Vulnerability Testing of RESTful APIs Against Application Layer DDoS Attacks

Sivakumar K, Santhi Thilagam P
Computer Science and Engineering, National Institute of Technology Karnataka, Surathkal, India 575025

***Abstract*—In recent years, modern mobile, web applications are shifting from monolithic application to microservice based application because of the issues such as scalability and ease of maintenance.These services are exposed to the clients through Application programming interface (API). APIs are built, integrated and deployed quickly.The very nature of APIs directly interact with backend server, the security is paramount important for CAP. Denial of service attacks are more serious attack which denies service to legitimate request. Rate limiting policies are used to stop the API DoS attacks. But by passing rate limit or flooding attack overload the backend server. Even sophisticated attack using http/2 multiplexing with multiple clients leads severe disruptions of service. This research shows that how sophisticated multi client attack on high workload end point leads to a dos attack.**

Keywords*—DDoS; rate-limiting; HTTP/1.1; HTTP/2; API; micro service; multiplexing; security; DoS; security testing

I. Introduction

The Application Programming Interface (API) acts as a software intermediary between modern mobile and web applications, providing a wide range of services shared across different platforms and consumers. APIs are built, integrated, and deployed quickly. API offers several advantages, including platform independence, scalability, flexibility, seamless integration, security, and cost-effectiveness. Because of the inherent advantages, APIs have emerged as a fundamental aspect of modern technology, enabling various applications and platforms to interact and share information. According to Akamai 83% of all internet traffic are API calls.

The API economy is a strategic approach where organizations utilize Application Programming Interfaces (APIs) to enhance accessibility to data and core capabilities, fostering innovation both within and outside the organization. By exposing APIs externally, businesses position themselves as platforms, inviting third-party innovation. This creates new avenues for market expansion, diverse monetization strategies, and the potential to seize opportunities not achievable through traditional methods. The API economy involves the controlled exchange of digital data and services through APIs, encompassing the value exchange between providers and consumers, both within and beyond a company. While an organization adopts an API-driven approach internally, the primary focus of the API economy is on business-to-consumer (B2C) and business-to-business (B2B) interactions. Prominent examples, such as Amazon Web Services (AWS), Twilio, Google Maps, and Stripe, illustrate the transformative impact of participating in the API economy, where companies build, consume, and expose APIs to accelerate development, enhance digital experiences, and capitalize on market opportunities.

APIs come in various styles based on their own characteristics and use cases. Those architecture styles are REST, GraphQl, gRPC, WebScokets, Webhooks, and SOAP. Among these REST is a widely adopted web service architectural style, that offers simplicity, scalability, adaptability, cache-ability, and security. REST uses HTTP methods(GET, POST, PUT, DELETE) to perform operations on resources, which are represented as URLs. Since it is stateless in nature, REST API facilitates easy resource addition and efficient traffic management. REST APIs are versatile, functioning across different platforms while supporting caching, security protocols, authentication, and authorization mechanisms, making it as a preferred choice for web service development.An API ecosystem that consists network of APIs that coexist and work together to provide a valuable and differentiated experience for customers. It uses tools, protocols, and standards to integrate and share data between software systems. API management ecosystems work to unite consumers and API providers to present a seamless experience to customers. Successful companies treat APIs as products and design, deliver, and manage it accordingly.

Robust API security is critical to protect sensitive user data from rogue cyberattacks. Unauthorized access attempts are frequent on APIs, and this has the potential to destroy the company's reputation as well as its finances. High-profile data breaches have highlighted the need for robust security measures. Hence, adequate security practices for APIs involves access control, monitoring of API activities, vulnerability testing, as well as covering security during the API development. API Gateways are often fully responsible for access control and rate limiting, but must cover vulnerabilities to avoid susceptibility to denial of service attacks resulting from misconfigured limits. API security is in the top 10 list for 2023, the main reason being also covered by Forbes [1]. As noted in an Imperva report [2], its annual estimate for global API-prompted cyber loss ranges between $41 and $75 billion. APIs are the number one attack vector, with conséquence on consumer privacy, public safety and intellectual property. Well known breaches include the current Twitter API breach [3] which exposed user personal data of as many as 200 million accounts, the Optus breach [4] which exposed the PII of 2.1 million ordinary Australians, and T-Mobile API data breach [5] affecting 37 million account holders. Apart from leaks, unsecured APIs poses risks to public safety, as described in the flaws found in the management system for Hyundai and Genesis cars [6], which allowed to take control without permission. Furthermore, there are API security weaknesses similar to the CircleCI breach [7], which facilitates stealing and exposing intellectual property.

Ignoring the security aspect, the API developers focus on design implementations and fast API deployment. Thatexposes a whole range of weaknesses that undermine the API. API security is of the utmost relevance due toits role in protecting sensitive data, safeguarding business reputation, ensuring regulatory compliance, allowing safe third-party integration, preventing DDoS attacks, ensuring data integrity, preventing monetary losses, managing authentication and authorization, defending against injection attacks, blocking phishing, and shielding intellectual property while facilitating safe DevOps. To bridge the digital divide these positives must be clouded with the right kind of API security measures that acts as a first defense barrier against unwanted access, breach, and downtime to maintain the data privacy, customer confidence, and operational continuity. Security testing helps with identifying weaknesses and vulnerabilities in the system allowing threats to be minimized and the system to continue operating unaffected by compromises. REST APIs are used by a lot of big companies so security testing of them is very important. But also, in the recent events, there have been denial of service attacks, bot/scraping, weaknesses, and authentication issues. Hackers uses such vulnerabilities to steal data, abuse accounts, or disrupt services. With the growing amount of internet traffic today and services that use APIs, it is important to protect against the OWASP top 10 [8] API security threats parameter through authentication and authorization like stealing a session by using APIs to research data and expose sensitive parts. Most of the papers in the literature focused on weak verification, data leakage, and validation attacks. However, what is missing is that resource exhaustion attacks by consuming the server resources that affects the availability of the services. This paper studies the application layer protocol security vulnerabilities and their impacts on the API server.

The contributions of this paper are as follows:

- Analyzing the OAS document, Discovering and identifying the target endpoints.
- Generating legitimate API requests based on attack types using single requests or multiple requests.
- Testing the API through the requests sent using either HTTP/1 or HTTP/2 protocol.
- Analyzing the results obtained from the experimental study.

The organization of the rest of the paper is as follows: Section II provides an overview of API testing and API vulnerabilities and attacks. Section III describes the problem which is addressed. Section IV describes the API security testing on the application layer. Section V specifies the attack methodology. Section VI describes the experimental setup and testing procedures. Section VII presents the findings, while Section VIII concludes this paper by highlighting the future research directions.

II. Related Work

A. RESTful API Testing

Most of the applications are not open source, the testing of Restful APIs is black box in nature. The bugs or errors generated from testing are either service unavailability or related to web security since the back end of API is similar to traditional web services. Many of the testing methods are trying to identify errors or bugs from the response status code 500.

1) General API testing methods: RESTTESTGEN [9] is a black box testing approach in which it reads the OpenAPI specification for identifying the operation dependencies among the parameter. It builds an Operation Dependency Graph(ODG) based on data dependencies between two operations. There it creates sequences of test cases to test APIs. It classifies based only on status code and does not incorporate a feedback mechanism. RESTLER [10] testing approach is a kind of bottom-up approach where it generates a test case for a single API and adds more API call sequences by trial and error by identifying resource dependencies between API endpoints. The limitation of this approach is that the search space for API testing is large since it doesn't have the knowledge of how APIs are connected.

Another approach MOREST [11] builds a Restful-service Property Graph(RPG) for single APIs, after each API testing, the graphs are dynamically updated. It is similar to RESTTESTGEN building graph based on resource dependencies but also has more details such as equivalence relation between schemas. Also, it incorporates an execution feedback mechanism to dynamically update the graph. Predicting the request parameter value or input parameter value for test case generation using the ML/DL model is another important aspect of testing where test case generation depends on the parameter value. MINER [12] uses a neural network model to predict the critical input or request parameter values. RESTest [13] is an open-source black-box testing framework for RESTful web APIs, addressing limitations of existing automated API testing tools that rely mainly on random fuzzing. RESTest enhances API testing by incorporating constraint-based testing, adaptive random testing, and fuzzing techniques, leveraging API specifications such as the OAS document.

Quickrest [14] finding faults by exposing misalignment between specification and implementation. It not only analyzes the response codes but also explores more properties of the response. It also tests the SUT(System Under Test) with agnostic input data and data that conformance to the parameter specification. The testing method [15] focuses on checking the robustness of the services, thereby identifying the bugs and security vulnerabilities. By giving unexpected or invalid input,it triggers a residual fault that is not detected during verification and validation. Commercial tools such as Postman [16], RESTAssured [17], ReadyAPI [18]and APIFortress [19] provide less automation since the test cases are written manually and then executed.The above-mentioned methods are black box techniques that are focused on parsing OpenAPI specifications, generating test case sequences, and predicting the input value for parameters. These testing strategies also look into the response or feedback on HTTP status code 500 but do not focus on security vulnerabilities.

2) Penetration testing methods: Simulated attacks are conducted to identify vulnerabilities in the System Under Test (SUT). These tests are conducted through human composition of test cases or executed automatically. Notable tools include ZAP (Zed Attack Proxy) and the Web Application Attack and Audit Framework (W3AF). These tools uncovers vulnerabilities, but only for specific API operations. Furthermore, it fails to identify dependencies, hence neglecting to recognize multi-API vulnerabilities in RESTful services.

NAUTILUS [20] incorporates annotations in the OpenAPI specification papers by recognizing the interdependencies of operations and parameters. Valid and modified payload sequences are generated as test cases. This work primarily addresses injection vulnerabilities, including SQL injection, XSS, and command injection, which are significant types of vulnerabilities resulting from inadequate management of user inputs. Nonetheless, it does not identify additional risks, including inadequate resource management, compromised access control, and absence of rate limiting.VoAPI [21] proposed vulnerability-targeted testing by identifying the API functions from the OpenAPI specifications that are vulnerable and conducting security testing on those functions. Instead of testing a large space of all API sequences, this method identifies the API interfaces which are having some keywords related to vulnerability. This reduces the time of indiscriminately traversing all API interfaces.

B. API Vulnerabilities and Attacks

Web Service Application Programming Interfaces (APIs) are essential to contemporary web development, facilitating smooth communication and integration across various software systems. The growing complexity and interconnectivity of these APIs provide considerable security threats, as it became targets for attackers aiming to exploit flaws and undermine the security of web applications. This literature survey seeks to examine the current research and methodologies pertaining to vulnerability detection.

The Open online Application Security Project (OWASP) published the 2017 top 10 critical security vulnerabilities for online apps, based on the contributions of over 40 application security organizations and an industry wide survey of more than 500 participants. This massive dataset contains vulnerabilities identified in various organizations, alongside over 100,000 real applications and APIs. The same goes for OWASP, which updated its TOP 10 threats in 2023 and then identified the latest risks and security issues in APIS so that developers and security professionals takes further steps to mitigate it.

1) Broken object level authorization: Broken object-level permission issue—this is when the API does not properly restrict object level actions based on user rights. This situation allows users to modify any API object regardless of rightful permissions [22]. As shown in the work of [23], malicious actors leverages this vulnerability to recover sensitive information and perform illicit operations. This vulnerability is due to insufficient methods to control access as well as poor tests of these controls. Malicious actors exploits this vulnerability by tampering requests for accessing unauthorized resources [24]. An attacker exploits a request to gain access to another user's data or escalate their privileges to perform actions not within their designated access level. This vulnerability was demonstrated, for example, in the Facebook Cambridge Analytica affair. A breach of Facebook's application-programming interface (API) was exploited by a third-party, providing the party without approval,access to and ability to extract user data. As a result, Facebook faced a major data breach and a huge hit on its brand [25].

2) Broken authentication: As per the research of [26], broken authentication is a vulnerability when an API does not adequately authenticate users, and attackers gain access to the system without valid credentials. Brute force attacks, session hijacking, and credential stuffing are some of the ways this vulnerability are exploited. The failure to utilize complex passwords that aren't easily deduced by potential attackers leaves accounts vulnerable, as failure to implement multi-factor authentication or secure session management. In general, attackers then exploits this vulnerability by stealing user credentials and using these credentials to access the system [27]. The Equifax data incident is a concrete example of this vulnerability. In this case, attackers exploited a vulnerability in Equifax's API to gain access to sensitive consumer data. Over 143 million people found that their personal data had been stolen from this breach, causing millions of dollars in damage and a loss of trust in Equifax [28].

3) Excessive data exposure: Researchers [29] illustrate that excessive data exposure becomes a vulnerability when an API discloses more data than necessary, encompassing sensitive information or user credentials. Malicious actors exploits this vulnerability to obtain unauthorized access to sensitive information or execute operations without appropriate authorization.The causes contributing to this vulnerability include insufficient data sanitization and validation, poor implementation of access controls, and the use of unsecured data storage. Attackers exploit this vulnerability by dispatching precisely formulated queries to obtain sensitive data, as detailed by [30].

4) Lack of resources and rate limiting: The inadequacy of resources and lack of rate limiting represent a vulnerability that arises when an API fails to sufficiently limit the number of permissible requests, thereby allowing attackers to overwhelm the system with requests and launch denial-of-service attacks.The causes contributing to this vulnerability include inadequate rate restriction, the use of susceptible or easily predictable API keys, and insufficient monitoring of anomalous traffic patterns. Attackers exploit this vulnerability by sending a large number of queries to the API, which overwhelms the system and causes it to become unresponsive, as noted by [31].

5) Broken function level authorization: The work by [22] demonstrates that the broken function level authorization vulnerability occurs when an API fails to limit access to certain functions or operations according to user roles or permissions. The author in [32] demonstrated that this vulnerability exposes the potential for attackers to execute unauthorized actions within the system, such as manipulating or erasing sensitive data. This vulnerability, generally resulting from insufficient access control implementation, typically occurs due to the inability to validate user permissions prior to allowing action execution.

6) Mass assignment: The study conducted by [33] touches upon the concept of mass assignment vulnerability. The vulnerability happens when a user modifies multiple properties of an object with one requestby the API. If the attackers exploit this vulnerability, it changes the sensitive information or gain unauthorized access. According to [34] this type of vulnerability is mainly triggered due to lack of user input validation or lack of proper access control mechanisms. Various approaches have been suggested to avoid mass assignment vulnerabilities. The work [35] proposed a rule-based solution

TABLE I. SUMMARY OF RELATED WORKS

Reference	Testing Approach	Payload Generation	Description	Limitations
Viglianisi et al. [9]	Model Based	Data Observed in Previous Response And Malformed Inputs	Builds Operation dependence graph and generate test sequence based on the graph	No feedback mechanism to update the graph and checks only HTTP 500 status code
Liu et al. [11]	Model Based	Last Successful Response Value, Example and Random	Building Restful service graph based resource dependencies and equivalence resource schema, test case generation using the graph and feedback mechanism to update the graph	Identifies only bugs/errors not identifying vulnerabilities
Stefan Karlsson et al. [14]	Model Based	Custom Input Generators	Property based on OpenAPI documents and responses from the request, finding faults or bugs by analyzing misalignment between specification and implementation	No security vulnerability detection capabilities.
Atlidakis et al. [10]	Fuzzing	Input Data Dictionary	Based on producer-consumer dependencies and dynamic response feedback mechanism	Testing space is large
Martin-Lopez et al.[13]	Fuzzing and Constraint Based	Test Data Generators	Testing using constrain based and fuzzing input generation , online testing and offline testing	Only 5xx and 4xx errors, not enough for testing complex API
lyu et al. [12]	Fuzzing	Dictionary and Previous Value	Deep learning model predicting the input or parameter values	No security testing
Laranjeiro et al. [15]	Fuzzing	Valid and Malicious Inputs	Testing the robustness of service by giving valid, boundary values and malicious inputs, detection of user input related vulnerabilities such as SQL injection, XSS	Other than injection vulnerabilities, OWASP TOP 10 API vulnerabilities not verified
Deng et al. [20]	Penetration Testing	Data Observed in Previous Response, Example, Mutated and Random	Identifying the dependencies of operation and parameter, valid and mutated payload sequences of test cases are generated	It fails to find other vulnerabilities such as improper resource management, broken access control, and lack of rate limiting
Du et al. [21]	Penetration Testing	Previous Response, Example, Random	Identifying the API functions from the OpenAPI specifications that are vulnerable and conducting security testing on those functions	Detection and verification vulnerability is limited, supports only OpenAPI formats.

to identify and mitigate mass assignment vulnerabilities in RESTful APIs. This is done by defining rules about what characteristics of object types are modified by which user roles or permission. When it gets a request, the system checks the rights of the user and apply rules that are needed, if the user is allowed to change the properties. Attacks involving Mass Assignment generally consist of attackers sending adjusted requests with extra parameters, or changing the values of supplied parameters. For example, an assailant uses a user's account information and make a request with the boolean field on; if the API does not validate this parameter, the assailant is having admin rights.

A security misconfiguration is a type of vulnerability in which API is implemented with insecure settings like default passwords, extra functionality enabled. The attackers used this vulnerability to gain unauthorized access to the system or to perform malicious acts [36]. Such vulnerability is mainly due to the lack of configuration management techniques, such as not disabling unnecessary features, enabling unnecessary services, and using default passwords [37]. Security misconfiguration is described by [38] as a scenario wherein an API exposes certain resources or functionalities to everyone as a value. This is due to weak access control settings or incorrect API authentication methods configured by the developers. This vulnerability is exploited by attackers to acquire sensitive data or do actions on behalf of another user.

7) Injection vulnerability: Injection vulnerabilities happen when an attacker attempts to insert code into an API, including SQL or code injections, and incorporates these itself as such in the research of [39]. This flaw allows attackers to run random code throughout the system and get access to confidential data. This vulnerability often results from inadequate input-validation or missing access-control measures. Many researchers have come up with various approaches to mitigate injection vulnerabilities. For example, a technique was proposed in, that leveraged both static and dynamic analyzes to identify injection vulnerabilities in RESTful APIs. It consists of the static analysis of the APIs source code to find injectable points and dynamic analysis techniques to assess the APIs behavior based on different conditions.

C. Rate Limiting

API gateways are a type of tool that help organizations manage,and aggregate their APIs, addressing key components like access control, rate limiting, and IP block lists. Because it is reactive, developers must register the APIs that are managed manually. API gateways are usually deployed inside of your organizations infrastructure, departmental level, and in cloud env. For web services, a commonplace functionality, that is provided by API Gateways is rate limiting. Its used to limit how many times the client makes a request to an API within a specific time to avoid overloading the system and fair use among users. Here, rate limiting is done to prevent no of traffic or no of request which lead to server overload or downtime or degraded performance. Rate limiting prevents abuse of the API by controlling the number of requests that requests are received by the server in a specified timeframe, thereby ensuring that resources are fairly allocated among users. Various types of rate limiting configuration are implemented including rate of requests (per second, minute, hour) as well as user/client-based rate limits on API Gateways. In addition, some API Gateways enable you to define rate-limiting rules pr. API endpoint, which is helpful in situations where different endpoints have different usage patterns or needs. APIs without rate limiting are vulnerable to Denial-of-Service (DoS) or brute-force attacks on the API, causing extensive damage to the platform. In API Gateways, Rate limiting is implemented to protect such attacks and protect the underlying system from abuse or misuse.

There are several types of rate limiting mechanism available, including:

- IP-based rate limiting: This form of rate limiting confines the quantity of requests originating from a specific IP address within a designated time interval. This is effective in deterring abusive conduct from an individual user or a collective of users utilizing the same IP address.
- User-centric rate restriction: This form of rate limiting confines the quantity of requests submitted by an individual user within a certain time interval. This is effective in mitigating abusive conduct from users who submits several requests.
- Token-based rate restriction: This form of rate limiting confines the quantity of requests executed with a certain access token or API key inside a designated time period. This is effective in mitigating API misuse by a certain client.
- Request-based rate restriction: This form of rate limiting constrains the quantity of requests directed to a specific API endpoint during a designated time period. This is effective in mitigating abusive conduct that are directed against a specific endpoint.

While API gateways provide rate-limiting features to protect APIs from abusive behavior, some aspects of API usage are not fully captured by rate limiting alone. Some examples include:

- Malicious intent: Rate limiting is not be sufficient to protect against malicious intent, such as targeted attacks aimed at causing denial-of-service or brute-force attacks to guess authentication credentials. Additional security measures, such as authentication and access control, are needed to prevent such attacks.
- Complex use cases: Some API use cases involve complex workflows that involves multiple API calls within a short period of time. Rate limiting mechanism is able to distinguish between legitimate and abusive behavior in such cases, leading to false positives or false negatives.
- Traffic spikes: Rate limiting is typically designed to handle steady-state traffic patterns. It is not effective in handling sudden spikes in traffic, such as those caused by events like product launches or marketing campaigns.
- Geolocation: Rate limiting based solely on IP addresses are effective in preventing abusive behavior from users who are using VPNs or other proxies to hide their location. Because rate-limiting features of API gateways are not fully capture certain aspects of API usage, such as malicious intent, complex use cases, traffic spikes, and geolocation, the API gateways are vulnerable to attacks that exploit these limitations.

D. Rate Limit Vulnerabilities

Rate limiting is a security mechanism that restricts the number of requests made to an API or web application within a certain timeframe. There are several rate-limiting algorithms, each with its own advantages and limitations.

1) Token bucket algorithm: This algorithm allows a fixed number of tokens to be used within a fixed time interval. Tokens are generated at a constant rate and are stored in a ”bucket”. When a request is made, a token is removed from the bucket and the request is processed. If there are no tokens left in the bucket, the request is denied until more tokens are generated. This algorithm is simple and efficient but the challenge is to tune correctly for varying traffic patterns.

2) Leaky bucket algorithm: This algorithm works by collecting requests into a bucket at a constant rate, with excess requests overflowing from the bucket and being discarded. Requests are processed at a constant rate, and the bucket empties over time. This algorithm handles bursts of traffic but it is inefficient when dealing with smaller requests.

3) Fixed window algorithm: This algorithm allows a fixed number of requests to be made within a fixed time interval. If a client exceeds this limit, all requests are denied until the next time interval begins. This algorithm is simple and efficient but leads to bursts of traffic at the start of each time interval.

4) Sliding window algorithm: This algorithm is similar to the fixed window algorithm, but instead of a fixed time interval, the time window slides over time. This allows for a more even distribution of requests and is more responsive to changes in traffic patterns. However, it is more complex to implement and it leads to uneven traffic distribution if the sliding window is not appropriately sized.

However, rate-limiting mechanisms are vulnerable to attacks and when it is bypassed or circumvented , allows an attacker to send the requests exceeding the threshold and perform unauthorized actions.

The following are some common techniques that attackers use to bypass rate limits:

- Using null chars: Attackers uses null characters (%00, %0d%0a, %09, %0C, %20, %0) to bypass rate limits.

For example, appending a null character to an email address allows an attacker to continue brute-forcing.

- Adding spaces: Attackers add spaces to usernames or email addresses to bypass rate limits. Some web servers strip off extra spaces, allowing an attacker to continue brute-forcing by appending a space each time the attackers are blocked.
- Host header injection: Attackers modifies the Host header of the request to confuse the server after being blocked. Changing the Host to a different domain or IP address confuses the server, allowing an attacker to bypass the rate limit.
- Changing cookies: Attackers change the session cookie after being blocked by the server. By figuring out which request sets the session cookie, an attacker updates the session cookie each time an attackers are blocked.
- X-forwarded-for: Attackers changes the X-forwarded-For header to confuse the server or load balancer after being blocked. This technique allows an attacker to bypass the rate limit by forwarding the request to another host.
- Confuse server with correct attempts: Attackers confuses the server by performing just under the maximum number of attempts before using the correct credentials to log in. This technique allows an attacker to bypass the rate limit by appearing to be a legitimate user.
- Updating target paths: Appending a random parameter value to the target path sometimes allows an attacker to bypass the rate limit on the endpoint. This technique involves brute-forcing a target path until the attacker is blocked, then appending a new parameter value and repeating the process.
- IP-Based rate limits: Attackers bypasses IP-based rate limits by changing their IP address or using an IP-rotate Burp extension.

There has been significant research on how to effectively test REST APIs, with various methods and tools proposed in the literature as given in the Table I. One approach involves using a combination of manual and automated testing techniques. Automated testing methods typically include using API testing frameworks, such as JUnit, Postman, or SoapUI, to test various API endpoints and validate their responses.In addition, researchers have proposed various methods for generating test cases and test data for REST APIs. One such approach is the use of combinatorial testing, where a set of test cases is generated by combining different input parameters and values systematically. Another approach is the use of model-based testing, where a formal model of the API is used to automatically generate test cases based on different input and output scenarios.Despite these advances, there are several limitations to existing REST API testing methods. One major limitation is the lack of standardization and guidelines for testing REST APIs, which leads to inconsistencies and variations in testing approaches. Another limitation is the difficulty of testing complex APIs with multiple endpoints and dependencies, which makes it a challenging to validate all possible combinations of input and output scenarios. In addition, automated testing approaches are not able to catch all possible errors or bugs, as it relies on predefined test cases and miss edge cases or unexpected scenarios. Finally, the lack of effective monitoring and reporting mechanisms makes it difficult to track and analyze API performance and identify potential issues in real-time.

E. Research Gaps

In existing works, very few research works have considered the impact of resource exhaustion attacks in RESTful APIs, particularly in the context of application-layer DDoS attacks, but are not adequately covered in existing studies. While vulnerabilities such as weak authentication, data leakage, and injection attacks are well-covered, the way attackers exploit API endpoints to flood server resources and bringservices down, remains poorly understood. Half-baked research includes advanced tactics like HTTP/2 multiplexing, which allows attackers to send multiple high-load requests over a single TCP connection, and rate-limiting attack strategies. Most current threat detection systems are insufficient in their ability to provide focus on high-workload endpoints or simulate multi-client attacks, leaving a gaping knowledge gap on how orchestrated attacks could deplete server resources. Furthermore, existing tools cannot fully assess resource exhaustion, highlighting the need for more sophisticated solutions to manage these complex attack vectors. The current work endeavours to address this gap, investigating the exploitation of application-layer protocols (specifically HTTP/1.1 and HTTP/2) to trigger resource depletion attacks, thus providing guidance on mitigating such weaknesses in RESTful APIs.

III. Problem Description

More and more web applications are accessed through mobile, web, or devices, and cybersecurity is paramount, with relentless hackers targeting organizations daily. As the industry shifts towards microservices architecture from monolithic, the need for cutting-edge cyber threat detection remains crucial. Recent times have seen the emergence of Application Layer Distributed Denial of Service (DDoS) attacks, focusing on fundamental aspects like CPU, memory, cache, disk, and network within microservices which is called as resource exhaustion attacks. Yet, modern application complexity introduces intriguing attack vectors, as illustrated in scenarios where microservices interact through API Gateways. Simultaneously, implementing rate limiting in API Gateways is essential for shielding backends from traffic surges, but it must be done carefully to prevent overloading. By sending a low volume of requests which are asymmetric workload requests, exhausting the resources of the server. This study's objective is to perform a vulnerability testing on microservices through REST API requests, in the presence of Rate limiting in API Gateway.

IV. Security Testing of API based on Application Layer Protocol

When requesting to access a service through an API, a client application sends a request to the Origin server routed through an API Gateway that includes information such as the requested resource and any necessary parameters. The

Origin server then processes the request, which involves authentication, data retrieval or modification, and other tasks, it internally calls some external APIs before sending a response message back to the client. The specifics of how an API requests to a server varies, but the basic idea of sending a request and receiving a response remains the same. HTTP/2 multiplexing aims to minimize the overhead of requesting and receiving resources by serving it over various streams. However, multiplexing has introduced some security concerns. It eliminates the need for a large number of bots to launch attacks since it enables multiple requests to pass through a single TCP connection at the same time. Furthermore, there are no restrictions on the types of requests that are multiplexed together, allowing attackers to bundle multiple API Requests into a single connection and force the server to process it concurrently. This results in a denial of service (DoS) scenario if computationally expensive requests are combined to form an attack payload, rather than random base requests [40]. Although rate limiting is a commonly implemented measure to prevent DDoS attacks, it is not foolproof and has vulnerabilities that attackers exploit to bypass the threshold limit set. These weaknesses enables attackers to launch successful attacks, despite rate limiting being in place.

Little's law is a theorem in queuing theory, which provides a relationship between the average number of customers in a queue (L), the arrival rate of customers (λ), and the average time a customer spends in the system (W). The formula for Little's Law is typically expressed as:

$$L = \lambda * W \tag{1}$$

This is applied to API management infrastructure not directly but the principles are applied here to optimize the performance and capacity. The Eq. 1 is rewritten as

$$N = X * R \tag{2}$$

Where N is throughput, X is Request Per Second(RPS) and R is average response time.

For example, if the origin server throughput is 7(i,e.N) and the response time is 1 ms(R), then the request per second is 7(X). This shows that when the response time is within the normal time limit, the origin server provides the maximum throughput. However, in a large distributed system, this is not happening in real time, and these requests have to spend more time in places such as memory, CPU cores, queues, cluster interfaces, connection pools, disk space, and thread pools because of topology changes, network failures, high-workload requests, request dependencies, race conditions, and synchronization issues. When high-workload requests are sent to the server, the CPU takes more time to execute affecting the latency. The absence of a rate limit is even worse when multiple high-workload requests are sent to the server. So Rate limit is a better mechanism to overcome this situation. However, this mechanism is bypassed using various mechanisms. One of the methods is using the HTTP/2 multiplexing feature to send multiple high-workload requests to the server. Multiple requests combined in the form streams in a single request. API gateway which enforces a rate limit is not be able to differentiate it. This makes more number of requests going to the endpoints. This increases the load of the endpoint and that leads to the unavailability of services to the legitimate clients.Therefore, when the response time or latency increases for the above reasons, the requests per second decreases. This implies that the number of requests for the origin server process decreases. When more and more requests are queued this increases the latency and, eventually lead to the failure of the server. So even though the rate limiting is implemented in the API Gateway, that is not going to be the cause of the server failure. In this paper, it shows that high-workload API requests and dependency requests take more time to process, thereby increasing latency, subsequently affecting the throughput and leading to server failure [40].

A. Symmetric Attack

Introducing a security threat referred to as the "symmetric single-client attack". In this scenario, multiple identical attack requests are generated by the attacker, including the use of the same URL and parameters, primarily through the transmission of POST requests. These requests are executed within a single TCP packet. This type of attack poses significant risks, especially in scenarios where no rate limit is enforced on the services.Each of these requests demands significant computational resources to generate a response. Consequently, the absence of a rate limit on the server, combined with multiple symmetric high-workload requests, has the potential to overwhelm a server with just a few attacking systems. This becomes especially detrimental when targeting a single high-workload endpoint. Moreover, the threat intensifies when the HTTP/2 protocol is employed, as the attacker leverages its multiplexing feature to further strain the computational resources of the server when executing attacks on services lacking rate limits.The proposed attack is known as the "symmetric multiprocessor attack". In this attack, the aggressor concurrently creates multiple clients, each of which carries numerous similar attack requests that are permissible by the server within a single TCP connection. These requests were launched simultaneously, exerting a substantial workload on the server. Even in the presence of rate-limiting measures, this attack has the potential to disrupt server operation, particularly when a large number of processes run in parallel. Similar to a single-client symmetric attack, this threat is even more pronounced in the presence of the HTTP/2 protocol, both at the server and application levels.

B. Asymmetric Attack

In the "asymmetric single-client attack", the attacker initiates numerous unique attack requests, employing various URLs or parameters, all within the server's defined limits for a single TCP connection. These requests are executed sequentially. Much like symmetric single-process requests, this attack places substantial computational demands on generating responses. Furthermore, it presents a challenge when the server enforces a rate limit, as each request is distinct and stays within the defined traffic limit. In this 'asymmetric multi-client attack', multiple clients simultaneously send requests, each of which carries distinct attack requests. These requests adhere to the server's allowable limits for a single TCP connection and are executed concurrently. Similar to symmetric multi-client requests, this attack places significant demands on the computational resources for response generation. Additionally, it presents a challenge even when a rate limit is enforced at the server because each request is unique and does not exceed the specified traffic limit.

V. Attack Methodology

Prior to launching a symmetric or asymmetric attack on APIs for vulnerability Testing of REST API web services against application layer DDoS, certain prerequisites must be arranged. This testing has been done to find the vulnerability in rate limiting and the features of HTTP/2. The steps involved in this process are outlined below. OAS document which contains the API endpoints information such as operations and parameters. Also end points are identified from other sources such as client code which is basically Javascript code. Using the web scrapper go through the each every link and discover the end points which are not described in the OAS document. Similarly through reverse engineering the mobile application code the hidden endpoints are identified. Using all these as input for this algorithm which greatly added source to add more details about the endpoint and other meta data details which is helpful for generating requests as well as analyzing the endpoints which are heavily loaded or not.

A. Discovering Endpoints

To discover all the API endpoints of a web application, employ a comprehensive approach blending manual exploration and automated tools as given in Algorithm 1.

Algorithm 1 Discovering Endpoints

```
1: Input: OAS document, Client Code, Mobile App,
2: Output: Endpoint List
3: while Traverse Endpoints do
4:     Update the Endpoint list with operation and parameter values
5: end while
6: while Traverse Client-side code do
7:     Extract Links in the document
8:     while Traverse All Links do
9:         Update the Endpoint list with operation and parameter values
10:    end while
11: end while
12: Extract Endpoints from APK file using Diggy tool
13: Update the Endpoint list with operation and parameter values
14: Return: List of Endpoints
```

B. Identifying Target Endpoint

To pinpoint the crucial endpoints required for the optimal operation of the application or business, employing methods such as monitoring response times, identifying key business functions, and assessing error rates.The input parameters such as endpoint list with operations and parameter values is of much important to analyze the endpoints to identify the weak endpoint or heavily loaded endpoints in which any requests ends up with this endpoint as discussed in the Algorithm 18. After sending the requests to the API server, based on the responses and response code it is analyzed to identify the requests which is having more latency. These requests are maintained in the list for further use.

Algorithm 2 Identifying Target Endpoints

```
1: Input: Endpoint List with List of Operations and Parameter Values.
2: Output: Endpoint List High Response Time
3: while Traverse Endpoints do
4:     while All Operations done do
5:         Sends the request to the server with valid inputs
6:         Analyze the Status code
7:         if Response Code is 200 then
8:             Updates the response time for the current request with parameter value in the Response list
9:         else
10:            Add the Endpoint to the Error list with the count.
11:        end if
12:    end while
13: end while
14: while Traversing Response list, Error list do
15:     Update the Higher Response time Endpoints to Target list
16:     Update the Target list by selecting more error-prone Endpoints
17: end while
18: Return: List of Target Endpoints with Operations and Parameter values
```

One approach is to monitor the response time of microservices. High-workload endpoints typically have slower response times due to the high volume of requests as it receives. Monitoring tools like New Relic or AppDynamics to track response times and identify any endpoints that are taking longer than usual to respond. High workload endpoints are also identified by analyzing error rates. Endpoints that are experiencing high workload are prone to more errors, and status codes 503,520,509 and 429 are going to be analyzed. Look for endpoints that are critical to the business functions of microservices. These endpoints are more prone to high workloads due to their functionality.Using the abovementioned techniques, it is possible to detect certain high-traffic endpoints are selected as targets for launching an attack aimed at overwhelming the systems with an excessive number of requests.

C. Attack Approaches

After the target API endpoints are identified, then the next step is to select an attack vector to perform the attack. Analysis of the endpoint operations and parameter values is also very crucial to generating the attack request so the request is a high workload. Different kinds of possible attack scenarios are as follows: The high workload request of the target endpoint is taken and sent multiple times to the server. This is done with HTTP/1.1 or HTTP/2 requests. The request that has the highest computation workload is "w_x" where "x" is the request that took the highest computation power to respond. The operation of the type request must be unique since all requests are intended to perform the same operation (for example POST operation). These requests are sent in flooding or multiplexing mode. In an asymmetric attack, different types of multiple requests are sent to the target API endpoint. These multiple requests are sent to the server as flooding of requests or multiplexing mode. Here operation to be selected to perform the attack are combination of different operations such as GET/POST/PUT/DELETE.After selecting the type of attack the attacker goes with either single client or multi-client.

- Single client: When the attacker wants to attack by sending more requests one after another from one client then the attacker goes with a single client attack.
- Multi-client: When the attacker wants to attack by sending multiple requests from different clients.

When the server and application support the HTTP/2 protocol then the attacker utilizes the features of the HTTP/2 protocol to attack the target endpoint. One of the features chosen as attack vectors is:

- Multiplexing: HTTP/2's multiplexing feature enables attackers to send multiple requests as a single request with help streams by which a single TCP connection is only required to be sent, resulting in the server receiving and executing these requests nearly simultaneously.

D. Test Case for Testing Symmetric Attack with Single/Multiple Clients

In this testing scenario, the objective was to identify the endpoint that imposes the most significant computational load. Subsequently, a symmetric attack is initiated by dispatching multiple requests of the same type of operation to the identified endpoint, utilizing either a single client or multiple clients as given in the Algorithm 3. Throughout the attack, the application's response times, CPU usage, and error rates were continuously monitored. The primary aim is to validate the resilience of the application, ensuring that it withstands an attack without a substantial surge in error rates or system crashes. Additionally, it is vital to ascertain that the application is efficiently handling the heightened workload without adversely affecting the performance of the other endpoints within the system. The anticipated results involve the successful identification of the high-load endpoint, subjecting it to an attack without critical failures, and ensuring minimal disruption to the overall performance of the other endpoints.The parameter API request describes the request list which is used to generate the attack request either symmetrically or asymmetrically.The execution count parameter specifies the number of times the requests to be generated and sent to the server.Client count describes the number of clients used in this study, High workload request, weak endpoint, more number of requests sent and finally client count are greatly influencing the outcome of the results in which CPU usage is varying from different levels.

Algorithm 3 Symmetric Attack

```
1: Input: API Request, ExecutionCount, ClientCount
2: Output: CPU Load
3: Select the Endpoint from the Target Endpoint List created
   based on Algorithm 2
4: Construct the API Request using CURL or Shell Script
   consisting of Endpoint, Operation, and Query Parameter.
5: if Single Client then
6:     while ExecutionCount not NULL do
7:         Run the attack script
8:         Update the CPU usage
9:     end while
10: elseMultiple Client
11:     while ExecutionCount not NULL do
12:         For Each Client do
13:         Run the attack script
14:         Update the CPU usage
15:     end while
16: end if
17: Return: Final result
```

1) Test case for testing asymmetric attack with single/multiple clients: The objective is to launch an symmetric attack using multiple clients based on the Algorithm 4 shown, sending numerous requests to the identified rate-limited endpoint, with variations in the request headers or body content aimed at potentially circumventing these rate limits. Throughout the attack, there is continuous monitoring of the application response times, CPU utilization, and error rates. The primary objective is to determine whether the application is effectively handling an attack or whether it experiences a notable increase in error rates. Additionally, it is essential to establish whether the application sustains a heightened workload without adversely impacting the performance of other endpoints within the system. The expected outcomes encompass the successful identification of the rate-limited endpoint, execution of an attack challenging rate limit, potential response time delays, and an evaluation of the application's resilience in this testing scenario, including its impact on other endpoints.

Algorithm 4 Symmetric Attack

```
1: Input: API Requests, ExecutionCount, ClientCount
2: Output: CPU Load
3: Select an Endpoint from the Target Endpoint List and
   Select a set of workload API Requests from the list based
   on Algorithm 2
4: Construct the API Request using CURL or Shell Script
   consisting of Endpoint, Operation, and Query Parameter.
5: if Single Client then
6:     while ExecutionCount not NULL do
7:         Run the attack script
8:         Update the CPU usage
9:     end while
10: elseMultiple Client
11:     while ExecutionCount not NULL do
12:         For Each Client do
13:         Run the attack script
14:         Update the CPU usage
15:     end while
16: end if
17: Return: Final result
```

2) Test case for testing HTTP/2 multiplexed attack: An HTTP/2 multiplexed attack is initiated where multiple requests are transmitted to the identified endpoint within a single

TCP connection. Throughout this attack, there is continuous monitoring of the application response times, CPU utilization, and error rates. The primary objective was to validate whether the application effectively withstand an attack without encountering a significant increase in error rates or experiencing crashes. Furthermore, it is essential to evaluate whether the application manages the increased workload without adversely affecting the performance of the other endpoints within the system. The anticipated outcomes involve the successful identification of the HTTP/2-compatible endpoint, execution of the multiplexed attack to assess the application's resilience, potential response time deceleration, and an assessment of the attack's influence on the performance of other endpoints. The following Algorithm 5 describes the steps for this testing.

Algorithm 5 Multiplexed Attack

```
1: Input: API Requests, ExecutionCount, ClientCount
2: Output: CPU Load
3: Select an Endpoint from the Target Endpoint List and
   Select a set of workload API Requests from the list based
   on Algorithm 2
4: Construct the API Request using CURL or Shell Script
   consisting of Endpoint, Operation, and Query Parameter.
5: Send Multiple Requests as chunks and send as different
   streams with stream identifiers.
6: if Single Client then
7:     while ExecutionCount not NULL do
8:         Run the attack script
9:         Update the CPU usage
10:    end while
11: elseMultiple Client
12:    while ExecutionCount not NULL do
13:        For Each Client do
14:        Run the attack script
15:        Update the CPU usage
16:    end while
17: end if
18: Return: Final result
```

VI. Implementation Details

A. Application Analysis

To do attacks, the application should be micro-service-based. For this, the chosen application is the SockShop Application which is a micro-service-based demo application. The architecture of the application in Fig. 1 is as follows. The application comprises eight microservices, each with distinct responsibilities. The front-end microservice is responsible for user interface interactions, presenting information, and gathering user input. The User microservice manages user accounts, including authentication and authorization, and handles user-related data and access controls. Catalog oversees product information and catalog data, offering insights into available products. Carts are responsible for shopping cart management, enabling users to add, modify, and oversee items in their carts during catalog browsing. The payment handles payment processing, transaction management, and user payments. Shipping focuses on order fulfillment and logistics, including tracking and delivery. Order oversees the entire order lifecycle, from order recording to processing, and inter-micro-service communication coordination. Lastly, the Queue-Master likely manages message queues and orchestrates background tasks and event-driven processes across microservices.The application also contains data services comprising Users-DB, Carts-DB, Catalogue-DB, Orders-DB, and Shipping-DB. These data services are responsible for storing and managing user information, cart contents, product catalog data, order details, and shipping-related information within the application. Also, it plays a crucial role in ensuring the application functions smoothly by providing the necessary data to the micro-services when needed.

B. Experimental Test Setup

The test bed setup, as depicted in Fig. 2, involves the processing of requests through a series of systems. Initially, the requests encounter an Apache server, which is HTTP/2 enabled and serves as a reverse proxy. This server redirects the requests to a Tomcat embedded server, responsible for spring boot applications. A spring boot client serves as an API gateway and implements rate limiting for the business applications APIs. The rate limit policy is implemented based on three different aspects, namely, X-API-KEY, IP, and USER. For X-API-KEY, requests starting with "$AX001$" have a limit of 100 requests per minute, while those starting with "$BX001$" have a limit of 70 requests per minute. Requests starting with other characters have a limit of 50 requests per minute. The rate limit policy based on IP/USER allows a limit of 100 requests per minute for each unique IP/USER. This rate-limiting approach is inspired by the official Twitter API rate limit documentation. Valid requests that fall within the rate limit are processed further by the spring boot services, which contain the business logic.

In the proposed setup, the target is a Spring Boot framework-based application, specifically a Microservice Application hosting APIs on an Eclipse-embedded Tomcat server. This application runs on a Lenovo ThinkCentre M910t system, equipped with an Intel® Core™ i7-7700 CPU operating at 3.60GHz × 8, and it runs the Ubuntu 21.10 operating system. The objective is to subject these applications to attacks designed to overload CPU performance while considering the rate limits and usage of the HTTP/2 protocol. The setup was designed to accommodate both the HTTP/1.1 and HTTP/2 protocols, and rate limiting was enforced through the API Gateway. Additionally, a performance comparison was performed between HTTP/1.1 and HTTP/2 by attacks carried out with varying numbers of requests.

C. Attack Tools

Tools like Net-Hunter for API fuzzing with Wordlists and the ZAP scanner for endpoint identification were employed to identify all the endpoints. A Python, Shell, or Go script was crafted to evade rate limits by dynamically altering request headers (specifically, X-API-KEY in our scenario) for each new request, with the intention of overwhelming the target server. The decision on whether to use symmetric or asymmetric attack requests was made based on specific requirements. These attack requests were executed both individually and concurrently, utilizing multiprocessing on a Linux platform. Additionally, the combination of Burp Suite and Curl facilitated the sequential execution of request attacks.

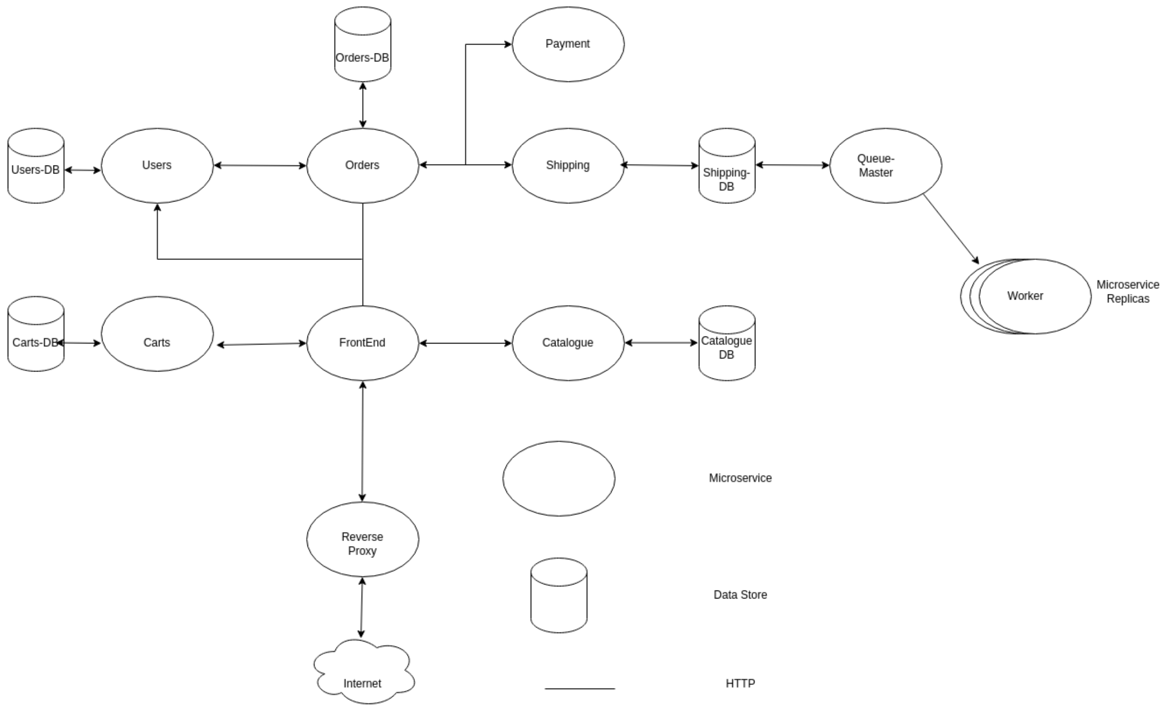


Fig. 1. SockShop application architecture.

D. Launching the Attack

The fourth and final step in an attack involves initiating the actual attack. This is done using any HTTP request generation tool that is compatible with HTTP/2.In this testing procedure, the process initiates with configuring an HTTP request generation tool, specifying the use of the HTTP/2 protocol, and setting it up to send requests to the designated target endpoint. The choice between a symmetric or asymmetric attack is made, and in cases involving multi-client approaches, the tool is configured to generate multi-client requests and establish multiple TCP connections. The attack is set in motion by instructing the tool to dispatch requests to the target endpoint, and simultaneous monitoring of the target endpoint's response times, CPU utilization, and error rates begins. Success is validated by observing a significant surge in response times and/or error rates, indicating the attack's effectiveness in overwhelming the target endpoint. Anticipated outcomes involve the tool's ability to execute the attack as intended, alongside notable spikes in response times, CPU usage, and error rates exhibited by the target endpoint, confirming the successful execution of the attack, resulting in the target endpoint's unresponsiveness or error generation.

1) Attack using HTTPX library: Python script was written to launch Symmetric and Asymmetric attacks using the HTTPX library and by changing the headers of the request to bypass the rate limit for HTTP/1.1 to test the CPU utilization of the servers and this attack brings down the targeted server with this attack.

2) Attack using CURL: The shell script was written to launch Symmetric and Asymmetric attacks using CURL and by changing the headers of the request to bypass the rate limit for HTTP/1.1 and HTTP/2 to test the CPU utilization of the servers and this attack brings down the targeted server with this attack.

3) Attack using GO multiplexing: Go language script was written to launch a Symmetric and Asymmetric attack using the multiplexing feature for HTTP/2 to test the CPU utilization of the servers and this attack brings down the targeted server with this attack.

4) Attack using Multiprocessing: Python script was written to launch a Symmetric and Asymmetric attack using a multiprocessing library to test the CPU utilization of the servers and this attack brings down the targeted server with this attack.

VII. Results and Discussion

A. Performance Comparison of HTTP/1.1 and HTTP/2 Under a Symmetric DDoS Attack

Fig. 3 displays a comparison of the performance between HTTP/2 and HTTP/1.1 in various scenarios under symmetric attack by sending multiple same requests where the rate limit is placed at the target server. In particular, Fig. 3a and Table II shows the CPU usage when the requests with low or normal workload and 3b and Table III show the performance of HTTP/1.1 with a single client symmetric attack with and without SSL under heavy workload request respectively. These HTTP/1.1 requests were generated using the CURL command and shell script, to send multiple same requests, but it incurs higher CPU usage combined with SSL than without SSL. Fig. 3c along with Table IV and 3d along with Table V exhibit the performance of HTTP/1.1 multi-client with and without SSL, respectively, using five and fifteen clients. When the number of clients increases the CPU consumption exponentially. Fig. 3e demonstrates the performance of HTTP/2 with multiplexing

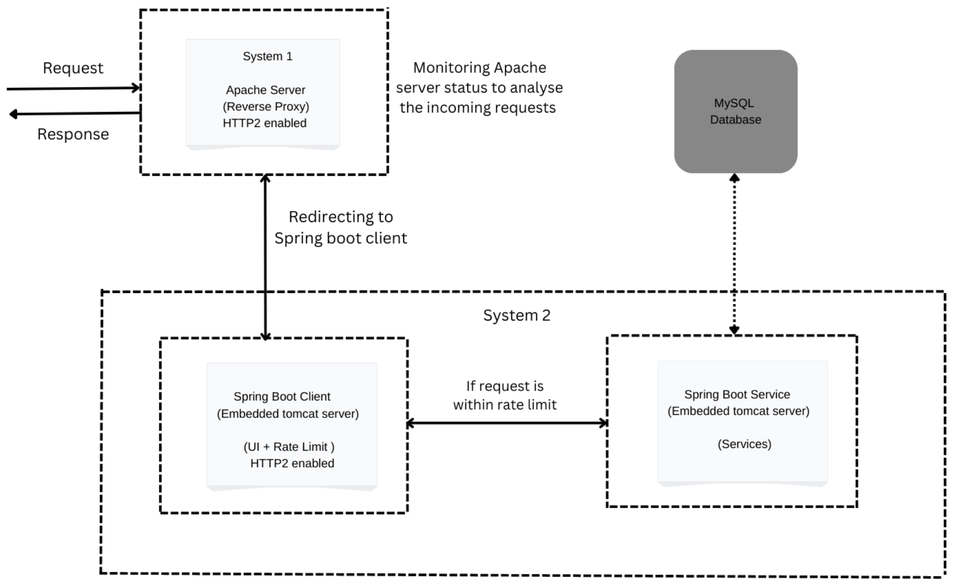


Fig. 2. Test bed setup.

using a Go language script and CURL shell script. HTTP/2 outperforms HTTP/1.1 CURL with SSL and exhibits higher CPU usage than HTTP/1.1 using the HTTPX library. The better CPU usage of HTTP/2 is due to various attributes, such as multiplexing, header compression, and server push compared with Fig. 3c and 3d. Fig. 3f displays the performance of HTTP/2 multiplexing with five and fifteen clients. For high workload requests, HTTP/2 multiplexed requests, result in high CPU usage and effectively bring down the target server. If no rate limit is placed at the target server it brings down the server with even less number of requests (Tables VI to IX).

TABLE II. Comparison of CPU Usage with and without SSL for Different Numbers of HTTP/1.1 Requests using HTTPX Client Library

No. of Requests	CPU Usage with SSL (%)	CPU Usage without SSL (%)
500	20	20
1000	20	25
1500	20	27
2000	20	30

TABLE III. Comparison of CPU Usage with and without SSL for Different Numbers of HTTP/1.1 Requests using CURL Script

No. of Requests	CPU Usage with SSL (%)	CPU Usage without SSL (%)
500	20	50
1000	25	60
1500	35	65
2000	30	60

B. *Performance Comparison of HTTP/1.1 and HTTP/2 Under a Asymmetric DDoS Attack*

Fig. 4 displays a comparison of the performance between HTTP/2 and HTTP/1.1 in various scenarios under asymmetric attack by sending multiple different requests by changing request headers where the rate limit is placed at the target server. In particular, 4a shows the performance of HTTP/1.1 with a single process asymmetric attack with and without SSL, respectively. The HTTP/1.1 CURL with and without SSL is written in shell script and uses the curl command

TABLE IV. Comparison of CPU Usage with and without SSL for Different Numbers of HTTP/1.1 Requests using 5 Multiple Clients

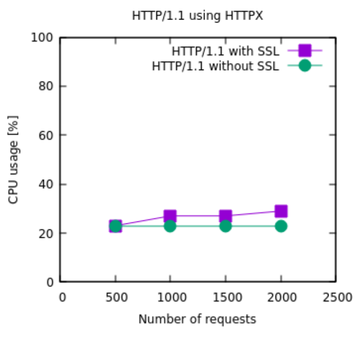
No. of Requests	CPU Usage with SSL (%)	CPU Usage without SSL (%)
500	20	50
1000	20	55
1500	25	55
2000	25	50

TABLE V. Comparison of CPU Usage with and without SSL for Different Numbers of HTTP/1.1 Requests using 15 Multiple Clients

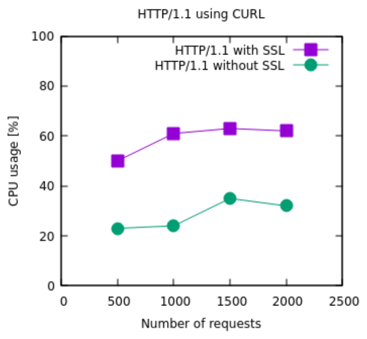
No. of Requests	CPU Usage with SSL (%)	CPU Usage without SSL (%)
500	30	90
1000	50	95
1500	45	95
2000	50	95

TABLE VI. Comparison of CPU Usage with and without SSL for Different Numbers of HTTP/2 Requests using GO and Curl Script

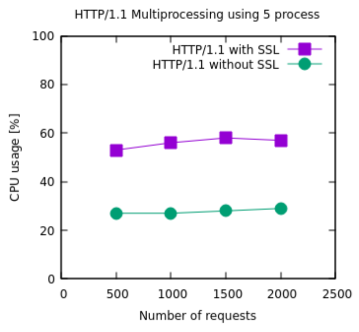
No. of Requests	CPU Usage using CURL Script (%)	CPU Usage using GO Script (%)
500	25	25
1000	30	35
1500	25	25
2000	30	35



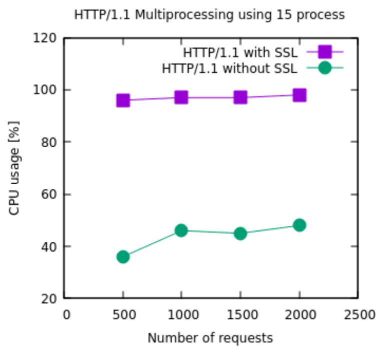
(a) HTTP/1.1 Using HTTPX



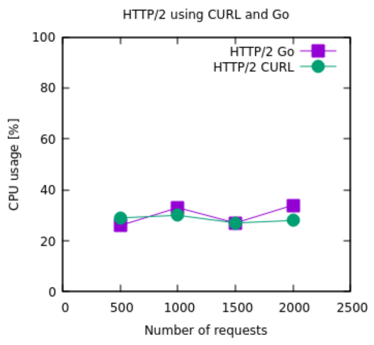
(b) HTTP/1.1 Using CURL



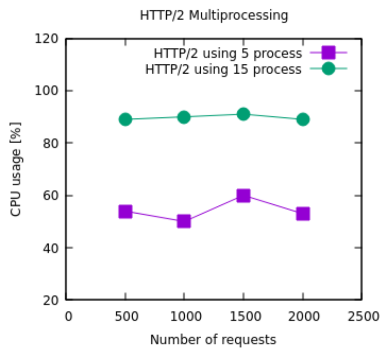
(c) HTTP/1.1 Using 5 Multiple Clients



(d) HTTP/1.1 Using 15 Multiple Clients

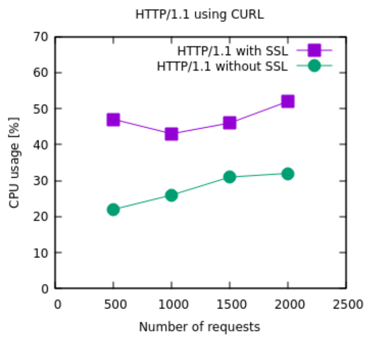


(e) HTTP/2 Using CURL and Go

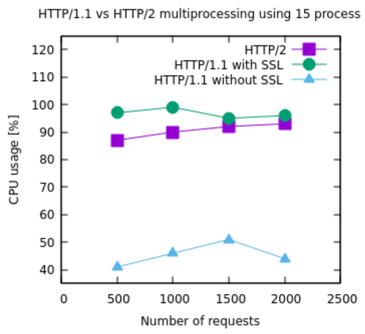


(f) HTTP/2 Using Multiple Clients

Fig. 3. The Correlation between CPU usage and the number of requests in an HTTP/2 server during a Symmetric DDoS attack.



(a) HTTP/1.1 Using Single Client (CURL)



(b) HTTP/1.1 and HTTP/2 Using Multiple Clients

Fig. 4. Comparison of CPU usage and request handling between single and multiple clients in HTTP/1.1 and HTTP/2.

to send multiple different requests, but it incurs higher CPU usage with SSL. Fig. 4b exhibit the performances of HTTP/1.1 requests when it is sent from multiple clients with SSL or Without SSL respectively. Also, it depicts the CPU usage when HTTP/2 multiplexing requests are sent from multiple clients. When the number of clients increased thereby increasing the number of asymmetric requests. When high-workload requests are processed by the server, latency increases thereby reducing the throughput of the system. When more and more requests are coming to the server, either those requests are queued or rejected even if it's a legitimate request. With a low volume of request rates, HTTP/2 multiplexing results in high CPU usage

TABLE VII. COMPARISON OF CPU USAGE WITH AND WITHOUT SSL FOR DIFFERENT NUMBERS OF HTTP/2 REQUESTS USING 5 AND 15 MULTIPLE CLIENTS

No. of Requests	CPU Usage for 5 Clients (%)	CPU Usage for 15 Clients (%)
500	50	90
1000	45	95
1500	60	95
2000	50	90

TABLE VIII. ASYMMETRIC ATTACK: COMPARISON OF CPU USAGE WITH AND WITHOUT SSL FOR DIFFERENT NUMBERS OF HTTP/1.1 REQUESTS USING SINGLE CLIENT

No. of Requests	CPU Usage with SSL (%)	CPU Usage without SSL (%)
500	20	50
1000	35	45
1500	30	50
2000	35	55

TABLE IX. ASYMMETRIC ATTACK: COMPARISON OF CPU USAGE WITH AND WITHOUT SSL FOR DIFFERENT NUMBERS OF HTTP/1.1 AND HTTP/2 REQUESTS USING 15 MULTIPLE CLIENTS

No. of Requests	CPU Usage for HTTP/1.1 with SSL (%)	CPU Usage for HTTP/1.1 without SSL (%)	CPU Usage for HTTP/2
500	40	95	85
1000	45	100	90
1500	50	90	95
2000	45	95	95

and effectively bring down the target server even if the rate limit is employed.

C. Discussion

Particularly with HTTP/2 multiplexing and multi-client in our experimental settings, the suggested method consists of a sequence of tests to find resource depletion attacks that can effectively stress RESTful APIs. This work compared with [40] where requests are generated and tested using web application and URL. But this work took the OAS document and identifying the endpoint which is heavily loaded and then it generated the API requests using the endpoint and its operations. From the CPU utilization statistics shown in the table and the picture, it is evident that HTTP/2 generates a higher server workload than HTTP/1.1. For symmetric attacks, for instance, HTTP/2 multiplexing with 15 clients caused CPU use spikes of 95% compared to HTTP/1.1, which reached 65% under the identical loading conditions. This highlights both HTTP/2's potential for misuse in DDoS attacks and its efficiency in letting several requests concurrently. Moreover, Asymmetric Attack findings revealed that several request-important aspects put the API server under great pressure since the CPU limit can reach 95% even with the Rate-limiting. Based on the findings, SSL with non-SSL scenarios shows that encryption causes fairly little overhead, suggesting that the true bottleneck is on the request-processing side rather than the encryption side. Research by Lookout emphasizes the need of using advanced detection techniques and stronger rate-limiting mechanisms to assist in defense against such complex application-layer DDoS attacks—including those using HTTP/2 capabilities.

VIII. CONCLUSION

DDoS attacks remain a serious threat to online services and continue to evolve in sophistication and scale. Recent years have seen an increase in the frequency and intensity of DDoS attacks, as well as the emergence of new attack vectors and techniques. Defending against DDoS attacks requires a combination of preventive measures, such as network and application layer defenses, as well as reactive strategies, such as monitoring and incident response. This paper proposed a security testing strategy to identify the vulnerability of API using a feature of HTTP/2 called multiplexing, which is exploited by a DoS attack. By trying to send a few requests in parallel from multiple clients through HTTP/2 multiplexing, the attack made the request consume a large number of CPU resources even though the rate limit was imposed on the gateway. It was observed from the experiments the requests sent using HTTP/1.1 consumed CPU usage relatively better than HTTP/2. It was also observed that the CPU usage of the target server was much more when performing testing based on Multi-client Symmetric/Asymmetric multiplexed on HTTP/2 was significantly higher that would make the services unavailable, which is justified by the multiplexing property of HTTP/2 as it tries to send multiple requests in one TCP connection.

The paper outlines several future directions for advancing RESTful API security, particularly in mitigating application-layer DDoS attacks and resource exhaustion vulnerabilities. Key areas include developing workload-based testing to assess the impact of high-computation requests, exploring interdependencies between APIs to understand complex attack vectors, and designing advanced rate-limiting mechanisms to counter sophisticated attacks like those leveraging HTTP/2 multiplexing. Additionally, research could focus on real-time monitoring and anomaly detection using AI/ML, integrating findings into API gateways, and evaluating the security implications of newer protocols like HTTP/3. Comprehensive tools for resource exhaustion testing and addressing regulatory compliance in API security are also highlighted as critical areas for future work. These directions aim to enhance API resilience against evolving threats and improve overall system robustness.

REFERENCES

[1] S.Levi, "Why api security is critical," https://www.forbes.com/sites/forbestechcouncil/2023/03/09/preventing-data-breaches-in-2023-why-api-security-is-critical, accessed: September 10, 2023.

[2] Imperva, "Quantifying the cost of api insecurity," https://www.imperva.com/resources/resource-library/reports/quantifying-the-cost-of-api-insecurity/, accessed: December 11, 2024.

[3] Twitter, "Twitter data breach," https://privacy.twitter.com/en/blog/2022/an-issue-affecting-some-anonymous-accounts, accessed: January 2, 2025.

[4] Optus, "Optus data breach," https://en.wikipedia.org/wiki/2022-Optus-data-breach, accessed: January 10, 2025.

[5] Salt, "T-mobile api breach what went wrong," https://salt.security/blog/t mobile api breach what went wrong, accessed: January 4, 2025.

[6] PortSwigger, "Critical vulnerability allowed attackers to remotely unlock control hyundai genesis vehicles," https://portswigger.net/daily-swig/critical-vulnerability-allowed-attackers-to-remotely-unlock-control-hyundai-genesis-vehicles, accessed: January 7, 2025.

[7] CircleCI, "CircleCI incident report for january 4 2023 security incident," https://circleci.com/blog/jan-4-2023-incident-report/, accessed: January 11, 2025.

[8] OWASP, "Owasp top 10 api security risks-2023," https://owasp.org/API-Security/editions/2023/en/0x11-t10/, accessed: December 11, 2024.

[9] E. Viglianisi, M. Dallago, and M. Ceccato, "Resttestgen: automated black-box testing of restful apis," in *2020 IEEE 13th International Conference on Software Testing, Validation and Verification (ICST)*. IEEE, 2020, pp. 142–152.

[10] V. Atlidakis, P. Godefroid, and M. Polishchuk, "Restler: Stateful rest api fuzzing," in *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*. IEEE, 2019, pp. 748–758.

[11] Y. Liu, Y. Li, G. Deng, Y. Liu, R. Wan, R. Wu, D. Ji, S. Xu, and M. Bao, "Morest: model-based restful api testing with execution feedback," in *Proceedings of the 44th International Conference on Software Engineering*, 2022, pp. 1406–1417.

[12] C. Lyu, J. Xu, S. Ji, X. Zhang, Q. Wang, B. Zhao, G. Pan, W. Cao, and R. Beyah, "Miner: A hybrid data-driven approach for rest api fuzzing," *arXiv preprint arXiv:2303.02545*, 2023.

[13] A. Martin-Lopez, S. Segura, and A. Ruiz-Cortés, "Restest: automated black-box testing of restful web apis," in *Proceedings of the 30th ACM SIGSOFT International Symposium on Software Testing and Analysis*, 2021, pp. 682–685.

[14] S. Karlsson, A. Causevic, and D. Sundmark, "Quickrest: Property-based test generation of openapi-described restful apis," 2019.

[15] N. Laranjeiro, J. Agnelo, and J. Bernardino, "A black box tool for robustness testing of rest services," *IEEE Access*, vol. 9, pp. 24 738–24 754, 2021.

[16] Postman, "Postman," https://www.postman.com, accessed: December 5, 2024.

[17] RestAssured, "Restassured," https://www.rest-assured.io, accessed: December 10, 2024.

[18] smartbear, "Readyapi," https://smartbear.com/product/ready-api/, accessed: December 13, 2024.

[19] APIFortress, "Apifortress," https://saucelabs.com/products/api-testing, accessed: December 17, 2024.

[20] G. Deng, Z. Zhang, Y. Li, Y. Liu, T. Zhang, Y. Liu, G. Yu, and D. Wang, "Automated restful api vulnerability detection," in *32nd USENIX Security Symposium (USENIX Security 23)*, 2023, pp. 5593–5609.

[21] 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."

[22] R. Haddad and R. E. Malki, "Openapi specification extended security scheme: A method to reduce the prevalence of broken object level authorization," *arXiv preprint arXiv:2212.06606*, 2022.

[23] T. Taya, M. Hanada, Y. Murakami, A. Waseda, Y. Ishida, T. Mimura, M. W. Kim, and E. Nunohiro, "An automated vulnerability assessment approach for webapi that considers requests and responses," in *2022 24th International Conference on Advanced Communication Technology (ICACT)*. IEEE, 2022, pp. 423–430.

[24] D. Votipka, K. R. Fulton, J. Parker, M. Hou, M. L. Mazurek, and M. Hicks, "Understanding security mistakes developers make: Qualitative analysis from build it, break it, fix it," in *29th USENIX Security Symposium (USENIX Security 20)*, 2020, pp. 109–126.

[25] M. Le Jeune, "Facebook and the cambridge analytica scandal: Privacy and personal data protections in canada," Ph.D. dissertation, Carleton University, 2021.

[26] M. Bach-Nutman, "Understanding the top 10 owasp vulnerabilities," *arXiv preprint arXiv:2012.09960*, 2020.

[27] M. A. Al Kabir and W. Elmedany, "An overview of the present and future of user authentication," in *2022 4th IEEE Middle East and North Africa COMMunications Conference (MENACOMM)*. IEEE, 2022, pp. 10–17.

[28] K. Dennis, M. Alibayev, S. J. Barbeau, and J. Ligatti, "Cybersecurity vulnerabilities in mobile fare payment applications: a case study," *Transportation Research Record*, vol. 2674, no. 11, pp. 616–624, 2020.

[29] L. Pan, S. Cohney, T. Murray, and V.-T. Pham, "Detecting excessive data exposures in web server responses with metamorphic fuzzing," *arXiv preprint arXiv:2301.09258*, 2023.

[30] S. Khan, I. Kabanov, Y. Hua, and S. Madnick, "A systematic analysis of the capital one data breach: Critical lessons learned," *ACM Transactions on Privacy and Security*, vol. 26, no. 1, pp. 1–29, 2022.

[31] B. Amin Azad, O. Starov, P. Laperdrix, and N. Nikiforakis, "Web runner 2049: Evaluating third-party anti-bot services," in *Detection of Intrusions and Malware, and Vulnerability Assessment: 17th International Conference, DIMVA 2020, Lisbon, Portugal, June 24–26, 2020, Proceedings 17*. Springer, 2020, pp. 135–159.

[32] O. B. Fredj, O. Cheikhrouhou, M. Krichen, H. Hamam, and A. Derhab, "An owasp top ten driven survey on web application protection methods," in *Risks and Security of Internet and Systems: 15th International Conference, CRiSIS 2020, Paris, France, November 4–6, 2020, Revised Selected Papers 15*. Springer, 2021, pp. 235–252.

[33] D. Kornienko, S. Mishina, S. Shcherbatykh, and M. Melnikov, "Principles of securing restful api web services developed with python frameworks," in *Journal of Physics: Conference Series*, vol. 2094, no. 3. IOP Publishing, 2021, p. 032016.

[34] S. Aslam and M. Mrissa, "A framework for privacy-aware and secure decentralized data storage," *Computer Science and Information Systems*, no. 00, pp. 7–7, 2023.

[35] H. Gantikow, C. Reich, M. Knahl, and N. Clarke, "Rule-based security monitoring of containerized environments," in *Cloud Computing and Services Science: 9th International Conference, CLOSER 2019, Heraklion, Crete, Greece, May 2–4, 2019, Revised Selected Papers 9*. Springer, 2020, pp. 66–86.

[36] M. Aljabri, M. Aldossary, N. Al-Homeed, B. Alhetelah, M. Althubiany, O. Alotaibi, and S. Alsaqer, "Testing and exploiting tools to improve owasp top ten security vulnerabilities detection," in *2022 14th International Conference on Computational Intelligence and Communication Networks (CICN)*. IEEE, 2022, pp. 797–803.

[37] S. Loureiro, "Security misconfigurations and how to prevent them," *Network Security*, vol. 2021, no. 5, pp. 13–16, 2021.

[38] A. Rahman, S. I. Shamim, D. B. Bose, and R. Pandita, "Security misconfigurations in open source kubernetes manifests: An empirical study," *ACM Transactions on Software Engineering and Methodology*, vol. 32, no. 4, pp. 1–36, 2023.

[39] M. Hasan and M. M. Rahman, "Minimize web applications vulnerabilities through the early detection of crlf injection," *arXiv preprint arXiv:2303.02567*, 2023.

[40] A. Praseed and P. S. Thilagam, "Multiplexed asymmetric attacks: Next-generation ddos on http/2 servers," *IEEE Transactions on Information Forensics and Security*, vol. 15, pp. 1790–1800, 2019.