent with the model of Secure API Development Lifecycle (SADLC) proposed in this study. Practical API Security (2023) by Mohammed Aboullaite [8] adds an experimental approach to these theoretical frameworks with techniques such as fuzzing, manipulating parameters, and automated scanning.

Despite the large number of contributions made thus far, some significant areas of neglect have emerged. The first issue is that many papers tend to treat vulnerabilities individually and fail to provide a cohesive framework that can implement the OWASP taxonomy in practice. The second problem is that, although much attention has been paid to authentication and authorization vulnerabilities, business logic issues and automated detection tend to be overlooked. Lastly, the process of implementing security mechanisms throughout the lifecycle, from design to deployment, has received limited practical research attention.

METHODOLOGY

The security of APIs is a crucial component of modern digital infrastructure. APIs promote interoperability and integration of applications, services, and users in different sectors, including financial operations, medical care provision, e-commerce websites, and government entities. Digital transformation through the use of APIs has been swift and successful; however, it has created new opportunities for cybersecurity threats, which hackers might abuse. Recent studies show that more than 80% of all internet traffic is powered by APIs, demonstrating their ubiquity and potential vulnerabilities. The Facebook-Calibrating Analytica scandal, Twitter API flaw in 2022, and a recent T-Mobile incident are some of the examples that demonstrate that vulnerabilities in API systems can lead to severe financial damage, reputational issues, and data leaks that threaten personal privacy of millions of people. There are several measures that developers can take to avoid major mistakes in API development that could compromise its performance and safety. One of the best solutions in this regard is the creation of the OWASP API Security Top 10

[9] framework that identifies the top ten API vulnerabilities. Developed in 2019, with updates in 2023, the guidelines focus on frequent risks, including object level authorization flaws, excessive data exposure, and poor configuration. Despite having such an effective guideline, vulnerabilities still happen since there is usually a gap between theoretical knowledge and actual implementation. Some companies tend to implement APIs with weak authentication and authorization controls, poor input validation, lack of API lifecycle management.

The main goal of the current study is the creation of the Secure API Development Lifecycle (SADLC) framework based on principles outlined by OWASP. This methodology is a result of the comprehensive analysis of various sources on API security, including Secure APIs (Peralta), API Security in Action [7], Microservices Security in Action [4], Cloud Native Security [6], REST API Design Rulebook [2], and The Web Application Hacker's Handbook [1]. Scholarly publications and practical insights were carefully selected and integrated to create an efficient methodology to implement a secure API.

A. OWASP API Security Top 10 (2025)

Here is the detailed breakdown of each rule:

- **Broken Object Level Authorization (BOLA)**
- The Broken Object Level Authorization (BOLA) refers to the initial authorization level inside the

operating system (OS).

- **Risk:** Hackers can access object IDs by manipulating them to reach unauthorized data.
- **Challenge:** Defects in business logic are not easily detected and are often difficult to track.
- **Mitigation:** Implement server-side authorization checks, authenticate ownership, and use least-privilege policies.
- **Broken Authentication**
 - This may occur due to weak or incorrect authentication, leading to impersonation risks.
 - **Vulnerability:** Poor handling of tokens and absent Multi-Factor Authentication (MFA).
 - **Mitigation:** Use OAuth 2.0/OpenID Connect, secure storage, expiry policies, and multi-factor authentication.
- **Broken Object Property Level Authorization**
 - **Risk:** Unauthorized access to sensitive fields within objects.
 - **Test:** Developers accidentally leave internal fields uncovered.
 - **Access Control:** Utilize fine-grained accessibility, field-level authorization, and schema validation checks.
- **Unrestricted Resource Consumption**
 - **Risk:** Rate limit-free APIs may be overloaded, leading to denial-of-service (DoS).
 - **Challenge:** Performance optimization is often prioritized over security.
 - **Mitigation:** Adaptive rate limiting, quotas, caching, and anomaly detection.
- **Broken Function Level Authorization**
 - **Risk:** Invocation of privileged functions; attackers invoke functions without proper checks.
 - **Challenge:** Complex microservices make consistent authorization difficult.
 - **Mitigation:** Administrative endpoint segregation, centralized authorization logic, and forced privilege limits.
- **Free Access to Sensitive Business Flows**
 - **Risk:** Attackers use APIs to misuse critical workflows (e.g., payments, reservations).
 - **Challenge:** It is difficult to find bad business logic with automated tools.
 - **Mitigation:** Implement workflow checking, anomaly detection, and impose restrictions on sensitive operations.
- **Server-Side Request Forgery (SSRF)**
 - **Attack:** APIs that retrieve resources located elsewhere can be fooled into accessing internal systems.
 - **Hurdle:** This is especially hazardous in cloud environments.
 - **Outbound Security:** Block outbound traffic, reverse path validation, whitelisting, and URL validation.
- **Security Misconfiguration**
 - **Threat:** Default configurations, open debug endpoints, and weak TLS expose APIs.
 - **Suggestion:** Rapid deployment cycles often neglect to consider secure configurations.
 - **Reduction:** Automate configuration audits, implement safe defaults, and continuously re-scan for configurations.
- **Improper Inventory Management**
 - **Bogus APIs:** Shadow APIs, obsolete versions, and undocumented endpoints increase the attack surface.

- **Challenge:** Major companies cannot easily monitor all active APIs.
- **Mitigation:** Maintain dynamic inventories of assets, enforce version control, and prune unused endpoints.
- **Hazardous API Consumption**
 - **Risk:** Relying on insecure third-party APIs introduces vulnerabilities.
 - **Challenge:** External APIs might not follow secure practices.
 - **Mitigation:** Check third-party responses, enforce schema checks, and apply zero-trust principles.

B. Framework/Meta-model: The Secure API Development Lifecycle (SADLC)

The study utilizes the Secure API Development Lifecycle (SADLC) approach, embedding “security shift-left practices at each stage of the development lifecycle.” This framework is based on DevSecOps approaches, thus implying automated and verifiable security control implementations across all stages of the software development life cycle. The key concept within SADLC is Zero Trust, which implies that the network is never trustworthy, therefore requiring authentication and authorization for each and every request independently of its origin.

The SADLC framework utilizes Contract-Driven Development approaches based on OpenAPI specifications acting as the “single source of truth.” API Development Methodology is a structured process starting with a requirements phase; where API impact and need is defined. Developers will define user stories and terms are classified on how sensitive the information is. Threat modelling is completed at this stage. Typically done in a team like manner, to find potential stride (spoof, tamper, repudiation, data disclosure, deny service, elevate privilege) threats early on.

The design phase relies on resource modelling and architectural consistency according to the REST API design rulebook. URIs (Universal Resource Identifiers) are developed based on nouns, and then an appropriate hierarchy is created so granular path authorization becomes possible. Identifier integrity is achieved by having opaque or non-sequential identifiers which will assist in mitigating enumeration. Interaction policy governs URI path to be in lower case letters and prohibits a trailing slash and enforce compliance to hypertext transfer protocol (HTTP) methods without tunnelling.

The implementation phase builds in security as part of the business logic. We authenticate with JSON web token (JWT), or OAuth 2.0, and will use JWT's as if they were all fragile keys. Secure JWTs are created via a strong signature method like RS256. We use role-based access control (RBAC) and scopes to enforce authorization. Password hashing will occur with SCRYPT and we will use TLS 1.2+ for secure transfer of data.

C. Development Stages

Stage One: Requirements Phase: The requirements phase consists of defining the “problem” and what affect the API will have. User stories get created identifying key users involved and what level of sensitivity the data has. Threats are modeled through threat modeling, which is usually regarded as a “team sport,” early on to identify potential threats to APIs through STRIDE: Spoofing, Tampering, Repudiation, Information Disclosure, Denial of Service, and Elevation of Privilege.

Stage Two: Design Phase: The design phase is focused around resource modeling and architectural consistency and is done by defining URIs (Uniform Resource Identifier) following the REST API Design Rulebook. URIs should use nouns and hierarchical patterns for creating (and narrowing down) access in a manner that is reflective of path-based authorization.

Integrity of Identifiers: Use opaque or non-sequential IDs to avoid enumeration.

Rules of interaction: Lowercase all URI paths, no trailing slashes, and use each method of interaction

(GET, POST, etc.) according to their intended purpose (i.e., no “tunneling”).

D. Framework/Meta-model: The Secure API Development Lifecycle (SADLC)

The study utilizes the Secure API Development Lifecycle (SADLC) approach, embedding “security shift-left practices at each stage of the development lifecycle.” This framework is based on DevSecOps approaches, thus implying automated and verifiable security control implementations across all stages of the software development life cycle. The key concept within SADLC is Zero Trust, which implies that the network is never trustworthy, therefore requiring authentication and authorization for each and every request independently of its origin.

The SADLC framework utilizes Contract-Driven Development approaches based on OpenAPI specifications acting as the “single source of truth.” API Development Methodology is a structured process starting with a requirements phase; where API impact and need is defined. Developers will define user stories and terms are classified on how sensitive the information is. Threat modelling is completed at this stage. Typically done in a team like manner, to find potential stride (spoof, tamper, repudiation, data disclosure, deny service, elevate privilege) threats early on.

The design phase relies on resource modelling and architectural consistency according to the REST API design rulebook. URIs (Universal Resource Identifiers) are developed based on nouns, and then an appropriate hierarchy is created so granular path authorization becomes possible. Identifier integrity is achieved by having opaque or non-sequential identifiers which will assist in mitigating enumeration. Interaction policy governs URI path to be in lower case letters and prohibits a trailing slash and enforce compliance to hypertext transfer protocol (HTTP) URIs (Uniform Resource Identifier) following the REST API Design Rulebook. URIs should use nouns and hierarchical patterns for creating (and narrowing down) access in a manner that is reflective of path-based authorization.

Integrity of Identifiers: Use opaque or non-sequential IDs to avoid enumeration.

Rules of interaction: Lowercase all URI paths, no trailing slashes, and use each method of interaction (GET, POST, etc.) according to their intended purpose (i.e., no “tunneling”).

1. Stage Three: Implementation Phase: Security is “woven into the DNA” of the business logic of the API. Developers will build in the following core controls for the API:

- Authentication will be done via JSON Web Tokens (JWT) or OAuth 2.0. JWT tokens are “fragile keys” that require a secure signature (e.g. HMAC).
- Authorization will be done through Role-Based Access Control (RBAC) and scopes.
- Data Protection will be provided by bcrypt for password hashing and use of TLS 1.2+ for data in transit.

2. Stage Four: Testing Phase: The multi-layered approach for testing the interfaces follows the API Testing Pyramid:

- **Static Application Security Testing (SAST):** reviewing source code for hardcoded secrets or secrets stored improperly.
- **Dynamic Application Security Testing (DAST):** assessing methods through automation for improperly configured services and APIs, malformed URIs, improper resource definitions, etc.
- **Manual QA Testing:** examining the usability and functionality of services/products being developed through manual inspection of service/application code bases.

The Natter API (Social Network) prototype implemented several proactive security controls for the STRIDE threats, ensuring the system’s availability using the Guava rate limit library to quickly reject large amounts of requests. Authentication for the system was ensured through password hashing using

Script password hashing. HTTPS was used to protect all data in transit.

The Order Processing Microservice prototype demonstrated how to build a secure distributed architecture by decoupling security from the business logic using the Zuul API Gateway. OAuth 2.0 scopes were utilized for service-level authorization, and both Mutual TLS (mTLS) and JSON Web Tokens were used to secure “East-West” service communication and provide user context to all requests made from a single user request.

E. Technology/Methodologies

- **API Gateway:** Single point of control through usage of tools like Zuul and Istio to enable SSL termination, rate limiting, and routing.
- **Secret management:** Credential injection at runtime through HashiCorp Vault, which helps to prevent hard coded secrets.
- **Vulnerability scanning tools:** Usage of tools like Burp Suite (Intruder for fuzz testing), OWASP ZAP, and Postman for both automated and manual penetration testing.
- **Supply chain security:** Containers and third party library scanning for vulnerabilities through Trivy and Clair.

Observability Stack: Usage of Pino for JSON structured logging, along with Prometheus/Grafana.

F. Limitations and Scope

The research concentrates on RESTful architecture mostly. Although there are common rules for GraphQL and gRPC, these architectures have their own vulnerabilities such as query recursion in GraphQL, which cannot be analyzed in detail. The major drawback of the research is that the responsibility model is shared. Developers can ensure security at the API level, but the security of the cloud environment lies in the cloud provider’s area of responsibility. Another limitation is that polyglot microservices require a high level of security expertise.

G. Conclusion of Methodology

The integrated methodology transitions API security from a reactive “patchwork” approach to a proactive, lifecycle-integrated discipline. Through integration of design principles for architecture with OWASP Top 10 and utilization of automation tools in DevSecOps, companies can create robust API environments that will protect their sensitive information while remaining available in a hostile digital world.

RESULT

- **Bibliography/References:** The list of primary sources cited for research. These include API Security in Action by Neil Madden (2020) [7]; Microservices Security in Action by Siriwardena (2020) [4]; and REST API Design Rulebook by Mark Masse’ (2012) [2].
- **Future Research:** This is where the author discusses future directions for research. In our paper, we would highlight future research trends in AI-assisted API security and how AI gateways can be used in managing LLMs’ APIs.
- **Technical Appendix:** List of all the threats based on the STRIDE Threat Model for securing the APIs. The list includes Spoofing, Tampering, Repudiation, Information Disclosure, Denial of Service, and Elevation of Privilege threats.
- **Interpretation of the Results:** The analysis has examined the balance of performance versus security. It has been determined that the benefits associated with modularity outweigh the milliseconds of latency introduced through security measures.

- **Review of Previous Literature:** The review of previous literature indicated a transition from technical injection attack vectors to context based attack vectors including business logic violations and governance.
- **Significance of the Results:** The advantages of contract- oriented security and mixed token mechanisms (JWT and opaque) have been identified.
- **Restrictions:** The restrictions related to protocol diversity (the emphasis on REST), the shared responsibility model in cloud computing and the knowledge restrictions associated with polyglot programmers have been noted.

The report is now complete, having added the Conclusion portion which consists of three parts: Summary of Results, Recommendations, and Further Study Suggestion. This marks the end of the analysis of the primary literature and the OWASP API Top 10 (2023) model.

REFERENCES

1. D. Stuttard and M. Pinto, The Web Application Hacker's Handbook, 2nd ed. Wiley, 2011. M. Masse, REST API Design Rulebook. O'Reilly Media, 2011.
2. B. Jin, S. Sahni, and A. Shevat, Designing Web APIs. O'Reilly Media, 2018.
3. P. Siriwardena and N. Dias, Microservices Security in Action. Manning Publications, 2020.
4. A. Vexel, API Security for Beginners. Independently published, 2021.
5. C. Hughes and B. Laster, Cloud Native Security. O'Reilly Media, 2022.
6. N. Madden, API Security in Action. Manning Publications, 2020.
7. I. Aboullaite, Practical API Security. Packt Publishing, 2023.
8. OWASP Foundation, Owasp api security top 10, 2023.