

CHAPTER 4 - RESULT AND DISCUSSION

4.1. Introduction

This chapter presents the results of security testing conducted using the GSTF framework implemented in Python. The objective is to evaluate and validate the effectiveness of GSTF in identifying vulnerabilities and improving the testing process compared to traditional manual methods. The **security testing** approach follows the PoC scope defined earlier, which includes:

- **Manual security testing** using conventional human-driven techniques.
- **GSTF security testing**, leveraging automation for efficiency.
- **Comparison of manual and GSTF testing**, focusing on execution time
- **Comparison of GSTF with other security testing tools** to assess its advantages.

The testing is performed on a gRPC-based API, ensuring all functions defined in the proto file are covered. This structure ensures a clear, structured evaluation of GSTF's capabilities, limitations, and improvements over traditional methods.

4.2. Security Testing Execution

During the testing phase, the framework was transformed into a tool aligned with the GSTF framework and implemented using Python to automate the requests generated by GSTF. The expected outputs, which serve as **key parameters**, include execution time, identified vulnerabilities, coverage, false positive, success and error rates. In addition, manual testing was performed to collect execution time data, which was then compared with GSTF to calculate the efficiency achieved. The vulnerabilities that will be used as test cases in this test are based on API vulnerabilities that are often found in

studies conducted in 2023 (Amid Golmohammadi et al., 2023; Gelei Deng et al., 2023), including:

- SQL Injection
- Cross Site Scripting (XSS)
- Local File Inclusion (LFI)
- Server-Side Request Forgery (SSRF)
- Remote Command Execution (RCE)
- Xpath Injection
- Command Injection
- LDAP Injection

With the above vulnerabilities, testing will be done on every method including GSTF and manual testing. This testing is not based on vulnerabilities such as IDOR, Improper Input Validation which require humans to validate them manually and only common vulnerabilities can be validated based on patterns.

4.2.1. Manual Security Testing

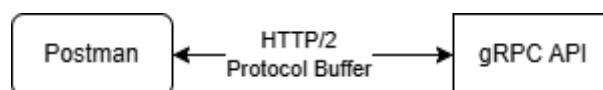


Figure 4.1 API Architecture in manual testing using postman

This section discusses the results obtained from manual security testing. This process involves periodically checking gRPC API endpoints, sending requests made, and

analyzing responses for potential vulnerabilities using the Postman application according to the architecture in Figure 4.1. Manual techniques such as input validation checks, authentication bypass attempts, and error message analysis were used to uncover security weaknesses.

The results uncovered several vulnerabilities, including improper input sanitization, weak authentication mechanisms, and inadequate error handling. While manual testing provides valuable insights, it is time-consuming and requires significant expertise to uncover complex issues. The challenges of manual testing include:

- Long time required for comprehensive coverage.
- Inconsistent results due to human error.
- Difficulty in detecting complex vulnerabilities such as race conditions or subtle data leaks.

A summary of the results of manual security testing is presented in the Table 4.1, highlighting the execution time for each endpoint and the vulnerabilities identified. This test was conducted 3 times by obtaining the average execution time using postman in initializing by importing the proto file which can be seen in Figure 4.2 and the sample request testing with manual validation which can be seen in Figure 4.3 and Figure 4.4.

Table 4.1 Time execution for manual security testing in gRPC

Endpoint	Execution time (ms) avg	Identified vulnerability
Initialization process	35000	• SQL Injection • Xpath Injection
Auth	315000	
Signup	302000	
VerifyToken	291000	

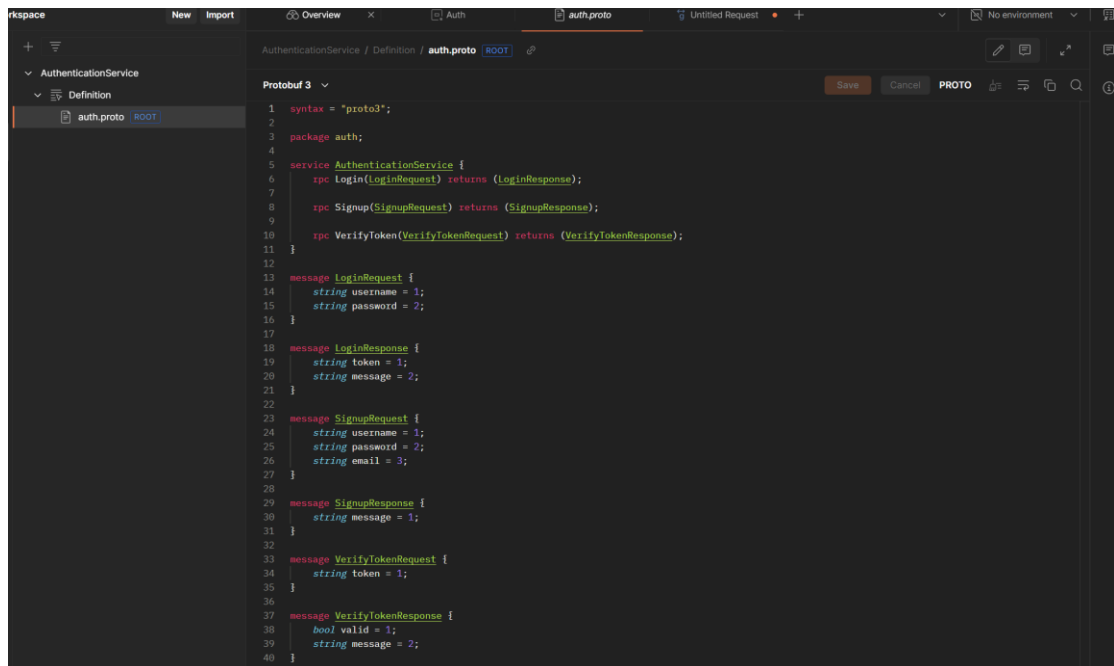


Figure 4.2 Initialize import auth proto file

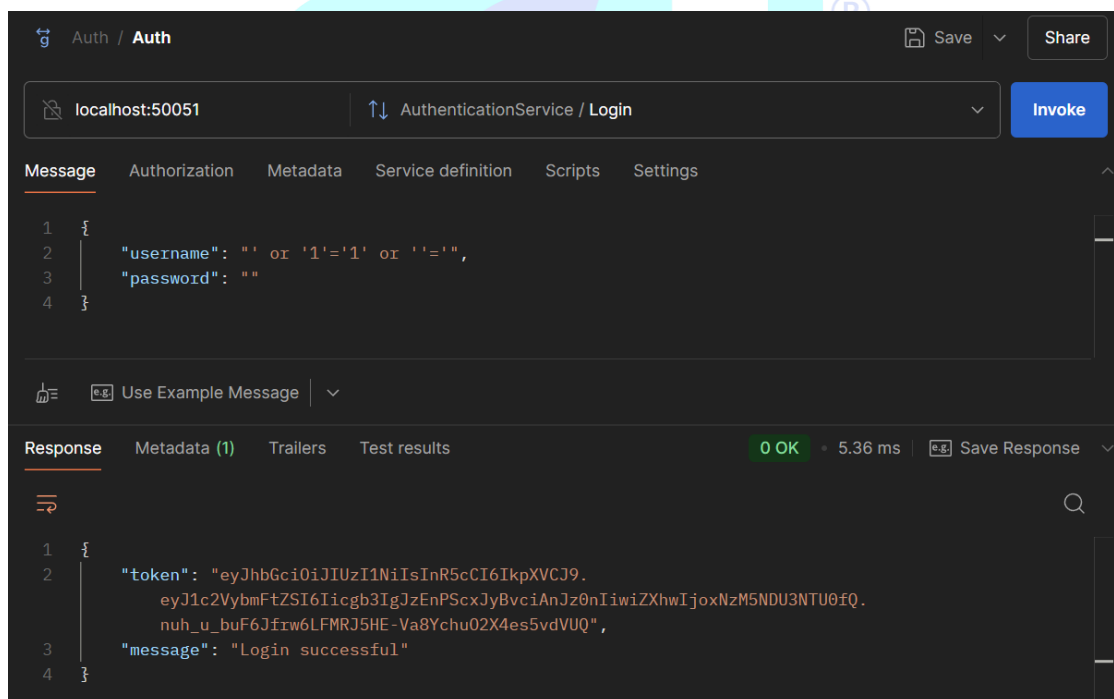


Figure 4.3 Sample security testing request in gRPC and validate manually for XPATH Injection



MUHAMAD ZAENUL HASAN BASRI

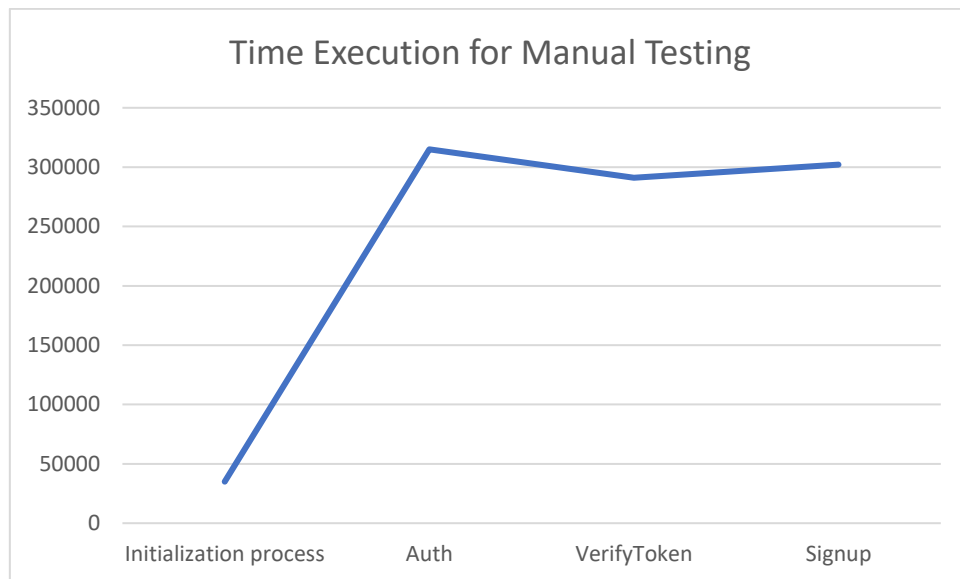


Figure 4.5 Time execution chart for manual testing

4.2.2. GSTF Security Testing

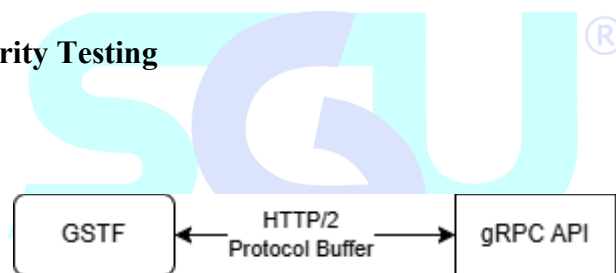


Figure 4.6 API Architecture in GSTF Testing

This section reviews the results of security testing conducted using GSTF (gRPC Security Testing Framework) implemented in Python and testing is done with the same test cases as manual testing. GSTF automates the testing process by generating payloads, creating stubs, sending requests, and validating responses in a structured manner with requests sent based on the architecture in Figure 4.6. The framework is designed to overcome the limitations of manual testing by increasing efficiency and improving vulnerability detection. The testing process involves the following major phases:

4.2.2.1. Extraction of Payload

In the first phase of the GSTF framework, payload extraction is performed from a YAML file, which is loaded based on its data type. This process is illustrated in the pseudocode in Figure 4.7 under the "Phase 1" tag and can be seen in the code section in Appendix A1, specifically in A1.2 and A1.4 and the sample payloads used in this testing can be found in A1.5 and A1.6.

4.2.2.2. Code Generation

In the second phase of the GSTF framework, the proto file is automate generated into a stub or function used to send requests and receive responses. This process is illustrated in the pseudocode in Figure 4.7 under the "Phase 2" tag and can be found in the code section in Appendix A1, specifically in A1.2 and A1.3 with proto file that can be seen in A1.7.

4.2.2.3. Test Case Creation

In the third phase of the GSTF framework, payload and stub combinations are generated based on data types to create comprehensive test cases. This process is illustrated in the pseudocode in Figure 4.7 under the "Phase 3" tag and can be found in the code section in Appendix A1, specifically in A1.2 and A1.6.

4.2.2.4. Execution

In the fourth phase of the GSTF framework, the generated test cases are executed, followed by automate evaluation and validation within the GSTF tools to gather insights on execution time and identified vulnerabilities with the output is report in excel format and further manual verification can be done to determine false positives that can be seen in section 4.4. This process is illustrated in the pseudocode in Figure 4.7 under the "Phase 4" tag and can be found in the code section in Appendix A1, specifically in A1.2.

```
FUNCTION start_scan(pathname, secure, url, metadata)
  SET global variables PROTO_PATHNAME, SECURE, URL, METADATA

  # Phase 1: Extraction payloads
  LOAD payloads from './core/modules/payloads.yaml' INTO temp_payloads

  # Phase 2: Code generation - Initialize gRPC Stub and Data
  INITIALIZE gRPC stub, request class names, and services using generateDataStub()

  FOR each service IN request class names:
    GET module and request parameters
    GENERATE test data for parameters

    # Phase 3: Test case creation
    FOR each parameter:
      IF parameter type is 'bool', SKIP
      GET relevant payloads for parameter type

      FOR each attack and payload:
        MODIFY test data with attack payload
        GET expected response

      # Phase 4: Execution, evaluation & validation
      CREATE gRPC request and CHECK for vulnerabilities

    PRINT scan completion time for service

  # Generate Excel Results
  CALL generate_excel_result()
END FUNCTION
```

Figure 4.7 Pseudocode for GSTF tool

The results of the execution phase 4.2.2.4 above explain the The GSTF framework efficiently identifies vulnerabilities such as SQLi, XSS, LFI, SSRF, RCE, XPath injection, command injection, and LDAP injection according to the payload used, surpassing the effectiveness of manual testing even though only a few were identified during testing. In addition, the framework uncovers complex issues, such as improper handling of edge cases, which are often overlooked during manual testing. The test results can be seen in Table 4.2, which was carried out 3 times that can be seen in Figure 4.8, with data collection based on the average test time.

Table 4.2 Time execution using GSTF security testing in gRPC

Endpoint	Execution time (ms) avg	Identified vulnerability
Initialization process	300	- SQL Injection - Cross Site Scripting (XSS) - Xpath Injection
Auth	1692	
Signup	2665	
VerifyToken	833	

```

PS D:\programming language\python\python_gRPC> python .\main.py -f .\proto\auth.proto -u localhost:50051
Initialize 303 ms

[#][LoginRequest] Data test: {'username': 'test', 'password': 'test'}
[+] [Vulnerable][SQL][LoginRequest] username = SQL Syntax Error: near "test": syntax error
[+] [Vulnerable][SQL][LoginRequest] username = SQL Syntax Error: no such function: SLEEP
[+] [Vulnerable][SQL][LoginRequest] username = SQL Syntax Error: no such function: BENCHMARK
[+] [Vulnerable][SQL][LoginRequest] password = SQL Syntax Error: unrecognized token: "test'"
[+] [Vulnerable][SQL][LoginRequest] password = SQL Syntax Error: no such function: SLEEP
[+] [Vulnerable][SQL][LoginRequest] password = SQL Syntax Error: no such function: BENCHMARK
[LoginRequest] Scan 1894 ms

[#][SignupRequest] Data test: {'username': 'test', 'password': 'test', 'email': 'test'}
[+] [Vulnerable][XSS][SignupRequest] username = Username 'test<script>alert('XSS')</script>' or email 'test' already exists
[+] [Vulnerable][XSS][SignupRequest] username = Username 'test'<script>alert('XSS')</script>' or email 'test' already exists
[+] [Vulnerable][XSS][SignupRequest] username = Username 'test' onmouseover=alert('XSS')' or email 'test' already exists
[+] [Vulnerable][XSS][SignupRequest] username = Username 'test<img src=x onerror=alert('XSS')>' or email 'test' already exists
[+] [Vulnerable][XSS][SignupRequest] email = Username 'test' or email 'test<script>alert('XSS')</script>' already exists
[+] [Vulnerable][XSS][SignupRequest] email = Username 'test' or email 'test'<script>alert('XSS')</script>' already exists
[+] [Vulnerable][XSS][SignupRequest] email = Username 'test' or email 'test' onmouseover=alert('XSS')' already exists
[+] [Vulnerable][XSS][SignupRequest] email = Username 'test' or email 'test<img src=x onerror=alert('XSS')>' already exists
[SignupRequest] Scan 2821 ms

[#][VerifyTokenRequest] Data test: {'token': 'test'}
[VerifyTokenRequest] Scan 844 ms
Excel file saved as vulnerability_list.xlsx

```

Figure 4.8 Sample GSTF testing results

The test results show that GSTF testing on gRPC runs very fast, especially during the actual testing process. In contrast, the initialization phase, such as importing proto files, is relatively slower compared to testing. In addition, the test duration is very efficient, with the longest test taking only 2665 milliseconds, as shown in Table 4.2 and Figure 4.9.

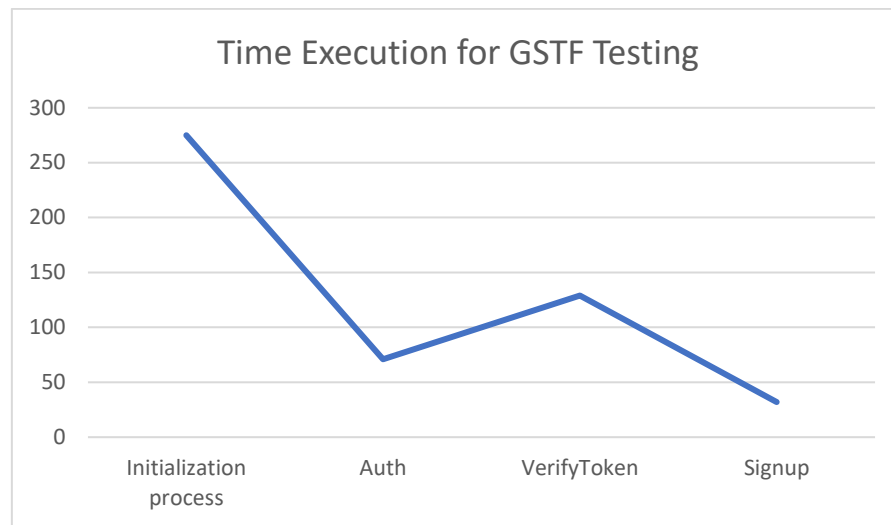


Figure 4.9 Time execution chart for GSTF testing

4.2.3. Comparison Between Manual and GSTF testing Based on Execution Time

The comparison between manual security testing and GSTF-based testing highlights notable distinctions. GSTF consistently outperformed manual testing in terms of execution speed and coverage. While manual testing was limited by human effort and time constraints, GSTF automated approach provided more comprehensive and consistent results. The comparison highlights significant differences in performance and efficiency between manual and GSTF testing approaches:

- **Time Efficiency:** GSTF testing is significantly faster than manual testing across all processes.
- **Consistency:** GSTF provides consistent test results with minimal errors, while manual testing is prone to human error and variability.

Figure 4.10 highlights the significant differences between GSTF and manual testing. GSTF shows superior efficiency, with an average initialization time of 300 milliseconds and a maximum test duration of 2665 milliseconds. In contrast, manual testing shows much slower performance, with an average initialization time of 35,000 milliseconds (35 seconds) and a maximum test duration of 315,000 milliseconds (315 seconds).

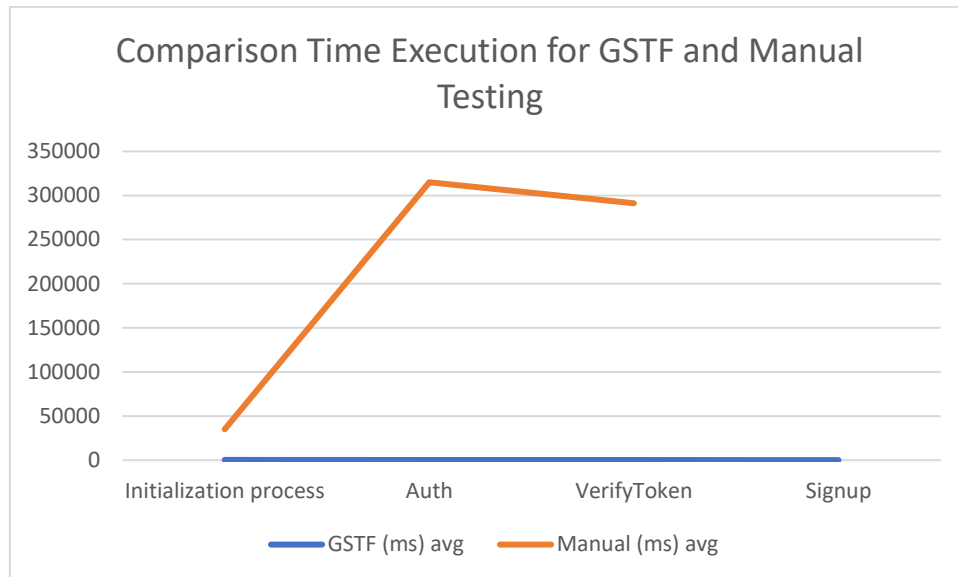


Figure 4.10 Comparison time execution between GSTF and manual testing

When comparing manual testing efficiency to the GSTF framework, the difference is stark. The data reveals a staggering 99% increase in testing speed with GSTF, calculated using formula (3.1) with details shown in Table 4.3. This dramatic jump not only confirms GSTF's potential to revolutionize testing efficiency but also surpasses initial expectations and this also answers the question from Q4 regarding this framework being able to provide significant efficiency improvements and even exceeds Hypothesis 4 which only ranges up to 80% according to the average test in the organization.

For companies that adopt GSTF, the future holds significant benefits. By drastically reducing testing time, resources can be reallocated to other critical areas, driving innovation and growth. Consistent and automated testing processes minimize human error, ensuring higher accuracy in detecting vulnerabilities. Additionally, GSTF's ability to seamlessly integrate with DevSecOps pipelines accelerates the software development lifecycle, enabling faster delivery of secure and reliable applications. This translates into cost savings, increased productivity, and a stronger competitive advantage in an ever-evolving technology landscape.

Table 4.3 Improvement Execution Time

Metric	Manual Testing	GSTF Testing	Improvement (%)
Execution Time (ms)	652000	5491	99%

4.2.4. Comparison of GSTF Security Testing Results with Other Tools

In this study, various tools were used to compare them with the tools developed by GSTF. However, the testing remains limited, as the tools used were primarily manual, and no other automated tools were identified for comparison. Table 4.4 presents the comparative results of similar tests conducted previously. The table highlights that the “Automate” test type is only applicable to GSTF, while other tools, such as Postman and gRPCurl, still require manual execution despite being tools. Although the comparison is not completely “apple to apple” due to the differences in testing approaches, the findings clearly show that GSTF provides significant advantages in testing efficiency and effectiveness.

Table 4.4 Comparison execution time of GSTF vs other tools which support gRPC

No	Tools	Type of Testing	Execution time (ms)
1	GSTF	Automate	5491
2	Postman	Manual	652000
3	gRPCurl	Manual	1008000

4.3. Evaluation

At this stage, re-evaluate the test results in section 4.2.2 above. To evaluate the GSTF framework, this study uses the success and error rate parameters in gRPC to measure its ability to handle invalid input, special cases, and vulnerabilities without causing any harm using data from GSTF test results 4.2.2 above. **The results** show that GSTF can manage tests with invalid input, even when errors occur that need to be converted to RPC status codes. Nevertheless, GSTF successfully completes the tests. The evaluation of the success and error rates is presented in Table 4.5, which shows a much lower success rate compared to errors.

Table 4.5 Success and error rate in testing

Metric	Total Execution	Output Rate	Percentage
Success Rate (%)	270	10	4%
Error Rate (%)	270	260	96%

However, all errors were successfully mapped by GSTF and used as validation references for the test results, as shown in Table 4.6. This indicates that, not only is it successful in executing each request, GSTF is also capable of handling errors as validation material.

Table 4.6 Status code in testing

Status Code	Total
OK	10
INTERNAL	22
UNAUTHENTICATED	99
UNKNOWN	4
ALREADY_EXISTS	135

4.4. Validation

At this stage, the process revalidates the test results from Section 4.2.2 above while proceeding to the next step after the validation method in GSTF Phase 4, which is the **manual verification phase**. The validation is performed by comparing the response data, both successful and error, with the expected results determined during the payload extraction phase and stored in a YAML file. GSTF **successfully validated** the response and identified vulnerabilities in the gRPC API reviewed in section 4.2.2 above. The output or results are stored in an Excel file for easy validation, with details status vulnerable only presented in Table 4.7 and full detail for GSTF Result in appendix A2. Notably, even when an RPC status error occurs, the response message can still be validated after being converted for readability. In the table presenting GSTF results with a vulnerable status, detected vulnerabilities include SQL Injection (sample row 1), XSS (sample row 101), and XPath Injection (sample row 31). Some of these vulnerabilities were also identified during **manual security testing**, although XSS was not detected during manual validation. Therefore, **verifying** each payload is essential to identify false positives, with detailed findings provided below.

- Manual verification – XSS

In this manual verification, testing is performed for all payloads marked as vulnerable by GSTF, specifically for the XSS attack type. When executing the request shown in Figure 4.11, the XSS payload was indeed reflected in the response. However, the payload or JavaScript did not execute as it would in a browser. Therefore, this vulnerability is considered a **false positive** since, despite being flagged as vulnerable, it could not be successfully exploited using the XSS payload.

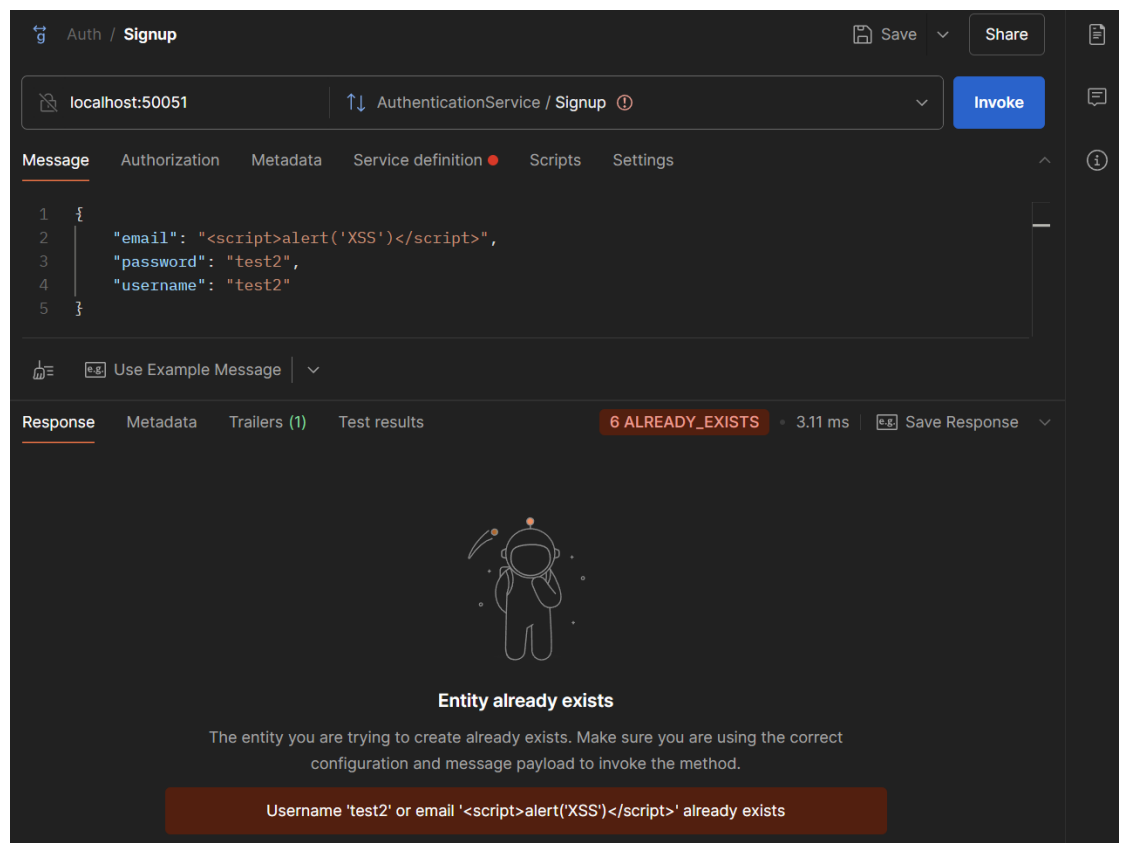


Figure 4.11 Sample request for manual verification - XSS

- Manual verification – SQL Injection

In this manual verification, testing was performed for all payloads marked as vulnerable by GSTF, especially for SQL Injection attacks. When executing the request shown in Figure 4.12, the API returned a database error, indicating that

an SQL Injection attack was possible. However, the attack performed by GSTF did not successfully retrieve the desired response, such as obtaining a valid token during login. Despite this, the vulnerability is still considered a **true positive**, as the API remains susceptible to SQL Injection.

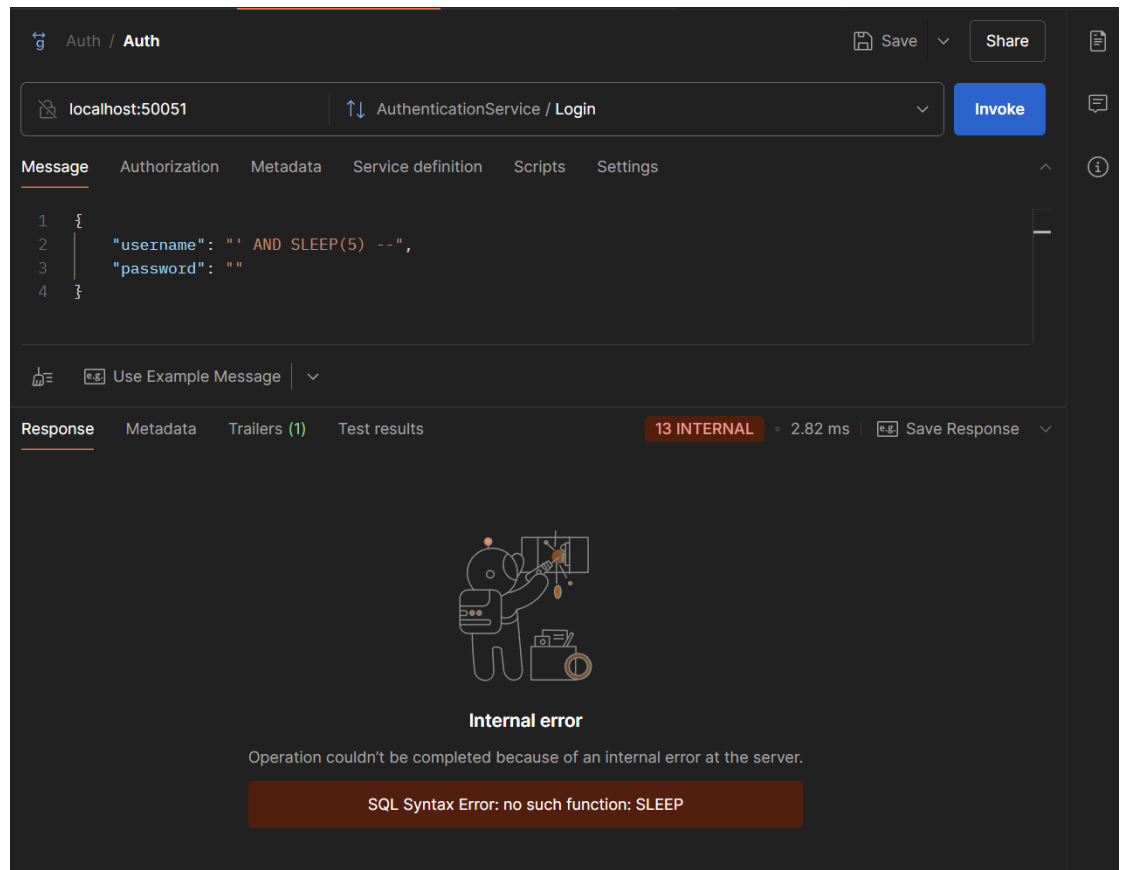


Figure 4.12 Sample request for manual verification - SQL Injection

- Manual verification – XPath Injection

In this manual verification, testing was performed for all payloads marked as vulnerable by GSTF, specifically for XPath Injection attacks. When executing the request shown in Figure 4.13 using one of GSTF XPath Injection payloads, the API immediately returned a login token, indicating that the payload successfully exploited the vulnerability. Therefore, this vulnerability is considered a **true positive**, as the exploitation was successful.



MUHAMAD ZAENUL HASAN BASRI

Table 4.7 GSTF test result coverage with Vulnerable status only

No	Status vulnerability	RPC code	RPC status	Attack title	Service	Vulnerable param	Payload
1	Vulnerable	INTERNAL	Error	SQLI	LoginRequest	username	'
5	Vulnerable	INTERNAL	Error	SQLI	LoginRequest	username	AND SLEEP(5) --
10	Vulnerable	INTERNAL	Error	SQLI	LoginRequest	username	AND BENCHMARK(50000000,MD5(1)) --
31	Vulnerable	OK	Success	XPATH INJECTION	LoginRequest	username	or '1'='1' or '='
32	Vulnerable	OK	Success	XPATH INJECTION	LoginRequest	username	' or 1=1 or '='
46	Vulnerable	INTERNAL	Error	SQLI	LoginRequest	password	'
50	Vulnerable	INTERNAL	Error	SQLI	LoginRequest	password	' AND SLEEP(5) --
55	Vulnerable	INTERNAL	Error	SQLI	LoginRequest	password	' AND BENCHMARK(50000000,MD5(1)) --
76	Vulnerable	OK	Success	XPATH INJECTION	LoginRequest	password	' or '1'='1' or '='
77	Vulnerable	OK	Success	XPATH INJECTION	LoginRequest	password	' or 1=1 or '='
101	Vulnerable	ALREADY EXISTS	Error	XSS	SignupRequest	username	<script>alert('XSS')</script>
102	Vulnerable	ALREADY EXISTS	Error	XSS	SignupRequest	username	'><script>alert('XSS')</script>
103	Vulnerable	ALREADY EXISTS	Error	XSS	SignupRequest	username	" onmouseover=alert('XSS')>
104	Vulnerable	ALREADY EXISTS	Error	XSS	SignupRequest	username	
191	Vulnerable	ALREADY EXISTS	Error	XSS	SignupRequest	email	<script>alert('XSS')</script>
192	Vulnerable	ALREADY EXISTS	Error	XSS	SignupRequest	email	'><script>alert('XSS')</script>
193	Vulnerable	ALREADY EXISTS	Error	XSS	SignupRequest	email	" onmouseover=alert('XSS')>
194	Vulnerable	ALREADY EXISTS	Error	XSS	SignupRequest	email	