Categories of possible attacks on GraphQL

1. Batching attacks

- A brute-force attack specific to GraphQL (not always for other APIs) https://graphql.org/faq/best-practices/ [What are the security concerns with GraphQL?] section
- Definition of batching: "Batching is the process of taking a group of requests, combining them into one, and making a single request with the same data that all of the other queries would have made. This is usually done with a timing threshold." https://www.apollographql.com/blog/batching-client-graphql-queries https://www.apollographql.com/docs/graphos/routing/performance/query-batchingBatch attack first defined in machine learning scenario: https://arxiv.org/pdf/1907.05587 and mentioned in https://arxiv.org/pdf/1908.07000 with the following content
  "Carlini et al. show that, defenders can iden- tify purposeful queries for adversarial examples based on past queries and therefore, detection risk will increase significantly when many queries are made [11]. We call these attack sce- narios *batch attacks*. To be efficient in these resource-limited settings, attackers should prioritize "easy-to-attack" seeds "
- GraphQL batching attack explanation: https://lab.wallarm.com/graphql-batching-attack/ and https://checkmarx.com/blog/didnt-notice-your-rate-limiting-graphql-batching-attack/
-

2. Code Injection attacks

- Brief/talking about different kinds: https://pmc.ncbi.nlm.nih.gov/articles/PMC10346793/pdf/sensors-23-06067.pdf
  Code injection attacks are a type of cybersecurity threat in which an attacker injects malicious code into a vulnerable application or software. This code is designed to exploit security weaknesses in the application or software, allowing the attacker to gain unauthorized access to the system, steal sensitive data, or carry out other malicious activities and attacks [22,23]. Code injection attacks can be performed in various ways, including Struc- tured Query Language (SQL) injection [24], Cross-Site Scripting (XSS) [25], and command injection [26], etc.
  With very detailed explanation for different types of injection from p2-15
- Formal/Intense: https://cse.usf.edu/~ligatti/papers/code-inj.pdf
- Casual examples: https://inigo.io/blog/graphql_injection_attacks
- Detection of injection attacks in general: https://homepage.iis.sinica.edu.tw/~skhuang/p148-huang.pdf

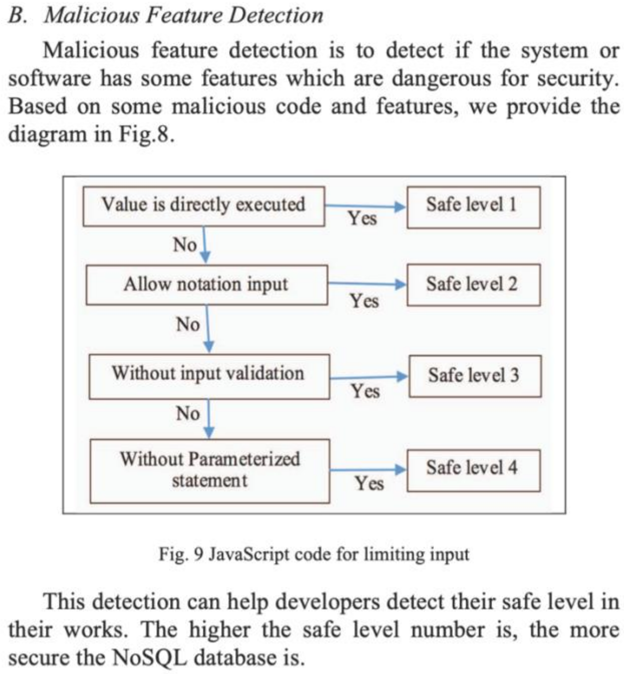
SQL injection

- specified as one of the GraphQL vulnerabilities [1]
- mentions about SQL injections with GrapQL queries: https://db.cs.uni-tuebingen.de/theses/2023/lukas-sailer/lukas-sailer-2023.pdf (p49)
- Description/Examples: https://owasp.org/www-community/attacks/SQL_Injection
- Related blog identifying SQL injection in graphQL: https://www.praetorian.com/blog/identifying-sql-injections-in-a-graphql-api/ [illustration]
- sqlmap: open-source penetration testing tool detecting and exploiting SQL injection https://github.com/sqlmapproject/sqlmap

- ML-based SQL injection detection tool: https://github.com/saptajitbanerjee/SQL-Injection-Detection
-

NoSQL injection

- specified as one of the GraphQL vulnerabilities [1]
- Explanation: https://www.invicti.com/blog/web-security/what-is-nosql-injection/
- MongoDB NoSQL injection detection: https://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=7545900 (p78)



B. Malicious Feature Detection

Malicious feature detection is to detect if the system or software has some features which are dangerous for security. Based on some malicious code and features, we provide the diagram in Fig.8.

Value is directly executed
Yes
Safe level 1
No
Allow notation input
Yes
Safe level 2
No
Without input validation
Yes
Safe level 3
No
Without Parameterized statement
Yes
Safe level 4

Fig. 9 JavaScript code for limiting input

This detection can help developers detect their safe level in their works. The higher the safe level number is, the more secure the NoSQL database is.

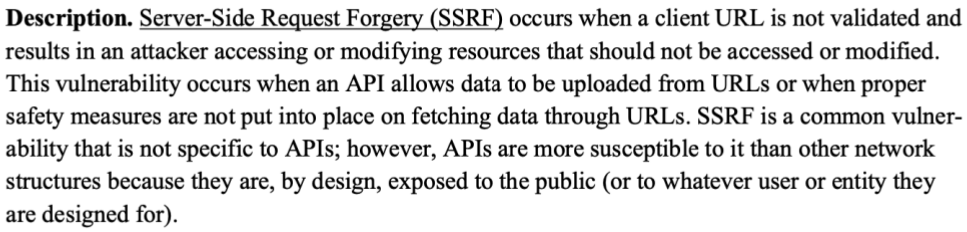
-

OS command injection

- specified as one of the GraphQL vulnerabilities [1]
- https://owasp.org/www-community/attacks/Command_Injection
- Commix: tool to detect command injection flaws https://citeseerx.ist.psu.edu/document?repid=rep1&type=pdf&doi=0f3bc42fda144276534bb1c5247342564474a041

SSRF Server-side request forgery injection

- Common API vulnerability [4] p11 and GraphQL vulnerabilities [1]



Description. Server-Side Request Forgery (SSRF) occurs when a client URL is not validated and results in an attacker accessing or modifying resources that should not be accessed or modified. This vulnerability occurs when an API allows data to be uploaded from URLs or when proper safety measures are not put into place on fetching data through URLs. SSRF is a common vulnerability that is not specific to APIs; however, APIs are more susceptible to it than other network structures because they are, by design, exposed to the public (or to whatever user or entity they are designed for).

- Definition: https://owasp.org/API-Security/editions/2023/en/0xa7-server-side-request-forgery/

3. Denial of service attack

- Identified as one of the GraphQL vulnerabilities [1] [3]

  Denial of Service Attacks. GraphQL enables clients to submit individual queries to the server for execution. This feature might be exploited by perpetrators generating excessive load through overly complex (single) queries possi- bly leading to service failure. Therefore, we need to introduce countermeasures securing GraphQL APIs against malicious queries.

  A naive approach against excessive requests is to **check the size of the query representation** before execution and restrict it to a certain limit. The GraphQL feature of persistent queries offers another simple way of protection. Here, all permitted query templates are stored on the server. Clients might only execute pre-defined queries, which can certainly been seen as benefit and drawback at the same time.

  More sophisticated approaches introduce cost metrics **quantifying the effort of query execution**. Resolver restriction builds on the weighted number of resolver calls for a query or, alternatively, the duration of resolver execution. If a request exceeds some given threshold, the resolver interrupts it. Obviously, harmless queries might be killed and malicious ones still run for some time.

  A better way would be to statically analyze queries prior to execution and disregard them in case of excessive complexity. Hartig and P´erez have shown that queries might lead to resulting object graphs of exponential size [10]. Luckily they also suggest that the size of a result object can be estimated with realistically low complexity. This would provide the theoretic foundation for effective DoS prevention to be build into GraphQL server frameworks.

  [10] Hartig, O., Perez, J.: An initial analysis of Facebook's GraphQL language. In: Pro- ceedings of the 11th Alberto Mendelzon International Workshop on Foundations of Data Management (AMW), Montevideo, Uruguay, 5–9 June (2017)

- GraphQL is susceptible to denial of service through complex queries

  https://link.springer.com/content/pdf/10.1007/978-3-030-33702-5.pdf (p3, p15)

  "Queries with super-linear response sizes can become security threats, overloading APIs or even leading to denial-of-service. For commercial GraphQL providers, exponential response sizes pose a potential security risk (denial of service). Even polynomial response sizes might be concerning — e.g., consider the cost of returning the (very large) cross product of all GitHub repositories and users."

- Tools for discovering DoS

  Wendigo: Deep Reinforcement Learning for Denial-of-Service Discovery in GraphQL

  Paper: https://fabio.pierazzi.com/assets/pdf/Wendigo.pdf

  Github: https://github.com/isneslab/Wendigo

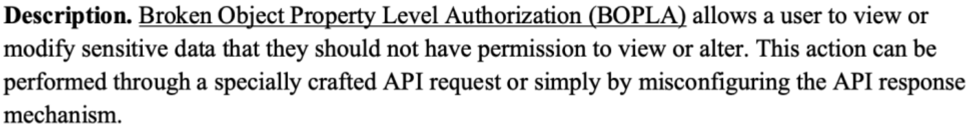
  Depending on the DDos attack vectors (Circular objects, field duplication, alias overloading,) derived from GraphQL queries, crafted and sent the complex query then waited for a full response before resending the query. This response time is measured, and the agent tries to maximize this time. This is done by iteratively modifies the query by selecting actions from the action space to increase the response time and sends this new query.

4. IDOR (access control)

- Common API vulnerability [2] and GraphQL vulnerabilities [1]
- Related blog in graphql field: https://cyberw1ng.medium.com/graphql-api-vulnerabilities-in-web-app-penetration-testing-2023-69821f5edc32 [Exploiting unsanitized arguments] section
- More

BOPLA Broken Object Property Level Authorization (an IDOR issue)

- Common API vulnerability [4] p6

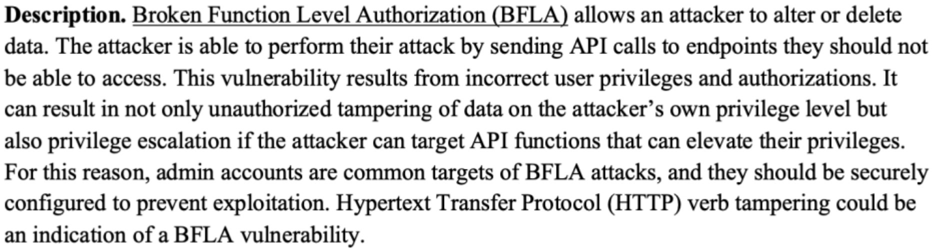


Description. Broken Object Property Level Authorization (BOPLA) allows a user to view or modify sensitive data that they should not have permission to view or alter. This action can be performed through a specially crafted API request or simply by misconfiguring the API response mechanism.

- Definition: https://owasp.org/API-Security/editions/2023/en/0xa1-broken-object-level-authorization/
- More

BFLA Broken Function Level Authorization (an IDOR issue)

- Common API vulnerability [4] p9



Description. Broken Function Level Authorization (BFLA) allows an attacker to alter or delete data. The attacker is able to perform their attack by sending API calls to endpoints they should not be able to access. This vulnerability results from incorrect user privileges and authorizations. It can result in not only unauthorized tampering of data on the attacker's own privilege level but also privilege escalation if the attacker can target API functions that can elevate their privileges. For this reason, admin accounts are common targets of BFLA attacks, and they should be securely configured to prevent exploitation. Hypertext Transfer Protocol (HTTP) verb tampering could be an indication of a BFLA vulnerability.

- Definition: https://owasp.org/API-Security/editions/2023/en/0xa5-broken-function-level-authorization/

5. Other possible related attacks

CORS; CSRF; ClickJack

- Common API vulnerability [2]
- Strong relationship with IDOR

[1] GraphQL cheat sheets talking about all the vulnerabilities:
https://cheatsheetseries.owasp.org/cheatsheets/GraphQL_Cheat_Sheet

[2] Common API vulnerabilities:
https://www.researchgate.net/publication/323817030_API_vulnerabilities_Current_status_and_dependencies

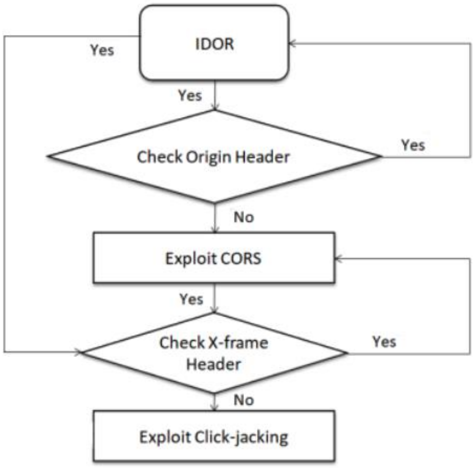


Fig. 3: How we tested to find out IDOR, CORS, X-Frame, and Click Jacking is present or not.

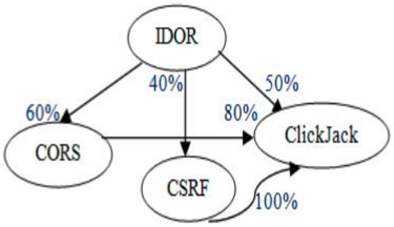
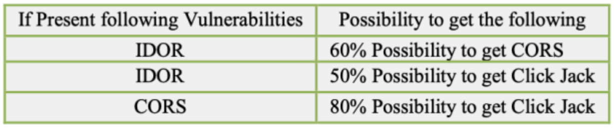


Fig. 4: Percentage to get one vulnerability if another is found.

Table I: Shows if one vulnerability is present then percentage of presenting another vulnerability.



If Present following Vulnerabilities	Possibility to get the following
IDOR	60% Possibility to get CORS
IDOR	50% Possibility to get Click Jack
CORS	80% Possibility to get Click Jack

[3] Dos was identified as an vulnerability for GraphQL in the following paper (p293):
https://link.springer.com/chapter/10.1007/978-3-319-91764-1_23

[4] API vulnerability and Risks CMU paper
https://insights.sei.cmu.edu/documents/5908/api-vulnerabilities-and-risks-2024sr004-1.pdf

GraphQL basics

6. http://olafhartig.de/files/KimEtAl_AMW2019.pdf There are five major components of GraphQL schemas that describes the supported operations: Query, Object, Mutation, Subscription, and Directive.
7. Security of GraphQL hasn't been largely talking about:
   "Due to difficulty and lack of well-defined solutions, security remains difficult and low-interest area, which can lead to vulnerabilities [48, 66]. Hence, efficient implementation of Security at various layers is a pressing research need. "
   https://arxiv.org/pdf/2408.08363 p23,30
8. https://dl.acm.org/doi/pdf/10.1145/3561818
   - Advantages of GraphQL (p28)
   - GraphQL procedure map (p4)

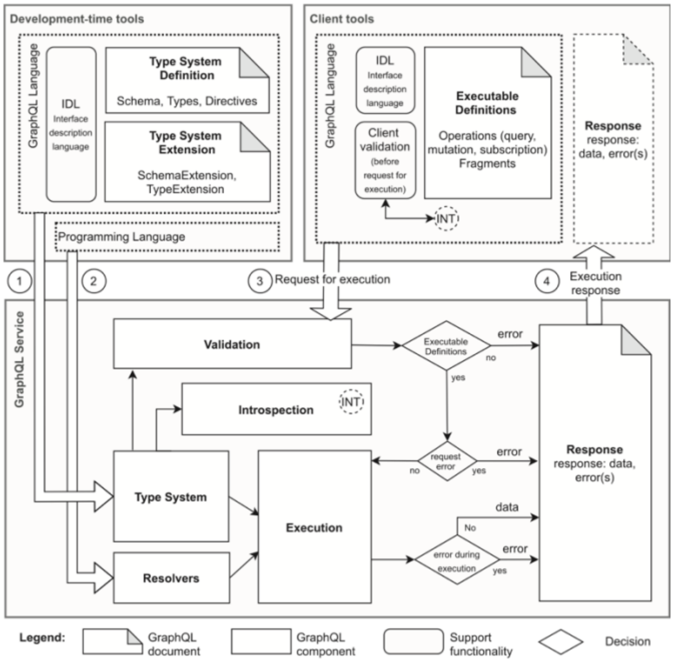


Fig. 2. Principal components interaction of the GraphQL paradigm.

9. Specification for GraphQL http://spec.graphql.org/June2018/#sec-Normal-and-Serial-Execution