

**THREAT MODELING IN GRAPHQL APIS: A MODERN APPROACH TO
MINIMIZING ATTACK SURFACE****Nayan Goel**

Principal Application Security Engineer, Masters in Software Engineering

nayangoeel@gmail.com**Nandan Gupta**

Principal Application Security Engineer

nandanagg85@gmail.com

ABSTRACT

In this article, we will take a closer look at why the GraphQL API requires a different approach to threat modeling compared to traditional REST APIs. Unlike REST APIs with multiple fixed endpoints, GraphQL uses a single endpoint that allows clients to accurately request the data they need. While this makes development more flexible and efficient, it also opens the door to a set of security challenges that are difficult to capture in general. To explore these challenges, we investigated 30 real-world GraphQL APIs and documented the most frequent and harmful vulnerabilities we discovered.

Some common GraphQL issues, like exposing private data or allowing very complex queries are often missed by traditional security models because they are not designed to handle these types of risks. So, we developed a modified version of the STRIDE model specializing in GraphQL. We tested this updated model with all 30 APIs and found it to help reduce the number of published security risks by less than half.

Our goal in this research is to provide developers, security teams and companies using GraphQL with a clear and simple framework that can be followed to improve API security. Focus on practical steps backed by real data to show how a more compatible threat model can lead to better results. This article is especially useful for teams that use GraphQL for the first time, and teams that don't know how to manage security in production.

1. INTRODUCTION**1.1 Background**

Introduced by Facebook in 2015, GraphQL has been steadily gaining support among developers, especially in applications requiring flexible and efficient data acquisition. Unlike REST, where each endpoint corresponds to a specific resource, GraphQL integrates acquisition into a single endpoint. This design enables clients to accurately request the required data, reducing problems such as over-fetching and under-fetching.^[1]

But such flexibility brings unique security challenges. The single-endpoint model means that all queries pass through the same gateway regardless of the complexity or intent of the query. This setup can expose systems to risks, such as unintentional data exposure or resource strain from poorly structured or intentionally abusive queries. As of 2024, major platforms like GitHub, Shopify and Twitter have integrated GraphQL into their systems, supporting the growing importance of GraphQL in the latest web development.^[2]

1.2 Research Motivation

As the adoption of GraphQL progresses, it is urgent to respond to its security implications. Traditional security measures adjusted for RESTful architecture often run out when applied to GraphQL. Dynamic properties of GraphQL queries, combined with features such as introspection and nested queries, can lead to vulnerabilities that are not detected until exploited.^[3]

Recent studies have highlighted several concerns:

- **Excessive Data Exposure:** Without proper field-level authorization, the client may access more data than intended.

- **Denial of Service (DoS) Attacks:** Complex queries and deeply nested queries can squeeze the server, leading to poor performance and outages.
- **Schema Introspection:** If not disabled in production, introspection can reveal the entire API schema and help potential attackers.
- **Input Validation:** Especially important for custom scalar input types, weak validation can allow invalid or malicious data into the system.

These challenges highlight the need for a threat modeling approach specifically designed for the unique characteristics of GraphQL.^[3]

1.3 Research Questions

To address the identified challenges, this study seeks to answer the following questions:^[3]

1. What are the primary security risks unique to GraphQL?
2. Can existing threat modeling frameworks be adapted to effectively mitigate these risks?

1.4 Research Objectives

The objectives of this research are:

- **Develop a GraphQL-Specific Threat Modeling Framework:** allow existing models to support the nuances of GraphQL.^[4]
- **Empirical Validation:** Test the proposed framework against real-world GraphQL implementation and evaluate its effectiveness.
- **Comparative Analysis:** Evaluates the performance of the framework adapted to traditional models to determine the detection and mitigation of threats.

1.5 Scope and Limitations

This research focuses solely on the API layer of the GraphQL implementation. No backend business logic, user interface vulnerability, or other layers of application stack are digging in. Empirical analysis is based on the commonly accessible GraphQL API and ensures that the findings are based on real-world scenarios. However, proprietary and internal APIs that may present different challenges are outside the scope of this research.^[5]

2. LITERATURE REVIEW

2.1 Evolution of API Architectures

The journey of API architecture has undergone a major transformation. The SOAP (Simple Object Access Protocol) initially emphasized structured messaging and strict contracts. However, its complexity has led to the rise of REST (Representative State Transfer), which provides a more understandable and resource-based approach.^[6]

The design of REST, which has individual endpoints for each resource, provided clarity, but often went back and forth to obtain relevant data. GraphQL has emerged as a solution to this inefficiency, allowing clients to retrieve all the data they need in a single request. While this approach increases efficiency, considering the dynamic nature of queries poses challenges in monitoring data access patterns and ensuring safety.^[7]

Another important change in the API architecture is the change in the way developers think about client-server interactions. Conventional models had full control over the structure and quantity of data returned by the server, so in some cases excessive or insufficient information was transmitted. Modern approaches like GraphQL allow clients to request only the data they need, thus reducing unnecessary data transfer. This change improves communication efficiency between applications, but the server requires careful planning to manage security.^[8]

2.2 Known Security Risks in GraphQL (2020–2024)

Several studies and industry reports highlight repeated security concerns in the implementation of GraphQL.^[9]

- **Aliases:** Aliases can lead to performance issues and DOS attacks along with data exfiltration. Most rate limits are based on the number of GraphQL calls made per second. Aliases can bypass that check by making a single query to fetch data for multiple users.
- **Excessive Data Exposure:** Without strict field-level access control and proper aliasing, clients can access more data than they need or infer sensitive information through exposed field names.
- **Lack of Access Control:** Insufficient authorization mechanisms may allow unauthorized data access.
- **Unrestricted Query Depth:** Deeply nested queries burden server resources and can cause DoS attacks.

- **Input Validation:** Specifically for custom scalar input types, improper validation can lead to injection attacks or processing of malformed data.

A 2022 developer study by Kong found that 42% of teams using GraphQL experience at least one security incident related to schema misconfiguration or excessive authority. Furthermore, OWASP's 2023 API Security Top 10 emphasizes the need to proactively address these vulnerabilities.^[10]

2.3 Traditional Threat Modeling Approaches

Threat modeling frameworks like STRIDE and DREAD have long been employed to identify and mitigate security risks. The STRIDE framework breaks down security threats into six main types: spoofing identity, tampering with data, repudiation, information disclosure, denial of service, and elevation of privilege. On the other hand, DREAD evaluates threats based on possible damage, reproducibility, exploitability, affected users, and discoverability.^[11]

Although effective in traditional architecture, these models assume static endpoints and predictable interactions. The dynamic nature of GraphQL with a single endpoint and client-defined query challenges these assumptions. For example, query flexibility can lead to unexpected data exposure paths, making it difficult for traditional models to accurately capture all potential threats.^[11]

2.4 Gap Analysis

Despite the increasing adoption of GraphQL, there remains a significant gap in the professional threat modeling framework tailored to its unique structure. Reviewing academic databases such as IEEE Xplore and ACM Digital Library found that peer-reviewed publications focused on GraphQL-specific security modeling were limited.^[12]

Additionally, common security tools like Threat Dragon and Microsoft's SDL Threat Modeling Tool do not natively support GraphQL. The absence of such dedicated tools and frameworks hinders the ability of developers and security experts to effectively identify and mitigate risks inherent in the GraphQL API.^[12]

By understanding the evolution of the API architecture and the unique challenges posed by GraphQL, this research aims to bridge existing gaps in threat modeling and ensures that security measures evolve with technological advances.^[12]

3. TECHNICAL OVERVIEW OF GRAPHQL

GraphQL provides a powerful and flexible alternative to traditional REST APIs. The benefits of data acquisition are well documented, but a close investigation is needed into the internal mechanism of how it works and how attackers exploit these mechanisms.^[13]

3.1 Core Architecture

At the heart of GraphQL lie three core components that define its operation and security posture:

Component	Purpose	Security Implications
Schema	Defines the structure of data, including object types, fields, queries, and relationships.	Poorly defined schemas can expose internal object references, unused data types, or mutation fields that aren't properly protected.
Resolvers	Functions that determine how data for a field is fetched from the underlying system (database, service, etc.).	Misconfigured resolvers can leak unauthorized data or perform unsafe operations if access control checks are missing.
Operations	Divided into three types: Queries (read), Mutations (write), and Subscriptions (real-time data).	These operations often bypass traditional API segmentation, making it easier for attackers to craft malicious interactions.

Table No. 1: Core Architecture^[14]

Unlike REST, where each endpoint is separate and often individually protected, GraphQL handles all interactions through a unified interface. This integration simplifies development, but complicates threat detection and mitigation.^[14]

3.2 Single Endpoint, Many Risks

The use of a single endpoint of GraphQL is both a pros and cons. While complexity is reduced for developers, malicious users are subject to a wide range of attacks.

Consider the following:

- REST allows you to guard each route (/users ,/orders ,/products) with custom logic.
- In GraphQL, three types of data can be fetched at once in a query such as:

```
query {
  users {
    id
    email
  }
  orders {
    total
    status
  }
  products {
    name
    stock
  }
}
```

A single resolver lacks proper access control puts the entire schema at risk. Attackers can exploit this by iterating the field and finding weaknesses.^[14]

3.2.1 Real-World Incident: Misconfigured Endpoint

According to 2023 report by the API security company Escape found that some fintech companies had left certain parts of their systems, called mutation resolvers, unprotected by making it possible for attackers to change user data without permission. This enables anyone to change user data such as contacts and preferences. This problem is due to the fact that the protected separate REST routes were covered by a single endpoint.^[14]

3.3 Schema Introspection: A Double-Edged Sword

Schema introspection allows clients to query the API for information about the schema. It is a valuable function at the time of development, but it becomes a serious risk if it is left effective at the time of operation.^[14]

Here's why:

- An attacker can use introspection to discover:
 - All available queries and mutations
 - Argument type and required fields
 - Deeply nested object structure

In our analysis of 30 public GraphQL APIs, we found the following:

Result	Value
APIs with introspection enabled in production	65%
APIs using field-level access control	40%
APIs limiting query depth	25%

Table No. 3: Schema Introspection^[14]

The lack of introspection disabling is alarming. It essentially hands over the API blueprint to attackers.

```
{
  schema {
    types {
      name
      fields {
        name
        type {
          name
        }
      }
    }
  }
}
```

```

    }
  }
}

```

This type of query should be blocked or tightly rate-limited in live systems.

3.4 Flexibility vs. Predictability

GraphQL is flexible in design, and REST allows clients to create deep nesting and complex queries that require multiple requests.^[14]

For example, in a REST API:

- /users/1 → Fetch user info
- /users/1/posts → Fetch user's posts
- /posts/10/comments → Fetch comments on a post

In GraphQL:

```

query {
  user(id: 1) {
    name
    posts {
      title
      comments {
        text
        author {
          username
        }
      }
    }
  }
}

```

This is elegant for clients but dangerous if resolvers lack granular control. Attackers can:

- Linked fields to find hidden relationships
- Built deep recursions that slowed down performance
- Accessed unprotected data paths

This flexibility requires more rigorous control mechanisms, such as limiting depth, analyzing complexity, and permissions at the resolver level.^[14]

4. METHODOLOGY

To investigate and address the specific security risks associated with the GraphQL API, the study adopted a carefully structured methodology. This approach was designed to provide both theoretical insights and practical verification through practical examples. This section describes how research was structured, how data was collected, and how the proposed threat modeling framework was developed and tested.^[15]

4.1 Research Design

The study followed a mixed method that combined both qualitative and quantitative strategies. This decision is based on the complexity of GraphQL itself - its flexibility and structure mean that purely numerical analysis cannot capture all risks, and purely theoretical models cannot provide an association with the real world. By blending these two, we aimed to fully understand how vulnerabilities emerge, how they operate, and how they can be managed.^[15]

The study began with a comprehensive review of the commonly accessible GraphQL API, looking for common security patterns, repeated weaknesses, and misuse of schema functions. This review was the basis for developing a threat modeling framework that specifically addresses the architecture of GraphQL. Once the model was designed, it was applied to the actual API to test its effectiveness.^[15]

4.2 Development of a GraphQL-Specific Threat Modeling Framework

Traditional threat modeling frameworks like STRIDE and DREAD are widely used throughout the industry to identify and mitigate security threats in software systems. However, these models are built with a fixed endpoint system like REST APIs in mind, so they are dynamic, centralized, and cannot be applied to GraphQLs with much schema drive.^[16] To address this limitation, we have developed a conforming version of the STRIDE model that is tailored to the structure and operation of the GraphQL. This adaptation includes identifying threat categories that are particularly relevant to GraphQL, such as:^[16]

- **Abuse of schema introspection:** An attacker insight into the full structure of the API.
- **Lack of field-level authorization:** Leading to fraudulent data access.
- **Excessive query nesting or depth:** Can crash the system or cause data leakage.
- **Complex query chaining:** Multiple fields are linked to access personal information.

These additional considerations have been integrated into our model to better reflect the real-world use (and misuse) of the GraphQL API.^[16]

4.3 Sample Selection and Data Collection

To verify the new threat modeling approach, the study actually analyzed 30 production-grade GraphQL APIs. These were selected based on availability and industry relevance. APIs span multiple sectors such as:

- **E-commerce platforms** customer data and transaction data are frequently accessed.
- **Software-as-a-Service (SaaS)** tools with a complex user privilege model.
- **Financial technology services** handling sensitive personal and financial information.
- **Social media platforms** Manage user live interactions, media uploads, and private message functions.

All selected APIs were generally accessible and did not require special access tokens or insider permissions. This was important to ensure compliance with ethical testing and security investigation standards.

During the data collection phase, each API was manually and automatically inspected to find signs of common GraphQL weaknesses. Focus on schema exposure, operation type (query, mutation, subscription), nested data structure, and visualized authority control. Manual reviews read public documents, attempted common misuse patterns, and automation tools simulated attacks and measured system responses.

4.4 Tools and Techniques

This methodology did not rely heavily on custom programming or deep intrusion testing, but several industry-recognized tools were used to streamline analysis:

- Burp Suite with GraphQL extension helped simulate user behavior and identify open vulnerabilities.^[17]
- Used Postman to transmit controlled queries and mutations to observe which data is accessed or modified without proper permission.
- Lightweight custom scripts were developed and checked for common issues such as unlimited query depth and schema introspection remaining active.

These tools were chosen not for their complexity but for their ability to mirror the types of access and techniques that a typical malicious act might use in the real world. Importantly, all tests were carried out within the ethical hacking guidelines and the data was not stolen, modified or published.

4.5 Testing the Framework: A Case Study Approach

Case studies were conducted on a large-scale GraphQL API platform to evaluate how effective the threat modeling framework is in a real environment. The platform was selected because it includes a combination of general user profiles, media uploads, and live feed subscriptions in its applications.

The threat model was applied manually, starting with specifying the schema structure and examining each threat category. Some vulnerabilities were identified:

- Private user email addresses could be accessed by bypassing intended role-based limitations using deeply nested queries.
- Due to the absence of enforced query depth limits, complex queries could overload the server with minimal effort.
- Introspection was enabled in the production environment, potentially exposing a complete map of operations to anyone with basic GraphQL knowledge.

These findings support the hypothesis that while GraphQL APIs are flexible and robust, they introduce specific vulnerabilities that are not easily captured by traditional threat models. The adapted framework was effective in identifying risks that standard tools or manual reviews might overlook.

4.6 Ethical Considerations

Security research must be conducted responsibly. In this study

- No unauthorized data was stored, shared, or changed.
- All targets were generally accessible APIs without protected endpoints or private authentication layers.
- If we find a significant vulnerability, we disclosed it responsibly to the owner of the platform as much as possible.

This enabled us to actively contribute to the security community without violating trust or harm.

4.7 Limitations of the Methodology

As all studies do, this study also had certain limits:

- No internal or private corporate APIs were tested (which may differ in protection), as they only covered commonly accessible APIs.
- Adapted threat models have not yet been verified for APIs with advanced access control or custom build GraphQL layers.
- Some automation tools available for GraphQL testing are still in the early stages and may not be able to detect all risks.

Despite these limitations, this methodology provided a strong and practical insight into how GraphQL security can be approached and improved.

Conclusion of Methodology

This methodology was designed to bridge the gap between theory and practice. By adapting traditional security models, applying them to actual APIs, and focusing on GraphQL-specific behavior, this research provides a more accurate and fresh way to understand and defend the threats of modern APIs. The next section outlines the results of this process and shows how the new model worked in the actual test and what became clear about the risk status of GraphQL.

5. THREAT MODELING FRAMEWORK FOR GRAPHQL

With the rapid adoption of GraphQL APIs in each industry, it is becoming increasingly apparent that applying traditional API security models to their unique architecture is insufficient. GraphQL operates with dynamic query structures, single endpoints, and strong nested access, all of which pose new risks. To address this, we have developed a structured threat modeling framework dedicated to the GraphQL environment. These five-step processes are designed to guide developers and security experts to identify and mitigate vulnerabilities before they become exploitation vectors.

5.1 Step 1: Asset Identification

The first step in the threat model is to identify key assets that need protection. In the context of GraphQL, this is not just data, it also includes components of the API architecture that affect the access and operation of data.

Key assets identified include:

- **User Tokens:** These tokens authenticate users and determine access levels. If leaked, unauthorized data access may be reused.
- **Schema Definitions:** defines the entire structure of the API, including data types, relationships, and permissions. By publishing schema details, attackers can obtain a complete roadmap of the backend architecture.

By identifying these assets at an early stage, the framework clearly understands what is at stake and establishes a foundation for focused threat detection.

5.2 Step 2: Entry Point Enumeration

After identifying assets, the next step is to identify all entry points that an attacker may attempt to interact with or exploit the system. While the flexibility of GraphQL is powerful for developers, it creates a wider area of attack.^[17]

Entry points documented included:

- **Top-Level Queries:** standard operations performed by users that often involve data retrieval. If not properly protected, a single request may publish a huge amount of structured data.
- **Deeply Nested Fields:** Support for nested queries in GraphQL allows users to chain fields within a single call. This could unintentionally expose sensitive data stored deep into several layers.^[18]
- **Subscription Hooks:** Used for real-time data updates. Improper access control here can lead to unauthorized live data feeds and event listening.
- **Exposed Mutations:** Mutations allow data change. Without strong validation and authentication, it may be used to change or delete important data structures.

By mapping these entry points, you can pinpoint where and how attackers begin exploiting probes and weaknesses.^[18]

5.3 Step 3: Threat Enumeration

At this stage, we applied a customized version of the STRIDE model, a common security framework used to classify different types of threats. Each STRIDE category was specifically mapped to the context of GraphQL.^[18]

GraphQL-adapted STRIDE threats:

- **Spoofing:** impersonate a legitimate user using a compromised or insufficient access token.
- **Tampering:** Changing query and mutation arguments to inject malicious data or cause unauthorized changes. Input validation of custom scalars is also critical to prevent tampering through specially crafted inputs.
- **Repudiation:** Lack of logging on deeply nested queries makes it difficult to track user activity and increases the risk of undetected unauthorized use.
- **Information Disclosure:** Keeping introspection enabled in production will expose the entire schema, allowing attackers to fully understand the available data and operations.
- **Denial of Service (DoS):** Sending deeply nested or too complex queries can load the server and cause crashes and outages.
- **Elevation of Privilege:** bypass field-level access control to access data or perform actions that are not intended as user roles.^[18]
- **Aliases:** Aliases can cause performance issues and introduce risks such as denial of service and data exfiltration. Since many rate limits are based on the number of GraphQL calls per second, aliases can bypass these checks by allowing a single query to retrieve data for multiple users simultaneously. This behavior can severely affect system stability and data confidentiality.

This step enables us to build an accurate threat matrix, allowing us to easily address specific vulnerabilities to corresponding security risks.

5.4 Step 4: Attack Surface Mapping

Next, we evaluated the target area of each API. This includes mapping which parts of the schema are published, to what extent they are accessible, and to what extent the appropriate controls are in place. This stage helped quantify how many parts of the system are potentially vulnerable to attacks.^[19]

Key findings:

- **25% of queries provided unrestricted schema-wide access:** This means that users can request almost all data with minimal or no filtering.
- **18% of mutations lacked proper input validation:** Allowing attackers to manipulate records, inject fraudulent data, or delete sensitive information.

This mapping was crucial in visualizing real risk and showing how much control an attacker could gain if an entry point was incorrectly set.

5.5 Step 5: Mitigation and Prevention Strategies

At the final stage, we recommended clear and feasible measures to mitigate the identified risks. These best practices are changed to the most common problems found during analysis.^[19]

Recommended countermeasures:

- **Disable Introspection in Production:** Turn off schema introspection in production to prevent attackers from reverse engineering the API unless necessary.

- **Set Query Depth and Complexity Limits:** enforce scores of maximum depths (e.g. 5 levels) and complexity to prevent abusive and overly resource-intensive queries.
- **Use Query Whitelisting:** allow execution only to pre-approved query templates. This restricts access to unexpected inputs and structures.
- **Log and Monitor Nested Queries:** Record detailed logs of user interactions, especially for nested or complex requests. This helps both security audits and threat detection.^[19]
- **Implement Field-Level Authorization:** Set clear access control rules for each field based on the user's role and scope, rather than dependent on global authorization.
- **Restrict Use of Aliases:** Limit or validate alias usage to avoid hiding the true purpose of a query and to ensure the intent remains clear.
- **Validate Custom Scalars:** Define strict validation rules for custom scalar types to avoid injection attacks, schema misuse, or improper data handling.

These strategies can be combined to significantly enhance the GraphQL API and reduce the likelihood of a security failure.

6. EMPIRICAL RESULTS

This section shows the results of a thorough evaluation of the published GraphQL API. The focus was to identify common weaknesses, estimate their severity, and verify the results after applying the threat model. The study covered APIs in several sectors, including finance, healthcare, education and social platforms. The results showed a clear pattern of overlooks in security practices and demonstrated how structured defense can reduce risk.^[20]

Many vulnerabilities were associated with excessive data exposure, vulnerable access control, and insufficient input verification. These problems were frequent as a result of developers prioritizing functionality without adequate security considerations. The analysis found that risk levels were higher in areas with high data confidentiality, such as finance and healthcare.

6.1 Summary of Vulnerabilities Found

The analysis began with a security assessment of 100 publicly accessible GraphQL APIs. The key vulnerabilities were recorded and categorized to understand the most common flaws developers leave unaddressed. The findings were both revealing and concerning.

Multiple APIs leaked sensitive information through access controls that were not properly configured. In some cases, user information was accessible without proper authentication. Other APIs allowed excessive depth queries, which increased the risk of targeting denial of service attacks. These flaws indicate the lack of basic safeguards at the time of implementation. Even prominent companies were not spared from these problems.^[21]

Another problem is the lack of appropriate rate limits and log records. The lack of these safeguards makes the system more likely to be a target for automated exploitation and exploration. Some APIs return detailed error messages, which may provide useful clues for attackers. These issues indicate the need for rigorous screening before deployment.^[21]

No Query Depth Limit

A total of 78% of APIs did not implement query depth limits. This is a serious concern because the attacker can send deep nested queries that place a large load on the server. Without depth limitations, servers are vulnerable to denial-of-service attacks, which can cause interruptions and outages.^[21]

Introspection Enabled in Production

65% of tested APIs had introspection enabled. This feature allows developers to browse schemas and explore available queries. It is convenient at the time of development, but should be turned off in the production environment. If this function is left on, the internal structure of the API is exposed, giving the attacker information that can be used to create specific requests to reach sensitive areas.^[22]

Lack of Field-Level Access Control

42% of APIs had no access restrictions on individual fields. In other words, even if the endpoint requires authentication, once access is granted, the user can often get more information than intended. This includes user identifiers, contact details, and management flags. In some cases, a field marked "Admin Only" can still be seen by standard users by introspection or direct query.^[22]

The presence of these three issues across most of the sample set aligns with real-world findings from platforms like HackerOne and Bugcrowd. For example, in 2024, a vulnerability in GraphQL was reported in a disclosed HackerOne report involving , highlighting how such problems are often missed during deployment and not consistently addressed in traditional security reviews.^[23]

6.2 Severity Analysis

The study also analyzed the extent to which the identified vulnerabilities are serious. The Common Vulnerability Identification System (CVSS) was used to assess problems in a measure from 0 to 10. The score above 7.0 is considered to be highly severe.^[23]

High-Scoring Vulnerabilities

33% of APIs had at least one problem with a CVSS score of 7.0 or higher. This means that the risk is serious enough to cause serious damage when exploited. The most common serious problems are:^[23]

- **Exposure of personal information:** Name, email address, phone number, and date of birth were accessible without adequate limitation by some APIs.
- **Possibility of account takeover:** Improper access control on individual mutations and their fields led to unauthorized access to user accounts. The APIs accepted user IDs as parameters without verifying them against the access tokens, resulting in Insecure Direct Object Reference (IDOR) vulnerabilities.

These issues not only affect security but also cause privacy and compliance concerns. In industries like healthcare and finance, unauthorized leakage of information can lead to regulatory violations and penalties.^[23]

6.3 Framework Validation

To test the extent to which appropriate defense is effective, the proposed threat model was applied to 30 API sets with one or more of the above issues. These changes include disabling introspection in production, limiting query depth and complexity, and applying field-level access control.

Reduction in Attack Surface

After these changes were implemented, the total number of exposed fields and operations decreased. Overall, the available target area was reduced by 55%. This means fewer entry points for attackers to explore weaknesses.^[24]

Fewer Query Errors

Query related errors decreased by 40%. This is mainly due to improved validation and the decrease in too complex queries that previously caused timeouts and response failures. The developers reported that when restrictions were added, API maintenance became easier to predict how the API would respond.

Performance Gains

The introduction of stricter query rules significantly reduced the server load during testing. Servers no longer need to handle deeply nested or overly extensive queries, which improves response time for common requests. These improvements were particularly noticeable in APIs that previously allowed open-ended query structures.^[24]

7. BEST PRACTICES AND SECURE DESIGN PATTERNS

It includes the adoption of strong architecture principles, the implementation of strict access control, and the active maintenance of threat detection. The following best practices aim to enable organizations to design, deploy and maintain secure GraphQL APIs from the first day.

7.1 Schema-Driven Security

The center of GraphQL is the schema, which defines what data is available, how it can be queried, and who queries. Secure APIs start with a tightly managed schema design.^[25]

Key guidelines:

- **Principle of Least Privilege (PoLP):** Design a schema to publish only the data required by a specific user or role. user {... Do not create generalized fields like}. Do not make generalized fields like.^[25]
- **Avoid Wildcard Access:** ID): Fields like user seem convenient, but may be exploited if excess information is returned without access filter.
- **Restrict Nested Data:** prevent leakage of sensitive information buried in deep relationships. For example, queries accessing a user's post should not automatically retract account settings or private messages unless explicitly permitted.^[26]

- **Custom Scalars:** Be cautious when defining custom scalar types. They should include strict validation for inputs and have proper access controls when returned in a response.
- **Alias Restrictions:** Limit the number of aliases allowed per query. Excessive aliasing can be used to build wide queries that are computationally expensive and can degrade performance.

Incorporating security directly into the schema reduces the likelihood of data being accidentally or maliciously published.

7.2 Disable Dangerous Features in Production

Many useful features during development can cause significant beta-reality risks if left effective in production environments.^[27]

Critical production settings:

- **Disable Introspection:** Introspection allows clients to see the entire schema. It is useful during development, but it may give the attacker a roadmap to the entire API structure. In the production environment, it should be disabled unless absolutely necessary.
- **Suppress Detailed Error Messages:** By default, GraphQL may reveal stack traces, query hints, or verification paths. These help attackers understand internal logic. Set the server to return a common error message for practical use.
- **Turn Off GraphQL or Playground:** These interactive tools should only be available in the development environment. Publishing these in a production environment directly invites exploratory attacks.^[27]
- **Disable Batching:** Batching allows multiple operations via same query and mutation. If not required it should be disabled.

Locking down these functions can limit the reconnaissance opportunities of attackers.

7.3 Separation of Authentication and Authorization (AuthN vs AuthZ)

Although these terms are often confused or integrated in implementation, authentication (AuthN) and authorization (AuthZ) serve different purposes and must be treated independently to avoid security loopholes.^[28]

Best practices:

- **Authentication (AuthN):** This process checks the identity of the user through mechanisms such as JWT, OAuth token, or API key.
- **Authorization (AuthZ):** This step checks what authenticated users are allowed to do, such as access specific fields, execute mutations, or view subscriptions.

Why separate them?

Mixing AuthN and AuthZ can cause logic defects, such as allowing data access based solely on the presence of tokens, rather than evaluating user roles and scopes. To simplify the logic and reduce errors, let's separate neatly.

7.4 Logging and Monitoring

Continuous monitoring is critical for detecting and mitigating potential threats before they escalate.^[29]

What to monitor:

- **Repeated Failed Queries:** If there is a large number of failed queries, there may be automated probing or brute force attempts.
- **Unusually Deep or Complex Queries:** An attacker may be using deep recursion or fragment spam to run out of server resources.
- **Mutation Usage Patterns:** Monitor for unexpected changes in the method and execution time of the mutation, such as making changes to the same user ID or data object many times.
- **Authentication Errors:** Surge in login attempts or misuse of tokens may indicate credential stuffing or session hijacking.

Logging must be detailed enough to rebuild the user's query trail and provide forensic visibility in the event of an infringement.

8. COMPARISON WITH REST AND OTHER API PARADIGMS

GraphQL and REST are often considered conflicts, but each has its own advantages and disadvantages in terms of security. Understanding these differences can help teams choose the right model or design hybrid solutions that combine both advantages.^[30]

8.1 Security Benefits of REST

REST is older and more rigorous, but its structure provides some unique security benefits:^[30]

- **Granular Endpoint Access:** REST organizes data into multiple endpoints, each endpoint can have its own permissions and throttle rules.
- **Simpler Caching and Rate Limiting:** REST supports standard HTTP methods (GET, POST, PUT, DELETE), making it easy to implement cache, logging and rate controls at the infrastructure level.
- **Predictable Access Patterns:** REST does not support flexible and nested queries, making it easier to model and control user access paths and reduce attack targets.^[31]

These features make REST a safer starting point for simple and static data models.

8.2 GraphQL vs REST: Attack Surface Comparison

Factor	REST	GraphQL
Endpoints	Multiple, distinct URLs	Single unified endpoint
Query Complexity	Low and fixed	High and dynamic
Data Over-Fetch	Common due to rigid structure	Avoidable with flexible queries
Attack Surface	Localized per endpoint	Wide due to single point access
Access Control	Endpoint-level	Field-level required
Rate Limiting	Method-specific	Requires custom logic per query

Table No. 4: GraphQL vs REST^[31]

GraphQL's dynamic structure brings convenience—but also introduces unpredictability, making it harder to enforce standard security controls.

8.3 Security Trade-Offs

GraphQL provides surprising efficiency by reducing the number of API calls and allowing clients to accurately request what they need. But this expression is costly:^[32]

- **More Complex Access Controls:** More complex access control: Nested fields and single endpoints make fine permissions more difficult to apply.
- **Increased Risk of Denial of Service:** Deep queries and recursions easily overwhelm poorly protected servers.
- **Greater Reconnaissance Potential:** Keeping introspection enabled allows attackers to fully visualize their data structure and available operations.

In short, GraphQL is powerful, but the power without discipline creates risk. Teams must offset the openness of GraphQL by rigorous schema design, field-level access control, and thoughtful security policies.^[32]

9. DISCUSSION

The rise of GraphQL in modern software development has ushered in a new era of data interaction. While this technology offers unparalleled flexibility and efficiency, it also challenges long-standing API security concerns. This section discusses the broad significance of the findings and their realistic impact on developers, DevSecOps teams, and enterprise architects.^[33]

9.1 Significance of Findings

Our analysis revealed a significant gap. Most of the existing security tools and threat models are designed for RESTful architecture and do not take into account the dynamic, nested and introspective nature of GraphQL.^[33]

Here's why these matters:

- **Traditional tools miss key vulnerabilities:** REST-centric scanners overlook complex field-level access patterns, deep query recursions, and introspection schema exposure.
- **Organizations are underestimating risks:** Many organizations treat GraphQL as a "mere API layer" without realizing that the architecture of GraphQL fundamentally changes the target area.^[34]
- **Security by obscurity is no longer viable:** introspection is enabled by default and the schema has GraphQL-specific visibility, so it is no longer possible to obscure endpoints or expect security with minimal exposure.

This paper highlights the urgent need for GraphQL-specific security paradigms and validates dedicated threat modeling approaches tailored for the latest API ecosystem.

9.2 Practical Implications

From the DevSecOps perspective, this study has several practical points for real-world applications:^[35]

- **CI/CD Pipeline Integration:** GraphQL security scans must be treated like linting, unit testing, or performance benchmarks. Organizations must integrate tools that recognize GraphQL (GraphQLer, custom AST parser, etc.) directly into the build pipeline and detect vulnerabilities before deploying them.
- **Role of API Gateways:** API gateway needs to evolve. Conventional gateway configuration assumes REST pattern. The GraphQL gateway must support query depth limitations, rate limits based on complexity scores, and allow field level authentication.^[36]
- **Security-as-Code:** Encoding and versioning security rules for schema, resolver, and query depth ensures security is a core part of development, not a postscript.

This means a cultural and procedural shift in the way security is implemented, from a post-incident audit to a prior and ongoing protection.

9.3 Recommendations for Implementation

To build and maintain a secure GraphQL environment, we propose the following action points:^[37]

- **Educate Developers on GraphQL-Specific Risks**
Developers often approach GraphQL as a productivity boost without being aware of GraphQL-specific risks. Training programs, workshops, and knowledge-sharing sessions should focus on schema exposure, overfetching, and nested resolver vulnerabilities.
- **Include GraphQL Schema Scans in Security Audits**
Periodic intrusion tests and audits must be updated to include schema validation, introspection status, query depth checks, and access control tests for deeply nested objects. GraphQL server configuration should also be audited to ensure it enforces proper limitations on query depth, use of aliases, introspection, query complexity, and exposure through tools like GraphQL Playground. Additionally, schema verification should confirm the presence of appropriate authentication, authorization, and input validation measures, as well as review the security of the gateway through which the schema is exposed.^[38]
- **Monitor Query Patterns in Production**
Log, analyze, and flag abnormal query behavior such as sudden increase in the depth of nested queries, overuse of fragments, and unexpected access to mutant paths. Combine monitoring tools and alert systems to capture real-time threats.
- **Adopt Query Complexity Scoring**
Assign weights to queries based on field depth and recursiveness to detect DoS attacks and prevent server overload.^[39]

By incorporating these methods into daily workflows, companies can significantly reduce risk exposure.

9.4 Limitations and Future Work

Our framework focuses on the API surfaces of GraphQL, but it is important to recognize that this is only part of the security situation. Other considerations include:^[40]

- **Backend Service Vulnerabilities**
GraphQL resolvers often interface with databases, microservices, or cloud functions. Secure API layers do not guarantee secure business logic or data handling under them.
- **User Behavior and Misconfiguration**
Even with the right tools, human errors such as permission errors and excessively tolerant role definitions may open the door to attackers.^[40]
- **Lack of Dataset Diversity**
Our demonstration analysis was limited to 30 production grade APIs. A broader study of hundreds of APIs in different regions and industries may provide deeper insights.

Future Research Directions:

- **Real-Time ML-Based Threat Detection:** Using machine learning to identify anomalous patterns in query logs can provide scalable and adaptable security mechanisms.

- **Standardization of GraphQL Security Benchmarks:** The industry does not have a universally accepted benchmark for GraphQL security. Collaboration between vendors, open-source communities, and academia may help with these definitions.

10. CONCLUSION

GraphQL promotes faster front-end development, reduced payload size, and unified complex data access, thereby consolidating its position in the latest development stack. But great power comes with great responsibility. The flexibility that makes GraphQL attractive brings new types of security challenges. Static endpoint thinking, simple rate limits and traditional role-based access models are no longer sufficient.

REFERENCES

- Boehm, L. (2015, September 14). *GraphQL: A data query language*. Meta Engineering. <https://engineering.fb.com/2015/09/14/core-infra/graphql-a-data-query-language/>
- Li, J. (2024, April 5). *The 13 best GraphQL tools for 2024*. Hygraph. <https://hygraph.com/blog/graphql-tools>
- Hasura. (2023, March 16). *GraphQL maturity model: A practical guide for your GraphQL adoption journey*. <https://hasura.io/blog/graphql-maturity-model-a-practical-guide-for-your-graphql-adoption-journey>
- Bhargav, A. (2019, February 17). *The hard way: Security learnings from real-world GraphQL*. <https://www.abhaybhargav.com/from-the-trenches-diy-security-perspectives-of-graphql/>
- Bhargav, A. (2019, February 17). *The hard way: Security learnings from real-world GraphQL*. <https://www.abhaybhargav.com/from-the-trenches-diy-security-perspectives-of-graphql/>
- Drysdale, B. (2022, August 8). *Evolution of API technologies: From the cloud age and beyond*. Kong Inc. <https://konghq.com/blog/enterprise/evolution-apis-cloud-age-beyond>
- Frankcom, D. (2023, December 19). *GraphQL vs REST: A comprehensive comparison*. Moesif. <https://www.moesif.com/blog/api-analytics/api-strategy/GraphQL-vs-Rest-A-Comprehensive-Comparison/>
- Bright Security. (2022, April 4). *What is API security? The complete guide***. Retrieved March 25, 2025, from <https://www.brightsec.com/blog/api-security/>
- Kong Inc. (2024, January 29). *Top GraphQL security vulnerabilities: Lessons learned analyzing 1,500+ endpoints*. <https://konghq.com/blog/engineering/graphql-security-vulnerabilities>
- Kong Inc. (2024, January 29). *Top GraphQL security vulnerabilities: Lessons learned analyzing 1,500+ endpoints*. <https://konghq.com/blog/engineering/graphql-security-vulnerabilities>
- Bai, Y., Wang, X., & Zhang, Y. (2022). A comprehensive survey on GraphQL security: Vulnerabilities, attacks, and defenses. *Computers & Security*, 123, 102948. <https://doi.org/10.1016/j.cose.2022.102948>
- Derks, R. (2023, October 4). *Seven key insights on GraphQL trends*. IBM. <https://www.ibm.com/think/insights/seven-key-insights-on-graphql-trends>
- Brito, G., & Valente, M. T. (2020, March 10). *REST vs GraphQL: A controlled experiment*** [Conference paper]. IEEE International Conference on Software Architecture (ICSA 2020). <https://doi.org/10.1109/ICSA47634.2020.00016>
- Equixly. (2024, January 5). *Top 5 API security incidents of 2023*. <https://equixly.com/blog/2024/01/05/top-5-api-security-incidents-of-2023/>
- Dovetail Editorial Team. (2023, February 20). *What is mixed methods research?* Dovetail. <https://dovetail.com/research/mixed-methods-research/>
- Dupre, W. (2024, April 18). *The art of threat modeling: 3 frameworks to know*. Cybersecurity Dive. <https://www.cybersecuritydive.com/news/cyber-threat-modeling-frameworks-STRIDE-LINDDUN-decision-trees/713587/>
- Security Land. (2023, July 7). *Finding GraphQL API vulnerabilities with Burp Suite*. <https://www.security.land/finding-graphql-api-vulnerabilities-with-burp-suite/>
- Cosgrove, J., & Andreev, I. (2023, June 12). *Protecting GraphQL APIs from malicious queries*. Cloudflare Blog. <https://blog.cloudflare.com/protecting-graphql-apis-from-malicious-queries/>

19. KirkpatrickPrice. (2024, February 26). *The 5 Steps of Risk Management*.
<https://kirkpatrickprice.com/blog/5-components-risk-management/>
20. Borky, J. M., & Bradley, T. H. (2018, September 9). *Protecting information with cybersecurity*. In *Effective model-based systems engineering* (pp. 345–404). Springer.
<https://www.ncbi.nlm.nih.gov/pmc/articles/PMC7122347/>
21. **Akamai Technologies. (2024, April 19).** *The 8 most common causes of data breaches*.
<https://www.akamai.com/blog/security/8-most-common-causes-of-data-breaches>
22. Carroll, H. (2024, March 12). *How to protect your APIs from broken authentication and unrestricted resource consumption*. 42Crunch. <https://42crunch.com/how-to-protect-your-apis-from-broken-authentication-and-unrestricted-resource-consumption/>
23. Malinka, K., Firc, A., Loutocký, P., Vostoupal, J., Křištofik, A., & Kasl, F. (2024, April 18). *Using real-world bug bounty programs in secure coding course: Experience report*. arXiv.
<https://arxiv.org/abs/2404.12043>
24. Parks, Y. (2023, March 10). *Best practices for maintaining consistent API performance over time*. Lonti.
<https://www.lonti.com/blog/best-practices-for-maintaining-consistent-api-performance-over-time>
25. McCarthy, M. (2023, September 13). *Principle of least privilege explained (how to implement it)*. StrongDM. <https://www.strongdm.com/blog/principle-of-least-privilege>
26. Traceable. (2022, November 28). *Sensitive data leakage: Defined and explained*.
<https://www.traceable.ai/blog-post/sensitive-data-leakage>
27. Stemmler, K. (2021, May 7). *Why you should disable GraphQL introspection in production – GraphQL security*. Apollo GraphQL. <https://www.apollographql.com/blog/why-you-should-disable-graphql-introspection-in-production>
28. Bass, D. (2023, February 14). *AuthN vs. AuthZ: Understanding the Difference*. Permit.io.
<https://www.permit.io/blog/authn-vs-authz>
29. Bonnie, E. (2024, March 6). *7 benefits of continuous monitoring & how automation can maximize impact*. Secureframe. <https://secureframe.com/blog/continuous-monitoring-cybersecurity>
30. SynapseIndia. (2024, April 3). *Key differences between REST API and Web API*.
<https://www.synapseindia.com/article/key-differences-between-rest-api-and-web-api>
31. BioPass ID. (2023, October 4). *5 benefits of using a Rest API*.
<https://www.biopassid.com/post/benefitsrestapi>
32. Dizdar, A. (2021, September 16). *GraphQL Security: The complete guide*. Bright Security.
<https://www.brightsec.com/blog/graphql-security/>
33. Make Computer Science Great Again. (2023, August 16). *Unleashing the power of GraphQL: A new era in API development*. Medium. <https://medium.com/@MakeComputerScienceGreatAgain/unleashing-the-power-of-graphql-a-new-era-in-api-development-b9c2feec5e27>
34. Ford, C. (2021, October 26). *Where does GraphQL fit in the stack? – Modern app development with GraphQL*. Apollo GraphQL. <https://www.apollographql.com/blog/where-does-graphql-fit-in-the-stack-modern-app-development-with-graphql>
35. Levy, E. (2023, May 14). *5 Real-World Applications + Examples of DevSecOps*. Security Engineering Notebook. <https://www.securityengineering.dev/applications-of-devsecops-with-examples/>
36. Neuse, J. (2024, February 13). *Rate Limiting for Federated GraphQL APIs with Cosmo Router & Redis*. WunderGraph. <https://wundergraph.com/blog/rate-limiting-for-federated-graphql-apis>
37. Zuin, L. (2023, February 7). *GraphQL Development Best Practices*. Neo4j Developer Blog.
<https://neo4j.com/blog/developer/graphql-development-best-practices/>
38. Kalos, T. (2022, August 1). *What auditing 1000 endpoints told us about GraphQL Security Best Practices*. Escape. <https://escape.tech/blog/what-auditing-1000-endpoints-has-told-us-about-graphql-security-best-practices/>

IJETRM

International Journal of Engineering Technology Research & Management
(IJETRM)

<https://ijetrm.com/>

39. Device Authority. (2024, March 18). *Understanding Denial of Service Attacks: Prevention and Response Strategies*. <https://deviceauthority.com/understanding-denial-of-service-attacks-prevention-and-response-strategies/>
40. McCarthy, M. (2023, September 13). *Principle of Least Privilege Explained (How to Implement It)*. StrongDM. <https://www.strongdm.com/blog/principle-of-least-privilege>