How to secure GraphQL APIs

Måns F. Franzén Nils Tyrén
Email: manfr689,nilty573@student.liu.se
Supervisor: Olaf Hartig, olaf.hartig@liu.se
Project Report for Information Security Course
Linköping university, Sweden

***Abstract*—GraphQL is a relatively new technology that allows for a more dynamic approach when it comes to sending queries to the back end of a web application. With new technologies comes new threats and this requires mitigations and control measures in order to secure an application that uses this API. The focus of this study has been to find threats to applications that uses GraphQL in the context of database security and how to mitigate these threats. The control measures found in this study can be divided into four different categories: Access control, data encryption, query analysis and input sanitizing. Some of these control measures are required in most web applications independent of what API is used and some are unique to GraphQL. Even though these control measure exist and can be used to secure an application that uses GraphQL, it is still up to the developer to make the right decisions and implement these measures depending on the security requirements of the application.**

I. Introduction

One of the most common web service APIs for communication between server and client is REST. REST is an architectural style that defines constraints for creating web services. One of the major problems with REST is that the endpoints are static in regard to what they return. This means that over and under fetching of data happens frequently. In 2015 a new web service API called GraphQL was released. GraphQL handles communication between client and server differently from REST. It was developed to be a more dynamic alternative to other web service APIs. Clients can write more complex queries and the GraphQL server will always return exactly what the client asked for.

Something that is still being examined is whether GraphQL is vulnerable to the same type of threats as other APIs. Perhaps some vulnerabilities are mitigated, and some new ones emerge. The goal of this project is to create a survey of approaches and techniques for securing GraphQL APIs against the typical threats of database security, which is loss of confidentiality, loss of integrity and loss of availability. Different threats and their respective mitigations are examined. Control measures to secure a GraphQL API are discussed. we also compare GraphQL with REST in terms of structure and security and discuss the use of GraphQL over HTTP and how the Apollo platform can be used with GraphQL.

II. Background

In order to understand the content of this report, short introductions to relevant concepts and techniques are presented in this section.

A. Database security

Database security refers to the techniques and measures taken in order to protect the contents of the database from being exposed or changed by unauthorized users. There are three main threats when it comes to database security:

B. Threats

1) Integrity

Loss of integrity can be described as unauthorized modification of data, meaning data in the database is changed without this being the intention. An example could be a criminal changing his criminal record in a database in order to receive a lesser punishment.

2) Confidentiality

Loss of confidentiality is the unauthorized disclosure of data. This means that data is exposed to someone that should not have access to that specific information. An example could be medical records being exposed to the public by someone taking advantage of a vulnerability in the database.

3) Availability

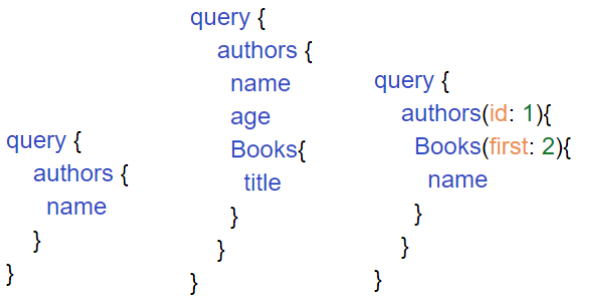
Loss of availability is when information that is supposed to be available for authorized users or programs cannot be displayed. This can happen for example if the server of an application shuts down due to some technical problem or if the server is overwhelmed by requests.

C. GraphQL

GraphQL is a query language for APIs that was released by Facebook as open-source in 2015. GraphQL was developed to provide a more efficient, powerful and flexible alternative to REST [1]. To be able to understand approaches and techniques for securing GraphQL APIs against the typical threats of database security we need to understand some core concepts and fundamental language constructs of GraphQL.

1) Query

A query begins with a field called root field. This root field can be seen as the entry point to the data graph. The following part of the query is the payload. In the payload you specify what information you want to request. In Figure 1 (a) authors is the root field and the only specified field in the payload is name. The response to this query will return the name of all the authors. Even though data of the authors age exists, this is not included in the response since it isn't specified in the



(a) Query (b) Nested query (c) Query with arguments

Figure 1: Query

payload. It is also possible to send multiple GraphQL queries with different root fields in one single request. By default, if you request the same root field twice in one single request the API will only respond with one response message with the requested information from both queries. It is however possible to send multiple queries in one single request and set them to be handled serially by GraphQL and by that if you send the same root field twice in a request the response will include two response messages from the same root field. This is referred to as batching [2] [3].

By having relations between types, you can query nested information. In Figure 1 (b) there is a relation between type author and type book. By that you can request the name of the author and the title of the authors books. With relations like this you can traverse through the data graph.
Queries can include arguments which makes it possible to request objects more specifically. In Figure 1 (c) the query will return a author with the id 1 and the authors first two books.

2) Mutation

Beside requesting information from a server, applications may need to add, update or delete information in the database. These changes can be made by using mutations. The structure of mutations is very similar to regular queries but the "mutation" keyword is used instead.

3) Introspection queries in GraphQL

Given an API, it is possible to ask a GraphQL schema for information about what queries it supports. By this the API somehow creates its own documentation [6] [7]. A real case example of introspection can be provided by using the tool GraphiQL and the Github API at https://developer.github.com [4].

D. GraphQL versus REST

Representational state transfer (REST) is a design architecture that is used in many web applications. With REST, data fetching is typically done by accessing multiple endpoints. These endpoints return a predetermined data set and is accessed by using different URLs. The data returned from a specific endpoint cannot be changed depending on the request.

In GraphQL on the other hand, the query sent to the GraphQL server would include all data requirements necessary. The data returned from the server depends highly on the structure of the query itself and can by that be sent to one single endpoint.
A common problem with REST is so called over/under-fetching. This means that the client receives more or less information than actually needed. If the client only needs a single data point and the only endpoint that returns the requested information also gives the client other data points, the client will receive more information than needed. This may not be a big problem in terms of functionality but will most likely decrease the efficiency of the program. It might also affect some aspects of security since more data is transmitted which could be intercepted by a potential attacker.
Another difference between REST and GraphQL is that introspecting is provided by default by GraphQL APIs. This feature is not available by REST APIs. Introspecting may be very useful for developers if the documentation of the API is defective [7].
Since GraphQL only uses one endpoint a failure of that endpoint will result in total failure to access data from the API. However, since this failure will occur due to server failure a REST API which uses multiple endpoints would have the same failure. By this no new security threat will emerge by using GraphQL compared to REST if the endpoint is well formed.

E. Server construction

At the core of every GraphQL-server is a so called "schema". The schema describes functionality that is available to the client application that connects to the GraphQL-server [11]. A schema can for example describe GraphQL-Object-types, define relations between these types and define root queries. In an application that handles books and authors a GraphQL-Object-type could for example be called a "book type". The creator of the application can define exactly what fields a book type should contain and the general structure of the data. An example of a book type written in JavaScript is shown in Figure 2. This type is called "Book" and contains four fields. An id, a name, a genre and an author which corresponds to an author ID. An author is in turn another GraphQL-Object-type that must also be defined. A resolve function is executed every time a query is used. This GraphQL-Object-type also defines a relation between a book and an author. When a user requests a book and also want to see the author of that book, the resolve function will look at the author ID from the requested book and find the correct author from where the authors are stored.
A root query can be defined as query that is directly available to the client when starting the application. If we continue with the previous example with books and authors, a root query could enable a client to fetch a specific book and a specific author based on certain arguments. An example written in JavaScript is shown in Figure 3. Here the only argument required for these queries is an ID. When a query is created with a correct ID, the resolve function will grab the object from where it is stored and return it.

```
const BookType = new GraphQLObjectType({
    name: 'Book',
    fields: ( ) => ({
        id: { type: GraphQLID },
        name: { type: GraphQLString },
        genre: { type: GraphQLString },
        author: {
            type: AuthorType,
            resolve(parent, args){
                return _.find(authors, { id: parent.authorId });
            }
        }
    })
});
```

Figure 2: Book type

```
const RootQuery = new GraphQLObjectType({
    name: 'RootQueryType',
    fields: {
        book: {
            type: BookType,
            args: { id: { type: GraphQLID } },
            resolve(parent, args){
                // code to get data from db / other source
                return _.find(books, { id: args.id });
            }
        },
        author: {
            type: AuthorType,
            args: { id: { type: GraphQLID } },
            resolve(parent, args){
                return _.find(authors, { id: args.id });
            }
        },
```

Figure 3: Root Query

F. GraphQL over HTTP

For communication between client and server HTTP is the most common protocol when using GraphQL. A GraphQL server operates over one single endpoint and should handle GET requests for queries and POST requests for queries or other operations [12] [13].
For HTTP to provide integrity and confidentiality it serves over TLS or SSL (HTTPS). These protocols provide communications security over the internet between client/server applications [8]. By using digital certificates client/server applications can authenticate each other and by that establish a secure connection [18].

G. GraphQL with Apollo

Apollo platform can be used to build a full-stack application by using GraphQL. On the server side you can set up a Apollo server which is a JavaScript GraphQL server and on the client side you can use Apollo client which is a complete state management library for JavaScript apps that will make it easier to fetch data via GraphQL [23]. Apollo also offers Apollo Link which makes it possible to manage over the network architecture [24]. Different techniques for authentication and authorization is offered when using Apollo [25]. These techniques are not implemented by default when using Apollo and shall therefore be implemented by the developer.

III. Method

In order to complete the goal described in Section I we followed a three-step approach when collecting and analyzing information.
The first step was to identify sources of information. We looked at blog posts, tutorials and documentation. All of the sources deemed to be relevant where collected in a shared document with a short description. This made it easier to track what information came from what source and finding information on a particular subject without having to perform another search.
The next step was to process the information. This step involved checking for validity, relevance and how intuitive the information was. It was at this step that much of the actual learning took place. Collaboration within in the project group was done by having multiple discussions about uncertainties in some of the topics and sharing information between project members.
The final step was to summarize the information. The report was structured in such a way that the goal of creating a survey of approaches and techniques for securing GraphQL APIs against the typical threats in the context of database security was met. Information was added successively along with their corresponding sources. Sources of information that may be relevant to the topic but not used as reference in this survey can be found in Appendix B.

IV. Result

One of the differences between REST and GraphQL is the possibility to do introspection queries. Even if this could be very useful for developers it also exposes GraphQL implementation details. This may not by itself be a threat, but indirectly, attackers could use these details to find exploits and cause loss of confidentiality, integrity and availability. For this reason it is sometimes recommended to disable introspection as a security measure [5] [10] [15].
One of the most common threats specific to GraphQL is the ability to send large nested queries [14]. Since GraphQL allows users to construct their own queries with a lot more flexibility than other APIs, this enables security threats that developers need to be aware of.

A. Confidentiality

1) Threats

Brute forcing. As in many applications, brute forcing can be used in GraphQL to send multiple mutations with different data to, for example, a login [10]. An attacker could send mutations to the GraphQL-server until the desired result is returned and the attacker could potentially compromise both confidentiality and integrity of the system.
Brute forcing can also be done specifically for GraphQL with batching attacks [19]. As mentioned previously this allows the

attacker to send multiple queries or mutations with the same root field within one request which can bypass some security checks.

If the GraphQL application uses an SQL-database and does not perform sufficient input-sanitizing it may be vulnerable to SQL-injections [22]. At some point the GraphQL server will communicate with the SQL-database after processing queries from users. An attacker could write an SQL-query where the program expects another argument. This could lead to information disclosure if the query ends up being sent to the database.

2) Mitigations

Data rate limiting. A solution to the problem of brute forcing is to implement some rate limit on the queries [10]. This could be a limit on the amount of queries a user can send over a specified time window. Rate limiting is used by the GitHub API. Each user has a rate limit of number of points the client is permitted to consume in a 60-minute window [26]. The more complex query or mutation a user makes the more points it costs. It is possible to query information about this limit and receive remaining points for the user. To get a better understanding of how effective GitHubs rate limiting is we created some test queries and mutations. As excepted the more complex queries resulted in a higher cost and using the same root field twice in a single request resulted in one single response message. We also tried batching multiple mutations in one request. We used the same root field for the mutations but different arguments. The response included one response message for each mutation as expected but the cost of the request was the same as for a request with a mutation with only one root field. This shows that the rate limit mitigation in this case wasn't effective against a batching attack.

When it comes to SQL-injections, the same mechanisms are involved in GraphQL as in other APIs. Input from users must be controlled and sanitized. This could be done by checking if common SQL-related symbols are used before processing a query.

B. Integrity

When it comes to integrity, the same kind of threats and mitigations that were presented in the confidentiality section applies. However, integrity is about ensuring data is not changed rather than unauthorized disclosure of data.

1) Threats

Batching attacks are also threats to the integrity of the application. If an attacker is able to access the system by brute forcing a login mutation for example, the validity of data may be compromised.

The same goes for SQL-injections. They can be used to send queries to change existing data in the database.

To maintain integrity some kind of authentication is often needed in the process of claiming an identity. Some kind of authorization that describes permission rules that specify users access right are also needed. Without these two there is a threat against integrity as well as against confidentiality.

While data is being transmitted between client/server applications it may be exposed to threats. Data can be read and modified by a third party via for example man-in-the-middle attack and by that be a threat against integrity and confidentiality.

2) Mitigations

The same mitigations for batching attacks and SQL-injection that where mentioned under confidentiality also apply here.

GraphQL by itself does not provide any authentication mechanism. Authentication can be done in different ways and how authentication should be done is up to the developer. The same situation arises when it comes to authorization. The GraphQL foundation recommends that all authorization logic should be delegated to the business logic layer and by that all entry points, no matter if it is REST or GraphQL, should be handled with the same authorization [17]. Suggestions and examples of how to implement authentication and authorization are provided by developer communities, documentation and blogs at for example Prisma, Apollo, Medium and the GraphQL foundation and can be found in Appendix A.

To securely transmit data between client and server HTTPS should be used as protocol. By using HTTP over TLS or SSL integrity is provided by calculating message digest and confidentiality is provided by data encryption. The client and server should have a digital certificate for authentication [8].

C. Availability

1) Threats

A potential attacker can make deeply nested queries that require extensive computations and large payloads. This can disrupt the network quality and lead to impaired performance [10]. This can be seen as a denial of service attack and affects the availability of the system.

Batching can also be used to perform denial of service attacks. The attacker can send multiple queries within one request and if the program processes these queries in parallel it can result in degradation of performance or even a complete shutdown of the server.

Another threat to the availability of the system is simply fetching many instances of an object. This will be an expensive operation regardless of the type of object [20].

2) Mitigations

A commonly used mitigation for deeply nested queries is depth limiting. A solution to this problem is to implement a limit on how deeply nested queries can be. There are several libraries that can perform this task, GraphQL Depth Limit is an example that let's the developer set their own depth limit.

Another mitigation for the use of complex queries is timeout. This strategy is fairly simple since it does not require the server to know anything about the structure of the arriving queries. All the server has to know is the maximum time that is allowed for a query [16].

To mitigate denial of service attacks performed using batching it is important to process all queries in one request serially (one after another). This will treat each query in the request as separate and will prevent overloading of the server.

A mitigation for the threat of queries that ask for multiple instances of an object is called amount limiting. Which can be implemented by setting the input argument to a custom scalar rather than an arbitrary integer. This scalar can be restricted to have a maximum value which would only allow a user to ask for a set number of instances of an object [20].

Despite the mitigations mentioned above there are still queries that could overwhelm the server by being right on the border of to deep or requesting too many objects. They would still be computationally expensive but would bypass the security checks. To solve this, you would have to analyze the complexity of the queries before running them. There are several ways of constructing complexity-calculating algorithms [21] [20].

V. Analysis

Since GraphQL is a relatively new technology on the market, new attacks and mitigations will most likely appear in the near future. However as discussed in the section IV a substantial number of threats and mitigations already exist. Some of these threats are not unique to GraphQL such as SQL-injections or Brute forcing. These kinds of threats are present in most web technologies. Some of the threats that are more or less unique to GraphQL includes deeply nested queries and batching attacks. These attacks take advantage of the fact that the queries in GraphQL are more dynamic and the user has more options when it comes to forming requests.

Despite the recent nature of these threats, many mitigations already exist as discussed in section IV. Mitigations such as rate limiting or timeout also exist for other web-APIs which may be the reason it was implemented so quickly. One mitigation that is fairly unique to GraphQL is the idea of calculating complexity of the incoming queries before processing them. This is a highly effective mitigation against several threats, such as ”deeply nested queries” or queries that request many instances of an object. An interesting point is that many of the functions that are implemented to increase usability and utility often lead to more threats. This is a point that is central in the field of database security. There has to be a balance between measures taken to increase security and the ease of use of the system. If security measures are to stringent, people will hesitate to implement them or find ways to avoid them. Introspection is an example of this where it exists to make things easier for developers but in turn introduces security vulnerabilities where potential attackers can get a better understanding of the underlying structure of the API. That is why the option to be able to disable introspection is important. This means that developers can make a conscious choice to either increase usability with the risk of introducing threats or to have a more confined system with less threats.

GraphQL is always used in the same context of other techniques. This survey has mentioned HTTP and Apollo. It is important to understand that such techniques may have its own security threats and other are used to mitigate these threats. Compared to database management systems (DBMS), which also provide database security, GraphQL is a relative new technology and by that there is no established control measures with approaches and techniques for securing GraphQL APIs that has been tested to provide the same protection as DBMS do.

VI. Conclusion

From the results and analysis of this survey these control measures are provided to maintain confidentiality, integrity and availability when using GraphQL.

- Access control:
  - Authentication and authorization of users to limit the access to the database or parts thereof
  - Authentication of client/server to establish a secure connection.
- Data encryption:
  - Preventing sensitive data when transmitting over the network.
- Query analysis:
  - Protecting your GraphQL API from malicious queries
  - Understanding the use of introspecting.
- Input sanitizing:
  - Protect against injections.

One of the most important conclusions drawn from this project is that the security of the application highly depends on the choices of the developer. GraphQL can be fairly secure compared to other technologies if the right measures are taken. The list above shows some of the most important control measures to avoid many of the most common security threats against GraphQL according to this study. To limit the access of the database some kind of authentication and authorization should be implemented. There are numerous ways of implementing this and new techniques may be developed in the future. HTTPS can provide authentication of clients/servers and also provides encryption of data when being transmitted over the network.

When it comes to query analysis, developers should at least use depth limiting and amount limiting as minimum protection since these measures are very easy to implement and still protects against many attacks. If the application has higher security requirements, query cost analysis can be used. It is a bit more difficult to implement but provides better coverage.

Input sanitizing is required for most web applications that takes user input. It involves checking for common symbols and characters used in languages such as SQL to prevent malicious code from executing.

In conclusion, GraphQL is a new and promising technology with many advantages.However, the simple fact that a technology is new and exciting does not mean developers can ignore to take necessary precautions. It is important to use the right control measures and be aware of the different risks to keep the application safe from security threats.

References

[1] Prisma.io *The Fullstack Tutorial for GraphQL* 2020-04-10. https://www.howtographql.com/

[2] Dawkins, Jake. *"Batching Client GraphQL Queries"*. Sept 19, 2018. https://blog.apollographql.com/batching-client-graphql-queries-a685f5bcd41b

[3] Facebook, Inc. 2020. GraphQL. Working Draft, Mar. 2020. Online at http: http://facebook.github.io/graphql/June2018/, retrieved on Apr. 24, 2020. (Mar. 2020).

[4] Github GraphQL API v4 2020. https://developer.github.com/v4/. (2020).

[5] Wallarm Inc. *Introspection*. Dec 5, 2019. 2020-04-13. https://lab.wallarm.com/why-and-how-to-disable-introspection-query-for-graphql-apis/

[6] Graphql.org. *"Why and how to disable introspection query for GraphQL APIs"*. 2020-04-12. https://graphql.org/learn/introspection/

[7] Chiazzo, Ignacio. *"Introspection in GrapQL"*. Medium, Feb 28, 2019. https://medium.com/@ignaciochiazzo/introspection-in-graphql-a5a5bd744a66

[8] IBM *"Cryptographic security protocols: TLS and SSL"*. 2020-04-17 https://www.ibm.com/support/knowledgecenter/SSFKSJ_8.0.0/com.ibm.mq.sec.doc/q009910_.htm

[9] Krawczyk, Hugo. "The order of encryption and authentication for protecting communications (or: How secure is SSL?)." Annual International Cryptology Conference. Springer, Berlin, Heidelberg, 2001.

[10] Nage, Tom. *"Protecting Your GraphQL API From Security Vulnerabilities"*. Medium, Dec 5, 2019. https://medium.com/swlh/protecting-your-graphql-api-from-security-vulnerabilities-e8afdfa6fbe4

[11] Tutorialspoint. *GraphQL-Schema*. 2020. https://www.tutorialspoint.com/graphql/graphql-schema.htm

[12] Apollograpgql.org. *"POST and GET format How to send requests to Apollo Server over HTTP."*. 2020-04-16 https://www.apollographql.com/docs/apollo-server/v1/requests/#post-requests

[13] GraphQL.org. *"Serving over HTTP"*. 2020-04-17 https://graphql.org/learn/serving-over-http/

[14] Nage, Tom. *Protecting Your GraphQL API From Security Vulnerabilities*. Dec 5 2019. https://medium.com/swlh/protecting-your-graphql-api-from-security-vulnerabilities-e8afdfa6fbe4

[15] Helfer, Jonas. *Disable Introspection in GraphQL-JS with a simple validation rule*. GitHub repository, Dec 7 2018. https://github.com/helfer/graphql-disable-introspection

[16] *Security and GraphQL* https://www.howtographql.com/advanced/4-security/

[17] Graphql.org. *"Authorization"*. 2020-04-12. https://graphql.org/learn/authorization/

[18] OWASP.org. *"Authorization"*. 2020-04-19. https://owasp.org/www-community/attacks/Man-in-the-middle_attack)

[19] Wallarm, Renata *"GraphQL Batching Attack"*. Dec 13, 2019. https://lab.wallarm.com/graphql-batching-attack/

[20] Stoiber, Max. *Protecting Your GraphQL API From Security Vulnerabilities*. Feb 21 2018. https://www.apollographql.com/blog/securing-your-graphql-api-from-malicious-queries-16130a324a6b

[21] Hartig, Olaf and Pérez, Jorge. *Semantics and Complexity of GraphQL*. Proceedings of the 2018 World Wide Web Conference. 2018.

[22] Choren, Matias. *Discovering GraphQL endpoints and SQLi vulnerabilities*. https://medium.com/@localh0t/discovering-graphql-endpoints-and-sqli-vulnerabilities-5d39f26cea2e.

[23] Apollo Docs. *The Apollo GraphQL platform*.2020-04-21 https://www.apollographql.com/docs/intro/platform/

[24] Huaser, Evans. *Apollo Link: The modular GraphQL network stack*. Jul 25, 2017. https://www.apollographql.com/blog/apollo-link-the-modular-graphql-network-stack-3b6d5fcf9244

[25] Apollo Docs. *Authentication. How to authorize users and control permissions in your GraphQL API*.2020-04-21 https://www.apollographql.com/docs/apollo-server/security/authentication/

[26] Github GraphQL API v4, *GraphQL resource limitations* 2020. https://developer.github.com/v4/guides/resource-limitations/graphql-resource-limitations (2020).

Appendix A
Authentication and authorization

Dawkins, Jake. *"Authorization in GraphQL"*. Apollo Blog, May 15, 2018. https://www.apollographql.com/blog/authorization-in-graphql-452b1c402a9

"Common Questions". HOW TO GRAPHQL. 2020-04-16 https://www.howtographql.com/advanced/5-common-questions/

Graph.cool. Apr 16, 2020. https://www.graph.cool/docs/reference/auth/overview-ohs4aek0pe

GraphQL.org. *"Authorization"*. 2020-04-12 https://graphql.org/learn/authorization/

GraphQL.org. *"Authentication and Express Middleware"*. 2020-04-12 https://graphql.org/graphql-js/authentication-and-express-middleware/

Apollo Docs. *Authentication. How to authorize users and control permissions in your GraphQL API*.2020-04-21 https://www.apollographql.com/docs/apollo-server/security/authentication/

Simha, Dotan. *"Authentication and Authorization in GraphQL (and how GraphQL-Modules can help)"*. Nov 7, 2018. https://medium.com/the-guild/authentication-and-authorization-in-graphql-and-how-graphql-modules-can-help-fadc1ee5b0c2

"Using OAuth 2.0 along with JWT in Node/Express)". Jul 3, 2019. https://medium.com/@rustyonrampage/using-oauth-2-0-along-with-jwt-in-node-express-9e0063d911ed

Quezada, Rodrigo. *"Authentication and Authorization Basics with GraphQL and REST"*. Prisma, 2020-04-8. https://www.prisma.io/tutorials/graphql-rest-authentication-authorization-basics-ct20

Sandoval, Kristopher. *"Security Points to Consider Before Implementing GraphQL"*. Nordic APIs, Apr 25, 2017. https://www.prisma.io/tutorials/graphql-rest-authentication-authorization-basics-ct20

Appendix B
Related articles

Sandoval, Kristopher. *"Security Points to Consider Before Implementing GraphQL"*. Nordic APIs, Apr 25, 2017. https://nordicapis.com/security-points-to-consider-before-implementing-graphql/

Mràz, David. *"GraphQL security in Node.js project"*. https://atheros.ai/blog/graphql-security-in-node-js-project

Helfer, Jonas. *Disabling Introspection*. https://webonyx.github.io/graphql-php/security/disabling-introspection

Bhargav, Abhay. *"The Hard Way: Security Learnings from Real-world GraphQL"*. Feb 17, 2019. https://www.abhaybhargav.com/from-the-trenches-diy-security-perspectives-of-graphql/

Szymanski, Matt *REST in Peace: Abusing GraphQL to Attack Underlying Infrastructure* 2020-05-01.

https://www.bugcrowd.com/resources/webinars/rest-in-peace-abusing-graphql-to-attack-underlying-infrastructure/