Eduvest – Journal of Universal Studies
Volume 5 Number 10, October, 2025
p- ISSN 2775-3735- e-ISSN 2775-3727

Contract Testing: A Framework for Security Evaluation in gRPC

Muhamad Zaenul Hasan Basri, Charles Lim, Kalpin Erlangga Silaen
Swiss German University, Indonesia
Email: muhamad.basri@student.sgu.ac.id, charles.lim@sgu.ac.id, kalpin.erlangga@lecturer.sgu.ac.id

ABSTRACT
The growth of APIs, including SOAP, REST, and gRPC, has made security a critical priority, with incidents such as those in the 2023 Paloalto report highlighting the financial losses resulting from API breaches. While existing tools focus on REST APIs, gRPC remains underserved, requiring time-consuming manual testing. This research aims to address this gap by proposing a security testing framework tailored to gRPC, integrating automated methods that DevSecOps can use to improve efficiency. gRPC, built on HTTP/2, uses a binary message format and client stubs generated from proto files, creating unique challenges for testing. The methodology involves extracting payloads, generating stubs from proto files, creating test cases, and executing automated tests for vulnerabilities such as SQL Injection and XSS. By analyzing gRPC components and adapting common API security practices, the framework identifies vulnerabilities, streamlines testing, and reduces manual effort. It automates processes such as payload generation and stub generation, enabling faster and more reliable testing compared to traditional methods. Results demonstrate that GSTF reduces testing time by 99% compared to manual methods while maintaining comprehensive coverage. Although some false positives were noted, the framework effectively identifies critical vulnerabilities and integrates seamlessly with DevSecOps pipelines. This approach not only improves testing efficiency by significantly reducing time but also sets a benchmark for secure API development. This study provides a practical solution for enhancing gRPC security, offering significant efficiency gains and establishing a foundation for future advancements in API security automation.

KEYWORDS *gRPC, API security, automated testing, security framework, DevSecOps*

INTRODUCTION

The development of APIs has expanded significantly, encompassing various types such as SOAP, REST, and gRPC (Newton Hedelin, 2024; Sharma, 2021). Based on Paloalto's 2023 report on the consequences of API security incidents, which can be seen in Figure Error! No text of specified style in document.., it was noted that team members suffered financial losses after the incident. This caused substantial financial setbacks, especially for the companies affected (Tan & Zhu, 2022). Currently, API security has become a very important requirement, and security testing needs to be carried out to mitigate risks (Alharbi & Moulahi, 2023; Jangam et al., 2022). The aspiration is that this research can assist other communities in enhancing the security of their own APIs, particularly focusing on the gRPC API (Owen, 2025).

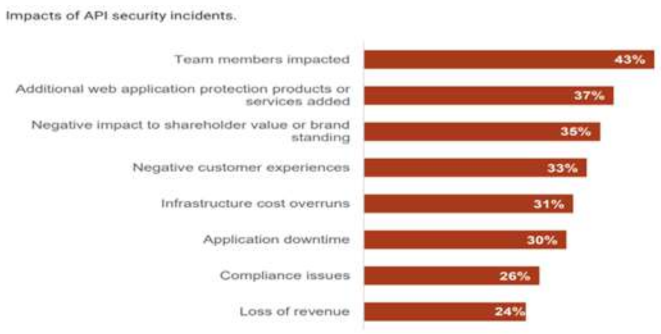


Figure Error! No text of specified style in document.. **API Security Incident 2023**

Currently, the researcher works as a security tester for applications, particularly focusing on APIs (Mousavi et al., 2025). Despite searching for a few years, the researcher found that API security testing tools are generally built for REST APIs and do not support gRPC natively, making testing more difficult (Basri & Hasan, 2024; Thiyagarajan et al., n.d.). Consequently, the testing process is conducted manually, resulting in a time-consuming and repetitive experience (Nama et al., 2021). This research aims to introduce a new framework for security evaluation in gRPC, then creating automated methods and combining with DevSecOps to automate them as well, to streamline and enhance the efficiency of the testing process. DevSecOps is a combination of development, security and operations, which makes the process faster and secure because the process is carried out automatically (Abiona et al., 2024; Sinan et al., 2025). With automated process and the proposed new framework, it can be used to carry out security assessments by carrying out security testing when the application has been developed to reduce the risk of vulnerabilities after the initial process and can reduce testing time significantly, especially in this project which focuses on developing API gRPC. Google Remote Procedural Call (gRPC) is an open-source RPC protocol developed by Google (Chen et al., 2023; Zhang et al., 2023).

gRPC has a very high processing speed and runs on the HTTP/2 protocol with messages converted into binary message format (Khan & Ahamad, 2024). gRPC also uses clients to generate functions using proto files provided by the server, which are called stub (client generated in a particular language) or it can be called a gRPC client (Giretti, 2022; Štefanič, 2021). Then, this function will later be used in making requests and getting responses. This makes the security testing of gRPC different from APIs in general, because the client will send a request using a function that has previously been created and contains a message which has been converted into a protocol buffer message and later the server will send back a response in the form of a protocol buffer that will be converted using the function response on the client (Frantz et al., 2024; Sangwai et al., 2023).

Conducting a thorough examination of how gRPC works compared to another APIs is essential (Ali, 2024). The results of this comparative analysis will produce a set of benchmark components used in gRPC testing and help in developing a new framework as well as being an important benchmark in determining which components will be tested. In addition, security weaknesses and vulnerabilities commonly identified in another APIs will be identified and adapted to facilitate analogous testing in the context of gRPC (Basri & Hasan, 2024). This project aims to develop a new security testing framework tailored to the gRPC process.

This research is intended to serve as a reference for security professionals in testing gRPC APIs by understanding the proposed framework in this thesis and to be able to solve the problem of long testing time efficiency that I get when doing gRPC testing manually. The approach involves comparing existing frameworks, identifying their weaknesses in relation to gRPC, and redesigning a new framework to align with gRPC's specific functionality.

Based on the book "Modern API Design with gRPC" by Arora (2024), gRPC encompasses various crucial aspects for its functionality. This research connects these aspects with the research problems faced, namely "Limited tools and methods for gRPC security testing" and "Security testing is manual and inefficient," ultimately leading to ineffective testing. These issues relate to error handling and security, which are the primary focus areas of this study. One contributing factor to the low efficiency of security testing in gRPC is the lack of available tools and methods, often resulting in manual testing. The current testing process is performed manually, making it time-consuming, especially when running multiple test cases for each service, which typically requires creating stubs from proto files each time a test is conducted. Additionally, repetitive testing, such as injection testing, further extends the time needed to execute the overall testing process.

The objectives of this research include identifying key gRPC components for security evaluation by developing customized test cases, highlighting gaps in existing security testing frameworks while demonstrating the benefits of the proposed framework, and defining a new framework for gRPC security testing that allows for automated test case generation by integrating stub objects with payloads. This study also aims to validate the applicability and effectiveness of the framework through implementation in a tool capable of automating security testing. Consequently, this research is expected to improve efficiency, reduce testing time, and provide better visibility of test results compared to manual methods. The study will also consider existing limitations, such as focusing on specific components, testing in controlled environments, and tailoring the framework specifically for gRPC, which may limit generalizability. The significance of this study lies in its role in addressing cybersecurity challenges associated with gRPC APIs, which are critical for organizations relying on inter-service communication. By identifying vulnerable components and developing targeted test cases, this research aims to enhance the security of gRPC APIs and reduce the risk of exploitation and data breaches.

METHOD

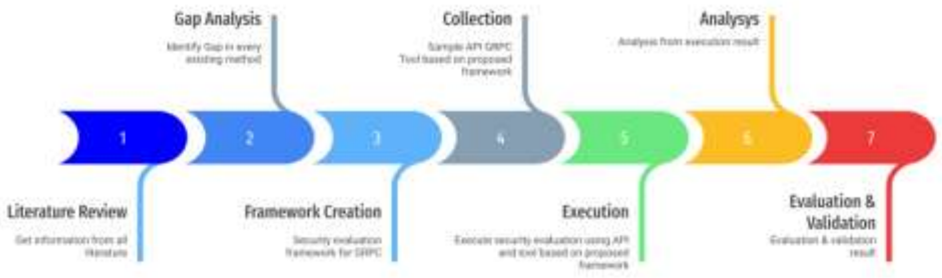

Gap Analysis
Identify Gap in every existing method
Collection
Sample API GRPC
Tool based on proposed framework
Analysys
Analysis from execution result
1
2
3
4
5
6
7
Literature Review
Get information from all literature
Framework Creation
Security evaluation framework for GRPC
Execution
Execute security evaluation using API and tool based on proposed framework
Evaluation & Validation
Evaluation & validation result

Figure 2. Research methodology

The research methodology in this case involves conducting a literature review and performing a security evaluation using the proposed framework developed in this thesis. The focus of these activities is outlined in the research framework, as depicted in Figure 2., which guides the process to achieve the final results in the security evaluation of gRPC.

The GSTF (gRPC Security Testing Framework) identified to be proposed at this stage can be seen in Figure Error! No text of specified style in document.. In this framework, it is divided into several parts including Extraction of Payload, Code Generation, Test Case Creation and Execution. In the "Execution" phase, automation will be implemented since the required objects, such as payloads, stubs, and test cases, have already been created in before process and the payload will be adjusted based on its data type based that will be created as a test case. At this stage, the process involves simply running these objects and validating the results. Regarding the success criteria for implementing this framework, since the primary issue in this research problem the inefficiency of manual testing, which results in prolonged testing times, it is expected that this framework will significantly reduce testing time by 60% until 80% or even more depending on the context. The exact percentage of improvement depends on factors such as:

a. Test Case Complexity: Automation is more effective for repetitive, time-consuming tasks, leaving complex, exploratory tests to manual efforts.
b. Coverage: Automated tools can increase test coverage, catching more issues than manual testing would in the same timeframe.
c. Speed: Automation speeds up execution, especially for regression testing and large test suites.
d. Consistency: Automation reduces human error, ensuring tests are run in the same way every time.

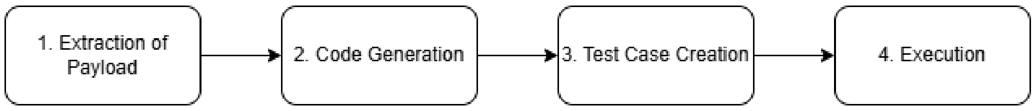


Figure Error! No text of specified style in document.. **GSTF (gRPC Security Testing Framework)**

Payload Extraction

This stage involves extracting the payload for testing, compiling it into a JSON/YAML file, and mapping important attributes such as data type, attack name, payload data, and expected results according to the Figure . The process includes:

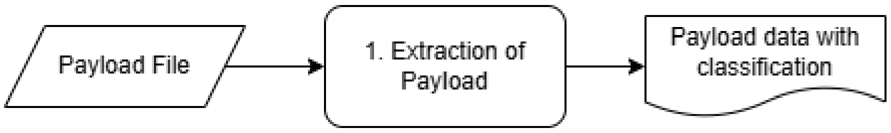


Figure 4. Phase 1 - Extraction of payload

a. Input: Collecting payloads for common vulnerabilities such as SQL injection (SQLi), cross-site scripting (XSS), local file inclusion (LFI), and server-side request forgery (SSRF) in JSON/YAML format.

b. Process: In this process, the input file will be extracted by mapping the payload on each attribute including data type, attack name, payload data, and expected results.
c. Output: The output that will be provided by the payload extraction process is an object that has been mapped based on attributes.

Code Generation

The proto file is processed to generate stubs, gRPC functions that handle requests and responses according to the Figure. Using Python, automation ensures all request methods are seamlessly integrated with the test payload. The steps are:

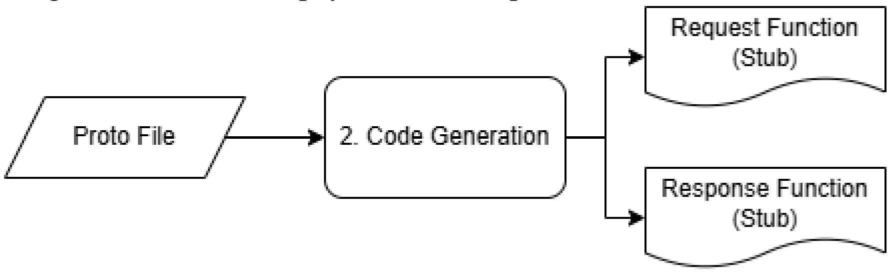


Figure 5. Phase 2 - Code generation

a. Input: The input to this process is a proto file from the gRPC service.
b. Process: In this process, a proto file will be generated to become a request and response function.
c. Output: The main output of this process is a stub or request and response function.

Test Case Creation

The stub parameters are used to generate test data, including valid and invalid inputs, based on vulnerabilities in payloads. The payload from the JSON/YAML file is paired with the stub parameters to create test cases and serialized into a protocol buffer according to the Figure 6.. The steps include:

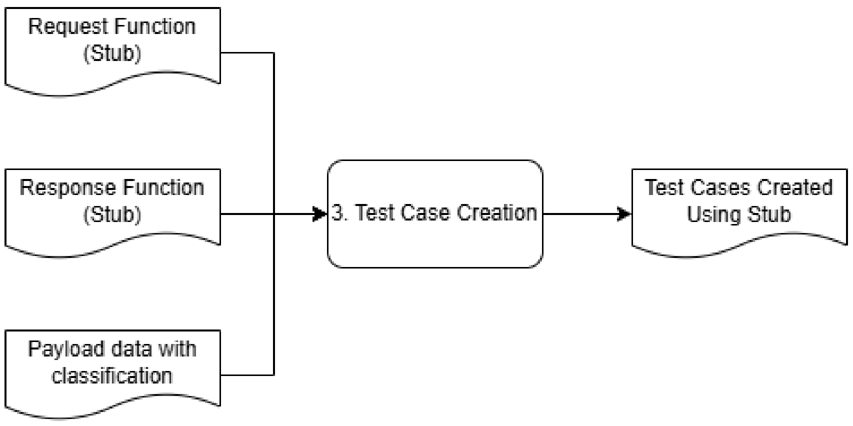


Figure 6. Phase 3 - Test case creation

a. Input: The input to this process is the payload generated from Phase 1 and the stub generated from phase 2.

b. Process: In this process, a payload will be combined with a stub or request and response function in gRPC based on the data type to become a test case and convert to protocol buffer.
c. Output: an object in the form of a combined test case and payload that can be used for the next phase.

RESULTS AND DISCUSSION

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:

a. Manual security testing using conventional human-driven techniques.
b. GSTF security testing, leveraging automation for efficiency.
c. Comparison of manual and GSTF testing, focusing on execution time
d. 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.

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, including:

a. SQL Injection
b. Cross Site Scripting (XSS)
c. Local File Inclusion (LFI)
d. Server-Side Request Forgery (SSRF)
e. Remote Command Execution (RCE)
f. Xpath Injection
g. Command Injection
h. 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.

Manual Security Testing

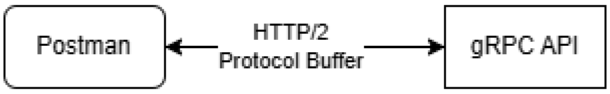


Figure 7. API Architecture in manual testing using postman

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 7.7. 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:

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

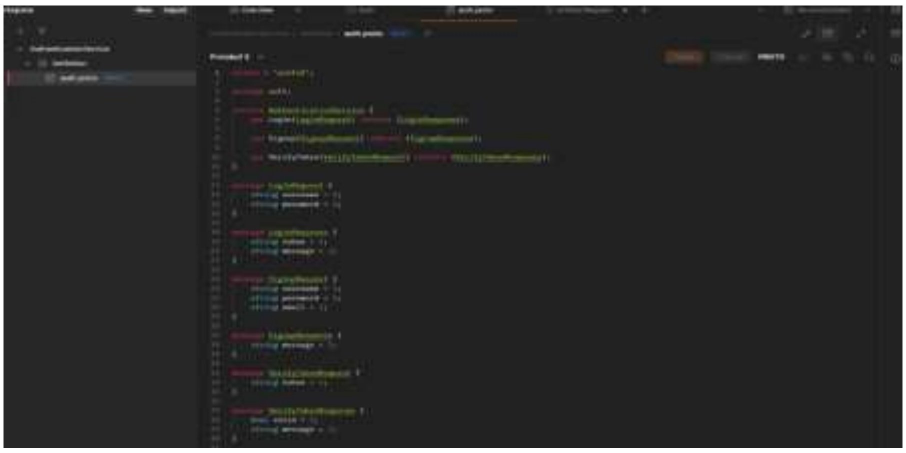
A summary of the results of manual security testing is presented in the

Table *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

Figure8 and the sample request testing with manual validation which can be seen in Figure 9.9 and Figure 10.10.

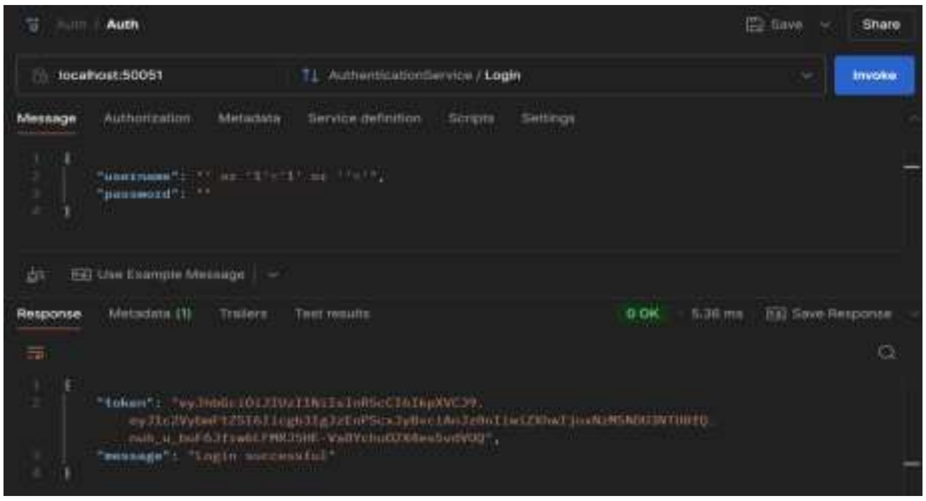
Table 1. Time execution for manual security testing in gRPC

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



```
syntax = "proto3";

package auth;

service AuthenticationService {
    rpc Login(LoginRequest) returns (LoginResponse);

    rpc Signup(SignupRequest) returns (SignupResponse);

    rpc VerifyToken(VerifyTokenRequest) returns (VerifyTokenResponse);
}

message LoginRequest {
    string username = 1;
    string password = 2;
}

message LoginResponse {
    string token = 1;
    string message = 2;
}

message SignupRequest {
    string username = 1;
    string password = 2;
    string email = 3;
}

message SignupResponse {
    string message = 1;
}

message VerifyTokenRequest {
    string token = 1;
}

message VerifyTokenResponse {
    bool valid = 1;
    string message = 2;
}
```

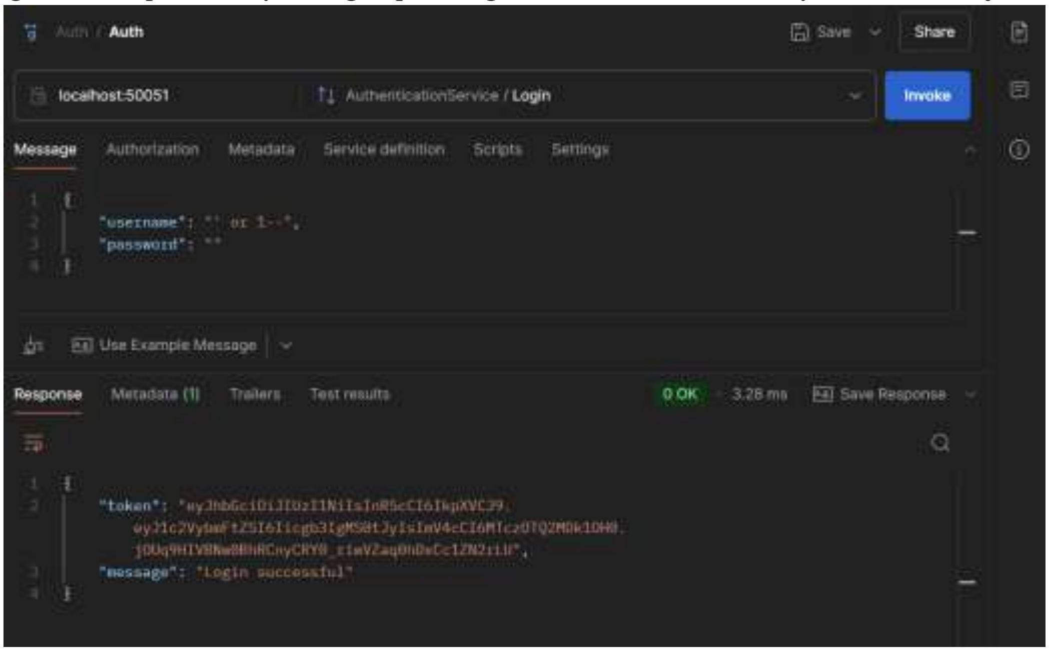
Figure 8. Initialize import auth proto file



```
{
    "username": "' or '1'='1' or ''='",
    "password": ""
}
```

```
{
    "token": "eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.
        eyJ1c2VybmFtZSI6Iicgb3IgJzEnPScxJyBvciAnJz0nIiwiZXhwIjoxNzM5NDQ3NTY0fQ.
        nuh_u_buF6JfswбLFMRJ5HE-VaBYchuOZX6ew5vdVOQ",
    "message": "Login successful"
}
```

Figure 9. Sample security testing request in gRPC and validate manually for XPATH Injection



```
{
    "username": "' or 1--",
    "password": ""
}
```

```
{
    "token": "eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.
        eyJ1c2VybmFtZSI6Iicgb3IgMS0tJyIsImV4cCI6MTczOTQ2MDk1OH0.
        jOUq9HIV8Nw8NhRCnyCRY8_ziwVZaq0nDeCc1ZN2i1U",
    "message": "Login successful"
}
```

Figure 10. Sample security testing request in gRPC and validate manually for SQL Injection

The test results show that manual testing in gRPC is very time-consuming, especially during the actual testing process. In contrast, the initialization process, such as importing proto files, is relatively fast. However, the overall test duration remains long, with the longest recorded time being 315,000 milliseconds (or 315 seconds) for eight test scenarios, as illustrated in Figure11.

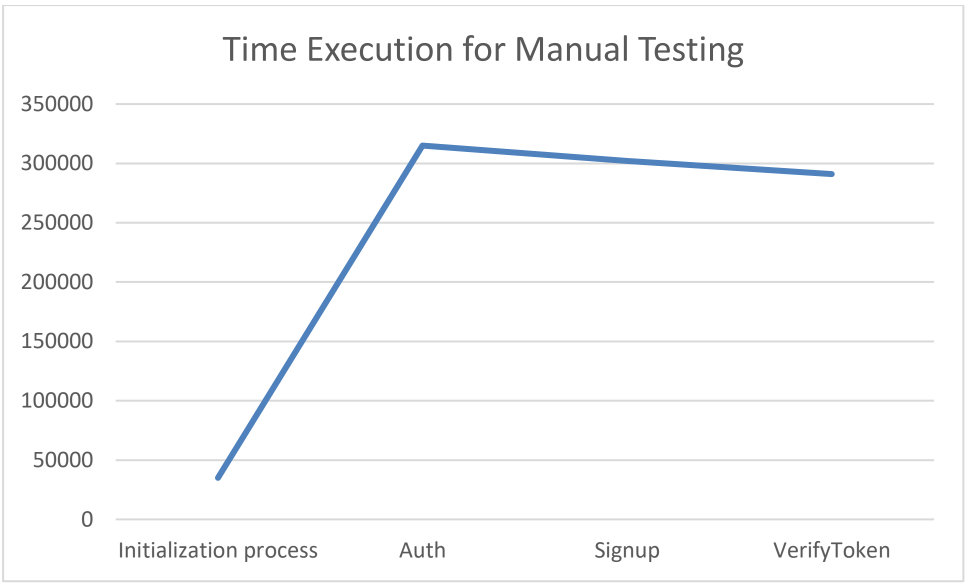


Figure 11. Time execution chart for manual testing

GSTF Security Testing

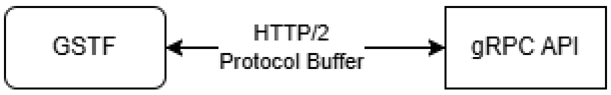


Figure 12. API Architecture in GSTF Testing

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 12. 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:

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 Figure13 under the "Phase 1" tag and can be seen in the code section in Appendix **Error! Reference source not found.**, specifically in **Error! Reference source not found.** and **Error! Reference source not found.** and the sample payloads used in this testing can be found in **Error! Reference source not found.** and **Error! Reference source not found.**.

Code Generation

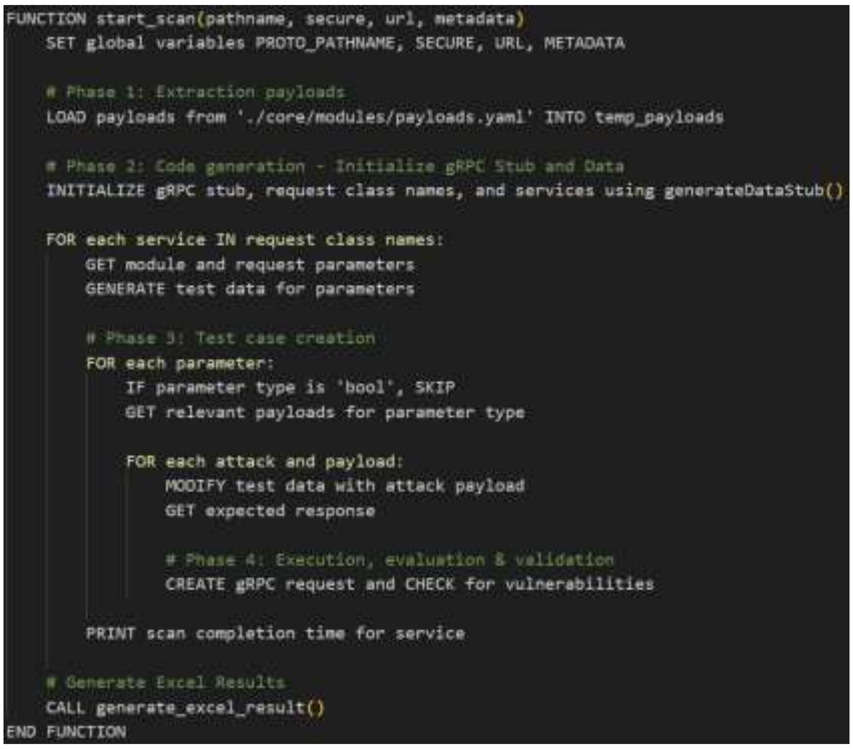
In the second phase of the GSTF framework, the proto file is automate generated into a stub or function used to send requests and receive responses. This process is illustrated in the pseudocode in Figure13 under the "Phase 2" tag and can be found in the code section in Appendix **Error! Reference source not found.**, specifically in **Error! Reference source not found.** and **Error! Reference source not found.** with proto file that can be seen in **Error! Reference source not found.**.

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 Figure13 under the "Phase 3" tag and can be found in the code section in Appendix **Error! Reference source not found.**, specifically in **Error! Reference source not found.** and **Error! Reference source not found.**.

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. This process is illustrated in the pseudocode in Figure 13 under the "Phase 4" tag and can be found in the code section in Appendix **Error! Reference source not found.**, specifically in **Error! Reference source not found.**.



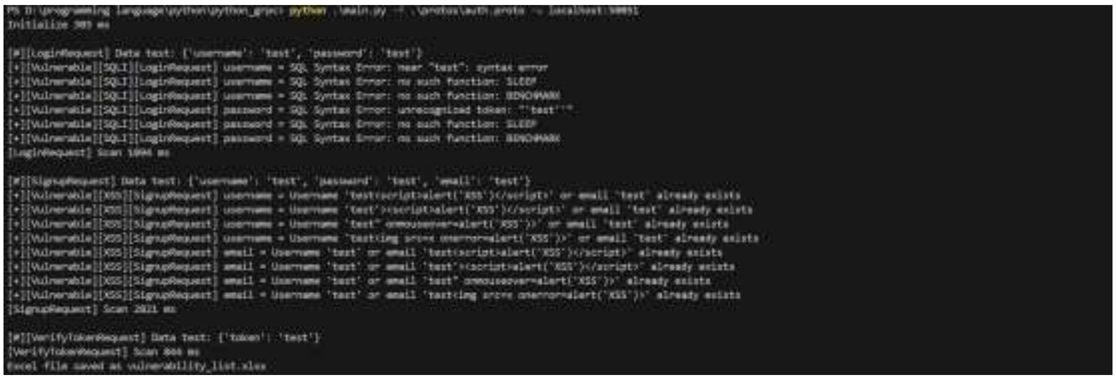
```
FUNCTION start_scan(pathname, secure, url, metadata)
    SET global variables PROTO_PATHNAME, SECURE, URL, METADATA

    # 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 13. Pseudocode for GSTF tool

The results of the execution phase 0 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 2, which was carried out 3 times that can be seen in Figure **1***Error! No text of specified style in document.*.14, with data collection based on the average test time.

Table 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 .\protos\auth.proto -u localhost:50051
Initialize 305 ms

[#][LoginRequest] Data test: {'username': 'test', 'password': 'test'}
[+][Vulnerable][SQLI][LoginRequest] username = SQL Syntax Error: near "test": syntax error
[+][Vulnerable][SQLI][LoginRequest] username = SQL Syntax Error: no such function: SLEEP
[+][Vulnerable][SQLI][LoginRequest] username = SQL Syntax Error: no such function: BENCHMARK
[+][Vulnerable][SQLI][LoginRequest] password = SQL Syntax Error: unrecognized token: "'test'"
[+][Vulnerable][SQLI][LoginRequest] password = SQL Syntax Error: no such function: SLEEP
[+][Vulnerable][SQLI][LoginRequest] password = SQL Syntax Error: no such function: BENCHMARK
[LoginRequest] Scan 1004 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 866 ms
Excel file saved as vulnerability_list.xlsx
```

Figure 1 Error! No text of specified style in document.. **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 *2*3 and Figure15.

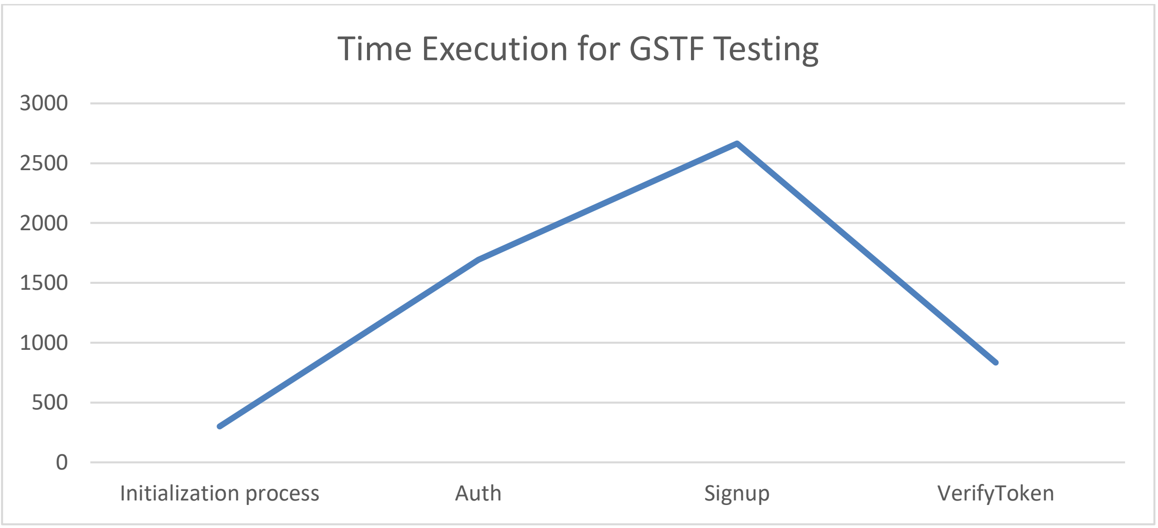


Figure 15. Time execution chart for GSTF testing

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.

Figure16 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).

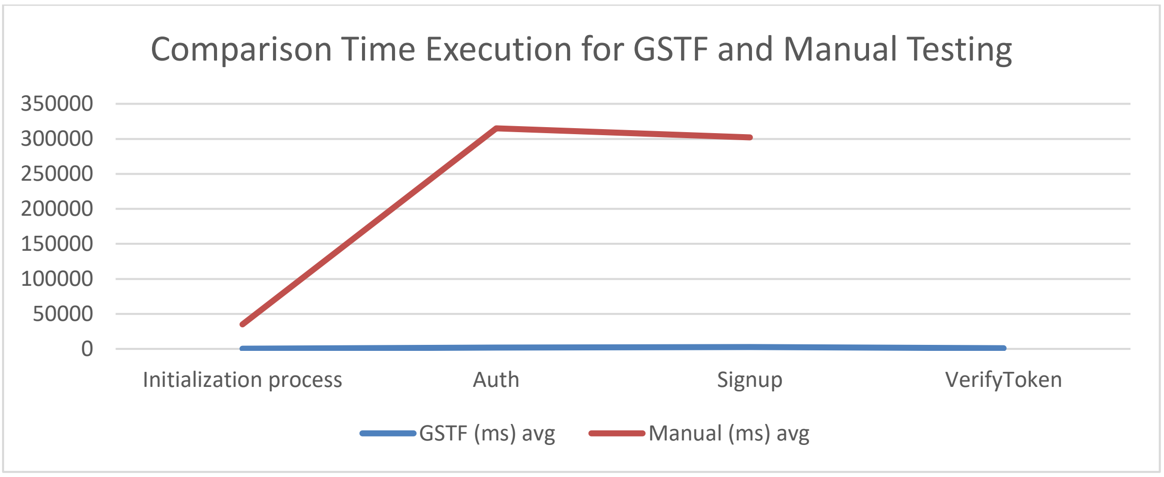


Figure 16. Comparison time execution between GSTF and manual testing

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 **Error! Reference source not found.** with details shown in Table 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 **Error! Reference source not found.** 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 3. Improvement Execution Time

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

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 Error*! No text of specified style in document.*. 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 Error! No text of specified style in document.. **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

Evaluation

At this stage, re-evaluate the test results in section. 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. 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, which shows a much lower success rate compared to errors.

Table 4. 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 5. This indicates that, not only is it successful in executing each request, GSTF is also capable of handling errors as validation material.

Table 5 Status code in testing

Status Code	Total
OK	10

INTERNAL	22
UNAUTHENTICATED	99
UNKNOWN	4
ALREADY_EXISTS	135

CONCLUSION

GSTF demonstrates a clear advantage over manual testing for security evaluation, achieving a 99% reduction in execution time while consistently detecting vulnerabilities across multiple endpoints, including complex issues such as improper handling of edge cases. Unlike manual testing, which is slow, inconsistent, and heavily dependent on human expertise, GSTF offers scalability and reliability, despite occasional higher error rates caused by invalid inputs requiring RPC status code conversion and the possibility of false positives confirmed through manual proof-based verification. Future research could focus on enhancing GSTF's accuracy by improving input validation mechanisms and developing advanced false-positive mitigation techniques to further increase trust and effectiveness in large-scale automated API security testing.

REFERENCES

Abiona, O. O., Oladapo, O. J., Modupe, O. T., Oyeniran, O. C., Adewusi, A. O., & Komolafe, A. M. (2024). The emergence and importance of DevSecOps: Integrating and reviewing security practices within the DevOps pipeline. *World Journal of Advanced Engineering Technology and Sciences*, *11*(2), 127–133.

Alharbi, S. J., & Moulahi, T. (2023). API security testing: the challenges of security testing for restful APIs. *International Journal of Innovative Research in Science Engineering and Technology*, *8*(5), 1485–1499.

Ali, O. (2024). *Popular API Technologies: REST, GraphQL, and gRPC*.

Arora, S., Bhardwaj, A., Kukkar, A., & Kaur, S. (2024). A Comparative Analysis of Communication Efficiency: REST vs. gRPC in Microservice-Based Ecosystems. *2024 International Conference on Emerging Innovations and Advanced Computing (INNOCOMP)*, 621–626.

Basri, M. Z. H., & Hasan, M. Z. (2024). Analysis and security testing for grpc. *No. January*, 2020–2023.

Chen, J., Wu, Y., Lin, S., Xu, Y., Kong, X., Anderson, T., Lentz, M., Yang, X., & Zhuo, D. (2023). Remote procedure call as a managed system service. *20th USENIX Symposium on Networked Systems Design and Implementation (NSDI 23)*, 141–159.

Frantz, R., García, J. S., Copik, M., Monroy, I. T., Olmos, J. J. V., Bloch, G., & Di Girolamo, S. (2024). Protocol Buffer Deserialization DPU Offloading in the RPC Datapath. *SC24-W: Workshops of the International Conference for High Performance Computing, Networking, Storage and Analysis*, 886–895.

Giretti, A. (2022). Create a gRPC-web service from a gRPC-service with ASP. NET Core. In *Beginning gRPC with ASP. NET Core 6: Build Applications using ASP. NET Core Razor Pages, Angular, and Best Practices in. NET 6* (pp. 395–418). Springer.

Jangam, S. K., Karri, N., & Muntala, P. S. R. P. (2022). Advanced API Security Techniques and Service Management. *International Journal of Emerging Research in Engineering and Technology*, *3*(4), 63–74.

Khan, I., & Ahamad, M. K. (2024). *Enhancing Security and Performance of gRPC-Based Microservices using HTTP/3 and AES-256 Encryption*.

Mousavi, Z., Islam, C., Babar, M. A., Abuadbba, A., & Moore, K. (2025). Detecting misuse of security APIs: A systematic review. *ACM Computing Surveys*, *57*(12), 1–39.

Nama, P., Meka, N. H. S., & Pattanayak, N. S. (2021). Leveraging machine learning for intelligent test automation: Enhancing efficiency and accuracy in software testing. *International Journal of Science and Research Archive*, *3*(01), 152–162.

Newton Hedelin, M. (2024). *Benchmarking and performance analysis of communication protocols: A comparative case study of gRPC, REST, and SOAP*. KTH Royal Institute of Technology.

Owen, A. (2025). *Microservices Architecture and API Management: A Comprehensive Study of Integration, Scalability, and Best Practices*.

Sangwai, A., Sapale, S., Ghodake, S., & Jadhav, R. (2023). Barricading system-system communication using gRPC and protocol buffers. *2023 5th Biennial International Conference on Nascent Technologies in Engineering (ICNTE)*, 1–5.

Sharma, S. (2021). *Modern API Development with Spring and Spring Boot: Design highly scalable and maintainable APIs with REST, gRPC, GraphQL, and the reactive paradigm*. Packt Publishing Ltd.

Sinan, M., Shahin, M., & Gondal, I. (2025). Integrating Security Controls in DevSecOps: Challenges, Solutions, and Future Research Directions. *Journal of Software: Evolution and Process*, *37*(6), e70029.

Štefanič, M. (2021). *Developing the guidelines for migration from RESTful microservices to gRPC*. Brno.

Tan, Y., & Zhu, Z. (2022). The effect of ESG rating events on corporate green innovation in China: The mediating role of financial constraints and managers' environmental awareness. *Technology in Society*, *68*, 101906.

Thiyagarajan, G., Bist, V., & Nayak, P. (n.d.). Strengthening gRPC Security in Microservices: A Proxy-based Approach for mTLS, JWT, and RBAC Enforcement. *International Journal of Computer Applications*, *975*, 8887.

Zhang, L., Pang, K., Xu, J., & Niu, B. (2023). High performance microservice communication technology based on modified remote procedure call. *Scientific Reports*, *13*(1), 12141.