Assessing the Effectiveness of StackHawk in Detecting GraphQL Security Vulnerabilities

Anusheh Mustafeez, Akshyan Anandakrishnan, Dheeraj Nayak, Sai Sushmitha Sanduri
Information Networking Institute
Carnegie Mellon University
Pittsburgh, USA, PA 15213
{amustafe, aanandak, dheerajn, ssanduri}@andrew.cmu.edu

***Abstract*— GraphQL is an open-source query language for APIs that allows clients to specify the data they require and receive only that data in response. However, like any other web technology, GraphQL is not immune to security vulnerabilities. In this research paper, we aim to evaluate the accuracy of fuzzing tools in detecting security vulnerabilities in GraphQL APIs. We use StackHawk, a modern application security testing tool that automates vulnerability detection and remediation, to fuzz GraphQL APIs and detect security vulnerabilities. We test the effectiveness of StackHawk in detecting vulnerabilities by comparing the results with the known vulnerabilities listed in the Damn Vulnerable GraphQL Application (DVGA). To measure the accuracy of the results, we use metrics such as true positive, true negative, false positive, and false negative. Our results show that StackHawk does not check for a majority of the vulnerabilities listed in DVGA. The tool also produces some false positives and false negatives, indicating room for improvement in its accuracy. We also found that StackHawk was effective in detecting vulnerabilities of varying severity levels. Our research provides insights into the effectiveness of fuzzing tools for detecting security vulnerabilities in GraphQL APIs. The results can help developers and security professionals make informed decisions about the use of fuzzing tools for their security testing needs.**

I. INTRODUCTION

GraphQL is a query language that provides a declarative syntax for clients to request the exact data they need from an API. It was created by Facebook and has gained significant popularity due to its ability to improve API efficiency and simplify client-server communication. It is a query language that allows clients to describe their data requirements and receive precisely what they ask for. It provides a complete and self-documenting API schema, which enables clients to construct queries and mutations dynamically without relying on external documentation [1]. It is a runtime for APIs that provide a more efficient, powerful, and flexible alternative to REST. It allows clients to specify the exact data they need, and the server to provide a complete and self-documenting schema, reducing the complexity and overhead of API communication. However, the unique characteristics of GraphQL, such as its complex type system and ability to merge queries, present new challenges for securing GraphQL-based systems. One approach to achieve this is through testing, specifically black box testing, which involves testing the system from an external perspective without any knowledge of its internal workings. Through our research, we hope to provide valuable insights and best practices for developers and testers working with GraphQL-based systems.

A GraphQL query consists of three main parts[2]: the operation type, the operation name, and the selection set. The operation type can be one of three keywords: "query", "mutation", or "subscription". "query" is used to retrieve data from the server, "mutation" is used to modify data on the server, and "subscription" is used to receive real-time updates from the server. The operation name is an optional identifier that can be used to make the query easier to understand and reference later. If a name is included, it must follow the same rules as any other identifier in GraphQL. The selection set is the heart of the query and consists of one or more fields. Each field corresponds to a piece of data that the client wants to retrieve from the server. The name of the field is a string, and it can have optional arguments that modify the behavior of the query. For example, a "first" argument might be used to limit the number of results returned. Fields can also have sub-fields, which allow for nested queries that traverse the relationships between different data types. For example, a query for a user might include a sub-field for the user's posts, which in turn might include sub-fields for each post's comments. By combining these different parts of a query, clients can retrieve exactly the data they need from the server in an efficient and flexible way.

GraphQL fuzzing is a technique for automated testing of GraphQL APIs that involves generating a large number of random or semi-random queries and mutations in an attempt to uncover vulnerabilities or weaknesses in the API implementation[3]. It is a form of black box testing that seeks to explore the behavior of the system under unexpected or unusual inputs. Here are some examples of how GraphQL fuzzing is defined in research papers: GraphQL fuzzing involves generating random or semi-random GraphQL queries and mutations in order to stress-test the API and uncover any vulnerabilities or errors in the implementation. It is a technique for testing GraphQL APIs by generating a large number of random queries and mutations, with the goal of discovering unexpected behaviors or errors in the system. It is a black box testing technique that involves the generation of random or semi-random queries and mutations, with the aim of uncovering potential security vulnerabilities or other weaknesses in the API implementation.

II. Problem Definition

The adoption of GraphQL has skyrocketed in the past few years. Since its inception in 2015, the user base has increased to over 30,000 tech stacks.

Because of its usability and the features that it provides, its adoption in microservice architecture has made it the default adoption for endpoint exposure. But with such a large user base and rapid adoption, there is evidently going to emerge a problem that cannot be reverted. To give an example, when GitHub adopted GraphQL, for its architecture, some of the endpoints did not follow the same kind of security principles that the REST API performed. During the migration, they had a layer of service that either called GraphQL or the REST endpoint. GraphQL takes in the query in a body and executes it, while the REST endpoint will authorize it before actually executing the query. Since the migration was unsuccessful, the endpoints that were not supposed to be accessed could be easily queried using the GraphQL endpoint while it failed in REST. This was a famous problem in 2019 when GitHub tried to prevent access to unpaid members from using certain repositories and blocked certain countries from accessing repositories due to export licenses [4]. This could have easily been a big issue where the microservice architecture that adopted GraphQL did not authenticate properly and had severe issues. The information flow shown in Figure 1 gives us a better explanation of why there was an issue with GraphQL microservice architecture.

The above is just one example of an attack and one parameter. There have been attacks in the real world where the implementation of GraphQL led to a Denial of Service. GraphQL allows you to send out requests in fragments and a long list of requests at once. If the schema or the request that is being sent is not validated properly there could be a huge impact, like memory leaks or bringing the server down. By sending a very large request, which has a recursion the application would try to retrieve the same data forever. If we try to perform an operation that takes forever, the server might get stuck in one operation instead of performing this asynchronously, and since there are multiple operations done by GraphQL, it could take forever to just refresh a page, finally leading to a DoS.

In a talk about the architecture of Reddit by the Director of Engineering, he mentioned how they could not hire junior-level employees because of microservice architecture that they adopted did not have a mechanism in place for testing vulnerabilities in code. These kinds of issues are recently being solved using automated tools like ZAP, StackHawk, Snyk, and other tools. These tools although don't have the highest accuracy and have a high false positive rate, they do provide a lot of valid information and give a gist of how the program can be attacked. GraphQL is an emerging technology with a very high adoption rate. But there aren't many tools that are in the market that performs GraphQL fuzzing or security checks. There are not many static scans that check the validity of how GraphQL is implemented. The only thing that these tools check is either the Authorization or injection tools. This authorization is extremely superficial and doesn't have every option. So the idea in our paper is to get a complete understanding of each of these tools and come to a conclusion of which of these tools has the best accuracy and understand what could be done better in each tool. Finally, we are trying to explain how to add these features and how much they could improve these tools.

III. BACKGROUND

Past attempts have been made to develop effective toolkits for GraphQL black box testing. Belhadi et al. [4] conducted a study on fuzz testing techniques for GraphQL APIs, using both white-box and black-box approaches. They found that the combination of these two methods resulted in higher code coverage and detection of more vulnerabilities than either method alone. Their research showed that both techniques can effectively identify vulnerabilities in GraphQL APIs. Furthermore, they found that black-box fuzzing can detect more unique vulnerabilities compared to white-box fuzzing, which is limited by the available schema information. These findings support the importance of comprehensive security testing for GraphQL APIs and provide useful insights for the development of effective fuzzing strategies.

This work serves as a preliminary to our study but much work still needs to be done in this space. Belhadi et al. proposed a framework that has been implemented as an extension to the open-source EvoMaster tool. This extension was evaluated by running black-box tests on 31 APIs and 825 endpoints. Given the low number, generalization may be a problem. Moreover, the authors mention that there are many queries/mutations for which they could not get back valid data. The extension also lacks input validation and depends heavily on the EvoMaster software.

Most common security tools today have added GraphQL testing as an extension to their base software but these have their fair share of issues. We will discuss some of these when discussing selected tools.

IV. Prominent GraphQL Attacks

According to OWASP[5] some of the most common vulnerabilities present in GraphQL are listed below. Here's a generic GraphQL schema to better explain the vulnerabilities listed below.

```
type User {
  id: ID!
```

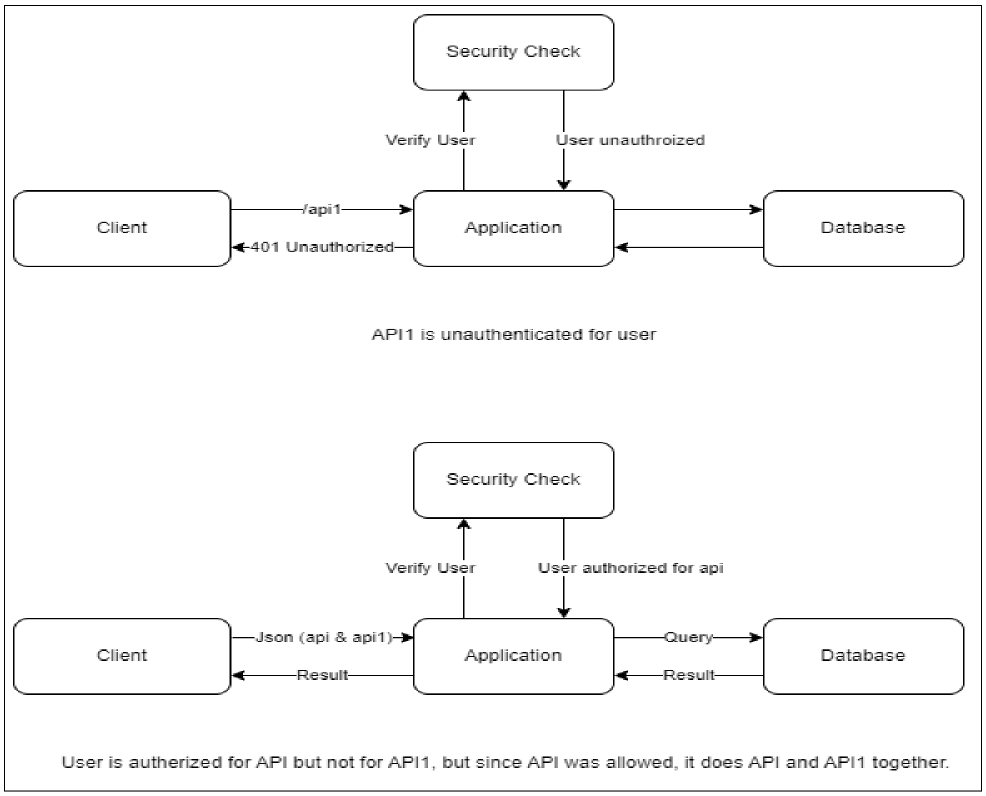


Fig. 1. Microservice Model difference of REST and GraphQL

```
  name: String!
  email: String!
  friends: [User!]!
  image(url: String!): Image!
}

type Image {
  url: String!
}

type Mutation {
  systemUpdate(version: String!):
     String!
}

type Query {
  users: [User!]!
  user(id: ID!): User!
}
```

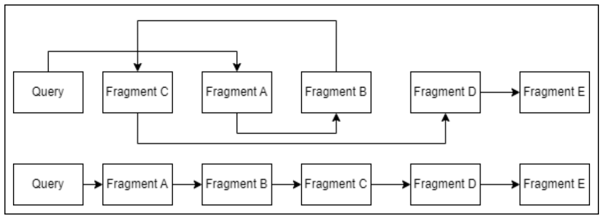


Fig. 2. Fragmentation

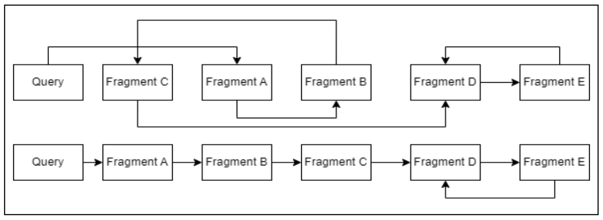


Fig. 3. Circular Fragmentation

- **Circular Fragmentation**: Fragmentation is a concept where each query or mutation is divided into multiple JSON queries with a fragment name. Each query would have "..<fragment name>" that would point to the next fragment. When some fragment points to an already visited fragment this would lead to a circular fragmentation as shown in Figure 2. Circular Fragmentation is a type of Denial of Service attack where a malicious client sends a query containing fragments that reference each other in a circular manner. This results in an infinite loop and causes the server to consume a lot of resources, leading to a denial of service. Here's an example query that can cause circular fragmentation:

```
query {
  users {
    ...UserFields
  }
}
```

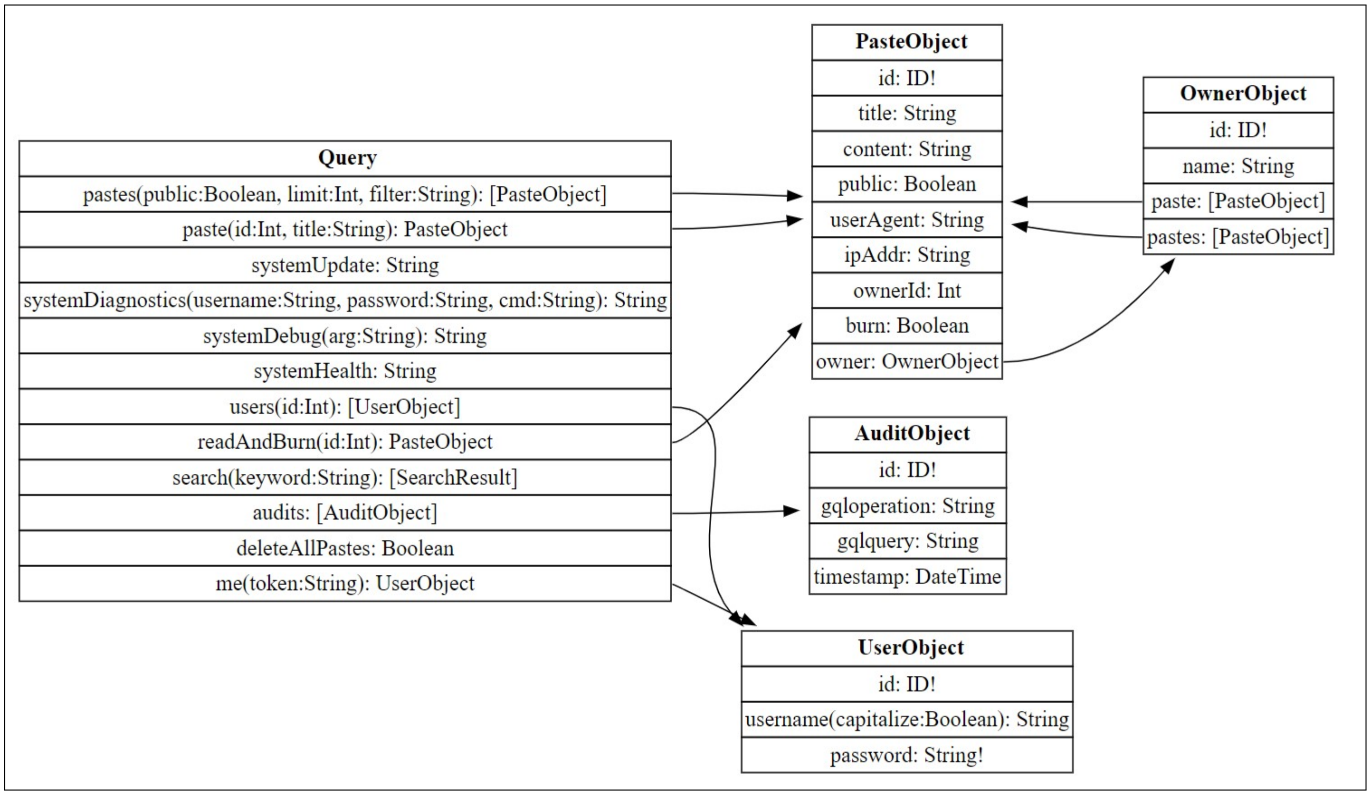


Fig. 4. DVGA Schema

```
fragment UserFields on User {
  name
  email
  ...UserId
}

fragment UserId on User {
  id
  ...UserFields
}
```

In this query, the UserFields fragment includes the UserId fragment, which in turn includes the UserFields fragment, creating a circular reference. When the server attempts to resolve this query, it will enter an infinite loop, causing a denial of service.

- **Deep Recursion**: Deep Recursion is a type of Denial of Service attack where a malicious client sends a deeply nested query that exceeds the server's maximum recursion depth. This can cause the server to enter an infinite loop, consume a lot of resources, and eventually lead to a denial of service. Here's an example query that can cause deep recursion:

```
query {
  users {
    name
    email
    friends {
      name
      email
      friends {
        name
        email
        friends {
          name
          email
          # This can go on for many
              levels...
        }
      }
    }
  }
}
```

In this query, the users field includes a nested friends field, which itself includes a nested friends field, and so on. If the server's maximum recursion depth is exceeded, it will enter an infinite loop, causing a denial of service.

- **Batching Query Attack**: GraphQL allows a user to send multiple queries in one single request. This is very similar to Authorization Bypass, but instead of bypassing, we send in multiple queries that load up the entire system. The batch query can create a problem with over-fetching and ex-filtration of data as mentioned in the Authorization bypass attack. It is a type of Denial of Service attack where a malicious client sends multiple queries as part of a single request, causing the server to execute them all at once and consume a lot of resources. For this, having a heavy query like systemUpdate e.g. can be useful since a small number of nested queries could consume a lot of resources. Another advantage of a Batching Query Attack is that an attacker could brute force information by sending a single batched query. Here's an example query that can cause batching query:

```
mutation {
  update1: systemUpdate(version: "
     1.0.0")
  update2: systemUpdate(version: "
     2.0.0")
  update3: systemUpdate(version: "
     3.0.0")
  # This can go on for many queries
     ...
}
```

The systemUpdate mutation takes a version parameter and returns a string indicating the result of the update. This mutation is vulnerable to a batching attack if a client sends multiple requests to update the system with different versions. In this query, This attack sends three requests to the systemUpdate mutation, each with a different version number. If the server processes these requests as separate transactions, it could be overwhelmed by the number of updates being performed simultaneously. If the server does not have appropriate rate-limiting or resource allocation measures in place, it may execute all the queries at once, leading to a denial of service.

- **Authorization Bypass**: This can be a crucial vulnerability in GraphQL. In GraphQL we can send mutations or queries as a bunch to retrieve various values. The user might not have the same amount of privileges to access all the endpoints and data they were not allowed to. For example in the GitHub attack, the user was not allowed to access the database, but they could information about the repositories, but were not allowed to delete or view the entire code. It is a type of security vulnerability where a malicious client can bypass access controls and gain access to sensitive data or functionality. Here's an example query that can cause authorization bypass:

```
query {
  user(id: "1") {
    name
    email
  }
}
```

In this query, the client is requesting information about a specific user with an ID of "1". If the server does not have proper authorization checks in place, a malicious client may be able to bypass them and gain access to the user's sensitive information.

- **Code Execution**: Any endpoint is vulnerable to an attack of Code Execution if the validation is not performed on inputs. A combination of Injection with Code execution could be a severe vulnerability. It is a type of security vulnerability where a malicious client can inject arbitrary code into the server and execute it. Here's an example query:

```
query {
  user(id: "1") {
    name
    email
    # This field is vulnerable to
       code injection
    image(url: "https://example.com
       /image.jpg; rm -rf /") {
      url
    }
  }
}
```

In this query, the image field includes a URL parameter that is vulnerable to code injection. If a malicious client sends a URL with a command like rm -rf /, it could be executed on the server, causing significant harm. Proper input validation and sanitization are critical to preventing code execution vulnerabilities.

- **Injection**: The most common attack in any web-related application is the injection. If the user-provided input is not properly validated and sanitized it would lead to issues where the information stored in the database would get leaked. It can also modulate the information on the database. In GraphQL the mutations allow you to POST something onto the server, leading to changes that are not allowed. Injection is not just limited to databases and websites, an attacker can add additional logs leaking information like passwords through logging systems. An example of a query injection attack in GraphQL could involve injecting malicious code into a query parameter, like the id parameter in a user query, as shown below:

```
query {
  user(id: "1; DROP TABLE users; --
     ") {
    name
    email
  }
}
```

In this example, the attacker has added a semicolon (;) followed by a malicious SQL command (DROP TABLE users) and a comment (–) to the end of the id parameter in the user query. If the server does not properly sanitize or validate user input, it may execute this SQL command along with the original query, causing the users table to be dropped from the database, resulting in data loss.

V. Tools

We researched various tools for GraphQL that are currently in the market. Most tools have add-ons and are not specific to GraphQL. The only tool specific to GraphQL is InQL which is the recommended tool by the OWASP foundation. Zap and StawkHawk fuzzing tools also have support to detect graphQL vulnerabilities.

A. inQL - Introspective GraphQL

inQL is an open-source, OWASP-recommended tool that is used to fuzz GraphQL [16]. It is specifically designed to detect vulnerabilities in GraphQL APIs by sending a series of malformed requests and observing the server's responses. InQL can be used as a stand-alone script, or as a Burp Suite extension. The tool makes use of the GraphQL language's built-in introspection query to dump queries, modifications, subscriptions, fields, and arguments as well as retrieve both standard and custom objects. This data is gathered, processed, and then used to create HTML and JSON schema descriptions for API endpoints. The ability to create query templates for every known type is another feature of InQL. Basic query types can be recognized by the scanner, and their replacement with placeholders makes the query ready for ingest by a remote API endpoint. One of the main advantages of inQL is that it automatically generates a list of mutation and query keywords for fuzzing. This makes it easy to quickly test an endpoint for vulnerabilities without having to manually generate a list of inputs. The tool also includes a variety of injection attacks that can be used to test for common vulnerabilities, such as SQL injection and cross-site scripting (XSS) attacks.[16]

- Schema Introspection: InQL uses the GraphQL introspection feature to fetch the schema of a GraphQL API and provides an interactive interface to explore the types, fields, and relationships in the schema.
- Query Execution: IntrospectiveQL allows you to execute GraphQL queries directly from the tool, which can be useful for testing and debugging queries without making actual API requests
- Autocompletion: InQL provides auto-completion for fields and arguments based on the schema, making it easy to navigate and explore the GraphQL API.
- Mutations and Subscriptions: IntrospectiveQL supports executing mutations and subscriptions, allowing you to test and experiment with write operations and real-time updates in the GraphQL API

Although InQL is an OWASP [5] recommendation for security tools for GraphQL the current version incorrectly parses [16] graphQL schemas which does not detects attacks like deep recession attacks and Batching attacks leading to DoS and making brute-forcing of passwords much easier [23].

B. OWASP ZAP

OWASP Zed Attack Proxy (ZAP) is a widely used, open-source, web application security tool [6]. It is one of the most popular tools used by developers and security testers to perform automated security testing. In March 2023 alone, there were 5,194,372 active ZAP scans [7]. Moreover, ZAP is highly customizable, allowing users to configure and extend its functionality using plugins and scripts. It has a user-friendly GUI and supports many web technologies and platforms. So, given ZAP's growing popularity and how powerful of a tool it is, it was important to discuss it in our work. Moreover, StackHawk, the tool of the primary focus of this work, is built on top of Zap which makes Zap all the more important [8].

Although ZAP is a powerful tool, its GraphQL support is still in its early stages. ZAP's GraphQL add-on allows testers and developers to launch automated attacks on open GraphQL endpoints. [9]. Currently, ZAP supports importing GraphQL schemas to allow GraphQL introspection. ZAP will then parse this schema and generate queries from it. The add-on also provides support for GraphQL scripting. This enables developers to represent the nodes for a GraphQL request in the sites tree and incorporate GraphQL queries in Active Scan Input Vectors [10]. Finally, in case, an endpoint and/or schema is not specified before a scan, the add-on has an automation framework that will import GraphQL schemas using introspection if endpoints are found while spidering [11].

The add-on is a welcome step towards improving GraphQL testing but is still relatively new and has limited functionality. Due to this, there are multiple areas where ZAP falls short. Firstly, there is a lack of GraphQL-specific scan rules that test GraphQL-specific vulnerabilities like those discussed in this paper. [12]. Moreover, the Zap community prides ZAP as being easily integrated into CI/CD pipelines but this support is yet to be extended to GraphQL integration tests [13].

C. StackHawk

Stackhawk is a tool that was built on top of OWASP ZAP [14]. It gives a comprehensive and more user-friendly application that is easier for analysis. StackHawk - Hawkscan is a dynamic analysis tool, created by the StackHawk Organization to help companies test their code before production [14]. They have an extremely interactive User Interface and a wide variety of features to integrate with applications. It allows you to integrate with your cloud products and integrate Snyk scans into your CI/CD pipelines. StackHawk HawkScan has a specific configuration that allows you to run a dynamic analysis.

Some of the key factors that make Stackhawk standout are:

- Integration: It allows the integration of applications to the cloud and deployment, preventing multiple the need for multiple processes and manual testing.
- Automated Scanning: The HawkScan allows you to create a configuration based on the application and runs the entire testing by itself
- Reporting: The report generated by the HawkScan is extremely concise and informative.

To set up StackHawk and make it run for GraphQL we would require the schema which we can get by using the introspection query. As shown in the documentation of StackHawk [15] we can give our configuration as:

```
app:
host: http://localhost:5013
graphqlConf:
enabled: true
filePath: schema.json
operation: MUTATION
requestMethod: GET
```

The above configuration of HawkScan would run a fuzzing scan on the endpoint(host) It provides a comprehensive report with a detailed explanation of each of the vulnerabilities it found. We can check the table 4 that gives the table of all the vulnerabilities it found. When we analyze this we notice it only reports Injection, Authentication Bypass, and Code Execution. It does not check for the Denial of Service Attack at all.

Working of StackHawk: StackHawk works in a multi-step process. The first part of the process is to perform a discovery using the Base and Ajax Spider, which use crawling algorithms to find all the different paths in the endpoint. The second step is to run a fuzzer on each of these paths that it has found. These fuzzers modulate and generate intuitive inputs based on the configuration we have provided. Since ours is a GraphQL, it uses schema.json to generate various inputs to find the vulnerabilities. Based on the error rates it generates a human-readable report which has details about the attack parameters for every specific vulnerability category.

The scan has some built-in flags that it tries to fuzz. There are specific graphQL vulnerabilities like circular fragmentation and deep recursion which would cause a denial of service, is an input that needs to be specific and cannot be generalized, it is hard for the fuzzer to find a path that would generate input for denial of service.

VI. Experiments

In this research paper, we evaluate the effectiveness of the GraphQL analysis tool StackHawk. To conduct our evaluation, we chose StackHawk for deep analysis as it is an enterprise app sec tool built on top of Zap, which is a widely used open-source web application scanner. We decided not to use InQL because its repository had a lot of open, unresolved issues and the contributors have mentioned that the tool incorrectly parses certain GraphQL schemas and introspection query results [16]. To benchmark StackHawk's performance, we selected the Damn Vulnerable GraphQL Application [17], which is a known repository containing various GraphQL vulnerabilities. In addition, we chose three real-world GraphQL applications from the GraphQL APIs repository as our test cases [24]. These applications are open source, which allowed us to retrieve their schemas easily. The selected applications were TOSKA conferences application [18], TCGdex Pokemon Card Database [19], and the Brazilian Deputies Chamber [20]. To evaluate the effectiveness of StackHawk, we used several metrics, including true positive, true negative, false positive, false negative, omission, and irrelevant omission. Here are the formal definitions of each:

- True Positive: StackHawk checks and catches a vulnerability that exists.
- True Negative: StackHawk checks but doesn't catch a vulnerability that does not exist.
- False Positive: StackHawk checks and catches a vulnerability that doesn't exist.
- False Negative: StackHawk checks but doesn't catch a vulnerability that exists.
- Omission: StackHawk doesn't check for a vulnerability that exists (counted as a sub-category of False Negatives for our analysis).
- Irrelevant Omission: StackHawk doesn't check for a vulnerability that doesn't exist.

To conduct our evaluation of StackHawk, we ran each of the four selected applications in their own docker containers and directed StackHawk to the HTTP endpoints running the applications. We used the development environment on StackHawk and selected GraphQL API as the application type. We specified the URL path for the introspection schema and provided the respective file paths for the introspection schemas [21] of DVGA and TCGdex, which we retrieved. For Brazilian Deputies Chamber and TOSKA conferences, we simply provided the URL path to '/graphql'. These configurations generated a stackhawk.yaml file that stored our selected configurations. We placed this file in the relevant application directories in our docker instances and ran "hawk scan" to start the StackHawk scans. Once the scans were completed, we viewed and analyzed the results from our StackHawk dashboard, which provided us with detailed information on the vulnerabilities detected in each application. We ran scans on each application thrice to ensure the results were consistent across scans. Additionally, to test for GraphQL-specific vulnerabilities that StackHawk was not checking for, we manually ran handcrafted queries using Postman and Curl to replicate the attacks discussed in Section IV.

A. Damn Vulnerable GraphQL Application

To test and learn about security attacks on a product we need a vulnerable application that explores as many security attacks that are discovered in GraphQL as possible. So we used an existing application that is made intentionally vulnerable for security research purposes. It's called Damn Vulnerable GraphQL Application which is developed by Dolev Farhi (GitHub username: dolevf) [17]. The application is written mainly in Python where it exposed a vulnerable endpoint for GraphQL. It also gives a user interface that provides a way to interact with the schema. The architecture of the entire DVGA is shown in 5.

Before we could understand what attacks could be performed on the DVGA, we retrieved the schema and analyzed the information flow. This gave us a detailed understanding of how to generate the queries, which led to interesting outputs to explore. We have two options to generate the schema. The first one is to go through the code manually and figure it out. The other option is both efficient for attackers and compatible with black box testing using the introspection query. For retrieving the schema we used an introspection query because we knew introspection was enabled for this vulnerable application [22]. By running the query we got the entire schema which was then used to generate prominent GraphQL attacks discussed in Section IV. This gave us a deep understanding of each query and possible inputs we could try while manually testing for all of our applications. The schema for DVGA can be seen in Figure 4.

Based on the StackHawk report as well as the vulnerabilities in Section IV, we found the evaluation metrics for DVGA to be that as presented in Figure 6.

B. TOSKA React Finland Conference Application

The application [18] was built specifically to track all the conferences that happened in Finland. This is an open-source application that uses GraphQL to extract information from the database about previous conferences, speakers, and their details. It also gives the information about the attendees and their contact for each conference. The frontend is written in React and the backend is using Node.js. By using the setup given previously, this application can be set up similarly and we can run the scan. The schema of this application can be found in the appendix 13. The report StackHawk presented to us is attached in appendix 11.

Testing for prominent GraphQL Attacks

- The injection attacks and code execution is tested by the StackHawk
- StackHawk skips the authorization bypass. But when we try to create a query for the contact of the speaker through the conference, we can bypass authorization successfully.
- Stackhawk does not test for circular fragmentation either. But when we perform the test, it gives an error that says "cannot be spread via a different fragment". This is a good case of an irrelevant omission test.
- StackHawk skips deep recursion test too. Creating a query with multiple conferences attended by the speaker and the speakers in the conference, we can create an infinite query, causing a deep recursion attack leading to a DoS attack.
- It also does not test for the batching attack, but there is no operation that takes a long time to complete, which prevents the existence of batching attack.

By analyzing the results of the scan the major outcomes are derived and the confusion matrix is shown in Figure 7. There are a total of two omissions and two irrelevant omissions.

C. Pokemon TCG Cards Database

The TCGdex cards database is a GraphQL API that serves as a comprehensive database for trading card games [19]. It includes data on thousands of cards across various games, including Pokemon Cards. The API allows users to search and filter through the database based on various parameters such as language, game, card type, rarity, and more. The API also includes features such as pagination and sorting for easier navigation through the vast database. The application is open source and is maintained by the TCGdex team. When querying the database, we could filter cards, sets, and series among other things. The detailed visualization of the schema can be found in figure 14 in the appendix.

Testing for prominent GraphQL Attacks

- StackHawk checks for injection already so we did not craft queries for this attack.
- Out of the six attacks, only deep recursion was successful when we called the cards fields inside the set field which was nested inside a cards field. This led the application to stall. This attack was accounted as an omission and thus a False Negative in our analysis.
- Circular fragmentation was unsuccessful as the application did not allow setFragment to spread within itself via a cardFragment.
- Authorization bypass wasn't relevant as this application does not take use tokens for authentication.
- Since the query time for this application was relatively efficient, there were no queries that when batched together could lead to a batching query attack. Moreover, this application does not hold any sensitive information for a brute-force attack to be relevant.
- For the code execution attack, the schema was not configured to allow mutations thus that was not possible. We tried using custom fields but that attack was also not feasible in this scenario.

So overall, we found four irrelevant omissions and one omission for this application. The detailed StackHawk report for this application can be found in Figure 12 in the appendix. The metrics after analyzing query results and the StackHawk report can be found in Figure 8.

D. Brazilian Deputies Chamber Application

The Brazilian Deputies Chamber Application was built with the primary objective to provide a user-friendly platform for developers to access data of the Brazilian deputies chamber and utilize the data in building innovative applications that can offer useful information to the citizens of Brazil and beyond [20]. The GraphQL schema captures relevant information from the Chamber of Deputies database. This will involve defining the types and fields that represent the various entities in the dataset, such as deputies, parties, sessions, bills, and expenses. The schema as shown in figure 16 also includes the necessary queries and subscriptions to enable efficient and flexible data retrieval and manipulation. The setup is similar to

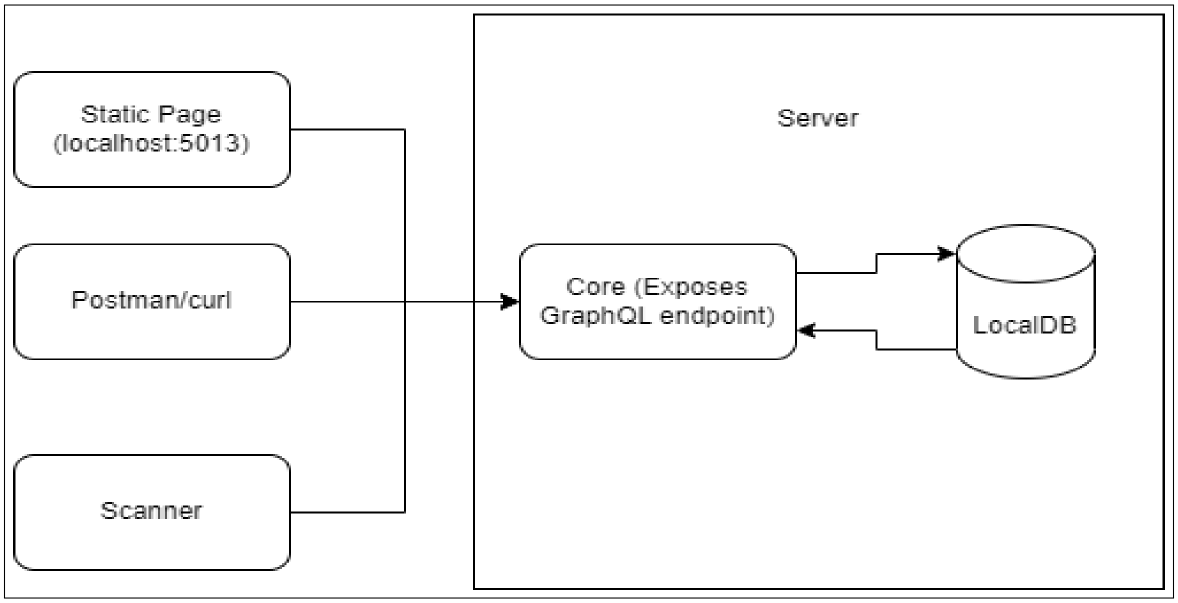
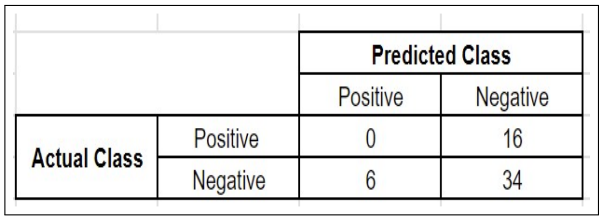
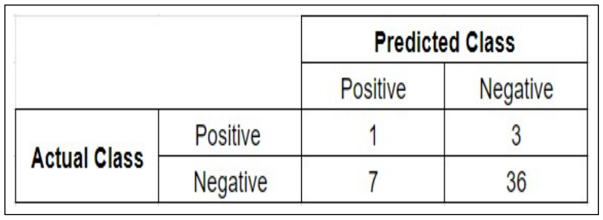


Fig. 5. DVGA Architecture



		Predicted Class	
		Positive	Negative
Actual Class	Positive	0	16
	Negative	6	34

Fig. 6. DVGA Confusion Matrix



		Predicted Class	
		Positive	Negative
Actual Class	Positive	1	3
	Negative	7	36

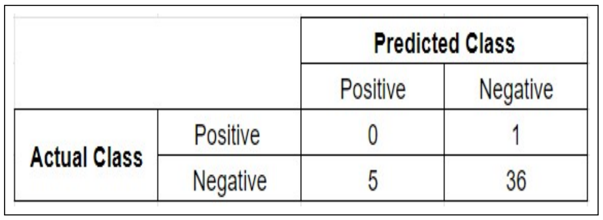
Fig. 7. TOSKA Confusion Matrix

that explained earlier

Testing for prominent GraphQL Attacks

- The Schema for Brazilian Deputies does not contain any circular references between the different endpoints hence graphQL specific vulnerabilities such as Circular fragmentation, Deep Recursion are not a possibility
- Authorization bypass is not relevant for this application as no tokens have been used for authentication
- Batching query cannot be performed as the querying time is relatively very insignificant to launch a DoS attack
- Code Executions is not a possibility either as mutation are not allowed on the Schema
- Injection was tested for by StackHawk

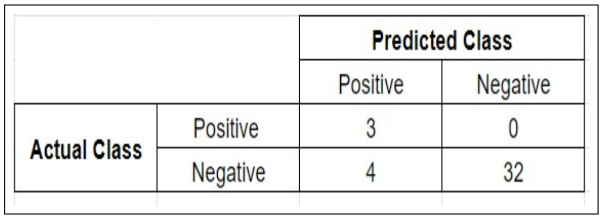
So overall, the Brazilian Deputies Chamber Application



		Predicted Class	
		Positive	Negative
Actual Class	Positive	0	1
	Negative	5	36

Fig. 8. Pokemon TCGdex Confusion Matrix

does not have any omissions but has five irrelevant omissions. The detailed StackHawk report for this application can be found in 13 in the appendix. The metrics after analyzing query results and the stackHawk report can be found in Figure 9



		Predicted Class	
		Positive	Negative
Actual Class	Positive	3	0
	Negative	4	32

Fig. 9. Brazilian Deputies Chamber Confusion Matrix

E. Overall Evaluation

Based on the experiments that we ran on three different open-source applications, we can see a huge number of true negatives and a very small number of true positives. In some cases, the true positives are 0. This suggests that StackHawk as a tool does not perform well in identifying the GraphQL-specific vulnerability. Based on figure 10 the accuracy is about 75% on average and the precision is varying and is lesser than 50%. This gives us strong evi-

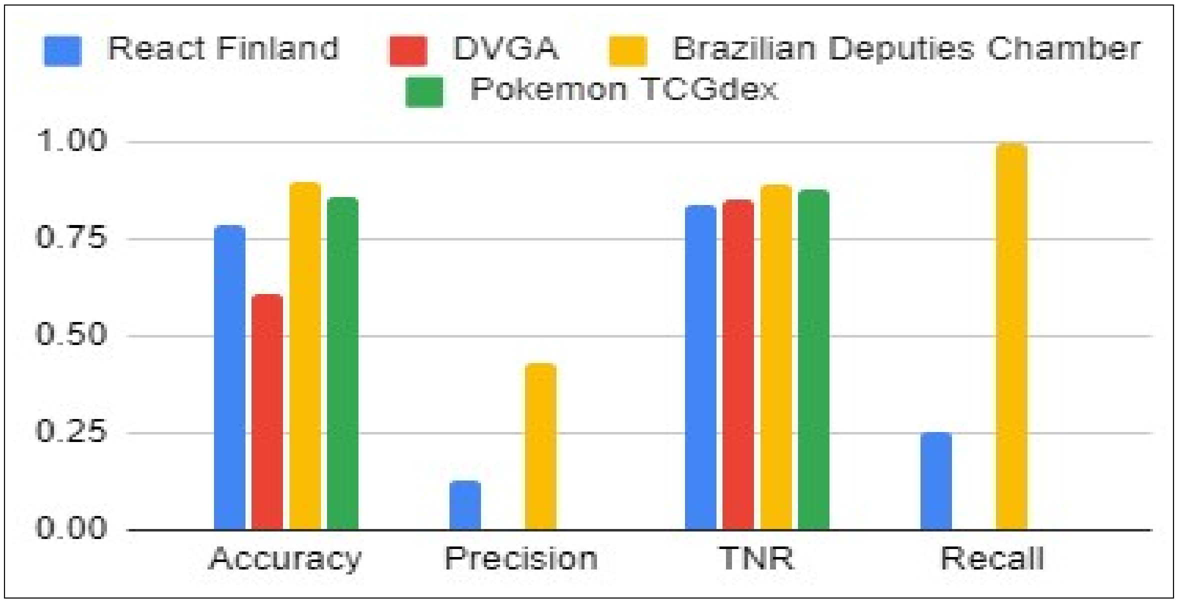


Fig. 10. Overall Performance Metrics

dence that StackHawk built on ZAP is performing poorly with respect to GraphQL specifics.

VII. DISCUSSION & LIMITATIONS

Our evaluation of StackHawk focused on a limited set of four applications and it is important to note that there may be other applications out there with different vulnerabilities that StackHawk may not be able to detect. We handpicked open-source applications that had introspection available, but this may not be the case for an average GraphQL application. Additionally, StackHawk itself has limitations that should be considered when using the tool for GraphQL analysis. Since StackHawk is built on top of Zap which itself has preliminary support for GraphQL, we must take its results with a grain of salt. One limitation of our evaluation is that we could not exhaustively test for False Negatives as we did not have a single source of truth and had to manually test for vulnerabilities. Furthermore, we do not have proof for True Negatives, other than the 5 vulnerabilities we manually checked for. Considering only the Damn Vulnerable GraphQL Application, there may be more vulnerabilities that StackHawk did not detect. We also did not analyze network-level vulnerabilities, which may be relevant for some applications. In our evaluation, we focused on the number of vulnerabilities detected by StackHawk, as the number of queries results in a highly imbalanced confusion matrix. However, it is important to note that to patch vulnerabilities security professionals should take into consideration each query that could lead to an attack. Although StackHawk identifies the severity of each vulnerability, we felt it was out of scope for this particular work so did not take that into consideration. Despite these limitations, our evaluation showed that there is still a lot of work to be done in the area of GraphQL security analysis. While StackHawk is arguably the best enterprise tool available, our evaluation highlighted its limitations. Furthermore, GraphQL is still in its early stages and there is not yet enough research done in this area.

VIII. MITIGATIONS

The attacks mentioned in this paper can be prevented by disabling introspection on GraphQL servers to prevent sensitive information from being exposed to the public. Additionally, increasing awareness and providing training on secure setup and configuration of GraphQL servers can help to mitigate vulnerabilities [23]. To mitigate Circular Fragmentation and Deep Recursion you can set a maximum depth for query execution and also limit the number of fields or fragments that a query can include. Additionally, you can monitor query execution times and abort queries that exceed a certain threshold. To mitigate batching query attack, the server can limit the number of requests that can be batched together or implement rate limiting to prevent a single client from making too many requests within a given time period. Additionally, the server can implement validation checks on the version parameter to ensure that it meets certain criteria, such as being a valid version number and not containing any malicious code. To mitigate Authorization Bypass vulnerabilities, you should ensure that proper access controls such as role-based access controls (RBAC) are in place, and that client requests are authenticated and authorized before accessing sensitive data or functionality.To mitigate Code Execution vulnerabilities, you should validate and sanitize all client input to prevent malicious code from being injected into requests.

IX. CONCLUSION

Black box testing is an important technique in software testing that involves generating and sending a large number of inputs to a program to detect faults or vulnerabilities in its code. As GraphQL grows in popularity, it becomes increasingly important to fuzz it accurately. In this paper, we demonstrate the gaps and limitations in today's popular web-testing tools namely StackHawk and OWASP ZAP when it comes to GraphQL testing. Our evaluation has highlighted that there is still a lot of work to be done in the area of GraphQL security analysis. While StackHawk is arguably the best enterprise tool available, our evaluation has highlighted its limitations. Furthermore, as GraphQL is still in its early stages, there is not yet enough research done in this area. Overall, our research provides insights into the effectiveness of StackHawk in detecting security vulnerabilities in GraphQL APIs, but there is still much more to be explored and improved upon in this field.

X. Acknowledgment

We would like to thank Dr. David Brumley for his valuable comments and suggestions to improve the quality of the paper and for reviewing our work regularly. We would also like to thank Ben McWilliams from the StackHawk team for guiding us on StackHawk internals and their workings.

References

[1] blog.octo.com (2023). GraphQL: A new actor in the API world. [online]. Available: `https://blog.octo.com/en/graphql-a-new-actor-in-the-api-world/`.

[2] Apollo GraphQL. "The Anatomy of a GraphQL Query." Apollo GraphQL Blog. [Online]. Available: `https://www.apollographql.com/blog/graphql/basics/the-anatomy-of-a-graphql-query/`.

[3] Belhadi, A., Zhang, M., & Arcuri, A. (2021). White-Box and Black-Box Fuzzing for GraphQL APIs. Kristiania University College.

[4] Yazdipour, S. (2020, May 27). Github Data Exposure and Accessing Blocked Data using the GraphQL Security Design Flaw. arXiv.Org. https://arxiv.org/abs/2005.13448

[5] owasp.org (2023). GraphQL Cheat Sheet. [online]. Available: `https://cheatsheetseries.owasp.org/cheatsheets/GraphQL_Cheat_Sheet.html`.

[6] owasp.org (2023), OWASP ZAP [Online]. Available: `https://owasp.org/www-project-zap/`.

[7] zaproxy.org (2023), OWASP ZAP Documentation: Statistics [Online]. Available: https://www.zaproxy.org/docs/statistics/.

[8] stackhawk.com (2023). ZAP vs. StackHawk: Dynamic Application Security Testing Tool Comparison [online]. Available: `https://www.stackhawk.com/blog/zap-vs-stackhawk-comparison/`

[9] zaproxy.org (2023), OWASP ZAP Blog: Introducing the GraphQL add-on for ZAP [Online]. Available: https://www.zaproxy.org/blog/2020-08-28-introducing-the-graphql-add-on-for-zap/.

[10] zaproxy.org (2023), OWASP ZAP Documentation: GraphQL Support Script [Online]. Available: https://www.zaproxy.org/docs/desktop/addons/graphql-support/script/.

[11] zaproxy.org (2023), OWASP ZAP Documentation: GraphQL Support Automation [Online]. Available: https://www.zaproxy.org/docs/desktop/addons/graphql-support/automation/.

[12] OWASP ZAP GitHub: Issue #7324 [Online]. Available: https://github.com/zaproxy/zaproxy/issues/7324.

[13] OWASP ZAP GitHub: Issue #7323 [Online]. Available: https://github.com/zaproxy/zaproxy/issues/7323.

[14] stackhawk.com (2023). StackHawk [online]. Available: `https://www.stackhawk.com/product/`.

[15] docs.stackhawk.com (2023). StackHawk GraphQL Configuration [online]. Available: `https://docs.stackhawk.com/hawkscan/configuration/graphql-configuration.html`.

[16] doyensec. (n.d.). GitHub - Doyensec/inql: InQL - A Burp Extension for GraphQL Security Testing. GitHub.

[17] Dolev Farhi. "Damn-Vulnerable-GraphQL-Application." GitHub, 14 Apr. 2021, commit 1a2b3c4, https://github.com/dolevf/Damn-Vulnerable-GraphQL-Application.

[18] ReactFinland. "graphql-api." GitHub, 13 Apr. 2021, commit 1a2b3c4, https://github.com/ReactFinland/graphql-api.

[19] TCGDex. "cards-database." GitHub, 16 Apr. 2021, commit 1a2b3c4, https://github.com/tcgdex/cards-database.

[20] Vitor Salgado. "camara-deputados-graphql." GitHub, 20 Apr. 2021, commit 1a2b3c4, https://github.com/vitorsalgado/camara-deputados-graphql.

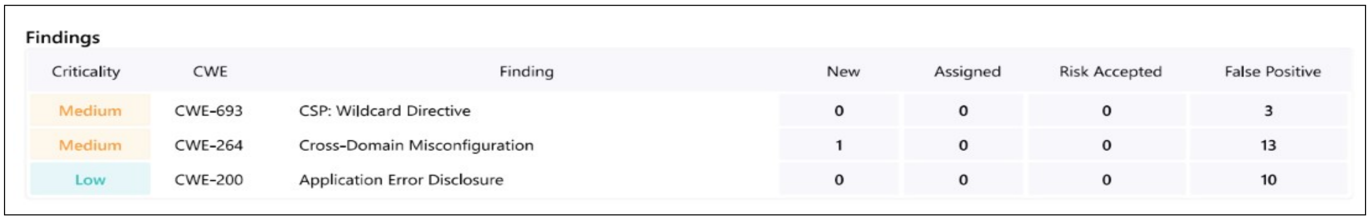
[21] nathanrandal.com. (2023). Introspection Query. [online]. Available. `http://nathanrandal.com/graphql-visualizer/`.

[22] "Introspection" graphql.org. [Online]. Available:`https://graphql.org/learn/introspection/`

[23] Neuse, J. (2021, September 1). The complete GraphQL Security Guide: Fixing the 13 most common GraphQL Vulnerabilities to make your API production ready. WunderGraph.

[24] graphql-kit. (n.d.). GitHub - Graphql-kit/graphql-apis: A collective list of public GraphQL APIs. GitHub. from https://github.com/graphql-kit/graphql-apis

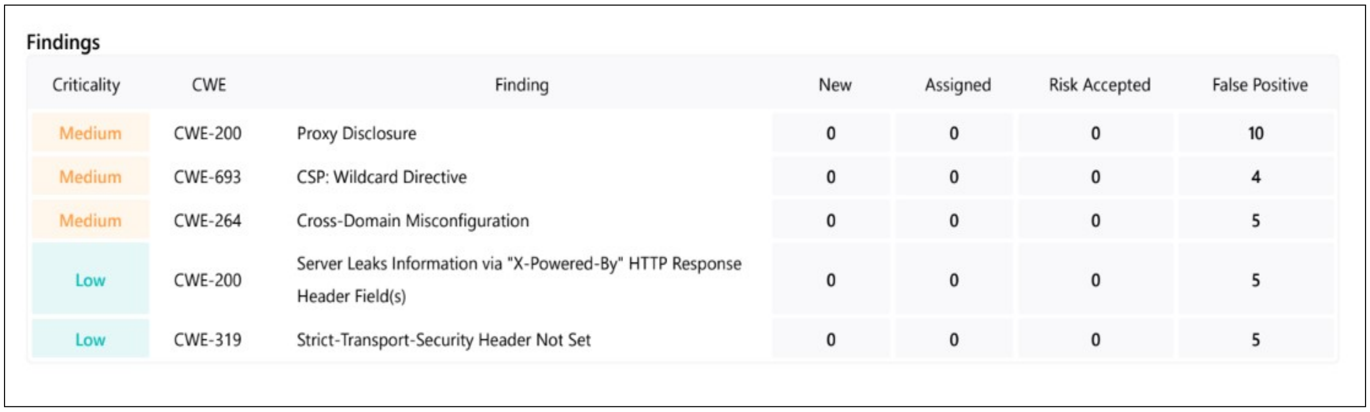
XI. APPENDIX



Findings

Criticality	CWE	Finding	New	Assigned	Risk Accepted	False Positive
Medium	CWE-693	CSP: Wildcard Directive	0	0	0	3
Medium	CWE-264	Cross-Domain Misconfiguration	1	0	0	13
Low	CWE-200	Application Error Disclosure	0	0	0	10

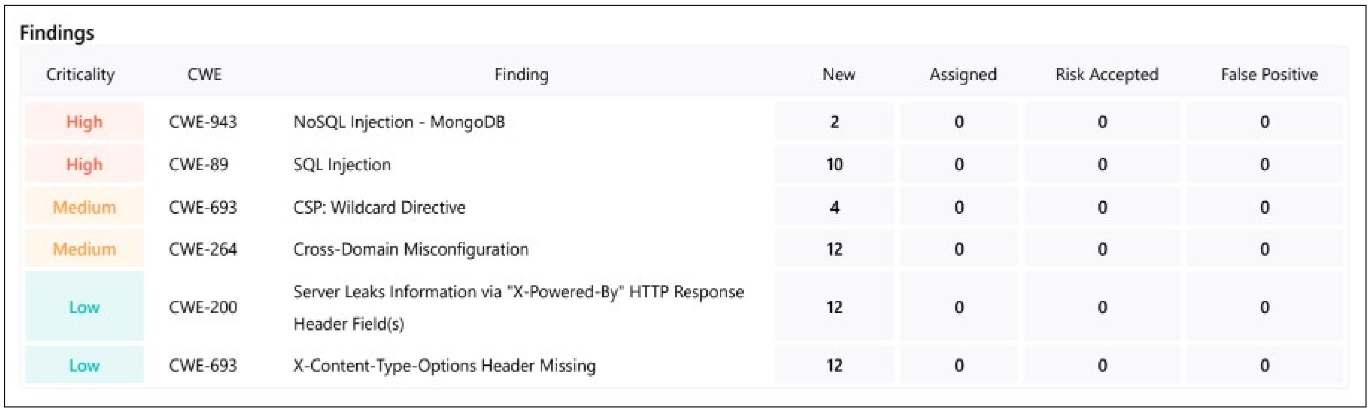
Fig. 11. TOSKA React Finland Report



Findings

Criticality	CWE	Finding	New	Assigned	Risk Accepted	False Positive
Medium	CWE-200	Proxy Disclosure	0	0	0	10
Medium	CWE-693	CSP: Wildcard Directive	0	0	0	4
Medium	CWE-264	Cross-Domain Misconfiguration	0	0	0	5
Low	CWE-200	Server Leaks Information via "X-Powered-By" HTTP Response Header Field(s)	0	0	0	5
Low	CWE-319	Strict-Transport-Security Header Not Set	0	0	0	5

Fig. 12. Pokemon TCG Cards StackHawk Report



Findings

Criticality	CWE	Finding	New	Assigned	Risk Accepted	False Positive
High	CWE-943	NoSQL Injection - MongoDB	2	0	0	0
High	CWE-89	SQL Injection	10	0	0	0
Medium	CWE-693	CSP: Wildcard Directive	4	0	0	0
Medium	CWE-264	Cross-Domain Misconfiguration	12	0	0	0
Low	CWE-200	Server Leaks Information via "X-Powered-By" HTTP Response Header Field(s)	12	0	0	0
Low	CWE-693	X-Content-Type-Options Header Missing	12	0	0	0

Fig. 13. Brazilian Deputies Chamber Application

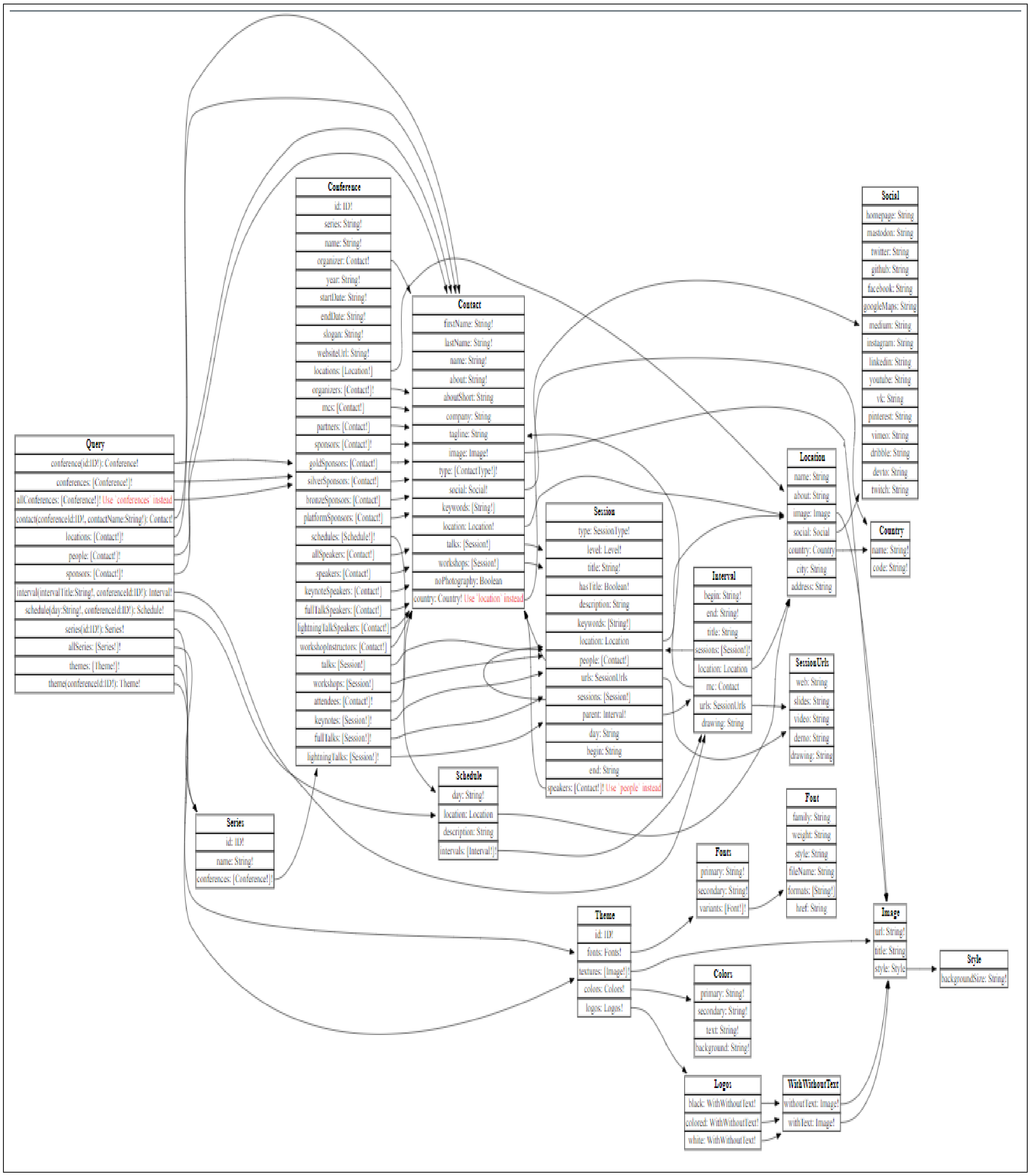


Fig. 14. TOSKA Conference Application

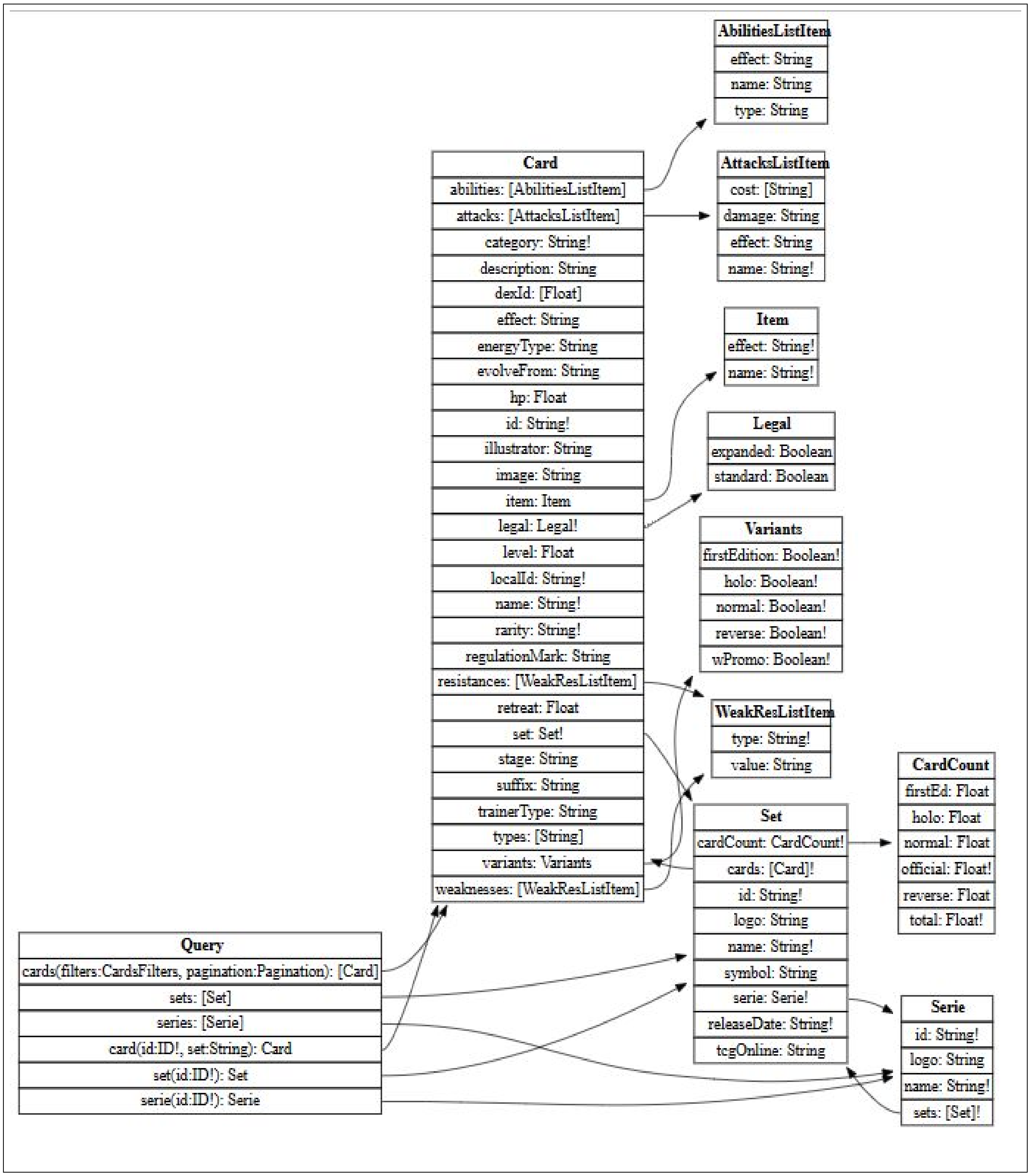


Fig. 15. Brazilian

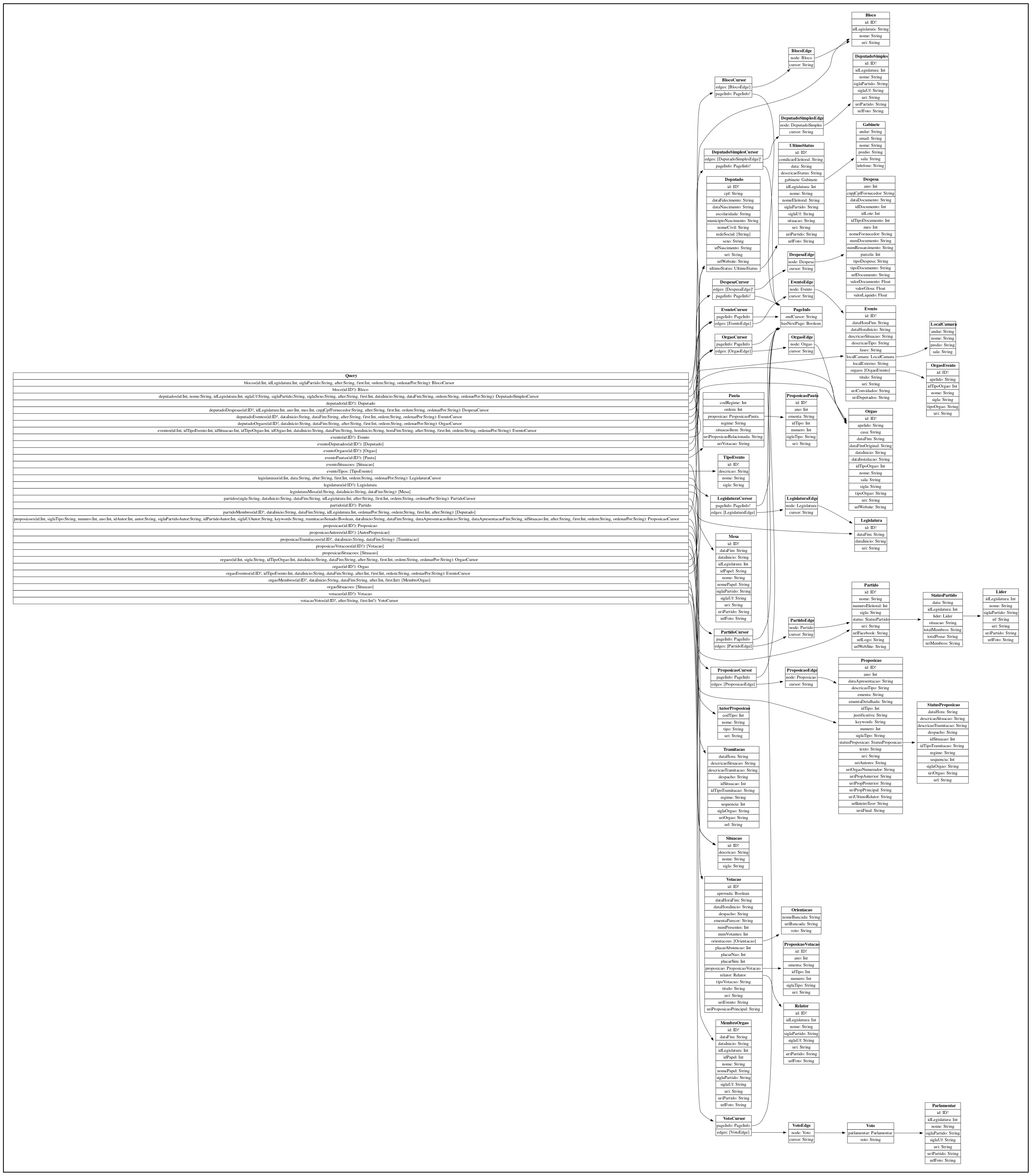


Fig. 16. Brazilian Deputies Chamber Application