THE COMPUTERTECH
(An International Peer Review Journal)
VOLUME 11; ISSUE 1 (JAN-JUNE); (2025)
WEBSITE: THE COMPUTERTECH

Advanced AI Techniques for Protecting GraphQL APIs Against Injection and DoS Attacks

Katarina Vrettos[1*]

[1]Zagreb Institute of Computational Intelligence, CROATIA

Abstract

GraphQL's adaptability is great for speedy data retrieval, but it also brings new security risks that standard API safeguards don't always cover. Attacks like denial-of-service and data exfiltration through injection can be caused by malicious GraphQL queries that take advantage of the language's dynamic nature. When it comes to protecting against complex, context-aware assaults, current solutions like rate restriction, static analysis, and general-purpose Web Application Firewalls fall short. This study introduces a new method for detecting malicious GraphQL queries in real-time using artificial intelligence. Our approach integrates static analysis with machine learning techniques. These techniques include CNNs, Multilayer Perceptrons, and Random Forests for classification, Sentence Transformers (SBERT and Doc2Vec) for contextual embedding of query payloads, and Large Language Models (LLMs) for dynamic schema-based configuration. Our system architecture is described in depth, along with optimization solutions for production scenarios, such as ONNX Runtime and par-allel processing. We also assess how well our detection models and the system perform under stress. The results show that a number of threats, such as SQL injection, OS command injection, and XSS attacks, can be accurately detected with a high degree of accuracy, and that threats like DoS and SSRF attempts may be effectively mitigated. An effective and flexible method for strengthening the security of GraphQL APIs is developed in this study.

Keywords: Advanced AI; Techniques for Protecting; GraphQL; APIs; DoS Attacks

Introduction

The efficiency with which GraphQL enables customers to request precise data, which optimizes data transport, has led to its increased usage. On the other hand, traditional security methods for static APIs sometimes fail to protect GraphQL queries due to their dynamic nature, which poses new security issues because of this adaptability, bad actors may launch sophisticated assaults. Server-Side Request Injection (SSRI), Denial-of-Service (DoS) via resource-intensive queries, XSS, schema introspection for vulnerability discovery, and injection attacks (e.g., SQL, OS commands) are common vectors. Falsification (SSRF). Protecting GraphQL-based systems from these dangers is of the utmost importance.

Web application firewalls (WAFs) that are designed for general use, static analysis, and rate limitation are not enough when it comes to security. Their inability to offer real-time analysis of query payloads, difficulty configuring properly, lack of knowledge of GraphQL's semantics, and trouble with dynamic assaults are some of their major drawbacks. This emphasizes the obvious necessity for smarter, more adaptable security measures that can analyze GraphQL queries in real-time, even though they are dynamic.

In order to strengthen GraphQL's defenses against injection, DoS, and complexity-based threats, this article suggests a system that is powered by artificial intelligence. This framework combines machine learning with static analysis. Most notably, there have been:

• A hybrid detection framework designed specifically for the structure of GraphQL.

GraphQL Schema Definition Language (SDL) analysis and dynamically generated context-aware static analysis rules is a unique use of Large Language Models (LLMs).

Effective contextual vector embedding of potentially malicious query payloads enables pattern-based detection through the employment of Sentence Transformers (particularly SBERT for injection attacks and Doc2Vec for XSS).

• Detecting injection and XSS attacks by the use of Sentence Transformers (SBERT, Doc2Vec) to include contextual vectors in query payloads.

To accurately detect threats based on payloads, an evaluation of multiple machine learning classifiers (CNN, Random Forest, MLP) was conducted.

• An architecture for the system that is both scalable and designed to function well in production settings.

The rest of this paper describes the relevant literature, the approach that was suggested, the results of the experiments, and finally, the discussion that follows.

part two.Tasks Connected

The flexibility of GraphQL makes data fetching efficient, but it also introduces new security risks that typical API security techniques don't always handle, such as injection attacks and denial of service. Insufficient defense against complex assaults 1 is provided by current methods such static analysis, rate limiting, and general-purpose Web Application Firewalls (WAFs). Worst case scenario: WAFs aren't built for GraphQL's structure, rate limiting and depth limitations don't examine content, and static analysis can miss minor patterns. Instead of analyzing each incoming query in real-time, many existing GraphQL security tools concentrate on server-side analysis. We have GraphQL Cop and Graph00f as examples. Intelligent and adaptive security systems that can comprehend the context and possible malevolence of real-time dynamic GraphQL queries are clearly required.

Because they depend on patterns inside queries instead than readily banned keywords, attack vectors including SQL injection, OS command injection, and XSS attacks are difficult to detect with static approaches. With the capacity to analyze large datasets and identify patterns, machine learning (ML) has become an attractive option for vulnerability detection [14]. Several ML methods may be used efficiently. A branch of machine learning, deep learning (DL) makes excellent use of architectures such as recurrent neural networks (RNNs) and convolutional neural networks (CNNs) to detect intricate patterns in data and code [14]. In cybersecurity, transformer-based models have great promise for accurate threat detection since they use self-attention techniques and can handle complicated dependencies. They are adept at handling data in both sequential and multi-dimensional formats [2].

In addition, GRAPHQLER and similar tools are investigating dependency-aware testing for GraphQL. GRAPHQLER examines the connections between queries, changes, and objects in order to find vulnerabilities using context-aware testing [21]. There are still obstacles to overcome when using ML and DL, including as obtaining high-quality labeled datasets, controlling overfitting, making sure models are interpretable, and keeping up with new threats [14].

To solve the specific security concerns given by GraphQL APIs in real-time, sophisticated methodologies are needed. These strategies include integrating static analysis with machine learning and natural language processing models.

part three.METHOD

Our AI-powered technology employs a hybrid strategy to identify potentially harmful GraphQL queries within an API gateway in real-time. The integration of static analysis and machine learning is showcased through the use of WSO2 API Manager. The configuration is dynamically determined by the GraphQL schema. The architecture of the system and its components for threat detection are described in depth in this section.

A.System Architecture as a Whole

Equipped for effective parallel processing, our system architecture is depicted in Figure 1. When a GraphQL query is received, the schema is used to validate it by the system. At the same time, an LLM examines the schema's SDL according to established criteria, and then it creates a configuration file that includes static check thresholds and schema field complexity values.

An Abstract Syntax Tree (AST) is created when the query is processed. This AST makes it easy to run several detection modules in parallel, including static analysis, SSRF detection, and machine learning inference for injection and XSS. At last, a thorough evaluation of the query's security is produced by combining the outcomes of all modules.

Dynamic Smart Query Difficulty

In order to prevent Denial-of-Service (DoS) attacks, our system uses dynamic schema analysis to detect queries that are too complicated. To achieve this goal, we created two separate and adaptable complexity estimators:

• Simple Estimator: This estimator provides a simple, depth-based estimate of query complexity by multiplying the depth of the query by a pre-configured fixed number.

• Directive Estimator: This advanced estimator examines the GraphQL with the help of a Large Language Model (LLM) schema. Based on the data type and potential size of the fields, the LLM automatically assigns different complexity values to them. It also sets a dynamic complexity threshold according to pre-defined criteria. This approach more accurately portrays real resource usage since it is more subtle and takes context into account.

A configuration parameter gives the user the opportunity to choose between a directive and a basic complexity estimator, so they may customize the difficulty assessment to their individual requirements and schema properties.

C.Limiting Frequent Denial-of-Service Attacks

Our solution applies dynamic thresholds to mitigate many Denial-of-Service (DoS) vectors, in addition to query complexity. These thresh-olds are produced by a Large Language Model (LLM) by examining the Schema Definition Language (SDL) in accordance with a set of rules that describe how to determine upper limits for things like batch sizes, aliases, and circular dependencies. This fixes the following vulnerabilities with an adaptive defense that is API-specific:

• Alias Overloading: If a query uses too many field aliases, it will cause the server to work harder.

A single batched request may contain a high number of queries, which is known as batch overloading.

• Deep Circular Queries: building resource-intensive, deeply nested queries by taking use of cyclical schema relationships.

• Directive Overloading: When directives like @include and @skip are used excessively, it causes a lot of extra processing work.

Crafting queries with an extreme amount of nesting takes unnecessary server resources; this is known as excessive query depth.

• Query Payload Inflation: Placing an excessive burden on server resources by requesting a huge amount of data.

These relevant, contextual, and LLM-generated constraints are enforced when an Abstract Syntax Tree (AST) traverses the nodes of a parsed query.

D.Artificial Intelligence for Identifying Injection Attacks

Identifying injection threats necessitates a more thorough comprehension of user input content, in contrast to structural vulnerabilities that may be effectively addressed by static analysis. Using vector embeddings and the handmade features, we train machine learning models to do this.

Statically identifying injection attacks is not feasible due to the impracticality of re-stricting common keywords for each attack type. These keywords are present in many harmless inquiries that do not cause harm. Thus, instead of depending just on keyword-based detection, pattern analysis is necessary for detecting such injection assaults. The project successfully handles the following two kinds of injection attacks.

One such vulnerability is operating system command injections, which let an attacker take control of the application server and run whatever command they choose. With GraphQL, an operating system

Having a mutation or query that uses user-supplied input in a system command might lead to a command injection vulnerability.

An attacker can run arbitrary SQL queries against a backend database using SQL injection (SQLi) attacks, a security vulnerability in GraphQL APIs. By inserting malicious SQL code inside a GraphQL query, attackers can exploit this vulnerability and cause the backend database to execute the query.

In order to detect injection assaults, a two-stage machine learning process is used.

1) Creating payload contextual vector embeddings

2) Foretell potential exposure

1) Phase 01 - Contextual Vector Embeddings for the Payloads: Several embedding approaches were investigated in the beginning to construct vector embeddings of the user payloads that were retrieved, such as BERT, CodeBERT from Microsoft, Doc2Vec, and FastText from Gensim. In the end, the model with the highest accuracy on the final validation set was the SBERT pre-trained all-MiniLM-L6-v2 with 384-dimensional embeddings.

Our injection attack strategy begins by creating an initial base embedding vector using the input data, as seen in Figure 2. Then, two other vectors are generated by appending this vector with certain handmade characteristics; one of these vectors is optimized for SQL Injection both for operating system command injection and SQLite. A specialized 1D convolutional neural network (CNN) then processes each vector. The two models for injection detection are a product of the shared design of the CNNs and their training on separate attack-specific datasets.

The second phase, "predict vulnerability," involves using a 1D convolutional neural network (CNN) using the combined feature vector to identify SQL injection and operating system commands using the same models. Figure 3 shows that it uses three 1D convolutional layers, each with Batch Normalization and MaxPooling1D (2). These layers include twelve, twenty-four, and fifty-one filters, kernel 3, and ReLU. Prior to the last sigmoid neuron, which outputs the malicious probability, there is a GlobalMaxPooling1D layer. Then, there is a dense layer with 256 neurons and 0.5 dropout. Using Adam (0.001) and binary cross-entropy, the model was trained for 32 batches of 20 epochs each, with early stopping at 5 patience levels and best-weight restoration.

Deep Learning for Cross-Site Scripting Attack Detection

Cross-Site Scripting (XSS) vulnerabilities, similar to injection assaults, cannot be detected by blocking certain phrases. Similar to the last instance, a two-stage machine learning method has been used to detect these vulnerabilities.

1) Creating payload contextual vector embeddings

2) Foretell potential exposure

XSS detection is easier than injection assaults because of typical exploit patterns in Phase 01, which involves building contextual vector embeddings for the payloads. Since injection assaults can take many forms, high-dimensional embeddings would be superfluous for XSS and would just add additional work.

The Doc2Vec embedding model from Gensim is utilized to produce an embedding vector of size 20, as seen in Figure 4, in order to improve performance. The Doc2Vec model used in this work is not a pre-trained version, but rather a custom-trained version that was trained using a dataset of malicious and benign scripts. Additional characteristics derived by hand from the query are then appended to this vector. In order to ascertain if the question is harmful or not, the models are then fed the combined feature vector.

The length of the payload is represented by its payload length; inputs with a greater payload length may contain obfuscation or embedded scripts.

• Count of Obfuscated Script Variants - This metric measures the number of flagged or obfuscated variants of <script> that are used to circumvent filters, such as %3Cscript and <script.

This function counts the number of times certain characters are used, such as <, >, ", ', and encoded forms (%3C, %3E).

They are essential for injecting HTML and JS.

In order to prevent the loading of remote scripts or the redirection of visitors to harmful domains, the following feature is available: • External Resource Count.

2) Phase 02 - Predict Vulnerability: To improve the efficiency of malicious XSS detection, an ensemble method of a Random Forest classifier and a Multilayer Perceptron (MLP) was used. This approach was chosen because the handcrafted and embedded feature vectors have low dimensionality, making traditional machine learning models a good fit. Traditional models have faster training and inference with minimal performance trade-offs, and this design helps reduce computational overhead while still maintaining high detection accuracy. The ensemble of Random Forest and MLP strikes a good balance between precision and efficiency. Thus, Phase 02 - Predict Vulnerability was successfully implemented.

Defending Against Server-Side Request Forgery Attacks

In order to prevent Server-Side Request Forgery (SSRF) attacks, where attackers alter requests made by the server, our system runs multiple security checks in parallel. In response to a query, we check the Abstract Syntax Tree (AST) for URLs. If any URLs are found, we compare them to the following attack vectors:

1) Detection of Local IP Attacks: This safeguard prevents unauthorized access to resources on the internal network by blocking requests to local addresses, private IP ranges, and their disguised forms, such as encoded IPs and DNS redirection.

2) Protecting Cloud Metadata: This feature blocks requests to specified IPs and hostnames used by major cloud providers (AWS, GCP, Azure) to access their metadata services, hence protecting critical cloud credentials.

3) SSRF Prevention Based on Parameters: We look for query parameters that are often used in SSRF attacks (e.g., url, redirect) and run our other SSRF checks on them. This stops attackers from concealing dangerous URLs in trusted parameters.

Fourthly, we may protect ourselves from encoded payload attacks by keeping an inventory of known encoded attack patterns. This way, we can identify malicious URLs that try to evade detection by encoding them using Base64, Unicode, or URL encoding.

Phase G: Production-Oriented Implementation and Optimization

In this part, we will go over the important implementation techniques and optimizations that were used to make sure the detection system is efficient, scalable, and works well with other systems. It is a high-performance service that is modularly built and designed for real-time analysis in a production setting.

1) Asynchronous Design: The foundation of the service is FastAPI, an asynchronous architecture that allows it to process real-time API traffic effectively and without stopping, even when dealing with a large number of concurrent requests.

2) Centralized Model Loader: All machine learning models are managed by a singleton class, which is loaded once during system startup and is then easily available for inference across all subsequent requests. This minimizes memory and loading overhead.

3) Optimizing for ONNX Runtime: To achieve maximum inference speed, we convert all trained models to the ONNX format and quantize them to INT8 precision. Then, we execute these models using the ONNX Runtime, configured with aggressive graph optimizations (Figure 5). This greatly speeds up predictions by reducing memory footprint and taking advantage of hardware-level optimizations.

4) Streamlined Preprocessing and Parsing: Parsing GraphQL queries and expanding fragments is done once per query in the initial preprocessing stage. This eliminates the need for redundant computation by building the Abstract Syntax Tree (AST) efficiently before passing it on to the different detection modules.

5. Vulnerability Detection in Parallel and Concurrent Streams: To optimize throughput, the system employs a hybrid execution model. One thread pool is responsible for processing operations that are heavy on the central processing unit (CPU), such as ML model inference. Another thread pool handles operations that are bound to the input/output (I/O) bus. The main asyncio event loop coordinates these tasks, keeping the application responsive even as it efficiently processes activities that are computationally intensive or blocking.

6) Centralized Logging: A bespoke logging system allows for clear insight into all important processes while reducing latency and I/O overhead by writing logs to a file in batches. External libraries are suppressed to keep the logging process simple.

7) Production Deployment Architecture: As a process manager, Gunicorn ensures robust concurrency and process supervision in the FastAPI application running in production. Uvicorn provides a high-performance Asynchronous Server Gateway Interface (ASGI) for the application, guaranteeing scalability under substantial user load.

Gathering and Arranging Data

For the purpose of training and evaluating the machine learning models, this section describes the approaches used to gather and prepare the datasets. Three distinct datasets were created for the intended attack vectors: SQL Injection, OS Command Injection, and Cross-Site Scripting (XSS).

1) Acquiring Data: Gathering Attack-Specific Payloads from Established Sources was Part of the Data Acquisition Phase.

The dataset used for SQL Injection detection came from a public Kaggle repository [30]. The OS Command Injection dataset was obtained from a resource specified in [27]. For XSS exploits, diverse payloads were sourced from [28], [29], [31], and [32]. Data pre-processing was done to ensure consistency in the datasets. Each instance had a payload and a binary label (1 for malicious, 0 benign). Hundreds of samples that were mislabeled were removed after manual review. For the

limited OS Command Injection dataset, LLM-based augmenta-tion generated synthetic payloads that combined OS commands and natural language to make them more robust.

With the use of negative sampling, we were able to improve the model's specificity for SQL Injection, OS Command Injection, and XSS detection by reducing the number of false positives generated by code-like patterns across categories.

Results

We first describe the performance of the machine learning models developed for detecting injection attacks and XSS exploits. Then, we present the results of load testing and system profiling that were done to assess the performance and scalability of our AI-driven GraphQL security detection system in a simulated production environment.

Evaluation of Machine Learning Models

Tables II and III, along with their respective confusion matrices, summarize the results. The ML models for SQL Injection, OS Command Injection, and XSS detection were tested on a held-out test set (Section III.H) using Accuracy, Precision, Recall, and F1-score. The matrices detailed true/false positives and negatives.

Dedicated convolutional neural networks (CNNs) processed by SBERT embeddings and handcrafted features successfully captured the intricate patterns of malicious payloads, resulting in high accuracy and robust detection for both the SQL Injection and OS Command Injection models.

Doc2Vec embeddings with handmade features are useful for XSS attacks, since both the Random Forest and MLP models exhibited excellent accuracy in recognizing these threats.

System Efficiency and Expandability

We ran load tests in a mock production setting to see how well and how scalable the system was.

1) Methodology and Environment for Load Testing: The following requirements were met during load testing in an Azure Virtual Machine environment:

We utilized Locust to model concurrent demand by flooding the API endpoint with a combination of benign and malicious GraphQL queries in POST requests. The test scaled up from zero to five hundred users at a rate of ten users per second over the course of two minutes. We conducted load testing in two stages to evaluate the functionality of various components.

3) Benchmarking All Security Checks: All tests were run in a non-GPU environment, and performance metrics (Figures 9, 10, 11) revealed significant overhead when all checks were enabled, including ML-based SQL Injection, OS Command Injection, and XSS detection. The average response time increased dramatically with higher concurrency, primarily because of the computational cost of ML inference.

Conclusion

Our hybrid solution to real-time GraphQL security utilizes LLMs for dynamic configuration and Sentence Transformers with CNNs for effective threat detection. We provide a revolutionary AI-driven system in this study. However, load testing showed that real-time ML inference imposes

computational complexity, especially in CPU-only contexts, underscoring the necessity for hardware acceleration, and the evaluation showed great accuracy in identifying SQLi, OS Command, and XSS attacks. We provide a flexible and strong approach to GraphQL API security that uses dynamic setup via LLMs to enable schema-specific rules and a combination of static and ML checks to cover all possible threats. Extending the dataset, improving models, testing on accelerated hardware, integrating XAI approaches, and investigating sophisticated feedback mechanisms for ongoing development will be the primary goals of future work.

References

[1] Kuntamukkala, N. K., & Thalary, S. (2021). Self-Optimizing Angular Applications: A Novel Framework for AI-Driven Performance Adaptation in Production Environments. *International Journal of AI, BigData, Computational and Management Studies*, *2*(2), 107-117.

[2] 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.

[3] Kuntamukkala, N. K. (2022). A Novel AI-Native Architecture for Enterprise Angular Using LLM-Orchestrated Signal Reactivity and State Isolation. *International Journal of Artificial Intelligence, Data Science, and Machine Learning*, *3*(3), 151-162.

[4] Babaey, V., & Ravindran, A. (2024). Gensqli: A generative artificial intelligence framework for automatically securing web application firewalls against structured query language injection attacks. *Future Internet*, *17*(1), 8.

[5] Katipelly, A., & Kuntamukkala, N. K. (2022). Mitigating Algorithmic Complexity Attacks in Federated GraphQL Architectures: A Depth-Bounded Semantic Rate Limiting Approach for Open Banking. *International Journal of Emerging Trends in Computer Science and Information Technology*, *3*(3), 112-121.

[6] Qazi, F. (2023). Application programming interface (API) security in cloud applications. *EAI Endorsed Transactions on Cloud Systems*, *7*(23), e1.

[7] Kuntamukkala, N. K., & Katipelly, A. (2022). Neural Component Libraries for Angular: AI-Generated, Self-Documenting UI Elements with Intelligent API Integration. *International Journal of AI, BigData, Computational and Management Studies*, *3*(3), 116-127.

[8] Qazi, F. (2023). Application programming interface (API) security in cloud applications. *EAI Endorsed Transactions on Cloud Systems*, *7*(23), e1.

[9] Thalary, S., & Kuntamukkala, N. K. (2022). Operationalizing Software Invariants: A DevOps-Driven Approach to Reliability in Cloud-Native Systems. *International Journal of Emerging Trends in Computer Science and Information Technology*, *3*(4), 157-168.

[10] Zhao, C. (2024). Api common security threats and security protection strategies. *Frontiers in Computing and Intelligent Systems*, *10*(2), 29-33.

[11] Kuntamukkala, N. K. (2023). Optimizing Enterprise SPAs: Angular Standalone Components and Signals. *International Journal of Emerging Trends in Computer Science and Information Technology*, *4*(1), 189-200.

[12] Augustine, N., Sultan, A. B. M., Osman, M. H., & Sharif, K. Y. (2024). Application of artificial intelligence in detecting SQL injection attacks. *JOIV: International Journal on Informatics Visualization*, *8*(4), 2131-2138.

[13] Kuntamukkala, N. K., & Katipelly, A. (2023). Predictive Angular Rendering: Machine Learning Models for Intelligent Client-Side Optimization with Adaptive Backend Coordination. *International Journal of AI, BigData, Computational and Management Studies*, *4*(2), 144-154.

[14] Kaul, D., & Khurana, R. (2021). AI to detect and mitigate security vulnerabilities in APIs: encryption, authentication, and anomaly detection in enterprise-level distributed systems. *Eigenpub Review of Science and Technology*, *5*(1), 34-62.

[15] Kuntamukkala, N. K. (2024). Self-Healing Angular Architecture: AI-Driven Autonomous Error Recovery and System Resilience. *International Journal of Artificial Intelligence, Data Science, and Machine Learning*, *5*(3), 219-230.

[16] Balaganski, A. (2015). API Security Management. *KuppingerCole Report*, *70958*, 20-27.

[17] Kuntamukkala, N. K., & Thalary, S. (2024). Intelligent Angular Architecture: Machine Learning-Based Component Recommendation Systems for Enterprise-Scale Development. *International Journal of Artificial Intelligence, Data Science, and Machine Learning*, *5*(4), 276-284.

[18] Enjam, G. R. (2024). AI-Powered API Gateways for Adaptive Rate Limiting and Threat Detection. *International Journal of Artificial Intelligence, Data Science, and Machine Learning*, *5*(4), 117-129.

[19] Kuntamukkala, N. K. (2025). Architectural Optimization of Performance and Security in Enterprise SPAs Using Angular Standalone Components and Signal-Based Reactivity. *International Journal of Emerging Trends in Computer Science and Information Technology*, *6*(2), 115-123.

[20] Peterson, G., & Bertino, E. (2022). API Security and Threat Intelligence Mechanisms for Protecting Digital Experience Platform Ecosystems.

[21] Katipelly, A., & Kuntamukkala, N. K. (2025). Hierarchical Multi-Agent Orchestration for Automated Dispute Resolution: A Game-Theoretic Approach to Policy Adherence in Digital Wallets. *International Journal of Artificial Intelligence, Data Science, and Machine Learning*, *6*(2), 195-204.