# Survey of How to Secure GraphQL APIs

Project Report for TDDD17 - Information Security Course - Linköping University, Sweden

Björn Edlund
bjoed010@student.liu.se

David Åström
davas593@student.liu.se

Supervisor: Olaf Hartig
olaf.hartig@liu.se

***Abstract*—GraphQL is a relatively new query language for Web APIs developed by Facebook which aims to offer a more flexible way to query data from a web server. As it grows in popularity and adaptation, the potential attack surface grows and thus security aspects of the language have been put under bigger scrutiny.**

**This report documents a survey trying to map the current state of GraphQL security research and discussions in 2020. The survey was made by searching through scientific articles, documentation, and tech blogs in order to find the threats and vulnerabilities currently present in the language. These were categorized according to the CIA triad, followed by searching for and documenting proposed solutions to these risks.**

**Many risks found were similar to those present in classic REST APIs, but the increased freedom in queries in GraphQL leads to some availability risks more unique to GraphQL.**

**Several proposed solutions to the risks were found through blog posts and documentation but the survey found that the field is relatively untouched in the formal research. This leads to the conclusion that although many risks are documented and solved, more formal research is needed in the subject, which this report could act as a basis for.**

## I. Introduction

This report is the result of an obligatory part of the course Information Security (Course code: TDDD17) at Linköping University.

### A. Background

After being built by Facebook and released as open source in 2015, GraphQL has quickly increased in popularity as an alternative to the classic REST API when building web pages. Today it is used by large companies and services such as Airbnb, GitHub, and Twitter [1].

As the relatively new technology emerges and some of the largest services in the world adopts it, one would argue that it is very relevant to look into the security implications the use of GraphQL brings with it. Since the technology is starting to mature, more methods and technologies for securing GraphQL applications are being discussed in forums, and more research is being made on the subject. This report is the result of a survey that aims to map the current state of GraphQL security research and method recommendations in the spring of 2020. What threats and vulnerability have been found, and what solutions are being proposed to these risks?

### B. Problem statements

In order to better understand the vulnerabilities of GraphQL the survey will first try to answer:

1) What security vulnerabilities does the use of GraphQL come with and how do they relate to the CIA triad.

Based on the vulnerabilities and risks found, the following questions will be explored:

2) How can Loss of Confidentiality be prevented in GraphQL.
3) How can Loss of Integrity be prevented in GraphQL.
4) How can Loss of Availability be prevented in GraphQL.

### C. Method

The survey was conducted in two main stages. First, the aim was to identify the security risks that GraphQL imply when implemented.

The search was conducted by first reading the official documentation for GraphQL, thereby getting the officially recognized risks.

In addition to these, an online search for tech blog posts, tutorials, and scientific reports which presents and discusses security in GraphQL was done. This search was conducted in a two stages:

- In order to find scientific reports on the subject, a search was made via Google Scholar, using the search query ”GraphQL security”. From the results, articles that seemed interesting for the survey were chosen to read and analyse. The references of the found articles were also used to find further information.
- To find blog posts and tutorials, two methods were used. First, a search on Google.com was made, also using the query ”GraphQL security”. Interesting posts from the results was analysed further. A search on Hacker News, a popular news site for computer science related subjects, was also made. Once again the same query as above was used, and interesting results chosen.

The found risks was then categorized according to which part of the CIA triad they belonged to. If said issue was not applicable to either one of the three categories, the issue would not be taken into consideration in this survey.

Once this was done, the survey moved into the second stage. Here the same material was analysed in order to try to find proposed solutions for every given issue. A second search similar to the first was also conducted by searching for each given problem specifically to find further proposed solutions. These solutions were then summarized in order to answer questions 2, 3, 4.

### *D. Limitations*

No first-hand experience of the API in question will be evaluated in this project.

## II. Theory

This section will cover some of the basic theory that acts as a basis for the rest of the report.

### *A. GraphQL*

GraphQL is a query language for web APIs developed by Facebook. It acts as an alternative to the more classic REST APIs, with the main aim to reduce the amount of over and under fetching of data that is common in REST applications. Instead of the server providing multiple endpoints for the client to send requests to, each with a specified set of data to return, a GraphQL server provides a single endpoint for the client to query. That endpoint accepts POST requests containing a GraphQL query which specifies what data the client wants to access. By doing this, the client can query exactly the data it needs, and thereby avoids over fetching data not needed, or under fetching resulting in more queries needed [2].

This method also helps with separating the client and server from each other. As the client is no longer dependent on certain endpoints and the data they provide, client development can be done relatively independently of the server, as the client can simply change the queries sent to the single endpoint [2].

*1) Client:* The client part of a GraphQL service sends request to the server. The requests are always POST requests, and the requested data is specified in the request body as a GraphQL query. In its basic form this query specifies what types and what fields of that type the client wants to retrieve [3]. A small example can be seen below:

```
{
  user {
    name
    email
  }
}
```

The query can be expanded further, for example specifying which instances of the object to retrieve, as well as retrieve related types, such as messages that a user has posted.

```
{
  user {
    name
    email
    messages {
      message
    }
  }
}
```

In addition to retrieving data, the query has two more specific use cases, mutations and subscriptions. A mutation query is used to mutate data, either by creating new types, or update fields in existing objects [1].

A subscription query opens a steady connection to the server, and is used to subscribe to real-time updates of the data. When a subscription is in place, the client receives the updated data whenever it is changed on the server, allowing for the real-time applications common today [4].

*2) Server:* Most of the GraphQL API is implemented on the server side. The main part of the server is the GraphQL schema. The schema defines what can be queried by the client. It defines the types, what fields the types contain, and the relations between types. In addition to this, the schema also defines the possible queries, mutations, and subscriptions the client can make [4].

The server must also define resolvers for each of the types and queries. However, GraphQL does not care about how these resolvers are implemented, as along as they return the promised data. This means that a GraphQL server can be implemented in pretty much any way, framework, or language. It can use any database system, and fetch data from other third party APIs, thereby connecting several systems and services to a single endpoint [5].

### *B. Definition of the CIA triad*

The CIA triad is a common way to categorize key objectives that need to be secured in any system. They consist of Confidentiality, Integrity, and Availability and can be described as follows:

1) *Loss of Confidentiality* - Assurance that information is not accessed by unauthorized entities such as programs and people [6].
2) *Loss of Integrity* - Information must retain its accuracy and be protected from unauthorized modification or removal [6].
3) *Loss of Availability* - Authorized users must be able to access information or information systems in a timely manner [6].

## III. Results

The survey resulted in a list of risks and vulnerabilities which can be studied in table I. Most of the risks were found in blogs and tutorial, but a few were found in the official documentation and articles as well.

### *A. What security vulnerabilities does the use of GraphQL come with and how do they relate to the CIA triad.*

The results clearly shows that most risks are either related to fetching or modifying data the user is not authorized to access, or about creating malicious queries that in some way overloads the server, thereby causing availability issues.

The official documentation for GraphQL mainly focused on specifying the underlying principles of the query language, rather than any specific implementations of it, and therefore did not touch many security aspects. It did however mentioned that GraphQL does not handle authentication or authorization by itself [7], [8], as well as also mentioning the issue with invalid and cyclic queries possibly crashing the server if the incoming queries are not validated properly.

TABLE I
RISKS AND VULNERABILITIES FOUND IN GRAPHQL

| ID | Description | C | I | A |
|---|---|---|---|---|
| **Official Documentation** | | | | |
| 1 | Unauthenticated queries [7] | X | X | |
| 2 | Unauthorized access to data [8] | X | X | |
| 3 | Invalid and cyclic queries [9] | | | X |
| **Scientific Articles** | | | | |
| 4 | Superlinear response sizes causing very large responses [10] | | | X |
| 5 | Unusually Big Queries [11]( [12]) | | | X |
| **Blogs and Tutorials** | | | | |
| 6 | Unusually deep queries [12], [13] | | | X |
| 7 | Queries requesting large amount of data [12], [13] | | | X |
| 8 | Unusually Frequent Queries [12] | | | X |
| 9 | SQL injection [14] | X | X | |
| 10 | NoSQL injection [15] | X | X | |
| 11 | Unauthorized access to data [16] | X | X | |
| 12 | Unauthorized access to data through parameter smuggling, insecure direct object referencing [17], [18] | X | X | |
| 13 | Cross Site Scripting [18] | | X | |

No articles that focused on security issues with GraphQL were found, but a few instances noted on individual security issues. Both mentioned the issue of unusually large queries causing very large responses, risking the stability and availability of the server [10], [11].

Blogs and articles proved the most valuable in the search. A few of these mentioned similar issues as the documentation and articles. These issues were mainly large and deep queries causing possible denial of service. HowToGraphQL also touched on the issue with a large amount of frequent requests overloading the server, as is the issue with most types of web services and APIs [12].

Many blogs also noted some vulnerabilities concerning confidentiality and integrity. Partially in general like the documentation with unauthorized access to data [16], but some more specific examples were also brought up. As GraphQL is an entry point to the server like any REST API, it may be utilized to do many common web attacks. Some of these are SQL injection and NoSQL injection, where GraphQL queries can contain executable code if not escaped properly. Attacks like XSS could also be possible this way [18]. Different types of direct object referencing were noted as well, where the user possibly can access data it is not authorized to see by smuggling parameters in the query if the data is not properly protected [17], [18].

### *B. How can Loss of Confidentiality and Integrity be prevented in GraphQL.*

Since most threats regarding confidentiality and integrity concerned both aspects, this subsection will address both Question 2 and 3.

1: Implementing user authentication in GraphQL can be handled similarly to in a REST API. As GraphQL utilizes HTTP requests just like REST, you can utilize a client-provided authentication token in the request header. The official documentation mentions this case in the context of an ExpressJS GraphQL server, and recommends using an Express middleware that processes the token in the resolver [7].

A blog post on Medium.com [19] discusses how to implement authentication in a GraphQL context, and concludes that authentication is best handled in the GraphQL server rather than the HTTP server or the Business logic, giving the best control of the authentication flow. In this example it is done by implementing a context in the server, which checks for a present token and fetches a user from it. This context is then present for all resolvers, which enables them to access the current user for further authorization [19].

2, 11, 12: All these risks are concerned with the authorization of users to access certain data. In order to implement authorization you would of course need to first implement authentication as above [19]. Although the documentation mentions that it is possible to implement authorization on a GraphQL level, it would require a lot of duplicate code as soon as a project grows past the experimental stage. This would result in both very tedious work developing, but more importantly the issue where changes or mistakes causing the authorization checks not being perfectly in sync throughout the server could introduce possible exploits like parameter smuggling and insecure direct object referencing [8].

Instead the documentation recommends that authorization should not be handled by GraphQL itself, but rather be delegated to the business logic layer of the server. This would provide a single point of truth for the authorization, thereby avoiding the issue with the authorization being out of sync throughout the server.

9, 10, 13: Both these points relate to injection attacks. Because of the way GraphQL is structured, it both solves and creates some common injection vulnerabilities. Each query or mutation is strongly typed in GraphQL. This means the system is already protected from malicious users injecting unexpected types in fields, simply rejecting queries that does not follow the specified structure [20]. It is however worth noting that by using naively implemented custom scalars to create more flexible queries you can open the server up for injection attacks. This can instead be mitigated by compromising on the flexibility in the query definition, and specify exactly which types of input you can expect to receive [21].

Like any API that accepts user input, GraphQL is also susceptible to more common SQL injection attacks, where finely constructed strings can modify a database query if passed on directly. This is also not a problem specific to GraphQL specifically, but must be solved by ensuring any strings are escaped when resolving the query, before passing the query to the database [14]. The same is true for defending against XSS exploits. The input strings must be properly escaped from characters that may cause scripts before being passed of to the database [18].

### C. How can Loss of Availability be prevented in GraphQL.

This subsection will list the methods found that solve the Availability issues found previously

4: In the article *An Empirical Study of GraphQL Schemas* by Erik Wittern et al. [10] the authors try to understand how GraphQL is used in open-source and in industry. However for our purposes Wittern et al. also examines response times and later goes into detail why this is a potential security problem. Since nested queries in GraphQL are applied to all returned objects of the parent node, nested queries can grow in size exponentially [10].

A cubic query can for example be achieved by asking for the friends of friends of friends:

```
query{
        friends(first 5){
            friends(first 5){
                friends(first 5){birthday}
                }
        }
    }
```

This query's response would, according to Wittern et al. [10], function regarding polynomial response for GraphQL be calculated in the following way:

$$\mathcal{O}((n-K)*D^K)$$

*where:*

1) $D$ is the length of the retrieved object list.
2) $K$ is the maximum number of nested lists.
3) $n$ is the size of the query.

Which gives us:

$$1+5*(1+5*(1+5*1))=156$$

Note that the $+1$ is there because GraphQL adds an additional name field to all lists. Queries with hefty responses such as these could be used in DoS attacks to overload the server [10]. These can be mitigated as described in 5, 6 and 7 utilizing depth and complexity. Wittern et al. [10] also introduces some ways to reduce exposure to DoS attack and making the server have a reasonable work load; namely paging. In which a limit is set, for example the maximum amount of friends of users you can request [22] [10].

6: Since GraphQL itself does not limit depth (or any part of the construction of a request) [12], it is possible to for example request the friends of friends of friends and so on from a database of users in a social media site.

```
{                               depth 0
  user {                        depth 1
    friends{                    depth 2
        friends{                depth 3
            [...]               depth n
        }
    }
  }
}
```

One way to limit deep queries or recursive queries is to introduce depth in which the server throws an error message and does not compute queries for all requests with a depth count over an arbitrary number n. This has the added benefit that it stops DoS attacks in which complex deep queries spends all server resources [18]. This does however not stop unusually large request with many nodes in the root layer.

7 5: As stated above, queries can still request large amounts of data if they simply pack the root node full of nodes, traversing *broad* instead of deep as above [12]. This can be mitigated with complexity layers. Where each node have a complexity value and the complexity of the whole query is the sum of all nodes' complexity value [18].

```
{
  user {                        complexity 1
    friends (first 5){          complexity 5
    }
  }
}
```

If the server is configured to allow a complexity of maximum 4 this query above would fail, since it has a complexity of 6.

8: This is most often solved by simply limiting the amount of requests available in a specific server side set time frame. For example letting each client have a "pool" of time which refills at a certain rate (for example 50 ms per second with a maximum pool of 1000 ms) [12]. Another, more elegant way is to throttle based on complexity as described at 7. Where clients have a pool of complexity which depletes based on the complexity of request. This makes the client implementation easier as they do not have to estimate server side computation time but instead exactly calculate complexity over time [12].

## IV. Discussion

When analysing the results, one can see some interesting patterns and draw some conclusions regarding the current state of GraphQL security. What has become clear is that GraphQL, in many aspects, is very similar to other APIs, being affected by many of the same threats and vulnerabilities. This may not come as a surprise as GraphQL is still based on HTTP requests, and need to fulfill many of the same functions as for example a REST API.

One can however note that GraphQL is simply a query language for server-client communication, and only a part of a complete web application. It is therefore naive to believe that GraphQL alone would be responsible for all security aspects of the system. Take the authorization aspect as an example, which had shown to probably be better to delegate to the business logic of the application.

What is more interesting to analyse is the security aspects that GraphQL and its principles more specifically entail. Having a more flexible way to query data gives more control to the client and user about what data to query. This does, not very surprisingly, seem to be the main security risks with GraphQL. The survey shows that most risks discussed or explored have been related to users sending unexpected queries. Either very large or deep queries which, if not handled correctly, may overload the server and cause availability issues, but also queries containing malicious input such as injections or scripts. Many risks seem to have been found and many ways to limit or better specify queries have been developed to counteract them. However, with GraphQL being relatively young compared to for example REST, there are great chances that more possible vulnerabilities exists, or will emerge from further development, meaning this area could be a good entry point for further studies on the subject.

This leads on to the next big point about the survey. There seem to have been very little formal research done into the subject of GraphQL security. This clearly shows from the lack of articles found discussing the subject when searching for materials for this survey. There is definitely room for further studies on the subject. As this survey mainly focuses on how GraphQL is discussed in the community, it could be interesting to study how security features are actually implemented on the web today, compared to the theoretical background of how they "should" be implemented. Similar studies have been done mapping the general use of GraphQL on the web [10], [11]. These have however mainly focused on how GraphQL performs regarding over and under fetching compared to REST, so a more security focused study could be interesting to see.

## V. Conclusion

As GraphQL increases in use, the research of how to develop secure GraphQL APIs is more relevant than ever. This survey has mapped the current landscape for risks and vulnerabilities related to the implementation of GraphQL in web services. It shows that GraphQL is concerned with similar threats as classic REST APIs, but that the more flexible queries that characterizes the language, also opens up the possibility of an attack, both in order to cause denial of service, or access or modify confidential information.

The survey also highlights the lack of formal research in the subject, indicating that this is a field to study further. With this work, we hope to have helped lay a foundation upon which this further research can be based upon.

## References

[1] "Graphql: A query language for apis." [Online]. Available: https://graphql.org/

[2] "Graphql vs rest - a comparison." [Online]. Available: https://www.howtographql.com/basics/1-graphql-is-the-better-rest/

[3] "Queries and mutations." [Online]. Available: https://graphql.org/learn/queries/

[4] "Graphql core concepts tutorial." [Online]. Available: https://www.howtographql.com/basics/2-core-concepts/

[5] "Graphql architecture big picture." [Online]. Available: https://www.howtographql.com/basics/3-big-picture/

[6] "Cia-triad," accessed: 2020-04-06. [Online]. Available: http://www.e2college.com/blogs/risk_management/confidentiality_integrity_availability.html

[7] "Authentication and express middleware." [Online]. Available: https://graphql.org/graphql-js/authentication-and-express-middleware/

[8] "Authorization." [Online]. Available: https://graphql.org/learn/authorization/

[9] "Validation." [Online]. Available: https://graphql.org/learn/validation/

[10] E. Wittern, A. Cha, J. C. Davis, G. Baudart, and L. Mandel, "An empirical study of graphql schemas," in *International Conference on Service-Oriented Computing*. Springer, 2019, pp. 3–19. [Online]. Available: https://arxiv.org/pdf/1907.13012.pdf

[11] T. Taskula, "Advanced data fetching with graphql: Case bakery service," G2 Pro gradu, diplomityö, 2019-03-11. [Online]. Available: http://urn.fi/URN:NBN:fi:aalto-201903172287

[12] "Howtographql," accessed: 2020-04-09. [Online]. Available: https://www.howtographql.com/advanced/4-security/

[13] "Securing your graphql api from malicious queries," accessed: 2020-04-06. [Online]. Available: https://blog.apollographql.com/securing-your-graphql-api-from-malicious-queries-16130a324a6b

[14] "Discovering graphql endpoints and sqli vulnerabilities," accessed: 2020-04-09. [Online]. Available: https://medium.com/@localh0t/discovering-graphql-endpoints-and-sqli-vulnerabilities-5d39f26cea2e

[15] M. T. You, "Learning graphql mongodb security vulnerabilities," Dec 2019. [Online]. Available: https://medium.com/@mrthankyou/learning-graphql-mongodb-security-vulnerabilities-b52f7e26ee24

[16] "The $30,000 gem: Part 1." [Online]. Available: https://www.hackerone.com/blog/the-30-thousand-dollar-gem-part-1

[17] "Bypass account level permissions through parameter smuggling," accessed: 2020-04-09. [Online]. Available: https://labs.detectify.com/2018/03/14/graphql-abuse/

[18] "Graphql - security overview and testing tips," accessed: 2020-04-09. [Online]. Available: https://blog.doyensec.com/2018/05/17/graphql-security-overview.html

[19] D. Simha, "Authentication and authorization in graphql (and how graphql-modules can help)," Mar 2019. [Online]. Available: https://medium.com/the-guild/authentication-and-authorization-in-graphql-and-how-graphql-modules-can-help-fadc1ee5b0c2

[20] P. Corey, "Nosql injection and graphql," Jun 2016. [Online]. Available: http://www.petecorey.com/blog/2016/06/13/nosql-injection-and-graphql/

[21] ——, "Graphql nosql injection through json types," Jun 2017. [Online]. Available: http://www.petecorey.com/blog/2017/06/12/graphql-nosql-injection-through-json-types/

[22] GraphQL, "Pagination," 2020. [Online]. Available: http://graphql.github.io/learn/pagination/