WWW.IJRPR.COM

International Journal of Research Publication and Reviews

Journal homepage: www.ijrpr.com ISSN 2582-7421

A Survey of REST API Vulnerability Detection

Pratik Shah[1], Suraj Prajapati[2], Pavitra Maligouda[3], Payal Shinde[4], Prof Deepali Dhadwad*[5]

Indira College of Engineering and Management, SPPU, Pune, India
pratikvshah12@gmail.com, suraj40praj@gmail.com, shindepayal306@gmail.com, pavitramaligouda@gmail.com, deepali.dagale@indiraicem.ac.in

A B S T R A C T :

Application Programming Interfaces (APIs), and in particular RESTful APIs, are a key building block of modern software systems. These expose structured interfaces that applications, services and users use to exchange data and invoke operations. Hence, REST APIs have become an important security critical attack surface due to their wide usage. Broken Object Level Authorization, Broken Authentication, Broken Function Level Authorization, Mass Assignment, Injection, Excessive Data Exposure, Security Misconfiguration These vulnerabilities can expose sensitive information, allow unauthorized operations, or compromise backend services. The number of studies on automated techniques to identify vulnerabilities in REST APIs has been increased. Existing approaches include specification-driven testing, black-box fuzzing, analysis of access-control, vulnerability-oriented test generation, static and dependency-level analysis, machine-learning based detection, and explainable artificial intelligence. But these approaches differ substantially in the inputs they require, the detection strategies they use, the classes of vulnerabilities they support, the scope of testing they conduct, and their ability to provide interpretable evidence. In this survey, we review representative research on REST API vulnerability detection and categorize the existing studies into five main research directions, i.e., specification-driven and black-box testing, vulnerability-oriented and access-control aware detection, static and dependency-level analysis, machine learning-based detection, and explainable artificial intelligence. The surveyed approaches are comparatively analyzed in terms of their objectives, testing strategies, vulnerability coverage, evidence generation, level of automation and limitations.Based on this synthesis, the survey points out significant research gaps in multi-step API dependences, dynamic API behavior, limited explainability, incomplete specifications, real-world validation, and the divide between vulnerability detection and developer focused reporting. Finally, we discuss future research directions, including dependency-aware testing, graph-based representations, machine-learning-assisted detection, explainable security analysis, continuous API security testing, and integration with DevSecOps workflows.

Keywords: REST API Security, API Vulnerability Detection, RESTful APIs, API Fuzzing, Vulnerability Testing, Machine Learning, Explainable AI, OpenAPI, Security Testing, API Security

Introduction

The Representational State Transfer architecture is one of the most popular methods to create APIs for web-based applications. The REST API exposes resources of an application in HTTP URLs and often uses operations like GET, POST, PUT, PATCH, and DELETE. Their simplicity, scalability, interoperability, and compatibility with different software make REST APIs popular. REST APIs are usually documented by machine-readable documents, like the OpenAPI Specification (previously known as Swagger). These documents may describe paths of endpoints, methods, parameters, schemas of requests and responses, authentication, etc. Growing reliance on APIs has contributed to the growing security attack surface of modern software applications. Vulnerabilities in APIs can have a direct impact on application data, business logic, authentication, and backend infrastructure. Some common vulnerabilities in APIs include BOLA (Broken Object-Level Authorization), Broken Authentication, Broken Function-Level Authorization, Mass Assignment, Injection, Security Misconfiguration, Excessive Data Exposure, and Unrestricted Resource Consumption. The above list of vulnerabilities is not language/framework specific but can be a result of the interaction of an API contract, logic, authorization, data, and state. Another way in which API vulnerabilities differ from many other conventional web vulnerabilities is that the former's exploitation may involve interactions between different operations. For instance, the detection of a broken object-level authorization vulnerability may require an attacker to first create a certain resource with one identity and then try accessing that resource with another identity. Moreover, operations like DELETE may have a dependency on a prior operation such as POST, which creates the resource. This means that it is possible for vulnerabilities to exist due to the sequence of operations carried out on APIs but testing each endpoint individually may not reveal such vulnerabilities. [1]. Automated security testing of REST APIs has seen progress in various research directions. Specification-based and black-box methods try to automate API request generation either from API specifications or their observations. While vulnerability-based approaches focus specifically on security properties, other research studies look into static analysis and vulnerability detection at a dependency level. Machine learning-based methods focus on finding vulnerable source code, network traffic patterns, and API behavior. Explainable artificial intelligence techniques, like LIME and SHAP, have also been explored to increase interpretability of machine learning predictions. [10], [3]. Even though there have been improvements, the field of study is still fragmented. For example, automatic testing techniques are capable of making a great number of requests but explaining their results. Machine learning algorithms can enhance classifications but function like black boxes. XAI techniques can give attribute at the level of features but do not necessarily turn these features into a full security

explanation or remediation guide. Moreover, according to the latest survey, the field of REST API vulnerabilities includes various vulnerability types and detection techniques, thus, a comprehensive view is important for comparison. [2]. Thus, there needs to be a clear understanding of the current studies to see how the various methodologies differ, what security vulnerabilities they address, what evidence they provide, and what issues remain unsolved.

A. Motivation

Reasons for this survey include growing complexity of REST API security testing and fragmentation of the research in this area. Existing studies target various aspects of the problem, including API specification processing and fuzzing, access control testing, and machine learning-based categorization of vulnerabilities. Yet, these techniques are evaluated in isolation and utilize various representations of API semantics. A systematic review may contribute to revealing methodological similarities and differences and answering the question whether the current research covers critical aspects of the problem, such as multi-step operation dependencies, API dynamics, explainability, and deployability. Another reason for conducting the survey includes the gap between the observation made by a security tester and the information required by the developer to fix the vulnerability. The API test may provide a test engineer with unusual status codes, a response that includes unauthenticated fields, or a sequence in which a restricted action became possible. However, a single observation does not imply the identification of a particular security requirement. Thus, a more comprehensive approach to detecting vulnerabilities should be developed. A useful system is able to make a decision about the security property and record the endpoint, principal, object identifier, payload, sequence, response behavior, and the policy A third incentive arises due to the rising inclusion of automation within software deployment. Application Programming Interfaces, application code, third-party packages, runtime logs, and continuous integration and deployment events all generate data pertinent to the investigation of security. The literature indicates that each of the sources holds data that the other sources may lack. For instance, static dependency analysis can be used to discover whether vulnerable third-party packages can be reached, whereas dynamic testing can confirm whether an endpoint displays the suspected behaviorr [4]. That is why hybrid methods work better than one detection method.

B. Objectives of the Survey

This survey has the following objectives:

1) To investigate current methods for automated REST API vulnerability detection.
2) The classification of literature on the basis of research methodologies.
3) To analyze the vulnerability classes addressed by existing techniques.
4) To compare specification-driven, black-box, static, machine-learning, and explainability-oriented approaches.
5) For recognizing limitations and research gaps in existing REST API vulnerability detection methods.
6) Identify potential areas for future research.

C. Contributions

The key contributions of the survey are as follows:

1) A classification scheme for various REST API vulnerability detection techniques that have been developed thus far.
2) A comparison of selected studies in terms of multiple security testing frameworks.
3) The investigation of the impact of the OpenAPI definition on API security testing.
4) The analysis of vulnerability-focused and access controlbased testing.
5) The investigation of the machine learning and explainable AI in vulnerability detection.
6) The identification of key limitations and open research issues.
7) Future research direct

Background

A. RESTful APIs and OpenAPI

A RESTful API is a way of exposing application resources via HTTP endpoints. Each of these HTTP endpoints corresponds to a particular resource and HTTP operation. Common methods include GET for retrieval, POST for resource creation, PUT and PATCH for updates, and DELETE for resource removal. Requests may contain path parameters, query parameters, headers, authentication tokens, and structured bodies. Responses commonly include status codes, headers, and serialized resource representations. OpenAPI specifications provide machine-readable descriptions of these interfaces. They can contain endpoint paths, HTTP methods, parameters, schemas, authentication requirements, and request or response definitions. This makes OpenAPI an important input for automated security-testing systems because security tools can construct a structured representation of the API without manually discovering every endpoint. Specification processing may also assist in generating valid base-line requests before applying any security-based mutations. Nonetheless, specification-based testing is reliant on the quality of the specification itself. Unspecified endpoints, incorrect schema definition, lack of authentication details, incorrect representation of optional fields, and discrepancies between the specification and the implementation undermine the efficiency of automated processing. This becomes more significant if there is a case where security relies on the application state that cannot be modeled by a static list of endpoints. Hence, OpenAPI should be regarded as a source of structure only.

B. Vulnerability Classes in REST API Security

Vulnerability classes found in relation to OWASP API Security Top 10 and similar security issues have been repeatedly mentioned throughout the reviewed literature in this survey.

1) Broken Object-Level Authorization (BOLA): BOLA can be defined as a situation when the API does not properly authorize access to the specific object. An attacker can change the ID of the object and access or modify the object owned by other users. In order to automatically

detect BOLA, there should be at least two principals, the set of object IDs and some kind of response/side effect comparison. BOLA is a great example of a vulnerability which is hard to detect through syntax-only fuzzing.

2) Broken Authentication: A broken authentication results from any weakness in identity verification process, tokens, sessions, or in the authentication process itself, leading to access by unauthorized entities. Testing can include invalid token, token replacement, missing authentication attempts, token expiration, and comparison between authenticated and non-authenticated operation sequences. This is more likely to be behavioral testing, because it may happen that the specification will state the need for authentication at a certain endpoint, but the implementation does not actually perform it.

3) Broken Function-Level Authorization: This defect happens in case when a certain privileged user's operation can be accessed by a user with less privileges or even with no authentication at all. Unlike Broken Object Level Authorization, which relates to the access to a particular object, the function level authorization deals with the permission to invoke a certain operation.

4) Mass Assignment: In mass assignment, client-controlled fields are automatically mapped to application objects in an unrestricted manner. In this case, sensitive fields like role and permissions could be controlled by the attacker. The specification mining approach can be used to detect possible suspicious operations based on the presence of fields that resemble privileged or internal state in the request schemas. Mazidi et al. investigated this issue by applying a light-weight specification mining approach and found that in a preliminary analysis of 100 APIs, 25 were at risk for mass assignment, and nine vulnerable operations were found in six APIs [8].

5) Injection: The vulnerability results from interpreting untrusted data as executable statements of SQL, operating system command, code, template, or anything else unexpected. Testing the API could execute the payload with respect to parameters and investigate the reaction to these attack vectors in the responses or external effects. Sahin, Zhang, and Arcuri incorporate security oracles for injection attacks and access policy violation in REST API fuzzing [6].

6) Exposure of Excess Information and Security Misconfiguration: Information may be exposed in APIs because of excessive exposure through responses, error messages, and misconfigurations of security measures. Such configurations as verbose errors, loose cross-origin settings, security misconfiguration, and administrative capabilities may cause such exposures. These issues may depend on the context; for example, an HTTP response may be syntactically correct but still violate some security expectations.

7) Lack of Restrictions on Resource Consumption: Lack of rate limitations, payload limitations, pagination limitations, or resource limitations allows attackers to consume excessive amounts of computational or network resources. Security testing for the current class is supposed not only to ensure that everything works correctly but also check how resources behave when overloaded with requests. It explains why comprehensive security testing should take into account test costs and number of requests.

C. Threat Model and Proof of Detection

In order to categorize the literature, the REST API is considered to be an interface which is accessible over remote channels and enables one or several principals to create, read, update or delete the application resources. The attacker will be able to alter the identifiers, parameters, request body, header, authentication context, and operations sequence, but will stay inside the channel of communications that are used by an ordinary API client. It is not required for the literature to have one universal adversary model, thus the current model is descriptive only. A good detection approach can be considered to provide evidence on multiple layers. The first layer of evidence is the request evidence that includes the HTTP request method, the endpoint, the parameters, the payload, and the authentication context. The second layer of evidence is the response evidence that involves status code, response body, response headers, and any observable side effect. The third layer of evidence is the relational evidence that establishes connections between operations within a certain chain of events, e.g., the creation of the resource followed by its access by another user. The fourth layer is the security interpretation that interprets the behavior as a vulnerability.

Systematic Literature Review Methodology

A. Research Questions

The literature review will be based on the following research questions:

1) RQ1: What approaches to REST API vulnerability detection have been developed?
2) RQ2: Which types of REST API vulnerabilities are most often detected by available methods?
3) RQ3: What is the role of API specifications such as OpenAPI for automatic vulnerability detection?
4) RQ4: How are complex multi-step API dependencies taken into account by existing security testing methods?
5) RQ5: How are machine learning approaches applied to REST API vulnerability detection?
6) RQ6: What is the role of explainability in vulnerability detection systems?
7) RQ7: What are the main problems of current research in REST API vulnerability detection?
8) RQ8: In what ways can future research in REST API security testing be further improved?

B. Information Sources

The literature was gathered from well-known academic databases, such as IEEE Xplore, ACM Digital Library, SpringerLink, ScienceDirect, Web of Science, Scopus, and Google Scholar. Moreover, other sources were utilized to identify relevant papers, which are not necessarily present in the mentioned databases. Information about bibliographies and methodological details was verified on publisher pages and freely accessible preprints.

C. Search Strategy

The search strategy included keywords concerning REST APIs, vulnerability detection, security testing, fuzzing, machine learning, and explainability. An example of a search string is:

(REST API OR RESTful API) AND (vulnerability detection OR security testing OR security analysis) AND (fuzzing OR machine learning OR static analysis OR dynamic analysis OR access control)

Other searches have been conducted concerning specific vulnerability classes, e.g., BOLA and mass assignment, and intelligent approaches, e.g., machine learning, transformers, SHAP, and LIME. The search strategy has been made rather wide since the same problem can be described differently in the literature.

D. Inclusion and Exclusion Criteria

A paper was included when it studied REST or RESTful APIs; addressed API security, vulnerability detection, security testing, or any related topic; suggested, discussed or analyzed a security-testing or vulnerability-detection method; was written as a scholarly article or conference or journal paper; and contained enough technical details for analysis. Papers were excluded if they solely concerned non-REST architectures without any related analysis; if they did not concern API security or vulnerability detection; were duplicate or inaccessible; or had nothing to do with the research questions.

E. Data Extraction and Synthesis

Key data from each chosen paper were collected to serve as a basis of comparison, such as the year of publication, research aim, API representation, input, testing technique, vulnerability categories, static/dynamic analysis nature, machine learning usage, explanation ability, and main disadvantages. This is a qualitative rather than a quantitative statistical meta-analysis. This is necessary since the studies under review use different data sets, API approaches, vulnerability approaches, test budgets, and evaluation methodologies. For instance, while one study may focus on discovering vulnerabilities in the case of actual services, another study may investigate specification patterns in a set of APIs, while the third study may assess the reachable vulnerable third party APIs in the case of Java applications. It would be misleading to combine these results to get a unified accuracy measure. Instead, the studies will be compared using common criteria, including observables, generators, security properties, evidence and assumptions.

Survey Of Literature and Classification

The surveyed literature can be classified into five broad categories:

1) Specification-driven and black box REST API testing. 2
4) Vulnerability-focused and access control aware detection.
3) Static and dependency focused vulnerability detection.
4) Machine learning and transformer based vulnerability detection.
5) Explainable artificial intelligence based vulnerability detection.

F. Specification-Driven and Black-Box REST API Testing

The specification-driven approaches leverage API documentation, like OpenAPI, to create requests and test the API's behavior. The main benefit of using the specification-driven methods is that the information about endpoints and parameters is available from the interface contract, hence eliminating the necessity to discover them manually. After the specification has been parsed, one can generate valid seed requests and vary parameters, authentication status, and operations. NAUTILUS by Deng et al. is an automated REST API vulnerability detection framework that considers the significance of interdependencies among several API operations. This framework uses security-oriented annotations over OpenAPI specifications to help with generating request sequences and parameters. The evaluation showed that NAUTILUS found considerably more vulnerabilities and tested more API operations than other frameworks on six RESTful services. Moreover, in another evaluation, NAUTILUS found 23 novel vulnerabilities on nine RESTful services corresponding to 12 CVE numbers [1]. This result highlights the general observation that sequence awareness can reveal faults that cannot be discovered if each endpoint is considered independently. The advantages of black-box fuzzing, however, are its low requirements to understand the implementation details. Nevertheless, a pure input mutation-based approach can still be ineffective regarding semantic authorization vulnerabilities. The server can respond with a valid 200 status code to an unauthenticated client and will not generate a crash or create any malformed state. Sahin, Zhang, and Arcuri try to improve this situation by developing security-oriented automated oracles that will cover access policy violations, SQL injection, and XSS vulnerabilities and integrate the oracles with EvoMaster [6] framework. The authors' evaluation on artificial, intentionally insecure and corpus-based APIs proves the usefulness of request generation combined with specific security validation. Request generation in the case of specification-driven and black-box approaches, therefore, does not constitute their main limitation but rather the lack of proper security semantics. While OpenAPI description language is good at describing the interface structure, it does not necessarily describe the right principal to access the right object, the process of validating the identifiers of an object and the order of operations.

A. Vulnerability-Oriented and Access-Control Aware Detection

Access-control weaknesses form one category of security issues in REST APIs, since a successful attack will result in a valid HTTP response. Fuzzers based on exception detection may not be able to determine whether the request is unauthorized when the attempt to perform an attack results in success. In their work on automated detection of access control vulnerabilities using API specification analysis, Barabanov et al. used a pattern-

based approach that identifies IDOR/BOLA attacks, categorizes attacks by similarity, and maps the attack patterns to the properties exposed in OpenAPI specifications. In addition, the authors developed an algorithm that automatically generates a list of vulnerable API endpoints and attack vectors [5].

Du et al. suggested VOAPI2, a vulnerability-oriented inspection framework founded on the premise that the functionality of the API interface provides insight into the vulnerabilities that could potentially lurk behind it. The VOAPI2 framework works with keywords to analyze the functionality of the API, builds up a stateful sequence of requests, inserts payloads, and checks the resulting behavior with the help of feedback-driven testing. For seven real-world RESTful APIs, the framework detected seven zero-day and nineteen disclosed vulnerabilities; 23 vulnerabilities were assigned CVE identifiers [9]. This finding strengthens the necessity of applying vulnerability-specific test generation and taking the functionality of the application into account when choosing test cases.

Both approaches, specification processing and vulnerabilityoriented testing, provide grounds for security testing to transition from asking a generic question "Does this endpoint work?" to posing a security-related question "Can this principal with this privilege acquire this object or perform this action in this sequence of states?" This transition influences the way the oracle should be designed: the oracle should describe the security-related behavior, not just the protocol compliance.

B. Static and Dependency-Level Vulnerability Detection

Static techniques inspect the source code, API specifications, or dependencies of the target API without running the API. These are important where the security analysis needs to take place before deploying the software or where it is costly to simulate runtime environments. Static representations can reveal sensitive functions, data flow, dependencies, and call graph reachability.

In their work, Zhang et al. examined the use of vulnerable third-party APIs and demonstrated how merely recognizing that a project uses a vulnerable third-party library would give an overestimate of the risk involved. Using the VAScanner technique, they traced the vulnerable root methods and used backward call-graph analysis to find out the vulnerable APIs that are indeed accessible via the project. They have reported a database with 90,749 vulnerable APIs from 362 third-party libraries and 14,775 versions, with a low false-positive ratio in their constructed data, and a large-scale experiment in which a significantly smaller number of projects using vulnerable libraries are observed to invoke vulnerable APIs [4]. The importance to REST API security methodology is that dependency existence and vulnerability reachability are two separate properties.

Static analysis still does not have access to information on deployment-time configuration, authentication state, external inputs and dynamic control flow. Theoretically accessible path in the source code may turn out to be inaccessible when viewed from the deployed perspective, and a seemingly safe endpoint can exhibit different behavior because of configuration or state assumptions. This makes hybrid static-dynamic analysis very natural. While static analysis can find potentially dangerous endpoints and dependencies, dynamic testing can verify their exploitability using realistic requests.

C. Vulnerability Detection Using Machine Learning

The machine learning approach has recently become increasingly common for vulnerability detection. Ahmad and Zhang looked into BERT-based approach to vulnerability detection with SHAP explanations, showing one possible way in which contextualized representations are employed instead of the pure pattern matching approach [7]. Despite the fact that this work was devoted to source code vulnerability detection and not to REST API endpoint behavior detection, this technique is relevant to our survey as API security systems might utilize structured text, source code snippets, request/response pairs or traces as features for their models.

Detection via ML allows for learning the relationships that are hard to represent manually. The model could be based on several weak signals like endpoint name, parameter types, patterns of status codes, structure of payloads, or request flows. At the same time, the quality of a model is largely defined by the representation and labeling of the training dataset. The classifier trained on one framework, set of vulnerabilities, or request pattern will not work for another API ecosystem. Imbalanced dataset could also make the model biased towards frequent classes.

The most relevant comparison would thus not be simply "ML versus non-ML". ML could fit into a particular stage of a larger pipeline. For example, rule-based checks could impose explicit invariants, static analysis could provide the context of implementation, dynamic testing could gather behavioral evidence, and ML could prioritize suspicious cases or identify more complicated patterns. Such an assignment of tasks takes some burden off the model by allowing it to concentrate on what really matters.

D. Explainable AI for API Security

Explainable AI strives to make ML predictions interpretable. LIME provides local explanations through an interpretable model fitted locally around the instance being investigated, while SHAP offers attribution scores from a unified additive explanation methodology [10], [3]. Both approaches could assist in determining which of the input attributes were responsible for a certain prediction.

In the field of cybersecurity, binary decisions such as "Vulnerability = True" cannot be enough. The analyst should know what triggered the detection process, what request attribute or source code feature was responsible for the decision, what security property was compromised, and which evidence supports the result. The work of Ahmad and Zhang on vulnerability detection through SHAP demonstrates how the attribution problem could be combined with security classification, but the feature attribution problem itself is not equal to the remediation explanation problem [7].

Thus, explainability needs to be looked at from various levels. The model level explainability is related to the impact of features, while the instance-level explainability explains the particular request, code snippet or trace leading to that particular decision. Finally, the system-level explainability relates to the security logic behind mapping evidence to vulnerabilities. Thus, ideally, the final reporting interface needs to connect all these different levels together.

Comparative Analysis of Studies

The above-discussed papers address different components of REST API security issues. The table below provides the key function played by each paper and the type of evidence utilized for validation of results.

A. Comparing Analysis Levels

One approach for comparing the literature is in terms of the level at which each analysis operates on the software stack. At the specification level, methods reason about endpoints, parameters, schemas, and asserted relationships. At the source level, methods examine implementations, call chains, dependencies, and reachability. At runtime, methods look at requests, responses, and state transitions. Intelligent analysis methods operate on representations based on one or more of these levels. This approach explains how apparently competing methods are in fact complementary.

It is worth noting that the comparison also brings out the common theme of automation versus semantics dilemma. Generic generation can explore a larger search space, but semantic vulnerabilities need some context. Targeted methods incorporate more knowledge and thus are capable of making meaningful checks, but this information needs to be provided, updated, and adapted to new APIs. Static analysis helps in reducing execution cost but produces assumptions which must be proven dynamically. ML techniques can help avoid feature engineering manually, but they have their own demands for data and interpretation.

B. Evaluation Dimensions

The surveyed methods cannot be compared solely based on accuracy. API security testing produces artifacts which are understood in terms of exploration technique and the evaluation framework adopted.

Some of the following dimensions are crucial:

Vulnerability detection rate: This is the number or proportion of known vulnerabilities detected within a certain test budget. It makes sense only if the vulnerable ground truth is known.

Coverage of endpoints and sequences: The percentage of endpoints, operations, parameters, or operation sequences tested during the process. The importance of sequence coverage arises particularly when testing stateful vulnerabilities, since many uncorrelated requests do not necessarily imply thorough investigation of sequences of requests.

False positives and false negatives: A security analysis tool has to be as little misleading and miss as few vulnerabilities as possible. As shown by the VAScanner paper, having the right reachability information can have significant impact on the interpretation of dependency vulnerabilities [4].

Cost of execution: Test generation and security analysis take network requests, CPU, memory, and possibly application state. In CI/CD environments, unnecessary testing can slow down the development cycle.

Reproducibility of the evidence: A valuable finding has to come along with a request or sequence that can reproduce it. Research on security oracles in REST API fuzzing focuses explicitly on the need to generate an executable test case for the discovered security vulnerability [6].

Explainability and usefulness: In machine learning-based tools, the possibility of feature identification is just one side of the story. The explanation provided must allow the security analyst or developer to grasp the found condition and know how to.

Taxonomy of REST API Vulnerability Detection Approaches

Given the literature, current techniques can be classified depending on the level at which they process the API. Figure 1 is an extension of the original taxonomy in that it illustrates the flow of structural, implementation, and runtime evidences into the rule-based, ML-based, and explainability layers.

The taxonomy makes a distinction between what is known about the API and how this knowledge is being used. The structure of an API is described by its specifications; its implementation – by the source and dependency code analysis, and its behavior – by the runtime monitoring. The rules can be applied to all three API representations, and machine learning algorithms can be used to rank/classify observations, while the explainable AI can help to find the proof.

TABLE I
SUMMARY OF REPRESENTATIVE REST API VULNERABILITY DETECTION STUDIES ON A STUDY LEVEL

Study	Input type	Strategy type	Target	Evaluation	Limitation identified in the summary
NAUTILUS [1]	OpenAPI + annotations	Sequence-aware generation	Multi-step API vulnerabilities	Comparison on six services; real-world study on nine services	Dependent on meaningful specification annotations and relation information
Barabanov et al. [5]	OpenAPI specification	Pattern mapping + rule analysis	IDOR/BOLA	Algorithm and tool evaluation	Strongly dependent on access-control patterns from the specification
VOAPI2 [9]	API specification + behavior	Function-aware, stateful, targeted testing	Vulnerability-oriented detection	Seven real-world APIs; zero-day and disclosed bugs	Targeted testing increases semantic requirements and costs
Sahin et al. [6]	Live API + generated requests	Security oracles integrated with fuzzing	Policy violations, SQLi, XSS	52 APIs in artificial, vulnerable-by-design, and corpus modes	Dependent on policy oracle coverage and test budget
Mazidi et al. [8]	API specifications	Lightweight specification mining	Mass assignment	100 APIs; 25 potentially affected; nine confirmed cases	Limited to a vulnerability category
Zhang et al. [4]	Source code + dependencies	Patch/root-method + call-graph analysis	Vulnerable third-party API usage	Dependency database and software project study	Needs implementation-level and language-dependent analysis
Ahmad and Zhang [7]	Source code	BERT classifier + XAI	Source-code vulnerabilities	Evaluation of model performance using SHAP and attention techniques	No direct generalization from source code analysis to API vulnerabilities
Lundberg and Lee [3]	Model predictions + features	Additive feature attribution	General interpretability	Interpretability framework for ML models	Feature attribution does not inherently have application-specific semantics

TABLE II
COMPARISON OF MAJOR REST API VULNERABILITY DETECTION APPROACHES

Approach	Primary Input	Main Strength	Major Limitation
Black-box fuzzing	Live API	Automated request generation and broad exploration	Limited semantic understanding
Specification-driven testing	OpenAPI	Structured endpoint discovery and valid request construction	Depends on specification quality
Access-control analysis	Spec / behavior	Explicitly targets BOLA/IDOR and privilege violations	Often vulnerability-specific
Static analysis	Source / dependencies	Early detection and reachability information	Limited runtime context
ML detection	Code / traces / traffic	Learns complex patterns	Dataset and generalization issues
XAI	ML predictions	Improves interpretability	Attribution is not full security reasoning
Hybrid analysis	Multiple layers	Combines structural and behavioral evidence	Higher implementation and execution complexity

Research Gaps

The synthesis of the reviewed literature highlights several gaps which still remain unresolved.

A. Multi-Step API Dependencies

Not all of the weaknesses can be discovered via endpoint testing only. The weakness may depend on the sequence of actions performed by the attacker. The connection is not always present in the specification. In this case, the discovery process turns out to be difficult. NAUTILUS shows how useful it is to represent action relations explicitly. VOAPI2 illustrates the significance of stateful targeted sequences [1], [9]. However, a unified representation of dependencies remains to be developed yet.

There are multiple ways to define a dependency. The first is the data dependency that appears when an identifier returned by the operation is used as an input parameter of another. The second is the control dependency when an authentication or initialization action must be done before performing a dangerous operation. The third type of dependency is the policy dependency between the operation and the particular user role. The fourth type of dependency is the state dependency that requires an earlier mutation to modify the server state before the operation is meaningful.

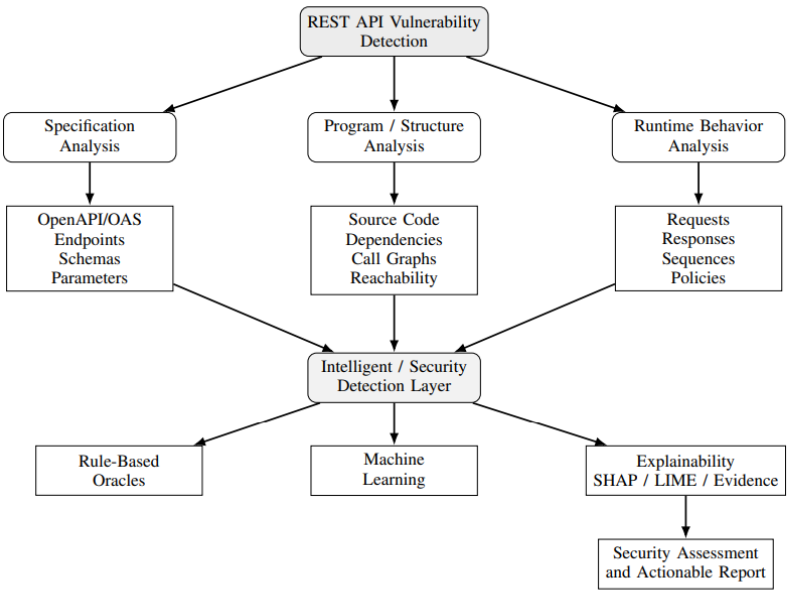


Fig. 1. Extended taxonomy of approaches to detection of vulnerabilities of REST API. The layers are complementary, not exclusive.

B. Inconsistency Between Specification and Implementation

Specification-driven testing presumes the completeness of the API specification. In reality, APIs include undocumented endpoints, outdated schematics, security requirements, and implementation-specific behavior, leading automated systems to ignore vulnerabilities beyond the surface that has been documented. On the other hand, a too relaxed schema may cause a tool to make up incorrect requests, thereby generating noise and wasting test budget.

A solution to the problem does not lie in discarding the specification. The specification may be taken as one of the pieces of evidence and compared with the runtime behavior. Discrepancies between the declared and observed behavior are of security interest by itself. Thus, for instance, a specification can declare the necessity of authentication in order to access an endpoint, whereas a live request with no authentication passes successfully.

C. Limited Explainability

Automated approaches generate binary or multi-class decisions. A good security tool must support the workflow of Detection → Evidence → Reasoning → Severity → Mitigation. The need for explanations in XAI models at the level of security analysis is yet to be fulfilled.

The explanation needs to make use of real-world objects. For a BOLA example, the report can contain the identifier of the object in question, two authorizations that were involved, the sequence of how the object was accessed, and the discrepancy of the response which means that the access was not authorized. In the case of source-code XAI models, the report will refer to the piece of code and the features responsible for making a decision.

D. Machine-Learning Generalization

Detecting tools trained on particular distributions of vulnerabilities will most likely be less efficient for detecting those that have not been seen before in terms of new APIs, new frameworks, and other novel request patterns. There is much work to do with regard to cross-API generalization. The problem encompasses domain shift and label shift: the same type of vulnerability can look very different depending on the technology and the prevalence of different vulnerability types may vary from one dataset to another.

The absence of any standard benchmarking suite makes direct comparisons hard. Different train/test splits, synthetic apps, real apps, or source code benchmarks can be used to test the performance of different models. In such a way, any seeming improvement in terms of detection can be nothing else but a result of dataset peculiarities and not better performing detectors. In future studies, more than just classification metrics should be reported.

E. Real-World Evaluation

There is a distinction between synthetic benchmarks and real-world APIs that are dependent on distribution, dynamic state, third-party services, and deployment policies. Evaluations at an industrial scale are necessary for verification of the proposed approach. What distinguishes NAUTILUS and VOAPI2 is that the real-world APIs have been evaluated in addition to toy examples [1], [9].

Real-world evaluations come with their own set of challenges. The data can be inaccurate, the APIs can evolve during the evaluation period, and aggressive testing could be risky and operationally disruptive. Reproducible evaluation, therefore, requires a combination of open benchmarking environments and well-controlled real-service evaluations. It is necessary for researchers to provide information about the budget of tests performed, authentication assumptions, API versioning, and ground-truth acquisition process.

F. Detection vs. Actionable Report

Simply detecting "BOLA" is not enough for developers. The reports need to make the findings actionable by identifying the exact endpoint, principal, and identifier involved in the attack, along with the recommended actions to fix the issue. This is more critical with automation as having a huge number of alerts will otherwise clutter the analysts.

The reporting problem can be viewed as an instance of the structured output problem. In the least, the report should include the endpoint, HTTP method used, security context, request sequence, response received, violated security property, confidence/evidence, and proposed remedy category.

Integrated Detection Approach

According to the literature, there should be an integrated approach where no one technique has to resolve all layers of the issue. Figure 2 illustrates a proposed workflow based on the discussed approaches.

In terms of the architecture, the first step is evidence normalization. The OpenAPI paths and schemas, functions in source code, dependencies and traces at runtime are translated into a form that allows for linking by common endpoint/operation identifier. Relationships between endpoints and data objects can then be expressed with a graph representation. Security policies define the expected behavior that is compared against the observations.

In the second phase, stateful test generation becomes possible. Valid baseline requests can be generated based on the API specification, whereas dependency edges will dictate which operations need to be invoked first. Based on the vulnerability under suspicion, targeted security payloads will be chosen. Rule-based oracle may check explicit invariants like authorization behavior or response field constraints. In ML layer, the priority will be given to traces or sequences that seem unusual, which will lessen the search space without substituting the deterministic verification.

At last, evidence gathered during execution, rules, and the results of model will allow to formulate the finding. In this architecture, explainability goes beyond the concept of feature attribution. The concrete request sequence and behavior observed should be preserved as well.

Future Research Directions

A. Dependency-Aware API Security Test Generation

The future approaches need to model the APIs as stateful graphs instead of separate endpoints in which the edges indicate the dependencies between different operations. The graph needs to incorporate data flows, creation of resources, authentication states, and changes in roles, if any, in the case where these dependencies can be detected or deduced. The use of graph structure will also help with prioritizing the testing paths; testing the paths that lead to critical operations or crossing trust boundaries can be considered first.

Another interesting problem is detecting the dependencies automatically. The presence of explicit dependencies in the OpenAPI specification is useful; however, the runtime value detection will uncover the dependencies which were not included in the OpenAPI specification. For instance, an identifier generated through the creation operation will depend on the deletion or retrieval operations.

B. Machine-Learning-Assisted Security Testing

The use of ML as an intelligent filtering layer that can identify promising attack paths and classify anomalies instead of outright replacing traditional security testing is very appealing since the security properties are deterministic in nature and the security testing itself can afford to execute fewer candidates.

More research needs to be done in creating representations that take into account the path structure, parameter semantics, authentication state, responses, and operation sequences. Transformers can potentially serve the purpose of contextualized representations but the evaluation should focus more on generalization across frameworks and APIs rather than just within the dataset.

C. Explainable Vulnerability Detection

The future work should go from only providing the attributions of features to complete explanations that relate model predictions, evidence from APIs observed, security context, and remediation actions. SHAP and LIME can be helpful for local attributions, but their results should be complemented by some security artifact. For instance, the system could link important features to the parameter/response field that causes the authorization violation in the API call.

Stability is another concern when developing explanations. In other words, an explanation should remain similar even when some non-essential information in the requests is changed. Thus, an evaluation can consider fidelity, stability, analyst understanding, and remediation utility along with model-based metrics.

D. Continuous API Security Testing

Vulnerability detection should be deeply embedded in CI/CD workflows to auto-retest endpoints as they change due to changes in schemas, dependencies, and authentication policies in the course of development. It might be useful to use the existing APIs to automate such testing since the API definition, codebase, and dependencies are already known in modern software engineering practices.

The other area for future work is incremental analysis of the software. It would not make sense to re-run full testing of the whole API after every change. One may build an automated system that could detect what parts of the API have been changed, which security aspects could be affected, and perform re-tests only in these parts.

E. Hybrid Static and Dynamic Analysis

The combination of static source-code analysis to detect sensitive operations, and dynamic fuzzing to confirm exploitability, can overcome the individual weaknesses of each of these techniques. Static analysis may be used to identify areas of code or dependencies which need to be looked at, while dynamic analysis can determine if the identified behavior is really exploitable using externally available requests.

A hybrid approach can also provide greater explainability of the results since the ultimate report could link the runtime behavior to its place in the implementation. The linking is particularly useful in case when dynamic analysis identifies some security issue, but finding the authorization control or validation of input becomes necessary.

F. Evaluation by Standardized Measures

Future research will need standardized criteria of assessment aside from accuracy, including endpoint and sequence coverage, resource consumption, reproducibility of results, and, for explainable approaches, analyst understanding and helpfulness of explanations. Evaluation must describe not only the advantages but also drawbacks of the approach being assessed.

The benchmark needs to encompass multiple frameworks, different authentication mechanisms, stateful processes, known and unknown behaviors, as well as different kinds of vulnerabilities. The benchmark needs to be able to differentiate between synthetic vulnerabilities and actual implementation errors, as well as keep enough details to be able to repeat each test.

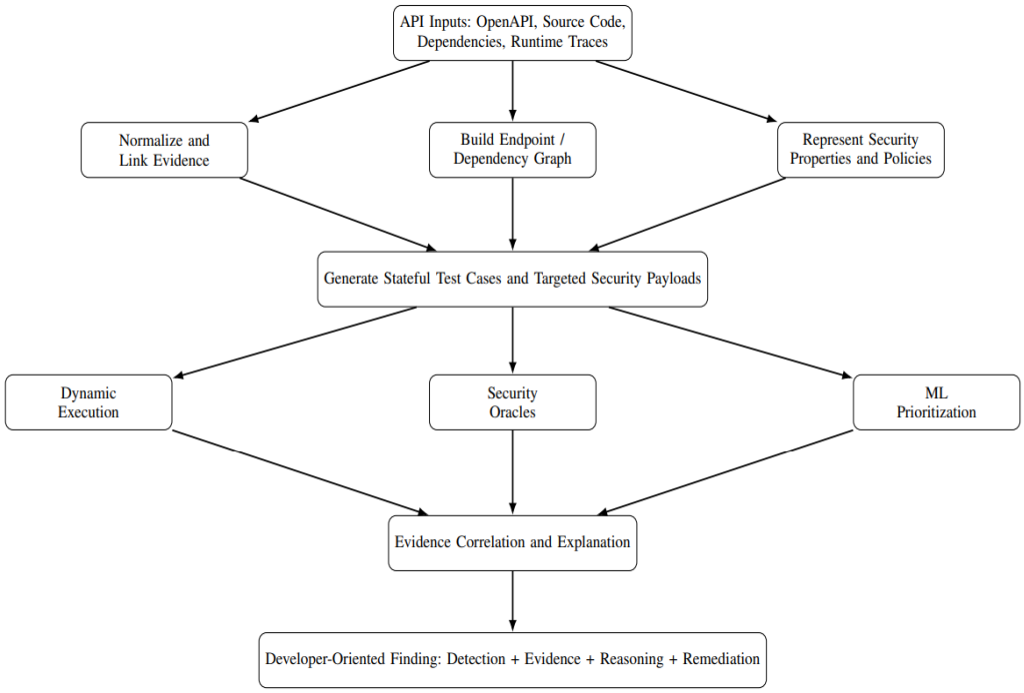


Fig. 2. Conceptual integrated workflow synthesized from the reviewed specification-driven, static, dynamic, ML, and XAI approaches.

Discussion

As can be seen from the literature, the evolution of REST API vulnerabilities' detection has progressed from a regular approach to request generation to more sophisticated and intelligent ways taking into account aspects of security. Initially, tools were created for crash generation purposes. Later, there was a move to access control attacks using static knowledge. Finally, machine learning (ML) and explainable artificial intelligence (XAI) were used to detect vulnerabilities in APIs.

It is important to emphasize that the result of the present survey is not a statement about superiority of any specific approach, but a conclusion about the layered nature of the security issue when each specific approach serves a different layer of it. Specification-based techniques can be useful for structured endpoint discovery and sequence generation; access-control techniques attack authorization semantics; static techniques give implementation and dependencies informa- tion; dynamic testing gives deployed behavior information; ML gives behavioral classification or prioritization; XAI helps interpret models' results.

Some examples among reviewed papers are also illustrative in this context. NAUTILUS highlights the significance of the relationships between multiple operations [1]; Barabanov et al. demonstrate the mapping of access-control attacks' patterns to the specification properties [5]; VOAPI2 illustrates the advantage of the targeted and stateful function-aware testing [9]; and VAScanner confirms that the dependency existence does not mean the vulnerable

code reachability [4]. Finally, the research on security oracles also implies that a fuzzing framework can be equipped with explicit security checks so that successful exploitations, not just crashes, could become test outcomes [6].

As a result, it could be concluded that a sophisticated REST API security framework should treat structural knowledge, implementation knowledge, behavral knowledge, classification, and interpretability as complementary components of a unified architecture. This finding also explains why the research gaps discussed above remain essential: with every added layer, there appears a new integration problem.

Conclusion

RESTful APIs constitute an important and growing surface for security attacks. In this survey, research on automatic REST API vulnerability detection was explored with a focus on specification-driven testing, access control analyses, static detections, machine learning classification, and XAI (Explainable AI).

Though impressive achievements have been made towards structured endpoint discovery and vulnerability identification, many difficulties persist with respect to the treatment of multisteps interaction, specification mismatches, ML generalization, real-world validations, and transformations from raw detection to meaningful developers' reports. Moreover, the literature suggests that dependency reachability, targeted security oracles, and vulnerability-specific test generation may greatly improve the quality of evidences for an automated security system.

In order to advance the state-of-the-art of REST API security research, future studies should aim at dependency-aware modeling, hybrid static and dynamic analyses, and deeply explainable and continuous DevSecOps integration. Rather than considering specification analysis, dynamic testing, machine learning, and XAI as alternative means for achieving a specific goal, future systems may combine them into a stacked process where each of them provides evidences that fit best for a particular task.

REFERENCES

[1] G. Deng, Z. Zhang, Y. Li, Y. Liu, T. Zhang, Y. Liu, G. Yu, and D. Wang, "NAUTILUS: Automated RESTful API Vulnerability Detection," in Proc. 32nd USENIX Security Symposium, 2023, pp. 5593–5609.

[2] F. Tanveer, F. Iradat, W. Iqbal, and A. Ahmad, "Towards Secure APIs: A Survey on RESTful API Vulnerability Detection," Computers, Materials & Continua, vol. 84, no. 3, pp. 4223–4257, 2025, doi: 10.32604/cmc.2025.067536.

[3] S. M. Lundberg and S.-I. Lee, "A Unified Approach to Interpreting Model Predictions," in Advances in Neural Information Processing Systems, 2017.

[4] F. Zhang, L. Fan, S. Chen, M. Cai, S. Xu, and L. Zhao, "Does the Vulnerability Threaten Our Projects? Automated Vulnerable API Detection for Third-Party Libraries," IEEE Transactions on Software Engineering, vol. 50, no. 11, pp. 2906–2920, 2024, doi: 10.1109/TSE.2024.3454960.

[5] A. V. Barabanov, D. O. Dergunov, D. N. Makrushin, and A. N. Teplov, "Automatic Detection of Access Control Vulnerabilities via API Specification Processing," Voprosy kiberbezopasnosti, no. 1, pp. 49–65, 2022, doi: 10.21681/2311-3456-2022-1-49-65.

[6] O. Sahin, M. Zhang, and A. Arcuri, "Enhancing REST API Fuzzing with Access Policy Violation Checks and Injection Attacks," Journal of Systems and Software, vol. 243, 2027, Art. no. 113060. A preprint was made available in 2026.

[7] N. Ahmad and C. Zhang, "Interpretable Vulnerability Detection in LLMs: A BERT-Based Approach with SHAP Explanations," Computers, Materials & Continua, vol. 85, no. 2, pp. 3321–3334, 2025, doi: 10.32604/cmc.2025.067044.

[8] A. Mazidi, D. Corradini, and M. Ghafari, "Mining REST APIs for Potential Mass Assignment Vulnerabilities," in Proc. 28th Int. Conf. on Evaluation and Assessment in Software Engineering (EASE), 2024, pp. 369–374, doi: 10.1145/3661167.3661204.

[9] W. Du, J. Li, Y. Wang, L. Chen, R. Zhao, J. Zhu, Z. Han, Y. Wang, and Z. Xue, "Vulnerability-oriented Testing for RESTful APIs," in Proc. 33rd USENIX Security Symposium, 2024, pp. 739–755.

[10] M. T. Ribeiro, S. Singh, and C. Guestrin, "Why Should I Trust You? Explaining the Predictions of Any Classifier," in Proc. 22nd ACM SIGKDD Int. Conf. on Knowledge Discovery and Data Mining, 2016, pp. 1135–1144