IOSR Journal of Computer Engineering (IOSR-JCE)
e-ISSN: 2278-0661, p-ISSN: 2278-8727, Volume 27, Issue 5, Ser. 4 (Sept. – Oct. 2025), PP 52-60
www.iosrjournals.org

Securing RESTful APIs With Middleware-Based Threat Mitigation

Mohammed Ali Rizvi
Mtech Scholar
Department Of Computer Science And Engineering
Jai Narain College Of Technology (JNCT), Bhopal

Neha Jain
Assistant Professor
Department Of Computer Science And Engineering
Jai Narain College Of Technology (JNCT), Bhopal

Abstract
Over the past decade, RESTful APIs have become one of the most widely used methods for connecting applications and services. They are now an essential part of mobile apps, web platforms, cloud services, and even IoT devices. This popularity comes from their simplicity, flexibility, and ability to work across many different systems and programming languages. However, as RESTful APIs have become more common, they have also attracted more attention from attackers. API endpoints often handle sensitive data, manage authentication, and control key business functions, which makes them high-value targets for exploitation. Despite the clear need for strong protection, securing RESTful APIs remains a difficult task. One challenge is the lack of a single, universal standard for API security. Developers often rely on different methods, tools, and frameworks, which can lead to inconsistent protections. Another difficulty is finding the right balance between performance and security. Adding extra security layers can slow down systems, but removing them increases the risk of attacks. Real world incidents have shown that poor API security can lead to severe consequences, including data breaches, service disruptions, and reputational damage. This review paper looks closely at the current state of RESTful API security. It gathers and compares information from existing studies, industry reports, and practical implementations to give a clear picture of what is working, what is not, and where there is still room for improvement. The review covers key security techniques such as token-based authentication (JWT, OAuth2), encryption methods, input validation, and threat detection tools like Burp Suite and OWASP ZAP. It also examines common attack patterns, such as injection attacks, broken authentication, and data exposure, to understand how they exploit weaknesses in API design and configuration. By organizing and analysing these findings, the paper identifies patterns, strengths, and limitations in current approaches. More importantly, it suggests a structured framework that puts security at the centre of the API development process, instead of treating it as an afterthought. This proposed direction encourages developers to adopt a "security-first" mindset, use layered defence mechanisms, and follow consistent best practices regardless of the platform or technology used. In doing so, this work aims to provide both researchers and practitioners with a practical guide to improving RESTful API security. It offers not only a summary of the state of the art but also actionable recommendations for future designs, testing methods, and standardization efforts. By following these insights, the next generation of APIs can be built to withstand evolving threats while continuing to deliver the speed, reliability, and flexibility that modern applications require

---
Date of Submission: 04-10-2025 Date of Acceptance: 14-10-2025

I. Introduction

In today's digital world, the way applications communicate has changed dramatically. Over the past decade, RESTful APIs have evolved from being a convenient option for developers to becoming the backbone of modern web, mobile, and cloud-based applications. They make it possible for different systems, built on different technologies, to work together smoothly. From e-commerce platforms that connect with payment gateways, to mobile apps that retrieve live data from cloud servers, RESTful APIs enable fast, reliable, and flexible integration. Their simplicity, scalability, and wide adoption have made them the preferred choice for many organizations, from startups to large enterprises. However, with this growth comes a challenge that cannot be ignored—security. Every API endpoint that is exposed to the internet becomes a potential entry point for attackers. As businesses rely more on APIs to handle sensitive data, financial transactions, and user authentication, the attack surface has expanded

significantly. Securing APIs has become more complex, as developers must not only defend against traditional web-based threats but also account for API-specific risks. The growing number of cyber incidents targeting APIs shows that vulnerabilities are not just theoretical—they have real and costly consequences. The motivation for this review is rooted in this reality. RESTful APIs are no longer optional components; they are core parts of critical systems. Yet, their security practices often lag behind their rapid adoption. Many implementations lack consistency, and there is no universally enforced security standard that developers can follow. Furthermore, the balance between performance and security often leads to compromises, leaving systems open to exploitation. These issues highlight the urgent need for a consolidated understanding of current API security approaches, their limitations, and possible paths for improvement. The objective of this review is threefold. First, it aims to highlight common vulnerabilities that put RESTful APIs at risk, including weaknesses in authentication, improper data validation, and misconfigured endpoints. Second, it seeks to assess the current standards, tools, and best practices used in the industry, identifying where they succeed and where they fall short. Third, it examines notable real-world breaches to understand how attackers exploit API weaknesses, and based on these insights, suggests practical improvements that can help prevent similar incidents in the future. The scope of this paper is focused on RESTful API security in the context of modern applications, especially those deployed in web, mobile, and cloud environments. It does not attempt to provide an exhaustive tutorial on building APIs, but rather concentrates on the security landscape surrounding them. The paper is organized as follows: Section 3 provides a review of existing API security techniques and compares REST with other models such as SOAP. Section 4 discusses the limitations of current approaches, supported by examples of security incidents. Section 5 presents a proposed methodology for enhancing API security, including architectural guidelines and testing strategies. Finally, Section 7 concludes with a summary of findings and recommendations for future work. By bringing together research findings, industry practices, and case studies, this review aims to serve as a valuable resource for both academics and practitioners who are committed to building secure, resilient, and reliable RESTful APIs.

II. Background And Fundamentals

Before diving into the security aspects of RESTful APIs, it is important to understand what they are, how they work, and how they compare to other API models. REST, short for Representational State Transfer, is an architectural style used for designing networked applications. When an API follows the principles of REST, it is called a RESTful API. These APIs rely on standard HTTP methods—such as GET, POST, PUT, and DELETE—to perform operations, making them lightweight, easy to use, and well-suited for web and mobile applications. Unlike older API protocols, REST does not require complex messaging standards or heavy data formats. Instead, it often uses JSON or XML to exchange information, which keeps interactions simple and fast. This simplicity has played a huge role in its widespread adoption across industries. When comparing REST to SOAP (Simple Object Access Protocol), the differences go beyond just data formatting. SOAP is a protocol with strict rules and built-in features like error handling and formal contracts (WSDL). It also has well-defined security specifications such as WS Security. REST, on the other hand, is more flexible but does not have a single, universal security framework—it relies on the underlying transport layer (HTTPS) and additional measures added by the developer. While this flexibility is an advantage for scalability and ease of integration, it can also mean that REST security is less consistent if not carefully implemented. A key principle of REST is statelessness. This means that each client request to the server must contain all the information needed to understand and process it. The server does not store any session data between requests. From a security perspective, statelessness has both benefits and drawbacks. On one hand, it reduces the risk of certain attacks that target stored session data and makes systems easier to scale. On the other hand, it requires careful handling of authentication and authorization, since user identity must be verified with every request. RESTful APIs are built from several essential components. Endpoints are the specific URLs through which clients interact with resources. HTTP methods define the type of action to be performed, such as retrieving data (GET), creating new records (POST), updating existing data (PUT/PATCH), or removing data (DELETE). Headers carry important metadata, including content type, authorization tokens, and caching instructions. Authentication mechanisms— such as API keys, Basic Authentication, OAuth2, or JWT—ensure that only authorized users and applications can access or modify resources. Each of these components plays a role in securing the API, and weaknesses in any one of them can expose the system to threats. Understanding these fundamentals is essential before exploring the specific vulnerabilities and security practices that will be discussed in later sections. Without a solid grasp of how RESTful APIs are structured and how they differ from other models, it is difficult to design strong and effective security measures.

III. Existing Security Standards And Protocols

Securing RESTful APIs is not a one-size-fits-all task. Developers have a variety of standards and protocols to choose from, each designed to handle different aspects of protection—from verifying user identities to ensuring that transmitted data remains confidential. While these mechanisms provide powerful tools, their implementation often varies widely between projects, leading to inconsistency and gaps in protection.

Authentication Mechanisms

Authentication is the first line of defence in any API security strategy, as it determines who is trying to access the system. The simplest method is Basic Authentication, where the client sends a username and password encoded in Base64 with every request. While easy to implement, Basic Auth is considered weak because credentials are repeatedly sent over the network and can be intercepted if not properly encrypted with HTTPS. API Keys offer another approach, assigning each client a unique identifier that must be included in requests. They are lightweight and easy to manage but do not inherently verify the identity of the requesting user—they simply confirm that the request came from an authorized application. More advanced methods include OAuth 2.0, a widely adopted authorization framework that allows applications to access resources on behalf of a user without sharing actual credentials. Building on top of OAuth, OpenID Connect adds a layer for verifying user identity, making it suitable for single sign-on scenarios. A popular companion to these frameworks is JSON Web Token (JWT), which packages authentication claims into a digitally signed token. JWTs are compact, easy to transmit, and can be verified without storing session data on the server—ideal for stateless RESTful APIs. However, they also have drawbacks: if a token is compromised, it remains valid until it expires, and improper signing or storage can lead to vulnerabilities.

Transport Security

Even the strongest authentication is meaningless if data can be intercepted in transit. This is where HTTPS and TLS (Transport Layer Security) come in. HTTPS encrypts communication between the client and server, preventing attackers from reading or altering the data. Proper TLS configuration is crucial—using outdated versions or weak ciphers can still leave a system exposed. Security-conscious implementations enforce the latest TLS standards, disable obsolete protocols, and apply certificate pinning to defend against man-in-the-middle attacks.

Access Control

Once a user or application is authenticated, the system must decide what they are allowed to do. Role-Based Access Control (RBAC) grants permissions based on predefined roles—such as admin, editor, or viewer—making it straightforward to manage large groups of users. However, RBAC can be too rigid for complex systems with varied permission needs. Attribute-Based Access Control (ABAC) offers more flexibility by making decisions based on attributes such as user department, resource type, or time of access. While powerful, ABAC can be harder to implement and maintain.

Limitations and Fragmentation

Despite the variety of available standards and protocols, there is no single, universally accepted method for securing RESTful APIs. Implementations often differ from one organization to another, even when using the same protocol. For example, two APIs might both use OAuth 2.0 but configure token lifetimes, scopes, and refresh mechanisms in entirely different ways. This lack of uniformity can create confusion, increase integration costs, and, in some cases, introduce security flaws.

IV. Common Vulnerabilities in RESTful APIs

While RESTful APIs provide flexibility and scalability, they can also introduce significant security risks if not designed and implemented correctly. Over the years, security researchers and industry bodies like OWASP have documented recurring weaknesses that attackers frequently exploit. Understanding these vulnerabilities is essential for building safer APIs and avoiding costly breaches.

Insecure Endpoint and Misconfigured APIs

One of the most common issues in RESTful APIs comes from insecure endpoints—parts of the API that are left exposed without proper security controls. This can happen when developers forget to enforce authentication on certain routes or leave debugging tools enabled in production. Misconfigurations, such as leaving unused endpoints active or failing to restrict HTTP methods, can give attackers unnecessary entry points into the system. Even seemingly harmless endpoints can reveal sensitive system information when not handled carefully.

Broken Authentication and Session Management

APIs often rely on authentication tokens or sessions to identify users. When these mechanisms are poorly implemented—such as using predictable session IDs, weak passwords, or failing to expire tokens—they open the door for attackers to impersonate legitimate users. In RESTful APIs, where statelessness means that credentials or tokens are sent with each request, failing to protect these credentials can have severe consequences. An attacker who steals a valid token can access the API until that token is revoked or expires.

Lack of Input Validation and Injection Attacks

RESTful APIs frequently accept user input through parameters, request bodies, or headers. If this input is not properly validated or sanitized, attackers can inject harmful code or commands. Common injection attacks include SQL injection, command injection, and XML external entity (XXE) attacks. These can allow attackers to bypass security controls, steal sensitive data, or even take control of the underlying system.

Improper Authorization (IDOR–Insecure Direct Object Reference)

A particularly dangerous vulnerability is Insecure Direct Object Reference (IDOR), which occurs when an API exposes internal resource identifiers without proper access checks. For example, if a user can change an account ID in a request URL to view another user's data, it means the API is not enforcing authorization correctly. IDOR vulnerabilities can lead to large scale data leaks with minimal technical effort from attackers.

Data Exposure (Sensitive Detain URI or Headers)

Another common issue is unintentional data exposure. Sensitive information—such as API keys, tokens, or personally identifiable data—should never be included in URIs or unsecured headers, as they may be logged in server files, browser histories, or analytics tools. Once logged, these details can be retrieved by anyone with access to those logs, significantly increasing the risk of compromise.

Case Studies from OWASP API Top10

The OWASP API Security Top 10 highlights many of these issues with real-world examples. For instance, several breaches have occurred due to overly permissive endpoints that allowed attackers to retrieve massive datasets without proper authentication. In other cases, token theft and replay attacks have led to unauthorized access to financial or healthcare records. These case studies underline a key point: vulnerabilities in APIs are not just theoretical—they are actively exploited, often with severe business and reputational consequences.

To support the discussion of common REST API vulnerabilities, I carried out practical security testing using the OWASP ZAP vulnerability scanner. My aim was to check whether the problems described in research and industry reports could also be found in real-world APIs.

For this, I tested a mix of APIs: some intentionally insecure applications such as Hackazon and Juice Shop, and some production-ready APIs that are publicly available through RapidAPI. This gave me a balanced view of insecure test environments on one hand, and real-world APIs on the other.

The scans revealed several issues. I did not find any high-risk vulnerabilities in the production APIs, but I did find medium- and low-risk issues across different endpoints. This matches what many reports already suggest: severe flaws are less common in well-maintained APIs, but mistakes like misconfigurations, missing security headers, or small leaks of information are still very common because many developers build APIs without deep security knowledge.

To make the results clearer, I summarized them in graphs:

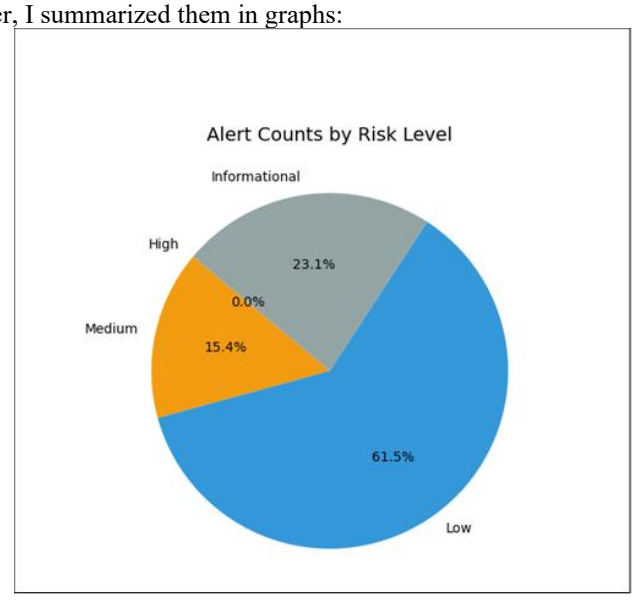

FIG 1

(Pie Chart: Alerts by Severity) shows that most of the problems were low risk (around 60%), while medium-risk issues made up about 15%. No high-risk issues were detected.

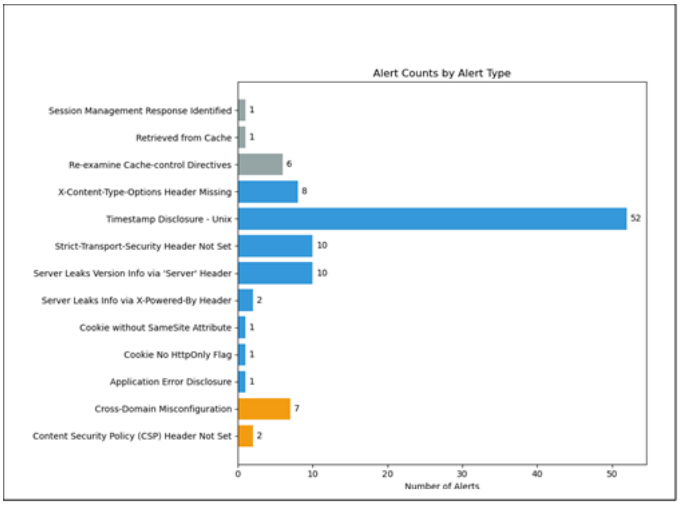

FIG 2

(Bar Chart: Alerts by Type) shows the specific kinds of issues that came up most often. Common examples included missing security headers, server response leaks, and timestamp disclosures. These are not always dangerous on their own, but they can help attackers gather information for larger attacks.

V. Real-World Breach Incidents (Case Studies)

While security guidelines and best practices for RESTful APIs are well documented, real-world incidents reveal that even major technology companies are not immune to serious vulnerabilities. Examining high-profile breaches helps uncover the recurring mistakes and oversights that allow attackers to succeed. These examples serve as practical lessons for developers, security teams, and organizations alike.

A. Facebook – Access Token Exposure via APIs

In 2018, Facebook suffered a large-scale security incident that affected approximately 50 million accounts. The breach stemmed from a vulnerability in the "View As" feature, which allowed users to see their profiles as others would. Due to a series of coding flaws, attackers were able to obtain access tokens—digital keys that allow continued access to a user's account without needing to log in again. These tokens were tied to Facebook's API authentication system, meaning the attackers could use them to access profiles, post content, and potentially connect to linked services. The incident highlighted how API-related vulnerabilities can turn into massive security failures when authentication tokens are not properly protected or invalidated.

B. Google+ User Data Exposure via API

Google+ experienced multiple API-related privacy issues before the service was ultimately shut down in 2019. One significant vulnerability exposed the personal data of hundreds of thousands of users, even if they had marked the information as private. A flaw in one of the Google+ APIs allowed external developers to access profile details, such as names, email addresses, occupations, and age, without user consent. While there was no confirmed evidence of abuse, the incident damaged trust and underscored the risk of over-permissive API endpoints. It also showed that even large companies with sophisticated infrastructures can misconfigure access controls in ways that go undetected for years.

C. T-Mobile and Instagram– API Flaws Leading to Data Leaks

In separate incidents, both T-Mobile and Instagram faced data breaches linked to API vulnerabilities. In T-Mobile's case, an API used for customer account management allowed unauthorized parties to retrieve sensitive account information by exploiting insufficient authentication checks. Instagram, on the other hand, had an API flaw that exposed user contact details, including phone numbers and email addresses, even for accounts set to private. In both scenarios, the core problem was the same: failure to properly validate and limit API responses based on the user's identity and permissions.

D. Patterns Observed Across Breaches

1) Token misuse and poor session invalidation leave APIs vulnerable to long-term unauthorized access.
2) Overly permissive endpoints expose more information than necessary, increasing the impact of a breach.
3) Weak or missing authorization checks allow attackers to bypass intended restrictions.
4) Insufficient logging and monitoring delay detection, giving attackers more time to exploit vulnerabilities.

These incidents serve as a reminder that security is not a one-time task—it must be continuously monitored, tested, and updated. Even industry leaders with advanced infrastructure can fall victim to API breaches when security assumptions go unchecked. By learning from these examples, organizations can better anticipate risks, tighten access controls, and design APIs that fail safely rather than catastrophically.

VI. Security Testing Tools And Practices

A secure RESTful API is not just the result of strong design principles—it also depends on thorough and continuous testing. Security testing ensures that vulnerabilities are identified and fixed before attackers can exploit them. In practice, API testing combines both manual exploration and automated scanning, each serving a unique purpose in the development and security lifecycle.

Manual Tools

Manual testing tools, such as Postman and Insomnia, are widely used by developers for building, sending, and inspecting API requests. While primarily designed for functional testing, these tools also allow security testers to manually craft requests, modify parameters, and observe responses for signs of vulnerabilities. For example, a tester might intentionally send malformed data or manipulate authorization tokens to see if the API responds in an unexpected way. Manual testing is particularly valuable for uncovering logic flaws—issues that automated scanners might miss—because it allows human intuition and domain knowledge to guide the process.

Automated Tools

Automated security scanners are essential for identifying common vulnerabilities quickly and at scale. OWASP ZAP and Burp Suite are two of the most widely used tools in this category. They can intercept API traffic, map endpoints, and run a series of vulnerability checks, such as injection attempts or misconfiguration detection. APIsec offers continuous API security testing, integrating into development pipelines to catch issues early. Tools like Postman Security and StackHawk extend automated testing capabilities by combining functional testing with security checks, making them well-suited for DevSecOps workflows where security is embedded into every development stage.

Static vs Dynamic Testing

Security testing can be broadly divided into Static Application Security Testing (SAST) and Dynamic Application Security Testing (DAST). SAST analyses the API's source code or configuration without executing it, helping detect vulnerabilities such as hardcoded credentials or insecure dependencies before the application is deployed. DAST, on the other hand, interacts with a running API to simulate real-world attacks and observe its responses. While static testing is useful for catching design flaws early, dynamic testing provides insights into how the API behaves under actual exploitation attempts. A well-rounded security program uses both approaches, ensuring that vulnerabilities are caught at multiple stages.

Limitations of Current Testing Strategies

Despite the availability of these tools and techniques, current testing strategies have limitations. Manual testing, while insightful, is time-consuming and heavily dependent on the tester's skill and experience. Automated tools can quickly scan large APIs but often produce false positives or miss context-specific vulnerabilities that require human judgment. Moreover, many security tests are run only once—just before deployment—rather than being integrated into a continuous security process. This creates gaps, especially as APIs evolve, endpoints change, and new threats emerge.

VII. Challenges In Current API Security Practices

Even with a variety of tools, protocols, and best practices available, securing RESTful APIs remains a challenging task. Many of the difficulties come not from a lack of awareness, but from the practical realities of building and maintaining APIs in fast-moving development environments. These challenges often combine technical limitations with human factors, creating persistent gaps that attackers can exploit.

Lack of Standardization Across Frameworks

One of the biggest hurdles in API security is the absence of a universal, enforced standard that applies across all frameworks and platforms. While guidelines like the OWASP API Security Top 10 provide general recommendations, the actual implementation of security measures varies widely. A feature considered secure in one framework might be absent or handled differently in another. This lack of uniformity leads to inconsistent protection, making it harder for teams to adopt a shared, proven approach to securing APIs.

Performance vs. Security Trade-offs

Developers often face a delicate balancing act between security and performance. Adding extra security checks—such as multiple authentication steps, encryption layers, or rate limiting—can slow down API responses. In high-traffic systems where milliseconds matter, these slowdowns can hurt user experience and scalability. As a result, some teams compromise on security for the sake of speed, inadvertently leaving the system more vulnerable to attacks.

Developer Misconfigurations and Human Error

Even the most secure frameworks can be undermined by simple mistakes. Common misconfigurations include leaving debugging endpoints exposed, failing to restrict access to sensitive APIs, or using default credentials in production. These errors are often the result of time pressure, lack of security training, or assumptions that "someone else will handle security." Human error remains one of the leading causes of API vulnerabilities, regardless of the technology stack.

Lack of Automated Analysis and Proactive Monitoring

While manual and automated testing tools exist, many organizations do not use them continuously. Security scans are often performed only during development or before a major release, leaving production APIs unmonitored for emerging threats. Without automated analysis and real-time monitoring, new vulnerabilities can go unnoticed for months, giving attackers ample opportunity to exploit them.

Inadequate Logging and Auditing in APIs

Effective logging and auditing are critical for detecting and investigating security incidents. Yet many APIs log only minimal information, making it difficult to trace an attacker's actions or determine the full extent of a breach. In some cases, logs are incomplete, inconsistent, or stored without proper security controls—risking both compliance violations and missed warning signs. Without strong auditing capabilities, even a detected breach may leave unanswered questions about how it happened and what was compromised.

In short, the challenges in current API security practices are not just technical—they are organizational, procedural, and human. Addressing them requires a shift toward security-first development, consistent standards, better training, and continuous monitoring. Without tackling these core issues, even the most advanced security tools will fall short of providing lasting protection.

VIII. Proposed Methodology

This section outlines the planned approach for building a robust and secure RESTful API system. The methodology integrates proven architectural practices with security-first principles, automated monitoring, and continuous testing. Each component is designed to work in harmony, ensuring that the final system is not only functional but resilient against modern security threats.

System Architecture Overview

The foundation of the proposed system lies in a modular architecture, where each layer—from the API gateway to the backend services—is clearly defined and independently manageable. The architecture will employ an API gateway as the first point of contact, handling authentication, request validation, and rate limiting before any request reaches the application logic. Backend services will follow the principle of least privilege, ensuring that components have only the minimum access necessary to perform their functions. Communication between services will be encrypted end-to-end using secure TLS configurations.

Security-First Design Principles

Security considerations will be embedded into the design process rather than added as an afterthought. This includes using secure defaults in configuration files, avoiding hard-coded secrets, validating inputs at both the gateway and service levels, and ensuring that error messages never leak sensitive information. API specifications (e.g., OpenAPI) will serve as a blueprint to enforce consistent request and response formats, which helps reduce attack surfaces and simplifies automated security testing.

Middleware-Based Threat Mitigation

Middleware components will act as security checkpoints throughout the request lifecycle. This includes modules for detecting suspicious patterns (e.g., repeated failed login attempts), sanitizing incoming data to prevent injection attacks, and automatically blocking or throttling requests from flagged IP addresses. The middleware will also integrate with third-party security threat intelligence feeds, allowing for real-time updates to blocklists and anomaly detection rules.

Role of Authentication and Authorization Layers

Authentication will be implemented using modern, secure protocols such as OAuth 2.0 or OpenID Connect, with token-based mechanisms like JWT for stateless session management. Authorization will follow either Role-Based Access Control (RBAC) or Attribute-Based Access Control (ABAC) depending on the complexity of the use case. This layered approach ensures that only verified and authorized users can access protected resources, and that access is tightly scoped to prevent privilege escalation.

Logging, Monitoring, and Alerting Systems

Comprehensive logging will be enabled across all layers of the system, capturing authentication attempts, failed requests, access to sensitive endpoints, and system errors. Logs will be aggregated in a centralized monitoring platform, enabling real-time analysis and correlation of events. Automated alerts will be configured to notify security teams of unusual activity, such as sudden traffic spikes or repeated failed logins, allowing for rapid response before potential threats escalate.

Security Testing Approach (Manual + Automated)

A dual approach to security testing will be adopted. Manual testing using tools like Postman will allow for targeted probing of specific endpoints and workflows, while automated scanners such as OWASP ZAP and Burp Suite will run periodically to detect common vulnerabilities. Static analysis tools will be applied to source code to catch insecure patterns early, and dynamic testing will be used to assess live API behaviour under simulated attack scenarios. This continuous testing cycle ensures that vulnerabilities are detected and addressed promptly throughout the development lifecycle.

IX. Future Research Directions

While recent advances in API security have introduced promising tools and techniques, there is still a considerable gap between current practices and the level of protection needed for increasingly complex, high-performance systems. Addressing these gaps will require both technical innovation and industry-wide cooperation. Future research can focus on several key areas that have the potential to reshape how RESTful APIs are secured.

Unified API Security Standardization

One of the most pressing needs in the API security space is the creation of a globally recognized, unified standard that defines how APIs should handle authentication, authorization, encryption, logging, and error handling. At present, security implementations vary greatly across frameworks, platforms, and organizations. A formalized, widely adopted standard—similar to how TLS became the default for web encryption—would bring consistency, reduce misconfigurations, and make security integration more predictable for developers. Research in this area could focus on defining core security requirements and ensuring compatibility across existing frameworks.

Lightweight Encryption and Authentication for High-Performance Environments

In industries such as finance, e-commerce, and real-time analytics, performance is critical. Heavy encryption and complex authentication flows, while secure, can introduce latency that impacts user experience. Future research could explore lightweight cryptographic algorithms and authentication protocols that offer strong protection with minimal performance cost. This could include adaptive encryption schemes that adjust their strength based on the sensitivity of the transaction or the trust level of the user's environment.

Integrating Security in the API Development Lifecycle (DevSecOps for APIs)

Many security issues arise because testing and protection measures are bolted on at the end of development, rather than built into the process from the start. Applying DevSecOps principles specifically to API development would ensure that security is considered at every stage—from design and coding to deployment and monitoring. Future work could explore automated security policy generation from API specifications, continuous vulnerability scanning in CI/CD pipelines, and developer-friendly security linting tools that run directly in code editors.

Automatic Vulnerability Mapping Using AI and OpenAPI

As APIs become more complex, manually tracking all possible vulnerabilities becomes unrealistic. AI-powered tools, combined with machine-readable specifications such as the OpenAPI Specification, could automatically map API structures, identify high-risk endpoints, and simulate potential attack paths. These systems could not only detect vulnerabilities but also recommend targeted fixes, helping developers respond quickly to threats. Over time, such AI-driven systems could learn from past incidents, improving their accuracy and predictive capabilities.

X. Conclusion

In this review paper, I explored the critical topic of RESTful API security, focusing on common vulnerabilities and the methods used to detect and prevent them. The goal was not just to list technical issues, but to understand the practical challenges developers face in designing and maintaining APIs that are both functional and secure.

Through this review, I highlighted recurring problems such as insecure endpoints, weak authentication, insufficient input validation, broken authorization, and unintentional data exposure. The increasing number of data breaches and unauthorized access incidents worldwide shows that these issues are not just theoretical—they pose real threats to businesses, organizations, and users every day. Observations from OWASP ZAP scans also reinforced that many APIs, even widely used ones, still have medium- and low-risk vulnerabilities that could be exploited if ignored.

To address these risks, I proposed a **structured methodology** for designing and securing RESTful APIs:

A. ***System Architecture Overview*** – embedding security into every layer of API design.
B. ***Security-First Design Principles*** – following best practices during development to reduce attack surfaces.
C. ***Middleware-Based Threat Mitigation*** – filtering and validating requests before they reach core services.
D. ***Authentication and Authorization Layers*** – enforcing strong, role-based access control to prevent unauthorized access.
E. ***Logging, Monitoring, and Alerting Systems*** – continuously tracking API activity to detect anomalies and potential attacks.
F. ***Security Testing Approach (Manual + Automated)*** – regularly evaluating APIs through penetration testing and automated scans to detect vulnerabilities early.

Given the growing prevalence of cyberattacks, this methodology is more relevant than ever. By adopting a **security-first mindset, layered defenses, and systematic testing**, developers can proactively reduce the risk of breaches, protect sensitive data, and build APIs that are resilient to evolving threats.

Acknowledgments

The author would like to express sincere gratitude to **Prof. Neha Jain**, CSE Department, JNCT Bhopal, for her valuable guidance, encouragement, and support throughout this work.

References

[1] Badhwar, R., 2021. Intro To API Security-Issues And Some Solutions!. In The CISO's Next Frontier: AI, Post-Quantum Cryptography And Advanced Security Paradigms (Pp. 239-244). Cham: Springer International Publishing.
[2] Pardal, M.L., Offensive Security Assessment Of A REST API For A Location Proof System.
[3] Ehsan, A., Abuhaliqa, M.A.M., Catal, C. And Mishra, D., 2022. Restful API Testing Methodologies: Rationale, Challenges, And Solution Directions. Applied Sciences, 12(9), P.4369.
[4] Mylläri, E., 2022. Introducing REST Based API Management And Its Relationship To Existing SOAP Based Systems.
[5] Bhateja, N., Sikka, S. And Malhotra, A., 2021. A Review Of Sql Injection Attack And Various Detection Approaches. Smart And Sustainable Intelligent Systems, Pp.481-489.
[6] Anugrah, I.G. And Fakhruddin, M.A.R.I., 2020. Development Authentication And Authorization Systems Of Multi Information Systems Based Rest Api And Auth Token. Innovation Research Journal, 1(2), Pp.127-132.
[7] OWASP Foundation, "OWASP Top 10: 2021 – The Ten Most Critical Web Application Security Risks," 2021. [Online]. Available: Https://Owasp.Org/Www-Project-Top-Ten/
[8] Sadqi, Y. And Maleh, Y., 2022. A Systematic Review And Taxonomy Of Web Applications Threats. Information Security Journal: A Global Perspective, 31(1), Pp.1-27.
[9] Dalimunthe, S., Reza, J. And Marzuki, A., 2022. The Model For Storing Tokens In Local Storage (Cookies) Using JSON Web Token (JWT) With HMAC (Hash-Based Message Authentication Code) In E-Learning Systems. Journal Of Applied Engineering And Technological Science, 3(2), Pp.149-155.
[10] Https://Developers.Google.Com/Identity/Protocols/Oauth2
[11] Wear, S., 2018. Burp Suite Cookbook: Practical Recipes To Help You Master Web Penetration Testing With Burp Suite. Packt Publishing Ltd.
[12] Kim, J., 2020. Burp Suite: Automating Web Vulnerability Scanning (Master's Thesis, Utica College).
[13] Maniraj, S.P., Ranganathan, C.S. And Sekar, S., 2024. SECURING WEB APPLICATIONS WITH OWASP ZAP FOR COMPREHENSIVE SECURITY TESTING. INTERNATIONAL JOURNAL OF ADVANCES IN SIGNAL AND IMAGE SCIENCES, 10(2), Pp.12-23.
[14] Soni, P., & Kumar, A. (2020). API Security Challenges In The Digital Finance Ecosystem. International Journal Of Cybersecurity And Digital Forensics, 2(2), 19-30.
[15] Mcdermott, M., & Harris, J. (2021). Defending Against Injection Attacks: A Comprehensive Review. Journal Of Cybersecurity, 18(4), 231-245.
[16] Coughlan, S., & Duggan, T. (2019). Denial-Of-Service Attacks In The Context Of Apis And Fintech. International Journal Of Information Security, 15(2), 114-126.
[17] Petrillo, F., Merle, P., Moha, N., & Guéhéneuc, Y.-G., 2019. Are REST Apis For Cloud Computing Well-Designed? An Exploratory Study. Université Du Québec À Montréal, Inria Lille-Nord Europe, École Polytechnique De Montréal, Federal University Of Rio Grande Do Sul.