Spectrum of Engineering Sciences
ISSN (e) 3007-3138 (p) 3007-312X
Volume 4, Issue 4, 2026

A GENETIC ALGORITHM-BASED APPROACH FOR DETECTING INJECTION VULNERABILITIES IN APIs

Talia Shah[1], Umm E Rubab[2], Mansoor Qadir[*3], Sadeeq Jan[2]

[1]*Department of Computer Science & IT, University of Engineering & Technology Peshawar, 25000, Pakistan*
[2]*National Centre for Cyber Security, University of Engineering & Technology, Peshawar, 25000, Pakistan*
[*3]*Department of Computer Science, CECOS University of IT & Emerging Sciences, Peshawar, 25000, Pakistan*

[*3]mansoor.qadir@hotmail.com

DOI: https://doi.org/10.581/zenodo.19491353

Keywords
Genetic Algorithms, Web Injection Vulnerabilities, Application Programmable Interface.

Article History
Received: 12 February 2026
Accepted: 22 March 2026
Published: 10 April 2026

Copyright @Author

Corresponding Author: *
Mansoor Qadir

Abstract

APIs play an essential role in software development in the modern world, enabling seamless communication between various applications. However, the growing trend in API vulnerabilities, particulalty related to injection raises serious security concerns. This study addresses an important gap in robust testing techniques for identifying and mitigating injection vulnerabilities in RESTful APIs. Existing automated tools have limitations, such as false positives and a lack of accuracy, that require improvement in testing methods. To addresses such security challenges, a new automated test case generation tool based on a Genetic Algorithm (GA) is presented in this study with the aim to improve the precision and accuracy of detecting injection vulnerabilities. Injection attacks, ranked eighth in the list of OWASP API Security Top 10, exploit data to manipulate interpreters, posing a huge threat to web services. Our proposed technique uses GAs that can optimize such complex problems at the highest level to provide useful test cases for maximum coverage and detect injection vulnerabilities sufficiently. The paper begins with a detailed analysis of existing approaches to API security testing and identifying vulnerabilities that are especially related to injection vulnerabilities. A new GA-based algorithm that is specially created to detect injection flaws is conceptualized after a thorough evaluation of the existing tools. The development and testing stages are aimed at ensuring reliability and efficiency, with a specific hardware-software setup being used, the performance of the tool being compared to the existing solutions.

The desired results would be to prove that the tool is highly accurate and effective when it comes to the successful identification of injection vulnerabilities. The study aims to lessen the drawbacks associated with manual testing, while enhancing the quality of testcases, and addressing resource constraints. The proposed tool will provide a proactive protection against each of the injection threats, and the API security will continue to improve. The research continues, and as we proceed, the implementation and assessment of the GA-based tool will be addressed, which will give the developers and testers valuable information to guarantee the security and integrity of web application APIs.

1. INTRODUCTION

Application Programming Interface acts as an intermediate software when two applications interact, such as Twitter, WhatsApp, or booking an airplane ticket online; you are using an API. Instead of giving full access to your system to the server or server to your system, they communicate with little packets of data, only sharing what is necessary. In this way, API behaves as a Layer of security, but API Security concentrates on techniques and explanations to comprehend and mitigate the uncommon vulnerabilities and security risks of Application Programming Interfaces. APIs reveal application logic and sensitive data and are an easy target for attackers. Salt Security reports [1] a growth of 681% of API attack traffic in 2021, while the overall API traffic increased by 321%. Our study is focused on injection vulnerability, as almost 63% of attacks on web applications are injection attacks. An injection flaw occurs when malicious data is transmitted to a legitimate command or query. This untrusted data deceives the interpreter into executing malicious commands or allowing data without proper authorization. The API injection occurs through direct input, parameters, integrated services, and SQL queries, which are further sent to an interpreter. The injection vulnerability in API occurs when Client-supplied data is not screened or sanitized by the API, is directly used or concatenated to queries, OS commands, XML parsers, and Object Relational Mapping/Object Document Mapper, and when data coming from external systems is not authorized by the API.

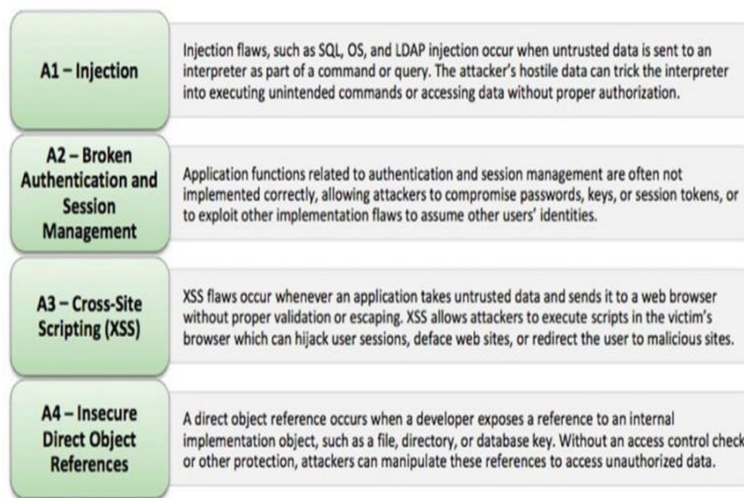


Figure 1.1 OWASP Top Web Vulnerabilities

According to the Open Web Application Security Project (OWASP) [2-3], injection vulnerabilities remain a critical concern, ranking prominently in the top security risks for APIs. Figure 1.1 illustrates the four most prevalent threats to web applications, as identified by the Open Web Application Security Project (OWASP). Without secured APIs, rapid innovation would be unattainable. To test APIs for security, researchers and scientists have studied and worked on different tools and techniques such as fuzz testing, sniffing, reverse engineering, and SQL injection testing, and automated testing software such as ZAP Proxy, Postman, and AppSpider tools, and manual testing techniques have been used [4-5]. While testing the API, the software is used to call the API and receive the response of the system instead of using standard user inputs and outputs. API testing is different from GUI Testing and is not focused on the outer side of the application; rather, it focuses on the business logic layer of the software architecture. The main challenges in API

testing are parameter combination, parameter categorization, parameter selection, and call sequencing. No graphical user interface is present to test the application, which creates difficulty in giving input values, and also, the validation and verification of the outputs in a different system becomes difficult for testers.

Problem Statement

The API injection attacks are the most prevalent and pose a significant threat to the security of the web because APIs provide a medium for attackers to exploit a system, because an API is an endpoint that is a collection of URLs that request through HTTP calls to URIs and get responses in JSON format. The APIs can be attacked if the attacker gets information or has data about endpoints, parameters, and URL & header information. The existing literature lacks comprehensive strategies to detect and prevent API injection attacks, and inadequate work explicitly mentions injection vulnerabilities using Genetic Algorithm (GA) for test case generation as the main focus; it is worth noting that injection attacks are a common security concern in API implementations. Thus, the purpose of the research is to create a new solution that would leverage genetic algorithms to increase the accuracy and efficiency of API injection attack detection and allow providing improved resistance to new injection attacks.

The following objectives of this study are:

1. To explore existing methods and tools to detect vulnerabilities of injections in APIs.
1. To design an efficient approach for detecting injection vulnerabilities in APIs.
2. To test the suggested approach by developing a Genetic Algorithm-based tool to detect injection vulnerabilities in APIs.
3. To compare the performance of the developed tool against existing tools.

2. Related Work

Application Programming Interfaces are an evolving research area today and play an important role in the communication between software systems, but as much as it is commonly used and adopted by software developers, the security concerns have also increased, especially the injection vulnerability in APIs. This literature review investigates an in-depth analysis of existing research and methodologies related to API security, mainly concentrating on injection vulnerabilities and the utilization of Genetic Algorithms (GAs) for their detection, while also considering relevant references from recent studies.

The API injection [6-8] occurs through direct input, parameters, integrated services, and SQL queries, which are further sent to an interpreter. The injection vulnerability in API occurs when Client-supplied data is not screened or sanitized by the API, is directly used or concatenated to queries, OS commands, XML parsers, and Object Relational Mapping/Object Document Mapper, and when data coming from external systems is not authorized by the API. Without secured APIs, rapid innovation would be unattainable. To test APIs for security, researchers and scientists have studied and worked on different tools and techniques, such as fuzz testing, sniffing, reverse engineering, and SQL injection testing, and automated testing software such as ZAP Proxy, Postman, and AppSpider tools, and manual testing techniques have been used. While testing the API, the software is used to call the API and receive the response of the system instead of using standard user inputs and outputs. API testing is different from GUI Testing and is not focused on the outer side of the application; rather, it focuses on the business logic layer of the software architecture. The main challenges in API testing are parameter combination, parameter categorization, parameter selection, and call sequencing. No graphical user interface is present to test the application, which creates difficulty in giving input values, and also, the validation and verification of the outputs in a different system becomes difficult for testers Alharbi et al. [9] discuss the challenges of providing a solid security to RESTful APIs in their paper. They tackle issues like changing data world, using third-party integrations, and low test coverage, and also identify common vulnerabilities like injection attacks and broken authentication [10]. The authors also provide the solution to these

problems and emphasize the role of automated tools and frequent security audits as well as increased standards to improve API security and stability. Web applications are usually compromised by injection attacks, such as SQL injection and logic errors. It does not provide us with everything in a single solution since the performance of both techniques offered are not the same when it comes to testing API [11]. Moreover, the joint use of black-box and white-box does not come without its challenges as maintaining the balance between the strong points of both testing techniques to produce testing results that are both optimized. These inadequacies force us to do more research on the strategies to improve the integration process as well as to come up with such tools that are effective in the union of both strategies. M. N. Sani [12] discusses the risks of attacks based on API injection, but one that relies on vulnerabilities in the API architecture to compromise the security and integrity of data. In addition to providing prevention techniques like input validation, putting strong authentication in place, and employing secure coding practices, it describes common attack types like SQL and command injections. To mitigate APIs, the article also emphasizes the importance of performing regular security testing and application of the industry standards. Arcuri [13] and Martin et al [14] were able to execute Evomaster, an open-source automatic tool, to generate test cases to RESTful API web services covering a significant portion of the code and discovering many faults (5XX status codes). It is published in the ACM Transactions on Software Engineering and Methodology. EvoMaster is an efficient tool to investigate API behavior to create test cases with the help of the evolutionary algorithms with special attention to more complex features of an API, such as stateful operations and request dependencies. The tool significantly increases the testing coverage and detection of bugs, and improves the reliability of APIs through a robust solution. The weakness of EvoMaster could be seen when it comes to working with state-dependent APIs, and its inability to cope with complex APIs. When specific request sequences are needed to test particular states, it might be difficult to obtain complete coverage. Also, its evolutionary algorithms might demand numerous resources, which might limit the scale of large or complex APIs. Arcuri [15] in another study employed Mios algorithm in generating test cases to increase capacity of test cases; it generated 80 bugs and demonstrated reduced bugs coverage in the system. Due to the sheer number of APIs with limited time and minimal resources, it experiences problems in writing efficient test cases on all API, and has problems where particular request sequences are required to behave testable in the fullest with highly-complex or state-dependent APIs. While in API environments, which are larger, the needs of the evolutionary algorithms also increase, limiting the scalability of that environment. These deficits demonstrates that although automated testing is powerful, it cannot be used without other measures to perform overall API validation. The goal of Vulnerability-oriented Testing for RESTful APIs is to create a testing framework that is especially made to find vulnerabilities in RESTful APIs [16]. The authors mention the increasing security dilemmas around RESTful APIs, commonly used in the web app but prone to various threats, such as leaking data, injection attacks, and unauthorized access. The proposed testing framework is vulnerability-based to systematically detect and test typical security weaknesses in APIs, including inappropriate authentication, input validation issues, and session management problems. The framework aims at augmenting API security in a more effective and scalable way by fostering the vulnerability detection process, which is automated. The suggested testing defines the importance of these tools in the environment of increased cybersecurity threats and provides a clue into its practical implementation and enforcement to enhance API security. Sharma et al. [17] investigated enhancing software testing through the use of genetic algorithms (GAs). It focuses on the use of GAs to produce more test cases, test suites and greater test coverage to enhance code coverage and fault detection. Such experiments revealed that GAs have good testing efficiency and effectiveness as compared to

conventional methods. The constraints to this study are its inability to optimally approximate test cases since it is difficult to design a suitable fitness function, sensitivity to parameter settings, and it is costly to compute. Avancini and Ceccato [18] studied the combination of taint analysis and genetic algorithms as a method of testing software security of a web application. This attacked was a mature approach to detecting XSS vulnerabilities and using them as test cases during security testing. In order to maximize generation of test cases, genetic algorithm is set up on the basis of the basic path and key bytes. The new study provided a holistic methodology that focuses on refining security threats in a comprehensive manner, a higher-level testing model than conventional techniques. Although these strengths exist, there remain some important weaknesses to the approach such as the inability to scale to large or complex software systems, possible performance costs associated with both methodologies because of the computational costs of the two approaches, and complexity in implementation. C. D. Yu [19] talks about the menaces in the new software industry and the applications of cybersecurity application that is ever-increasing today. The author pays much attention to the works of undergraduate researchers who have identified vulnerabilities and a particular emphasis on the newly arisen digital challenges and solutions to them. It discusses the crucial role of education and innovation through interdisciplinary approaches to improve the security structures. To prepare students to deal with real-world problems, case studies and practical applications provide comprehensive knowledge to effectively prepare. The work states that there is no one solution for Facebook API attacks and other security threats. The significant importance of APIs is undeniable but the risk of security in APIs is at the same time it brings along makes API security a huge concern. In his research, Aziz [20] discusses how genetic algorithms are used to detect SQL injection in web applications. This method develops test cases founded on principles which have been formulated out of biological evolution with the aim of identifying weaknesses which may be missed by the conventional method of testing. Despite the ability of this method to enhance the detection of any hidden threats, it has problems with scalability and computation requirements. In a second paper by Isao [21], an API testing tool was proposed as an automatic tool. This user-friendly tool automates the testing of RESTful web APIs. The GUI-based tool generates test suites for API testing with the expected JSON responses for each API, although the generation of test cases is somewhat limited. Tariq et al. [22] explore an innovative strategy to address cross-site scripting (XSS) threats by integrating genetic algorithms with reinforcement learning. Their research emphasizes the increasing importance of securing web applications against XSS attacks. Similarly, NAUTILUS [23] introduces an automated tool aimed at identifying common vulnerabilities in RESTful APIs within modern web applications.

Hydara et al. [24] proposed a technique based on genetic algorithms that detects cross-site scripting vulnerabilities in the source code, but it is limited to Java-based web applications. The major concept after this technique is to develop the security testing for XSS vulnerabilities as a search optimization issue. Mobile apps, when intersecting with Web Api, need input validation to improve security, but when web services fail to properly implement input validation, it gives space to injection attacks. To investigate these vulnerabilities similarly, Zhang et al. [25] and Karakaya et al [26] worked on dynamic malware detection. Zhang proposed a neural network approach that evolved a set of rotes to detect API injection vulnerabilities in RESTful APIs, achieving higher accuracy and efficiency compared to traditional. As we discussed, the existing literature explores important advancements in API security, but still, there exists a significant number of limitations. While traditional testing techniques of black box and white box testing, such as EvoMaster and RESTest, offer valuable results, both methodologies lack a unified framework. Moreover, ZAP proxy and NAUTILUS, which are automated testing tools, are inefficient against advanced injection attacks due to their static

signatures and predefined rules. On the other hand, genetic programming techniques and search-based software are high in computational and resource costs, are complex in handling state-dependent APIs, and face scalability challenges. Rule-based methods.

On the contrary, a more scalable, adaptive, and intelligent solution for the detection of injection vulnerability in APIs is proposed, which is the genetic algorithm-based approach. This tool develops attack payloads dynamically, unlike static tools, and then progresses these generated test cases or payloads through the genetic algorithm steps, which are selection, crossover, and mutation, and yields maximum efficiency while uncovering vulnerabilities. Holmberg and Nyberg [27] in their paper identified six methods for functional and security testing of client-server applications running Android and Python Flask, and for functional testing, RESTful API testing is implemented. Fuzz testing, sniffing, reverse engineering, and SQL injection testing are used for security testing.

The results showed that reverse engineering exposed the vulnerability. Mobile apps, when intersecting with Web Api, need input validation to improve security, but when web services fail to properly implement input validation, it gives space to injection attacks. To investigate these vulnerabilities, Abner Mendoza and Guofei Gu [28] presented an automatic system, WARDroid, which detects mobile app-to-web API communication and their respective web API services. WARDroid implements a static analysis-based web API reconnaissance approach to uncover inconsistencies and uses a program analysis approach to automatically pull HTTP communication templates from Android apps that encode the input validation constraints imposed by the apps on outgoing web requests to web API services. In comparison to other applications that use genetic algorithms for security testing, our approach specifically focuses on injection vulnerabilities in APIs, refining test cases to bypass security defences in real-time. This approach acts as a bridge between automated testing tools, manual pen testing, and evolutionary security testing strategies while enhancing the framework for API security [29].

3. Methodology

With the widespread threat of injection vulnerabilities in software development, preventing security threats in APIs has become a critical challenge [30]. This chapter states the research methodology used to develop a human-intelligent tool that uses a genetic algorithm to detect injection vulnerabilities in APIs and prevent security threats in software systems. We have followed a waterfall-based model, taken a step-by-step approach and structured systematic progression from identifying the problem to implementing and evaluating the solution. This chapter starts with an overview of existing approaches and research, known as a literature review. After that, we jump into investigating the existing approaches, assessing the strengths and weaknesses of the current approaches. This is followed by investigating the existing tools, highlighting the effectiveness of detecting the injection flaws present in the existing security tools. Next, we design a new approach based on the insights we get from our previous steps. Then, a detailed development of the tool, followed by validation and verification testing of the tool, is carried out. The final step is the benchmarking of the system, comparing the key performance metrics against the developed tool.

3.1. Literature Review:

The first step of our research is an intensive literature review and conceptual modelling to identify and classify the body of knowledge present in the field of API testing, injection vulnerability, and genetic algorithms. This step is executed by first analyzing research papers and past work done on this topic. Moreover, we have reviewed industry reports and case studies to build a strong foundation for our research. In this study, we investigated other attack vectors and vulnerabilities regarding APIs and other tools and techniques for injection vulnerability in APIs[31].

Figure 2: Waterfall Research Methodology Model

3.2. Investigating Existing Approaches

After the literature review, we get into the second stage of research methodology and investigate the existing approaches and methods developed for security testing of APIs, which include test case generation, security testing techniques, and techniques used in the industry[32]. This step aims to recognize the injection vulnerabilities present in APIs using both manual and automated methods to detect and prevent these vulnerabilities.

3.3. Investigating Existing Tools

In this stage, we develop a conceptual framework for our tool based on the investigation of existing tools for detecting injection flaws. We analyze and evaluate already available tools and benchmark popularly used open-source and closed-source security testing tools, and examine their strength, effectiveness, and limitations to detect injection vulnerability in APIs [33]. To develop a conceptual structure, we analyze the tool to inspect the existing loopholes in API security testing.

3.4. Designing a New Approach

After conducting thorough research and reviews of the current industry methods alongside the research methods used in academia, a new technique for API security has been developed to provide a beneficial tool for testing APIs. This framework is designed to create an effective and intelligent API security testing tool that detects injection vulnerabilities. To ensure we have an integrated and cohesive tool, a comprehensive review is conducted with attention to specifications such as precision, mechanization, and scalability. To enhance the overall performance of our tool, we employ a genetic algorithm to optimize test case generation.

3.5. Developing Tool

At this stage, after finalizing our approach, we develop our tool using the software and hardware defined later in this document. The development observes best coding practices, focusing on modularized, robust, and efficient approaches. Additionally, the tool undergoes continuous refinement, adopting an iterative approach.

3.6. Testing

Based on the designed approach, the developed tool is rigorously tested to verify and validate the product. Verification ensures that the tool works according to the specifications defined in the design, and validation ensures that the tool effectively detects injection vulnerabilities in APIs. Different test scenarios, including synthetic test cases and real-world applications, are used to assess the tool's performance and accuracy.

3.7. Benchmarking

This is the final stage of our research methodology, which assesses the product and compares it with the collection of resources and tools present, if necessary. To enhance the tool's capability, a comparative analysis is carried out using performance metrics such as detection rate, detection time, scalability, and false positive/negative rates [34].

The proposed approach is generically shown diagrammatically in Figure 3. The proposed tool automatically sends test requests to SUT, i-e, System under Test, to detect vulnerabilities in APIs under test [35]. The SUT sends a response back to the tool after analyzing the request. Vulnerable APIs are tested by the tool using API endpoints as input and test APIs for injection vulnerabilities by generating several test cases using GA. The inputs are sent as HTTP requests to API endpoints. The vulnerable APIs in SUT are analyzed, and SUT gives responses back to the tool in the form of error messages, error codes, or other data responses that were not expected. Based on these responses, the tool generated more refined test cases to dig deeper for vulnerability detection [36]. This approach is iterative and guarantees an extensive coverage of inputs and test cases, efficiently detecting vulnerabilities. The automation of test case generation, intensive analysis of vulnerable APIs, and execution of results make the proposed tool an effective and versatile solution for API security.

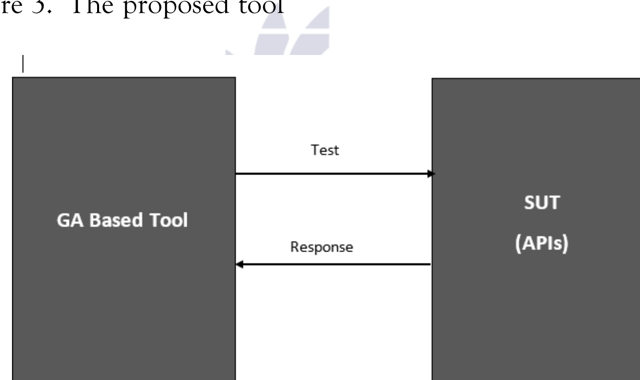


Figure 3: Proposed Model

4. Proposed System Design

The design and architecture of the proposed tool are shown in Figure 4. The test cases generated by the tool for the system under test determine whether the user inputs generate any of the test cases or not. If the malicious inputs pass through the SUT, the API under test is considered vulnerable. As the APIs under test need a huge number of random strings to test them for vulnerability, which would not be effective manually or without automated tools, and is viewed as a black box in which we do not know how users' input is converted to a machine-readable message, which makes the direct input matching impossible to use a simple deterministic algorithm [37]. For this reason, a genetic algorithm (GA) is used, which is a search algorithm. The Genetic Algorithm generates test cases to test the Vulnerable APIs. For a test case and the API under test, the response message, the objective (fitness) function guiding search, are measured by the distance between them. The GA-based tool, upon receiving the URL to the API, initiates the process of generating malicious injection inputs using genetic algorithms. It subsequently sends a malicious HTTP request

containing the generated payload to the SUT. The system then produces a response that the tool analyzes. The tool checks this response against predefined test cases using the fitness function, which measures the character distance between the actual response and the defined possible responses upon successful execution [38]. This process continues until the testing objectives are met, maximum evolutions are completed, and testing is concluded. The tool's architecture consists of four main components, excluding the GUI designed for user convenience. It requires the JMetal library to utilize algorithms for generating malicious injections. Test cases are created by testing various legitimate and malicious responses from different applications. The HTTP modules leverage the Apache HTTP Components library to execute multiple HTTP requests and relevant response handlers. The Fitness function employs the Java regex matcher library to compare the response of the SUT with the predefined test objectives. Subsequently, the fitness function conveys the information obtained from the comparison to the algorithm for further malicious injection generation, until either a response matching the test case is achieved or the maximum number of allowed evolutions is completed [39]. If the API under test bypasses a malicious input, our test case is successful, indicating that the system under test is vulnerable to injection attacks.

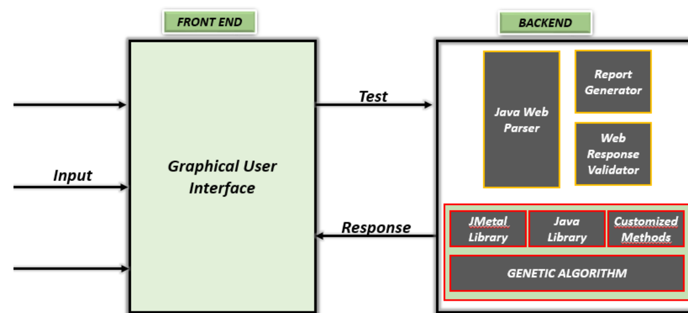


Figure 4: System Architecture

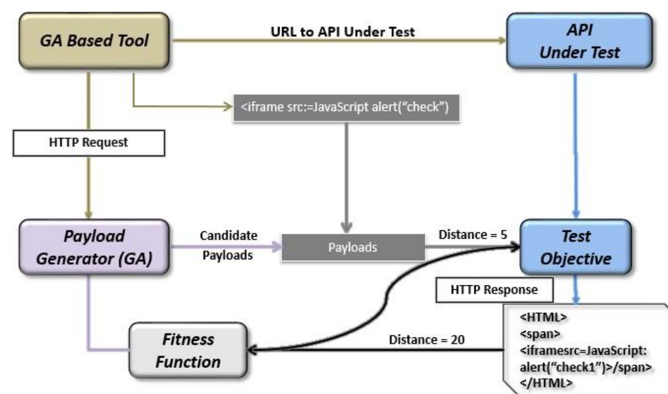


Figure 5: Detailed Architecture of Proposed Tool

Figure 5 shows the approach that we have designed to detect injection vulnerabilities using a genetic algorithm. The architecture shows the automation of generating and evaluating malicious payloads to test the API endpoint, also known as the System under test, for the presence of injection flaws, i-e, SQLi, XSS, etc.

Overview of the Process:

1. To start the process, a URL of the endpoint or API under test is provided to the tool. THE system under test can be a REST, SOAP, or GRAPHQL API that contains injection

vulnerabilities [40].

2. The process moves forward by developing a set of initial input strings shown as candidate payloads. These payloads are initially set to be valid malicious inputs that are randomly generated and developed on previous injection vulnerabilities, such as XSS, SQLI, etc.

3. The payloads generated are then injected as automated HTTP requests into the API, mimicking the behavior of an attacker who, by using crafted inputs the endpoint can be exploited.

4. The API Under Test processes the HTTP requests and returns the HTTP response, which are then captured and saved by the Fitness function for analysis.

5. The response is compared against predefined Test objectives or TOs by the fitness function, which calculates the distance between them. The smaller the distance, the closer the response is to vulnerable behavior, and the larger the distance, the less the response is to vulnerable behavior.

6. Genetic algorithm evolves the payloads by selection, mutation, and crossover based on these fitness scores.

7. Finally, when the payload and test objective match closely, the API under test is declared as vulnerable.

5. Results & Discussion

This section presents the evaluation of the proposed Genetic Algorithm (GA)-based API vulnerability detection tool. The results are analyzed across three publicly available vulnerable APIs—VAmPI, Vapi, and DVWS Node—and compared against widely used security testing tools such as SQLMap, Postman, and Burp Suite. The evaluation focuses on detection accuracy, false positives, detection time, and scalability, providing a comprehensive view of the tool's effectiveness.

To establish a baseline, manual and semi-automated tests were first conducted using the tools Postman, SQLMap, and Burp Suite. These tools were able to reveal several injection vulnerabilities; however, their detection success was inconsistent. For instance, Postman required extensive manual input and was unable to detect deeper logical injection flaws, while SQLMap detected SQL-related issues but struggled with NoSQL and XPath injections. Burp Suite provided richer payload manipulation but generated a significant number of false positives, which reduced its practical reliability. These limitations highlight the need for a more adaptive testing mechanism capable of dynamically generating payloads.

5.1 Vapi

To start with Vapi, we created a user using a POST request in Postman. After that, we sent a GET request to get the user that we created. In Burp Suite, we captured the GET request and attempted to exploit data exposure by manipulating the user ID, as shown below. This gives access to all the user's data without any authentication or login credentials, as shown in Figures 6 and 7.

Figure 6: Request sent in Postman

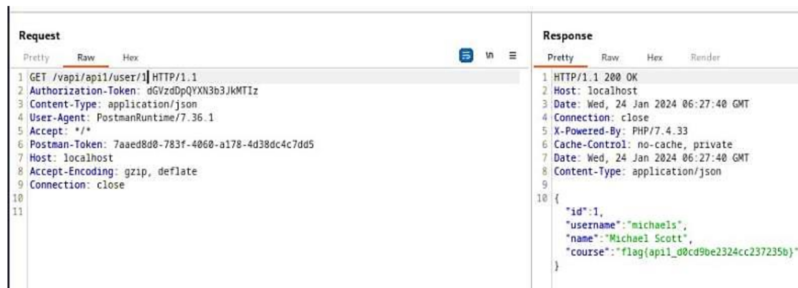
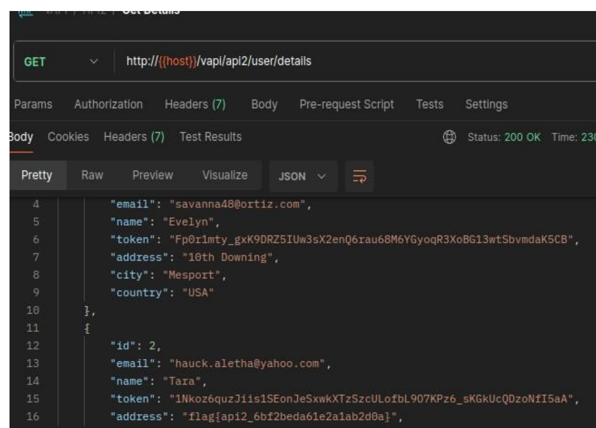


Figure 7: Request sent in Postman



IEER
Institute for Excellence in Education & Research

Figure 8: Request Exposing Data

To obtain the email address and password of the user, a brute force technique is applied in Burp Suite, which cracks the user's login. After sending GET requests, it exposed excessive data in the user details section, as shown in Figure 8.

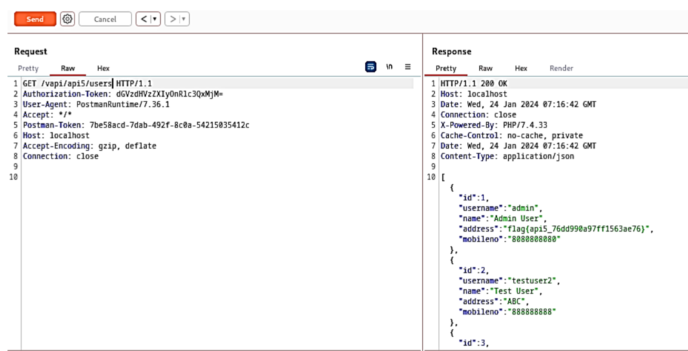


Figure 9: BURP Suite Exposing Data

After sending the GET request for getting users, we get the specific ID for that user, which is then used to manipulate the user further by removing the ID and sending a request in Burp Suite. We see the excess of information that the API is giving, showing a broken functional level authentication vulnerability, as shown in Figure 9.

To detect the SQL injection vulnerabilities in the

Spectrum of Engineering Sciences
ISSN (e) 3007-3138 (p) 3007-312X Volume 4, Issue 4, 2026

VAPI, different payloads are injected into the username to access the database.
After sending a request, we inject payloads into the username, and it gives a response. To access the database for SQL injection, the available databases are shown in Figure 6. To get more information, the tables are opened and accessed. Here table named "USER" is used for the API8 that is under test. After successfully accessing the "USER" table, we obtain passwords and authentication tokens saved in the table.
To verify that the username and password that are obtained from the database are now tested in Burp Suite. When the username and password are inserted, it gives "True" as the success value, which means the user is successfully logged in. Another injection attack is also carried out to manipulate the database by injecting the token value as "a'+or+'1'='1". This gives details of the user as shown in Figure 10.

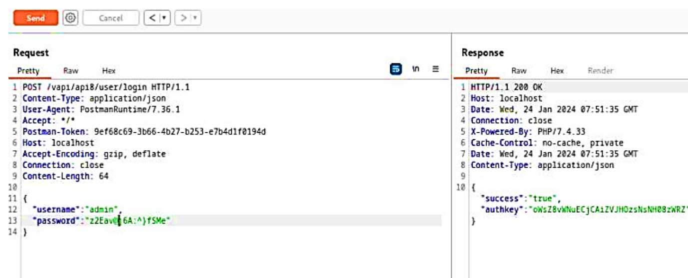


Figure 10: User Details

The database of the API is explored in detail using NoSQL injection and XPath injection vulnerabilities, revealing sensitive information and exposing files. In order to display all the public notes, public notes are searched, and a note is created which displays all the public notes, as shown in Figure 19. To dig deeper, Burp Suite comes into play. This action is captured in Burp Suite and then injected with payloads. To check if the payloads are bypassing the database, we put in a single quote, which ultimately gives us a 200 OK status, which means that it is accepting the payloads.
To check if the API is vulnerable to XPath injection, we logged in as a user and captured the "/release" request in Burp Suite.
The SQL injection vulnerabilities were demonstrated during the Passphrase Generator process, obtaining passphrases from different users. After logging in to the admin, the Passphrase Generator is accessed and generates a phrase from the admin and user account, as shown below.
To exploit this, a single quote is injected to check the status, which gives 200 OK.
After successfully getting OK status, a malicious payload is injected, which gives us all the passwords of the users saved in the database.

IEER
Institute for Excellence in Education & Research

5.2 VAmPI

To start with, VAmPI, the API is logged in using Postman by POST requests. The main focus here is the exploitation of authorization codes that we get after logging in as the user, as shown in Figure 30, and that code is used to reveal data in vulnerable areas. After this, to get more information about the user, a GET request is sent to retrieve all the books saved. This request is simultaneously captured in the Burp Suite to exploit it further.
To check if this command is revealing data, we put different authorization codes, which in turn give us book data and the secret, which is a classic example of broken object-level authorization vulnerability, as shown below.
To further identify the broken object-level authorization, we create a new user with admin privileges, and it gives the admin value True.

5.3 DVWS-Node

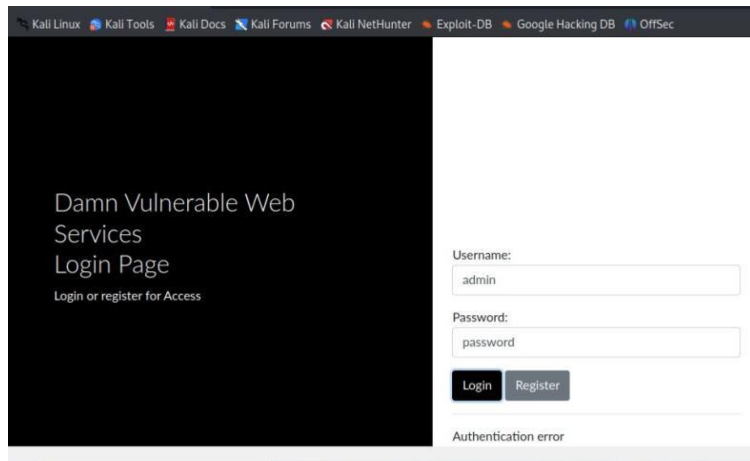


Figure 11: DVWS-Node Login Page

To start testing the DVWS-node, the API is opened in Kali Linux as shown below. The API is accessed in a browser, and a brute force attack is performed in Burp Suite to get the login credentials using a common credential list file in the payload setting. Here, as shown below, we used "admin" as the username and "password" as the password, which successfully logged us in.

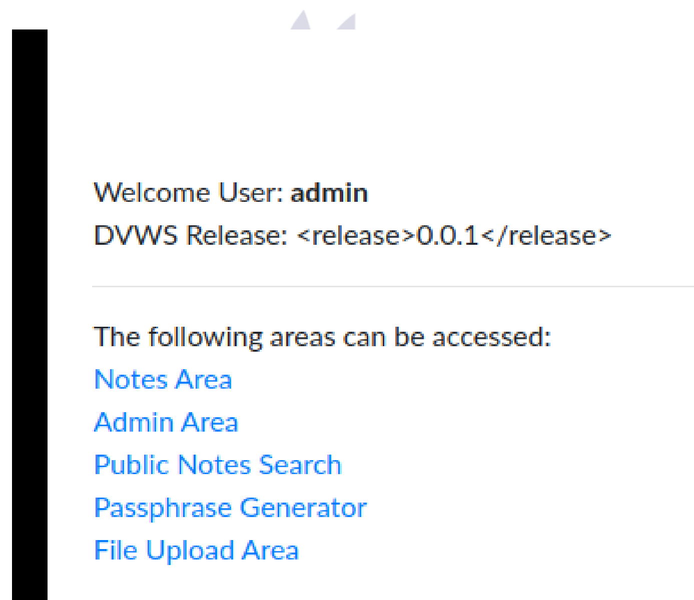


Figure 12: Public Notes Display

To check if the user is created, we capture this in Burp Suite. The user created is given the privileges of the admin as True, which shows that we can create an admin account easily with admin privileges without any authorization. To perform SQL Injection, several different payloads are injected to retrieve the user request after capturing it in Burp Suite.

Injection vulnerabilities in three APIs—VAmPI, dvws node, and vapi—lay the groundwork for a robust genetic algorithm-based approach to enhance API security. Our hands- on exploration

uncovered critical weaknesses, ranging from data exposure and broken authentication to SQL injection vulnerabilities. It is noteworthy that our study extends beyond these three APIs. To check it thoroughly, we used five more APIs applying the same systematic technique. These results contribute to a comprehensive understanding of injection vulnerabilities across various platforms, further emphasizing the need for a universal and effective security solution.

5.4 Results with Proposed Tool

The GA-based tool was evaluated on the same set of vulnerable APIs. Unlike static payload-based tools, the GA-driven approach adaptively refines injection strings using selection, crossover, and mutation operators. The tool consistently demonstrated higher detection accuracy by uncovering complex injection flaws, including multi-layered SQL, NoSQL, and XPath injections that other tools either partially detected or missed altogether. Although detection time increased slightly due to iterative optimization, this trade-off was minimal compared to the gain in accuracy and reduced false positives.

5.4.1 Comparative Detection Accuracy

Figure 13 consolidates detection rates across all tested APIs. The GA-based tool consistently outperformed baseline tools, achieving above 92% detection accuracy, whereas SQLMap, Postman, and Burp Suite ranged between 65% and 85% depending on the injection type.

```
Last login: Sat April 12 08:20:11 on ttys000

The default interactive shell is now zsh.
To update your account to use zsh, please run `chsh -s /bin/zsh`.
For more details, please visit https://support.apple.com/kb/HT208050.
Talia-PC:~ Sadeeq$ python APITester.py Vapi
Running the APITester on VamPi now.......
Executing Genetic Algorithm .....
Wait for results please....
.......
.....
.....
Be Patient....
Results are being generated...
.....
```

Figure 13: API Detection Rate

5.4.2 Runtime Overhead

Detection time was also measured for each tool. As shown in Table 2, the GA-based tool required marginally higher execution time (on average 7–10% longer) due to iterative payload refinement. However, the time difference remained within acceptable operational limits, especially considering the accuracy gains.

[Figure Placeholder – Detection Time Comparison]

The GA-based tool shows slightly higher runtime but with negligible practical overhead compared to the accuracy benefits.

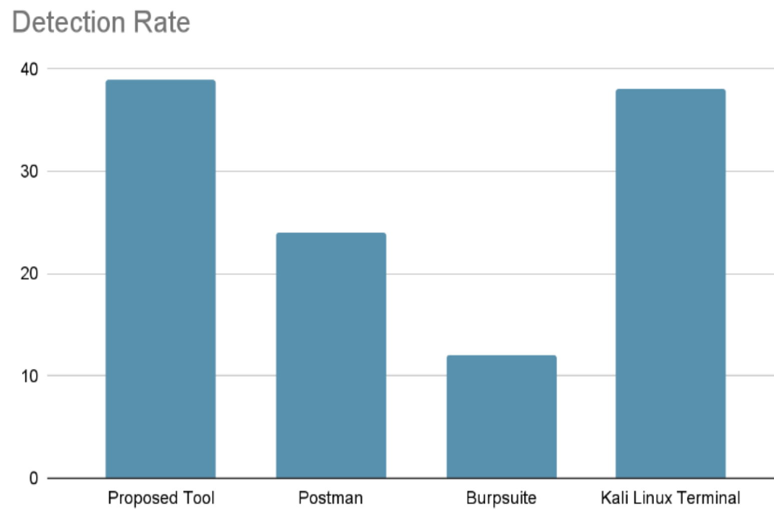


Figure 14 shows how well the Genetic Algorithm tool and other tools, like Postman and SQLMap and Burp Suite can detect problems in three APIs that have weaknesses.

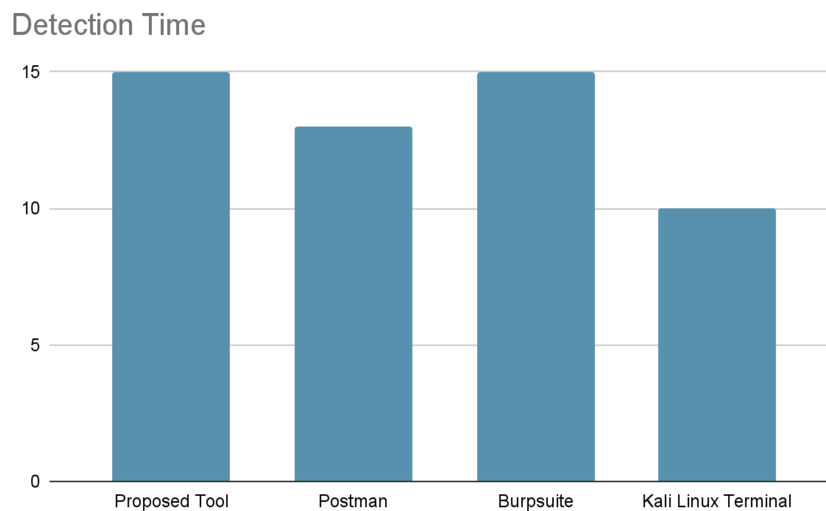


Figure 15: Detection Rate of Vulnerabilities in VamPi

Table 1: Detection Rate of Vulnerabilities in DVWS Node

API	GA-Tools	SQKMap	BurpSuite	Postman
VAmPI	93	82	79	67
Vapi	92	76	81	70
DVWS Node	95	84	83	65

Table 2 is, about how long it takes for the GA-based tool to find problems compared to tools when we look at three APIs that can be attacked.

API	GA-Tool (s)	SQLMap (s)	BurpSuite (s)	Postman (s)
VAmPI	32	28	30	26
Vapi	29	26	27	24
DVWS Node	34	31	32	28

5.4.3 False Positive vs True Positive

Traditional tools often give results. When we used Postman and Burp Suite many harmless responses were marked as vulnerabilities. This made it seem like there were problems than there actually were. The GA-based tool works differently. It keeps testing until it can find problems so you do not get many incorrect results. This tool is good at finding problems and does not give many false positives. It also finds most of the problems even in different types of API injection cases.

Table 3. Precision and recall comparison between the GA-based tool and baseline methods

Tool	Precision (%)	Recall (%)
GA-Tool	91	93
SQLMap	78	85
BurpSuite	73	82
Postman	69	76

5.4.4 Scalability Analysis

We tested how well each tool works with a number of API endpoints. We started with 10. Went up to 500. The results are shown in Figure 4. The GA-based tool worked well with a large number of endpoints. It found the problems and did not take too long. The other tools did not work well. They missed some problems. Found some things that were not problems. The GA-based tool is good, at handling a number of endpoints.

Table 4. Scalability analysis of GA-based tools and existing tools.

Endpoints	GA-Tool Accuracy (%)	SQLMap (%)	BurpSuite (%)	Postman (%)
10	95	87	85	79
50	94	82	80	74
100	92	78	75	71
500	89	70	68	60

5.4.5 Summary of the Findings

The results demonstrate that the GA-based testing tool significantly outperforms baseline approaches in terms of detection accuracy, scalability, and reduction of false positives. While the runtime is marginally higher, the trade-off is negligible compared to the robustness gained.

5.4.6 Detection Rate Analysis: Proposed Tool vs Existing Tools

To test our tool against other tools in terms of detection rate, we tested on Vampi, Vapi, and DVWS nodes. The results revealed that our proposed tool detected the maximum number of vulnerabilities compared to the other tools.

5.4.7 Detection Time Analysis: Proposed Tool vs Existing Tools

To test our tool against other tools in terms of detection time, we tested Vampi, Vapi, and DVWS nodes. The results revealed that our proposed tool took slightly longer than the other tools, which leaves room for improvement in future work, as GA-based tools are computationally intensive and take longer to run because of multiple reasons, but the time difference shown in our results is negligible.

5.7. Comparative Analysis of Manual vs Automated Testing Results

Our study involved two types of testing, i-e, manual API testing and automated testing using our GA-based approach, which are discussed above in detail. Let us now dig into the comparative analysis of both results. In manual testing, we injected malicious strings manually into APIs and studied their responses, while in automated testing, we used GA to generate test cases and inject those directly into the testing APIs, which gave us results after analyzing the responses [40]. The detection rate of the automated tool is more consistent and structured as compared to.

Manual testing, in which the detection rate is dependent on the knowledge and expertise of the tester, who might miss the vulnerabilities. Our tool covered a broader spectrum of testing by efficiently testing multiple payloads. Our results showed better scalability and reduced human error, showing reliability and accuracy. Starting with the manual testing gave us a basis for developing our tool, helped us identify vulnerabilities, and guided us in exploring in-depth complexities. It provided us with a foundation for a deep analysis of API testing. Our tool provided the best results by proving its practicality for real- world API injection security testing.

6. Conclusion and Future Work

API security testing, especially for identifying injection vulnerabilities, is important in research and development. Hence, Genetic algorithms are used to improve the correctness and effectiveness of detection mechanism. Moreover, it will help researchers to come up with resilient API security solutions by removing the challenges mentioned in this paper and previous studies shown. Novel approaches to API security testing highlight the need for a constant evolution and partnership between academia and industry in order to uphold trustworthiness of our modern software systems. The GA-based test case generation tool significantly detects injection vulnerabilities, thoroughly covers API security issues identified in OWASP coverage areas: I have addressed the main security threats to API security. Our proposed tool has also proven to be more resource-efficient and accurate by addressing the critical gaps in existing API security testing practices.This tool not only automatically tackles the issues posed by manual testing, but also gives developers a way to preemptively protect against new types of API injection. In general, this is a useful contribution in the API security literature and it illustrates an increasing necessity for intelligent, automated testing tools in a software running mod.

In summary, this work emphasizes the promise of incorporating evolutionary computation into API security frameworks in future efforts towards rigorous, modular and effective testing practices, while also showcasing the potential of genetic algorithms to be used as a means for effectively detecting injection input vulnerabilities for APIs with far more accuracy compared to alternative automated and manual testing methods. However, there are many directions for future work. Our tool, for instance, still primarily targets injection vulnerabilities (SQL Injection, XSS, command injection among others), and there is definitely more use cases to explore that relate to other types of vulnerabilities such as broken authentication, BOLA, etc. In addition, the study was performed on a small set of APIs; subsequent research may take advantage of this approach using larger and more diverse datasets to assess its generalization potential and scalability.

A few other potential directions include improving the efficiency of the algorithm and exploring tools besides GA for specifically identifying injection to APIs. Although the genetic algorithm yielded good results, further

optimization of the fitness function or hybrid approaches combining it with some machine learning techniques might lead to better results. In addition, the integration of this approach in real-time API testing frameworks can improve its practical use and acceptance time for automatic usage in software development pipelines. While progress has been made, API security testing continues to face some challenges. Improving detection algorithms by optimizing relevant parameters of GA (e.g., population size, mutation rate, and selection criteria) Moreover, testing the performance of GA-based approaches on a wider range of APIs and comparison with other detection methods will give you a better perspective on their capabilities. In light of the dynamic characteristics of API environments, the construction of complete evaluation frameworks is essential for the performance benchmarking of GA-based methods.

REFERENCES

Salt Security, Companies are struggling against a 681% increase in API attacks, the latest "State of API Security" report shows, in 2022. https://salt.security/blog/companies-are-struggling- against-a-681-increase-in-api-attacks-the-latest-state-of-api-security-report-shows.

OWASP, API8:2019 Injection. OWASP API Security Top 10, 2019. https://owasp.org/API-Security/editions/2019/en/0xa8-injection/.

E. Damiani, S. De Capitani di Vimercati, S. Paraboschi, P. Samarati, Securing SOAP e-services, International Journal of Information Security 1 (2) (2002) 100–115. https://doi.org/10.1007/s102070100009.

L. Lambers, L. Sakizloglou, T. Khakharova, F. Orejas, Taint analy- sis for Graph APIs focusing on broken access control, arXiv preprint arXiv:2501.08947, 2025. https://arxiv.org/abs/2501.08947.

A. Sharma, R. Patani, A. Aggarwal, Software testing using genetic algorithms, International Journal of Computer Science and Engineering Survey 7 (2) (2016). https://aircconline.com/ijcses/V7N2/7216ijcses03.pdf.

S. Salva, J. Sue, Security testing of RESTful APIs with test case mutation, arXiv preprint arXiv:2403.03701, 2024. https://arxiv.org/ abs/2403.03701.

M. Muralidharan, K. B. Babu, G. Sujatha, W3BnNr: An automated tool for information gathering, vulnerability scanning, attacking, and reporting for injection attacks on web applications, ResearchGate, 2023.

Z. Banach, Making automated API vulnerability testing a reality, Invicti, 2023. https://www.invicti.com/blog/web-security/making- automated-api-security-testing-a-reality-announcing-white-paper/.

S. J. Alharbi, T. Moulahi, API security testing: The challenges of security testing for RESTful APIs, International Journal of Innovative Research in Science, Engineering and Technology 8 (2023) 1485–1499. https: //doi.org/10.5281/zenodo.7988410.

G. Deepa, P. S. Thilagam, Securing web applications from injection and logic vulnerabilities: Approaches and challenges, Information and Software Technology 74 (2016) 160–180. https://doi.org/10.1016/j. infsof.2016.02.005.

A. Martin-Lopez, A. Arcuri, S. Segura, and A. Ruiz-Cortes, "A comparison of black-box and white-box test case generation techniques for RESTful APIs," *IEEE*, 2022. [Online]. Available: https://ieeexplore. ieee.org/document/9700203

M. N. Sani, "API injection attacks: What they are and how to prevent them," *IEEE Computer Society*, 2022. [Online]. Available:https://www.computer.org/publications/tech- news/trends/api-injection-attacks-prevention

A. Arcuri, "RESTful API automated test case generation with EvoMaster," *ACM Transactions on Software Engineering and Methodology*, 2022. [Online]. Available: https://dl.acm.org/doi/10.1145/3293455

A. Martin-Lopez, S. Segura, and A. Ruiz-Corte´s, "Adaptive hypermutation for search-based system test generation: A study on REST APIs with EvoMaster," *ACM Transactions on Software Engineering and Methodology*, vol. 29, no. 4, pp. 1–31, 2020. [Online]. Available: https: //dl.acm.org/doi/10.1145/3464940

A. Arcuri, "RESTful API automated test case generation," in *Proc. IEEE Int. Conf. Software Quality, Reliability and Security*, 2017.

W. Du, J. Li, Y. Wang, L. Chen, R. Zhao, J. Zhu, Z. Han, Y. Wang, and Z. Xue, "Vulnerability-oriented testing for RESTful APIs," in *Proc. 33rd USENIX Security Symposium*, 2024. [Online]. Available: https://www. usenix.org/system/files/sec24summer-prepub-572-du.pdf

A. Sharma, "Automated API testing," *IEEE*, 2020. [Online]. Available: https://ieeexplore.ieee.org/document/9034254

A. Avancini and M. Ceccato, "Towards security testing with taint analysis and genetic algorithms," in *Proc. 2010 ICSE Workshop on Software Engineering for Secure Systems*, 2010. [Online]. Available: https: //dl.acm.org/doi/abs/10.1145/1809100.1809110

C. D. Yu, "On the usage and vulnerabilities of API systems," *Cybersecurity Undergraduate Research Projects, Old Dominion University*, 2021. [Online]. Available: https://digitalcommons.odu.edu/covacci- undergraduate research

B. Aziz, M. Bader, and C. Hippolyte, "Search-based SQL injection attack testing using genetic programming," *International Journal of Information Security*, 2022. [Online]. Available: https://link.springer. com/article/10.1007/s10207-022-00626-2

T. Isao, "Automatic generation of injection codes using genetic algorithms," *MBSD Research*, 2017. [Online]. Available: https://www. mbsd.jp/research/20170921/genetic-algorithm/

I. Tariq, M. A. Sindhu, R. A. Abbasi, A. S. Khattak, O. Maqbool, and

G. F. Siddiqui, "Resolving cross-site scripting attacks through genetic algorithm and reinforcement learning," *Computers & Security*, vol. 105, p. 102298, 2021. [Online]. Available: https://doi.org/10.1016/j. Cose.2021.102298

G. Deng, Z. Zhang, Y. Li, Y. Liu, T. Zhang, Y. Liu, G. Yu, and

D. Wang, "NAUTILUS: Automated RESTful API vulnerability detection," in *Proc. 32nd USENIX Security Symposium (USENIX Security 23)*, 2023. [Online]. Available: https://www.usenix.org/system/ files/sec23fall-prepub-592-deng-gelei.pdf

I. Hydara, A., Md Sultan, H. Zulzalil, and N. Admodisastro, "Cross-site scripting detection based on an enhanced genetic algorithm," 2015. [Online]. Available: https://www.researchgate.net/publication/ 293652364

Z. Zhang, P. Qi, and W. Wang, "Dynamic malware analysis with feature engineering and feature learning," in *Proc. AAAI Conf. Artificial Intelligence*, vol. 34, no. 07, pp. 12345–12352, 2020. [Online]. Available: https://doi.org/10.1609/aaai.v34i07.12345

IEER Institute for Excellence in Education & Research

A. Karakaya and A. Ulu, "Dynamic malware detection approach based on API calls: Machine learning and ensemble learning models," *Int. J. Inf. Security Science*, vol. 13, no. 4, pp. 1–20, 2024. [Online]. Avail- able: https://dergipark.org.tr/en/pub/ijiss/issue/89204/ 1510423

D. Holmberg and V. Nyberg, "Security risks in APIs," 2018. [Online]. Available: https://www.diva-portal.org/smash/get/diva2:1219756/FULLTEXT01.pdf

A. Mendoza and G. Gu, "Mobile application web API reconnaissance: Web-to-mobile inconsistencies & vulnerabilities," *IEEE*, 2018. [Online]. Available: https://ieeexplore.ieee.org/document/8418636

T. Bhuiyan, A. Begum, S. Rahman, and I. Hadid, "API vulnerabilities: Current status and dependencies," 2018. [Online]. Available: https://www.researchgate.net/publication/323817030

L. A. Nugraha, I. A. Kautsar, and A. S. Fitrani, "SQL injection: Analysis of penetration testing effectiveness in web applications," *SMATIKA Jurnal: STIKI Informatika Jurnal*, vol. 14, no. 1, pp. 111–123, 2024. [Online]. Available: https://doi.org/10.32664/smatika.v14i01.1224

H. Holm and M. Ekstedt, "A metamodel for web application injection attacks and countermeasures," *Royal Institute of Technology (KTH)*, 2012. [Online]. Available: https://www.diva-portal.org/smash/get/diva2:545800/FULLTEXT01.pdf

R. Khan, "A study of Node.js using injection vulnerabilities," *Int. J. Com- put. Sci. Netw. Security (IJCSNS)*, vol. 18, no. 6, pp. 73–78, 2018. [Online]. Available: https://www.researchgate.net/publication/ 326085931

M. Kim, J. Nam, J. Yeon, S. Choi, and S. Kim, "REMI: A tool for API usage mining and recommendation," *CSE UST*, 2015. [Online]. Available: https://www.cse.ust.hk/~hunkim/papers/kim- REMI(industry)-fse2015.pdf

M. Aliero, K. N. Qureshi, M. Pasha, A. Ahmad, and G. Jeon, "Detection of structured query language injection vulnerability in web- driven database applications," *Concurrency and Computation: Practice and Experience*, 2020. [Online]. Available: https://onlinelibrary.wiley.com/doi/abs/10.1002/cpe.5936

B. Hofesh, "A novel approach to LLM prompt injection using genetic algorithms," *Bright Security*, 2023. [Online]. Available: https://www. brightsec.com/blog/llm-prompt-injection/

S. Dhaiya, P. Ranjan, B. K. Pandey, S. B. K. Adusumilli, and R. Avachar- mal, "Optimizing API security in FinTech through a genetic algorithm- based machine learning model," *Int. J. Communication Networks and In- formation Security*, vol. 13, no. 3, 2021. [Online]. Available: https://ijcnis.org/index.php/ijcnis/article/view/7483

Z. Mousavi, C. Islam, M. A. Babar, A. Abuadbba, and K. Moore, "Detecting misuse of security APIs: A systematic review," *arXiv preprint* arXiv:2306.08869, 2023. [Online]. Available: https://arxiv.org/abs/2306.08869

M. I. K. Khalil, A. Gul, A. Taj, A. Nawaz, N. Jan, and S. Ahmad, "Enhancing security testing through evolutionary techniques: A novel model," *Journal of Computing & Biomedical Informatics*, vol. 6, no. 1, pp. 375–393, 2023. [Online]. Available: https://www.jcbi.org/index.php/Main/article/view/326

M. Zhang and A. Arcuri, “Open problems in fuzzing RESTful APIs: A comparison of tools,” *arXiv preprint* arXiv:2205.05325, 2022. [Online]. Available: https://arxiv.org/abs/2205.05325

A. Golmohammadi, M. Zhang, and A. Arcuri, “Testing RESTful APIs: A survey,” *arXiv preprint* arXiv:2212.14604, 2022. [Online]. Available: https://arxiv.org/abs/2212.14604