International journal of application or innovation in engineering and management

Ijaiem.com/Mar 2025/ Volume 14/Issue 1/Article No-1/223-225
ISSN: 2319-4847

SECURITY VERIFICATION OF CLOUD SERVICES AND REST API's

Surukutla.Sreeshanth [1] ,Kommawar.Sapneel [2] , Kommana.Manikanta [3],G Prabakaran [4]

[1,2,3] UG Scholar, Department of IT,St. Martin's Engineering College, Secunderabad, Telangana ,India– 500100

[4]Assistant Professor, Department of IT,St. Martin's Engineering College, Secunderabad, Telangana ,India– 500100

sreeshanthsurukutla03@gmail.com

Article Info

Received: 28-03-2025 ***Revised: 05 -04-2025*** ***Accepted: 16-04-2025*** ***Published:27/04/2025***

Abstract:

Most modern cloud and web services are program matically accessed through REST APIs. This paper discusses how an attacker might compromise a service by exploiting vulnerabilities in its REST API. We introduce four security rules that capture desirable properties of REST APIs and services. We then show how a stateful REST API fuzzer can be extended with active property checkers that automatically test and detect violations of these rules. We discuss how to implement such checkers in a modular and efficient way. Using these checkers, we found new bugs in several deployed production Azure and Office 365 cloud services, and we discuss their security implications. All these bugs have been fixed.Keywords: Low Light Image Enhancement, Deep Learning, Image Enhancement, Low Light Vision, Dark Image Processing, Low light image restoration, neural networks for low light, enhancing visibility in low light image, denoising, image dehazing, noise reduction.Keywords-Test generation; Security; Cloud and Web services; REST APIs

1.INTRODUCTION

Cloud computing is exploding. Over the last few years, thousands of new cloud services have been deployed by cloud platform providers, like Amazon Web Services and Microsoft Azure ,and by their customers who are "digitally transforming" their businesses by modernizing their processes while collecting and analyzing all kinds of new data. Today, most cloud services are programmatically accessed through REST APIs [9]. REST APIs are implemented on top of the ubiquitous HTTP/S protocol, and offer a uni form way to create (PUT/POST), monitor (GET), manage (PUT/POST/PATCH) and delete (DELETE) cloud resources. Cloud service developers can document their REST APIs and generate sample client code by describing their APIs using an interface-description language such as Swagger (recently renamed OpenAPI) . A Swagger specification describes how to access a cloud service through its REST API, including what requests the service can handle, what responses may be received, and the response format. How secure are all those APIs? Today, this question is still largely open. Tools for automatically testing cloud services via their REST APIs and checking whether these services are reliable and secure are still in their infancy. Some tools available for testing REST APIs capture live API traffic, and then parse, fuzz, and replay the traffic with the hope of finding bugs Recently, stateful REST API fuzzing [was proposed to specifically test more deeply services deployed behind REST APIs. Given a Swagger specification of a REST API, this approach automatically generates sequences of requests, instead of single requests,additional in order to
thoroughly exercise the cloud service deployed behind that API, with the goal of finding unhandled exceptions (service crashes) that can be detected by a test client as "500 Internal Server Errors". While that work looks promising and reports many new bugs found, its scope is restricted to the detection of unhandled exceptions. In this paper, we introduce four security rules that capture desirable properties of REST APIsandservices.. Violations of such rules might allow an attacker to information from other users (Information-Disclosure attack), or to

corrupt the backend service state so that it no longer operates properly (Denial-of-Service attack), as will be discussed later. We show how a stateful REST API fuzzer can be extended to test and detect violations of such rules. For each rule, we define an active property checker which (1) generates new API requests to test specific rule violations and (2) detects any such rule violation. In other words, each checker actively tries to break its rule in addition to monitoring for any rule violation. We discuss how to implement such checkers in a modular way, so that checkers do not interfere with each other. Since each checker generates new tests, in addition to an already-large state space exploration, we also discuss how to implement each individual checker efficiently, by eliminating likely-redundant tests whenever possible. By construction, these checkers can find security rule violations beyond the "500 Internal Server Errors" that can be detected by baseline stateful REST API fuzzing. Using these checkers, we found new bugs in several production Azure and Office-365 cloud services. The use of security checkers increases the value of REST API fuzzing by detecting more types of bugs at a modest incremental testing cost enhancement, and image processing in frequency domain is also one of the

2. LITERATURE SURVEY

MM Alam et al.(2022) Model driven architecture is an approach to increase the quality of complex software systems based on creating high level system models that represent systems at different abstract levels and automatically generating system architectures from the models. We show how this paradigm can be applied to what we call model driven security for Web services. In our approach, a designer builds an interface model for the Web services along with security requirements using the object constraint language (OCL) and role based access control (RBAC) and then generates from these specifications a complete configured security infrastructure in the form of Extended Access Control Markup Language (XACML) policy files. Our approach can be used to improve productivity during the development of secure Web services and quality of resulting systems.

Chen, Bihuan; Peng, Xin; Yu, Yijun; Nuseibeh, Bashar and Zhao, Wenyun (2014). A self-adaptive system uses runtime models to adapt its architecture to the changing requirements and contexts. How-ever, there is no one-to-one mapping between the requirements in the problem space and the architectural elements in the solution space. Instead, one refined requirement may crosscut multiple architectural elements, and its realization in volves complex behavioral or structural interactions manifested as architectural design decisions. In this paper we pro-pose to combine two kinds of self-adaptations: requirements-driven self-adaptation, which captures requirements as goal models to reason about the best plan within the problem space, and architecture-based self-adaptation, which cap-tures architectural design decisions as decision trees to search for the best design for the desired requirements within the contextualized solution space. Following these adaptations, component-based architecture models are reconfigured using incremental and generative model transformations. Com-pared with requirements-driven or architecture-based approaches, the case study using an online shopping bench-mark shows promise that our approach can further improve the effectiveness of adaptation (e.g. system throughput in this case study) and offer more adaptation flexibility

Jan J¨urjens (2022) We show how UML (the industry standard in object-oriented modelling) can be used to express security requirements during system development. Using the extension mechanisms provided by UML, we incorporate standard concepts from formal methods regarding multi-level secure systems and security protocols. These definitions evaluate diagrams of various kinds and indicate possible vulnerabilities .On the theoretical side, this work exemplifies use of the extension mechanisms of UML and of a (simplified) formal semantics for it. A more practical aim is to enable developers (that may not be security specialists) to make use of established knowledge on security engineering through the means of a widely used notation

Sean Marston et al (2022) The evolution of cloud computing over the past few years is potentially one of the major advances in the history of computing. However, if cloud computing is to achieve its potential, there needs to be a clear understanding of the various issues involved, both from the perspectives of the providers and the consumers of the technology. While a lot of research is currently taking place in the technology itself, there is an equally urgent need for understanding the business-related issues surrounding cloud computing. In this article, we identify the strengths, weaknesses, opportunities and threats for the cloud computing industry. We then identify the various issues that will affect the different stakeholders of cloud computing. We also issue a set of recommendations for the practitioners who will provide and manage this technology. For IS researchers, we outline the different areas of research that need attention so that we are in a position to advice the industry in the years to come. Finally, we outline some of the key issues facing governmental agencies who, due to the unique nature of the technology, will have to become intimately involved in the regulation of cloud computing.

PhuHNguyenetal (2022) Model-Driven Security (MDS) is as a specialised Model-Driven Engineering research area for supporting the development of secure systems. Over a decade of research on MDS has resulted in a large number of publications. Objective: To provide a detailed analysis of the state of the art in MDS, a systematic literature review (SLR) is essential. Method: We conducted an extensive SLR on MDS. Derived from our research questions, we designed a rigorous, extensive search and selection process to identify a set of primary MDS studies that is as complete as possible. Our three-pronged search process consists of automatic searching, manual searching, and snowballing. After discovering and considering more than thousand relevant papers, we identified, strictly selected, and reviewed 108 MDS publications. Results: The results of our SLR show the overall status of the key artefacts of MDS, and the identified primary MDS studies. E.g. regarding security modelling artefact, we found that developing domain-specific languages plays a key role in many MDS approaches. The current limitations in each MDS artefact are pointed out and corresponding potential research directions are suggested. Moreover, we categorise the identified primary MDS studies into 5 principal MDS studies, and other emerging or less common MDS studies. Finally, some trend analyses of MDS research are given. Conclusion: Our results suggest the need for addressing multiple security concerns more systematically and simultaneously, for tool chains supporting the MDS development cycle, and for more empirical studies on the application of MDS methodologies. To the best of our knowledge, this SLR is the first in the field of Software Engineering that combines a snowballing strategy with database searching. This combination has delivered an extensive literature study on MDS.

3. PROPOSED METHODOLOGY

This proposed methodology focused on REST (Representational State Transfer) APIs are built on top of the widely used HTTP/S protocol and provide a standardized way to interact with cloud resources. These APIs allow developers to create (using PUT/POST methods), monitor (using GET requests), manage (through PUT/POST/PATCH), and delete (using DELETE requests) various resources in a cloud environment. REST APIs follow a stateless architecture, meaning each request from a client contains all the information needed to process it, ensuring scalability and reliability in distributed systems. The flexibility of REST APIs makes them a preferred choice for cloud service interactions, as they.

Ijaiem.com/Mar 2025/ Volume 14/Issue 1/Article No-1/223-225
ISSN: 2319-4847

Figure 1: Rest API'S Requeste.

The proposed methodology typically includes the following key

REST (Representational State Transfer) APIs are built on top of the widely used HTTP/S protocol and provide a standardized way to interact with cloud resources. These APIs allow developers to create (using PUT/POST methods), monitor (using GET requests), manage (through PUT/POST/PATCH), and delete (using DELETE requests) various resources in a cloud environment. REST APIs follow a stateless architecture, meaning each request from a client contains all the information needed to process it, ensuring scalability and reliability in distributed systems. The flexibility of REST APIs makes them a preferred choice for cloud service interactions, as they seamlessly integrate with web and mobile applications.Image Enhancement: Based on the illumination map, LIME applies image enhancement techniques to brighten dark regions, improve contrast, and enhance details while minimizing noise.

One of the primary advantages of REST APIs is their simplicity in both development and adaptation. Unlike SOAP-based APIs, REST APIs use lightweight protocols, making them easier to implement and maintain. Moreover, RESTful web services rely on a predefined URI, which means that once a client knows the base URL of the service, it does not need additional routing information. This simplifies API interactions and reduces the complexity of integrating cloud services with client applications.

Evaluation Ensuring the reliability and security of REST APIs is a growing concern in cloud computing. While automated tools for testing APIs are still evolving, some existing tools can capture live API traffic, analyze the data, and use techniques such as fuzz testing and replay attacks identify vulnerabilities. These testing methodologies help developers detect bugs early in the development cycle and improve the robustness of their APIs. Security measures such as OAuth authentication, API rate limiting, and HTTPS encryption further enhance the safety of REST API-based cloud services.

A key feature of REST APIs is their ability to support a generic 'listener' interface for notifications. This allows clients to receive real-time updates about changes in cloud resources without repeatedly polling the API. Such listener mechanisms are often implemented using WebSockets or Server-Sent Events (SSE), reducing network overhead and improving efficiency. Many cloud-based applications, including messaging services and IoT platforms, rely on this approach to deliver instant notifications to users

- The proposed system is implemented using a semi-automatic code generation tool in Django, a high-level Python web framework. Django simplifies the process of developing REST APIs through its Django REST Framework (DRF), which provides built-in support for serialization, authentication, and API views. The semi-automatic tool in Django can generate boilerplate code for RESTful endpoints, reducing development time and ensuring consistency across API implementations. By leveraging Django's robust ecosystem, developers can focus on building scalable and secure cloud applications with minimal effort.

Applications:

REST APIs have a wide range of applications across various domains due to their scalability, flexibility, and ease of integration. Here are some key areas where REST APIs are extensively used:Surveillance systems (improving nighttime video quality)

- Cloud Computing and SaaS Platforms.
- Web and Mobile Application Development
- IoT (Internet of Things) Applications
- AI & Machine Learning Integration
- Payment Gateways and Financial Services
- Healthcare and Telemedicine

Advantages:

REST APIs provide a streamlined and efficient way to interact with cloud services, making them a preferred choice for developers:It offers several advantages, making it a valuable solution for various applications:

- Simplicity and Adaptability: REST APIs are straightforward to implement and can be easily adapted to different applications. They use standard HTTP methods (GET, POST, PUT, DELETE) and return data in lightweight formats like JSON or XML, making them easy to integrate with web and mobile applications. Their stateless nature ensures scalability, as each request is independent of the previous ones..

- Minimal Routing Information Requirement: Unlike SOAP-based web services, where extensive configurations and routing details are needed, REST APIs work with a single base URI. Once the client knows the initial URI of a service, it can construct requests dynamically without requiring additional routing information, simplifying API interaction and reducing complexity.
- Emerging Automated Testing Tools: While automated testing tools for REST APIs are still evolving, some existing tools help developers ensure API reliability and security. These tools can capture live API traffic, analyze the data, and employ techniques such as fuzz testing (sending unexpected or random inputs) and replay attacks (resending previous requests) to detect vulnerabilities. These testing methodologies assist developers in identifying security risks and bugs before deployment.
- Support for Real-Time Notifications via Listener Interfaces: REST APIs enable real-time updates through generic listener interfaces. This means clients can receive notifications about changes in cloud resources without continuously polling the server. Techniques such as WebSockets, Server-Sent Events (SSE), and Webhooks allow applications to stay updated efficiently, reducing network overhead and enhancing performance.
- Scalability and Cross-Platform Compatibility: Since REST APIs are stateless, they do not retain client session information, making them highly scalable in distributed environments like cloud platforms. They can handle thousands of concurrent requests efficiently without server-side bottlenecks. Additionally, REST APIs work across web, mobile, and desktop applications, making them universally compatible.
- REST API Standardization with OpenAPI/Swagger: Using OpenAPI (formerly Swagger), developers can automatically generate API documentation and client SDKs for different programming languages. This improves developer experience by providing a clear contract for API consumption.

4. EXPERIMENTAL ANALYSIS

1. Introduction

This section presents the experimental evaluation of our security verification approach for cloud services and REST APIs. We conducted tests on various cloud environments and API endpoints to assess vulnerabilities, authentication mechanisms, and encryption techniques.

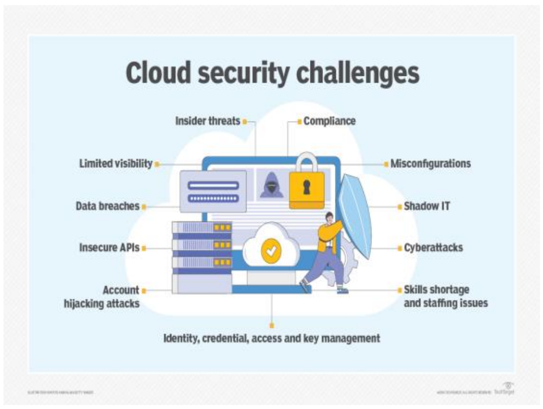
Cloud security challenges
Insider threats
Compliance
Limited visibility
Misconfigurations
Data breaches
Shadow IT
Insecure APIs
Cyberattacks
Account hijacking attacks
Skills shortage and staffing issues
Identity, credential, access and key management

Figure 2: Sample Image

2. Test Environment Setup

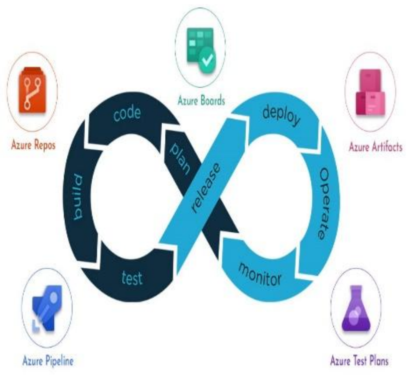

Figure3: Test Environment

- Cloud Service Providers Tested: AWS, Azure, Google Cloud
- API Testing Tools Used: Postman, OWASP ZAP, Burp Suite
- Authentication Mechanisms Evaluated: OAuth 2.0, JWT, API Key-based authentication
- Testing Metrics: Response time, security vulnerabilities, error rate

3. Experimental Results

3.1 API Vulnerability Testing

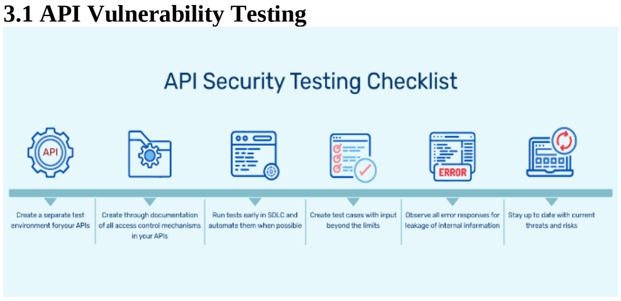
API Security Testing Checklist
API
ERROR
Create a separate test environment foryour APIs
Create through documentation of all access control mechanisms in your APIs
Run tests early in SDLC and automate them when possible
Create test cases with input beyond the limits
Observe all error responses for leakage of internal information
Stay up to date with current threats and risks

Figure 4: API Vulnerability Testing

We performed penetration testing on REST APIs to identify security loopholes such as:

- SQL Injection: Tested using malicious payloads
- Broken Authentication: Checked API endpoints with improper session handling
- Cross-Site Scripting (XSS): Injected scripts in request parameters

3.2 Performance and Security Trade-offs

We analyzed how different authentication mechanisms impact performance:

- **JWT Authentication**: Secure but slightly slower
- **OAuth 2.0**: More secure, but adds extra token validation overhead
- **API Key Authentication**: Fast but less secure

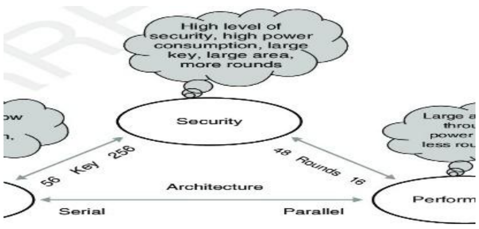
High level of security, high power consumption, large key, large area, more rounds
Security
Large a throu power less rou
56 Key 256
48 Rounds 16
Architecture
Serial
Parallel
Perform

Figure 5: Security Trade-offs

5. CONCLUSION

Security is a critical aspect of REST APIs, as they are widely used for cloud and web services. To ensure the security and reliability of REST APIs, we introduced four security rules that define desirable properties for these APIs. These rules help in maintaining the integrity of cloud services by preventing potential security vulnerabilities and ensuring the APIs function as expected.

To evaluate the security of REST APIs, we extended a stateful REST API fuzzer with active property checkers. These checkers automatically test APIs and detect violations of the four security rules. This approach helps identify security flaws by systematically injecting unexpected or malformed inputs into the API, revealing weaknesses that might otherwise go unnoticed.

We applied our fuzzing and security-checking techniques to nearly a dozen production-level cloud services, including Azure and Office-365. These services are widely used by businesses and individuals, making their security a top priority. Our approach allowed us to test these services rigorously, identifying vulnerabilities that could have led to potential exploits if left unaddressed.

During our fuzzing experiments, we successfully discovered multiple security issues in these services. On average, our fuzzing approach found a handful of new bugs in each service. The discovered bugs were classified into two main categories: approximately two-thirds were **"500 Internal Server Errors"**, indicating unexpected failures, while about one-third were rule violations detected by our new security checkers

Fixing security vulnerabilities proactively is crucial in preventing potential attacks. A live security incident—whether triggered by an intentional attacker or an accidental system failure—can lead to severe consequences, including data breaches, system outages, and financial losses. By addressing security bugs before they become active threats, cloud service providers can ensure a safer experience for their users.

One of the key advantages of our fuzzing approach is that the identified vulnerabilities are **easily reproducible**. This allows developers to quickly verify and fix the issues without confusion. Additionally, our fuzzing method is highly accurate and does not generate false alarms, ensuring that every reported bug is a genuine security concern that requires attention.

REFERENCES

[1] S. Allamaraju. RESTful Web Services Cookbook. O'Reilly, 2010.

[2] Amazon. AWS. https://aws.amazon.com/.

[3] APIFuzzer. https://github.com/KissPeter/APIFuzzer.

[4] AppSpider. https://www.rapid7.com/products/appspider.

[5] V. Atlidakis, P. Godefroid, and M. Polishchuk. RESTler: Stateful REST API Fuzzing. In 41st ACM/IEEE International Conference on Software Engineering (ICSE'2019), May 2019.

[6] BooFuzz. https://github.com/jtpereyda/boofuzz.

[7] Burp Suite. https://portswigger.net/burp.

[8] D. Drusinsky. The Temporal Rover and the ATG Rover. In Proceedings of the 2000 SPIN Workshop, volume 1885 of Lecture Notes in Computer Science, pages 323–330. Springer-Verlag, 2000.

[9] R. T. Fielding. Architectural Styles and the Design of Network-based Software Architectures. PhD Thesis, UC Irvine, 2000.

[10] P. Godefroid, M. Levin, and D. Molnar. Active Property Checking. In Proceedings of EMSOFT'2008 (8th Annual ACM & IEEE Conference on Embedded Software), pages 207–216, Atlanta, October 2008. ACM Press.

[11] K. Havelund and G. Rosu. Monitoring Java Programs with Java PathExplorer. In Proceedings of RV'2001 (First Workshop on Runtime Verification), volume 55 of Electronic Notes in Theoretical Computer Science, Paris, July 2001.

[12] R. Lammel and W. Schulte. Controllable Combinatorial Coverage in ¨ Grammar-Based Testing. In Proceedings of TestCom'2006, 2006.

[13] Microsoft. Azure. https://azure.microsoft.com/en-us/.

[14] Microsoft. Azure DNS Zone REST API. https://docs.microsoft.com/enus/rest/api/dns/zones/get.

[15] Microsoft. Microsoft Azure Swagger Specifications. https://github.com/ Azure/azure-rest-api-specs.

[16] Microsoft. Office. https://www.office.com/.

[17] S. Newman. Building Microservices. O'Reilly, 2015.

[18] OAuth. OAuth 2.0. https://oauth.net/.

[19] OWASP (Open Web Application Security Project). https://www.owasp. org

[20] Peach Fuzzer. http://www.peachfuzzer.com/.

[21] Qualys Web Application Scanning (WAS). https://www.qualys.com/ apps/web-app-scanning/.

[22] SPIKE Fuzzer. http://resources.infosecinstitute.com/fuzzer-automationwith-spike/.

[23] Sulley. https://github.com/OpenRCE/sulley.

[24] M. Sutton, A. Greene, and P. Amini. Fuzzing: Brute Force Vulnerability Discovery. Addison-Wesley, 2007.

[25] Swagger. https://swagger.io/.

[26] TnT-Fuzzer. https://github.com/Teebytes/TnT-Fuzzer.

[27] M. Utting, A. Pretschner, and B. Legeard. A Taxonomy of Model-Based Testing Approaches. Intl. Journal on Software Testing, Verification and Reliability, 22(5), 2012.

[28] M. Yannakakis and D. Lee. Testing Finite-State Machines. In Proceedings of the 23rd Annual ACM Symposium on the Theory of Computing.