

What Is Old Becomes New

Finding classic vulnerabilities in GraphQL APIs






Tech Talk May, 2025

Aleksa Zatezalo, Security Engineer, Praetorian



Presentation Overview

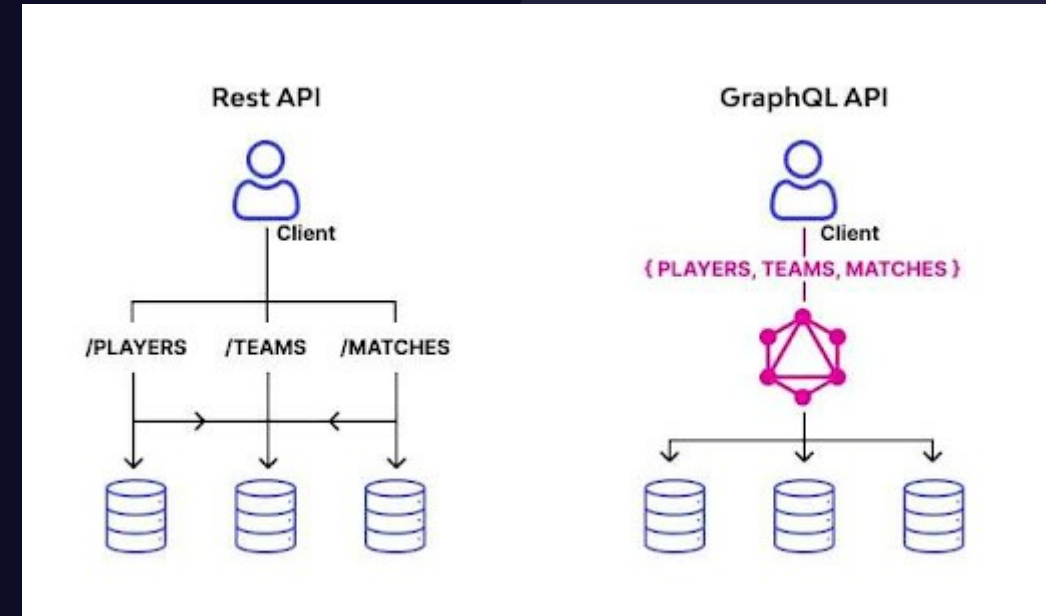
In an ever changing landscape learning to apply the fundamentals everywhere takes you a long way.

-  A Introduction to GraphQL
-  Setting up a test environment
-  Classic Vulnerability Refresher
-  Case Study
-  Introduction to GrapeQL

An Introduction To GraphQL

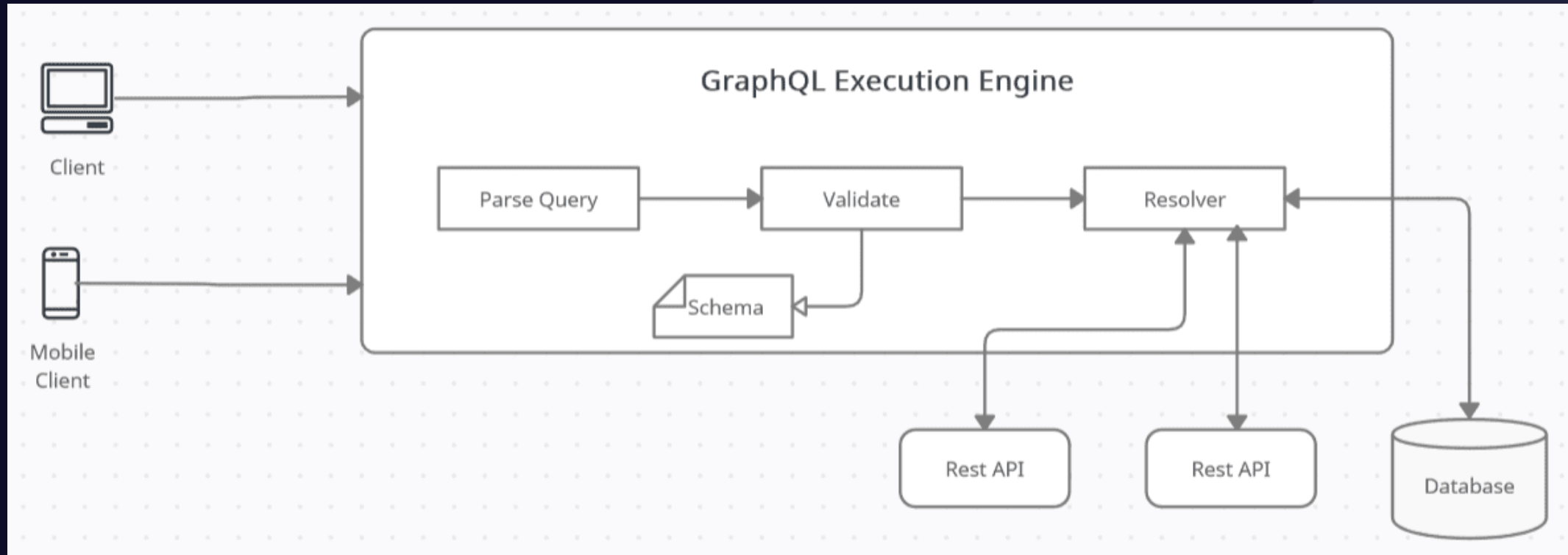
How do GraphQL APIs differ from traditional APIs?

- A query language for APIs developed by Meta
- Provides a single endpoint that handles all data operations
- Allows clients to request exactly the data they need
- Two primary operation types: queries and mutations
- Strongly typed schema known by client and server



An Introduction To GraphQL

How do GraphQL APIs differ from traditional APIs?



Setting Up A Test Environment

How do we begin testing GraphQL applications? What tools and techniques do we need?

 A Introduction to GraphQL

 **Setting up a test environment**

 Classic Vulnerability Refresher

 Case Study

 Introduction to GrapeQL

Setting Up A Test Environment

Setting up a GraphQL testing environment is surprisingly straightforward. All you need is Burpsuite, Python, a few select tools like GrapeQL, and a Docker container to host the Damn vulnerable GraphQL application.



Burp Suite

- Any GraphQL api can be tested entirely with BurpSuite
- A number of plugins, like InQ, exist to assist with testing
- GraphQL Voyager can be used alongside Burp Suite to help visualize schemas



Python

- Many automated GraphQL testing scripts have been made using python
- Sending and analyzing web requests is also quite easy



GrapeQL

- An automated testing tool that analyzes schemas and tests for a number of common "old-school" vulnerabilities such as CSRF, command injection, SQLi, and DOS.



Docker

- Will host a virtualized container with the Damn Vulnerable GraphQL application (DVGA).

Classic Vulnerability Refresher

How do common vulnerabilities differ in GraphQL applications? Is there anything different we need to look for?



A Introduction to GraphQL



Setting up a test environment



Classic Vulnerability Refresher



Case Study



Introduction to GrapeQL

Classic Vulnerability Refresher

Testing for these classic vulnerabilities does not deviate too far from normal API tests.



SQL Injection

- Look for variables like filter in queries and mutations
- Databases like postgres are commonly used








Authentication Bypass

- Closely analyze parameters like username & password, and if they are compared against a cookie during authentication

Case Study

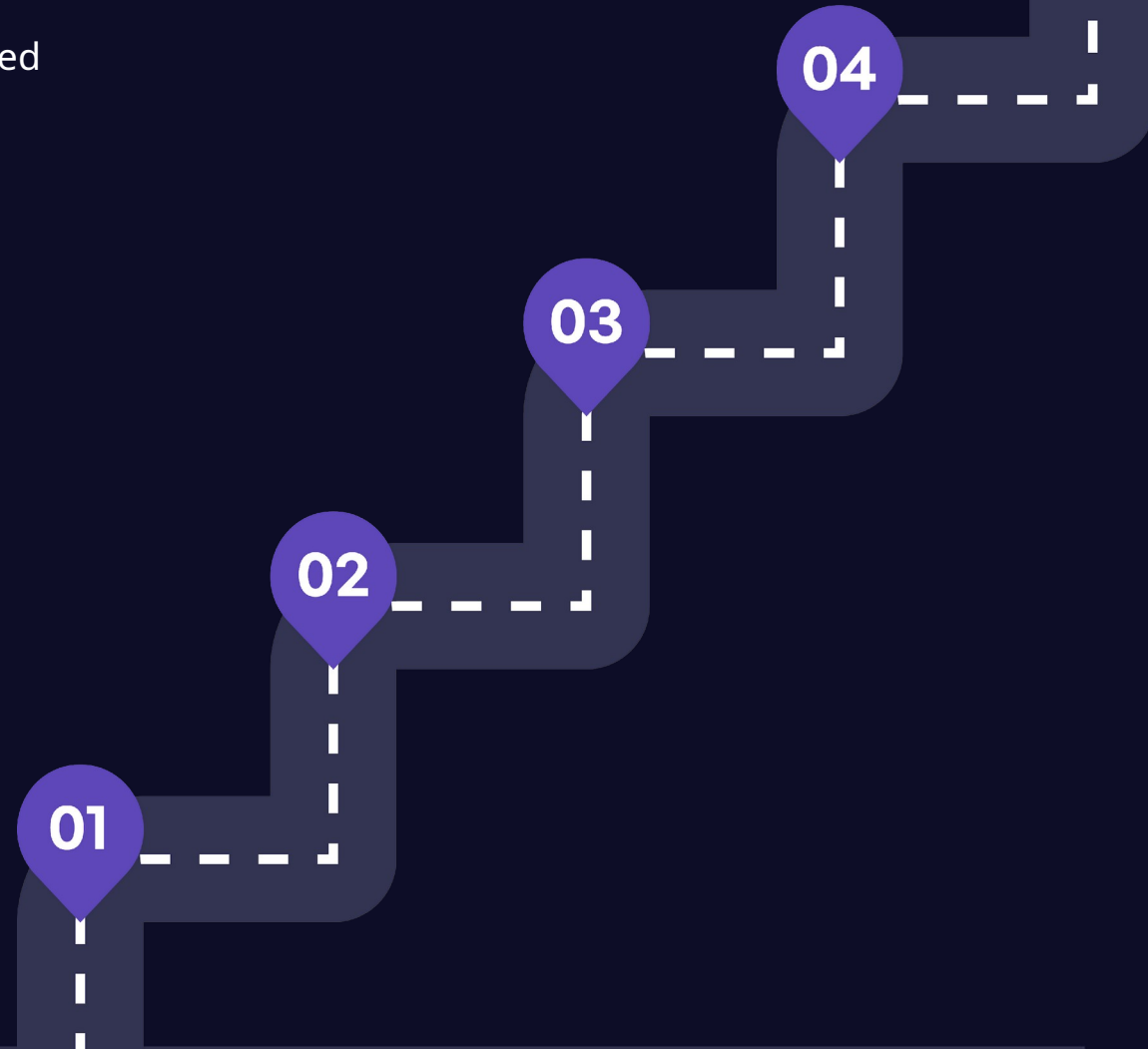
A GraphQL API with a privilege escalation vulnerability and SQLi was found on a client engagement.

-  A Introduction to GraphQL
-  Setting up a test environment
-  Classic Vulnerability Refresher
-  **Case Study**
-  Introduction to GrapeQL

Case Study: Privilege Escalation

Identified a medium severity privilege escalation vulnerability that evaluated permissions against a username and a username only.

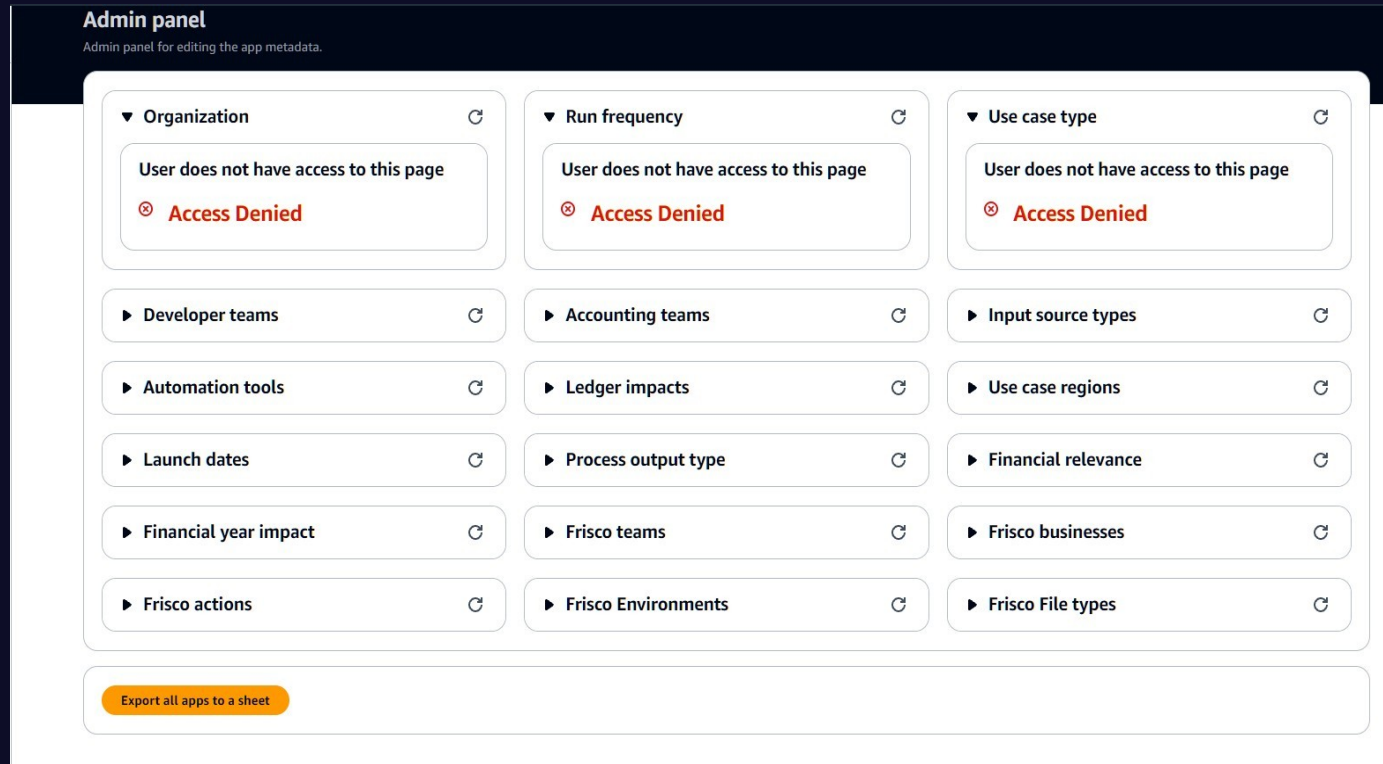
- 01 Identified an admin panel on the target website**
An admin panel was present on a target website. After proxying through burp I saw that one GraphQL request was used to evaluate access.
- 02 Collect a list of valid usernames**
Scraped internal company boards to generate a list of potential usernames. The username pattern was easy to guess.
- 03 Set up an Burp Intruder attack**
Used the username to iterate through all potential usernames to see which one would grant me access. All requests responded with an HTTP 200 so I had to look for a response that was uncharacteristically large.
- 04 Privilege Escalation**
I could access the websites admin panel and edit application metadata.



Case Study: Privilege Escalation

01

Identified an admin panel on the target website



Case Study: Privilege Escalation

02

Collect a list of valid usernames

Request

```
query MyQuery {  
  brassCheckAccess(  
    user: "zatezala"  
    id: "ABC_ADMIN"  
    accessLevelId: 1  
    stage: "beta"  
  ) {  
    is_authorized  
    optional_data  
  }  
}
```

Response

```
{  
  "data": {  
    "brassCheckAccess": {  
      "is_authorized": false,  
      "optional_data": null  
    }  
  }  
}
```

Case Study: Privilege Escalation

03

Set up an Burp Intruder attack

?

Choose an attack type

Start attack

Attack type: Sniper

?

Payload positions

Configure the positions where payloads will be inserted, they can be added into the target as well as the base request.

Target:

☒ Update Host header to match target

Add §

Clear §

Auto §

Refresh

```
1 POST /graphql HTTP/2
2 Host: efkypdtrnne37djo7ryeq1bwqm.appsync-api.us-east-1.amazonaws.com
3
4 {"query":
  "query MyQuery {\n  brassCheckAccess(\n    user: \"$szatezala$\n    id: \"ABC_ADMIN\"\n    acces
  sLevelId: 1\n    stage: \"beta\"\n  } {\n    is_authorized\n    optional_data\n  }\n}\n",
  "variables": {}}
```

Case Study: Privilege Escalation

04

Privilege Escalation

▼ Organization

Statutory X

Stores X

Tax X

Transportation X

Add

Save

▼ Run frequency

Ad-hoc/On-Demand X

Annually X

Bi-Annually X

Daily X

Monthly X

Multiple times a day X

Quarterly X

Working Day 1 X

Working Day 10 X

Working Day 2 X

Working Day 3 X

Working Day 4 X

Working Day 5 X

Working Day 6 X

Working Day 7 X

Working Day 8 X

Working Day 9 X

Add

▼ Use case type

Accrual X

Allocation X

Analysis X

Audit support X

Dashboard vizualization X

Intercompany X

MJE X

Other X

Reclass X

Reconciliation X

Reporting X

Reserve X

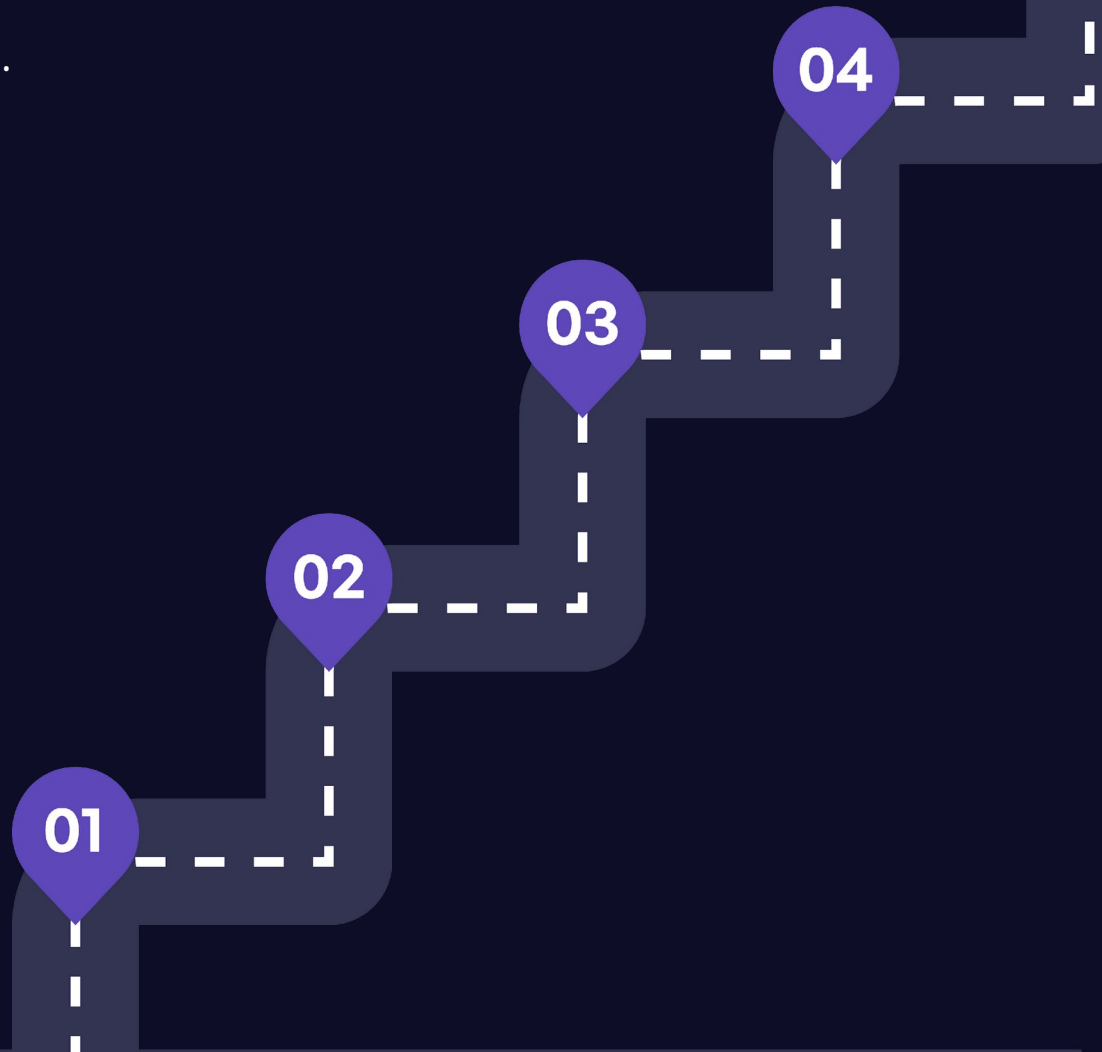
Roll forward X

Add

Save

Case Study: SQL Injection

Identified a blind SQL injection vulnerability that allowed us to extract data.

- 
- 01 Identified a suspicious request**
Function called getAppsV2 that contained a filter parameter.
 - 02 Generated a postgres error**
Used a traditional ' in the filter parameter to generate a SQL error.
 - 03 Identified it as a blind boolean SQLi**
I could use sleep statements to verify if a query was true or false. A series of requests could then be used to extract data.
 - 04 Data extraction with python**
SQLmap could be used against GraphQL APIs, in theory. In this case it could not pick up the vulnerability. A custom python script was used to extract data.

Case Study: SQL Injection

02

Generated a postgres error

```
HTTP/2 200 OK
Content-Type: application/json; charset=UTF-8
Content-Length: 347
Date: Tue, 21 May 2024 20:40:34 GMT

{
  // [REDACTED]
  "message":
    "{ 'S': 'ERROR', 'V': 'ERROR', 'C': '12P02', 'M': 'invalid input syntax for type bigint: '\\\"', 'P': '8970', 'F': 'numutils.
    c', 'L': '316', 'R': 'pg_strtoint64' "
  }
}
```


Case Study: SQL Injection

03

Identified it as a blind boolean SQLi

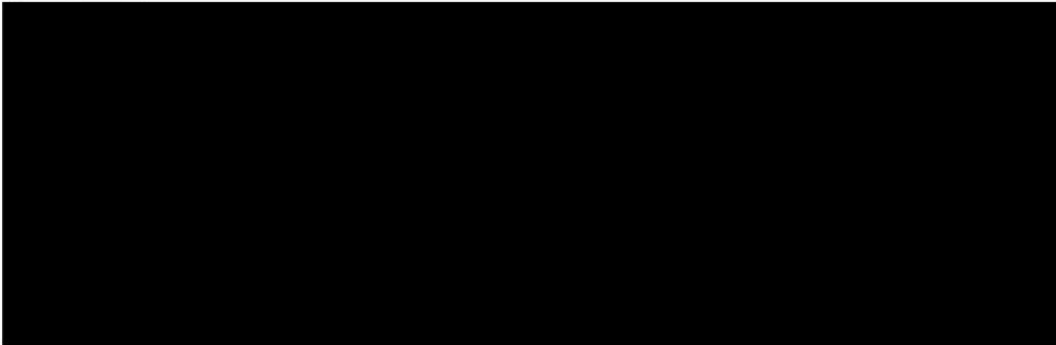
Request

Query

```
query (
  [REDACTED] ids: [], apps: [], [REDACTED] ";SELECT case when (SELECT current_setting('is_superuser'))='on' then pg_sleep(25) end;-- -" filters: [])
```

Response






HTTP/2 200 OK



```
{
  "data": {
    [REDACTED] [
      null,
      null
    ]
  }
}
```

Introduction to GrapeQL

In an ever changing landscape learning to apply the fundamentals everywhere takes you a long way.

-  A Introduction to GraphQL
-  Setting up a test environment
-  Classic Vulnerability Refresher
-  Case Study
-  **Introduction to GrapeQL**

Introduction to GrapeQL

Lots of decentralized scripts that caught low hanging fruit. Could the whole be greater than the sum of its parts?

Centralization

Many different tools, combined can become greater than the sum of their parts. Examples include:

- Burp's InQL plugin
- graphw00f
- graph-COP

Workflow

A simple workflow could be established by combining various scripts:

1. Obtaining the GraphQL schema
2. Fingerprinting the underlying server
3. Testing for CSRF
4. Testing for command injection and SQLi
5. Stress testing with circular queries and other DOS requests

GrapeQL

GrapeQL combines all aforementioned scripts and functionally as outlined in the workflow to produce a report containing all identified vulnerabilities.

Introduction to GrapeQL

```
C:\Users\zabum\Documents\code\grapeql>grapeql --api http://127.0.0.1:5013/graphql --proxy 127.0.0.1:8080 --report report.md --dos
```

EXAMPLE NOTIFICATIONS:

```
[+] Good news is printed like this.  
[!] Warnings are printed like this.  
[-] Errors are printed like this.  
[!] Logs are printed like this.
```

Introduction to GraphQL

```
=== Testing endpoint: http://127.0.0.1:5013/graphql ===

[+] Endpoint set: http://127.0.0.1:5013/graphql
[+] Proxy configured: http://127.0.0.1:8080
[+] Endpoint set: http://127.0.0.1:5013/graphql
[+] Proxy configured: http://127.0.0.1:8080
[+] Introspection successful

=== Fingerprinting GraphQL Engine ===

[+] Identified GraphQL engine: Graphene
[+] Endpoint set: http://127.0.0.1:5013/graphql
[+] Proxy configured: http://127.0.0.1:8080
[+] Introspection successful

=== Starting Information Disclosure Testing ===

[!] Testing for Field Suggestions...
[!] LOW: Field Suggestions Enabled
[!] Found issue: Field Suggestions Enabled
[!] Testing for GET-based Queries...

[+] GET-based Queries test passed
[!] Testing for GET-based Mutations...

[+] GET-based Mutations test passed
[!] Testing for URL-encoded POST...
[!] MEDIUM: URL-encoded POST Queries Enabled (Possible CSRF)
[!] Found issue: URL-encoded POST Queries Enabled (Possible CSRF)
```

Introduction to GraphQL

```
[+] Report written to report.md
```

```
=== Findings Summary ===
```

```
Severity Breakdown:
```

```
CRITICAL: 2
```

```
HIGH: 5
```

```
MEDIUM: 2
```

```
LOW: 2
```

```
INFO: 1
```

```
Total: 12 findings
```

```
[!] Critical/High Severity Findings:
```

```
CRITICAL: SQL Injection in pastes.filter - http://127.0.0.1:5013/graphql
```

```
CRITICAL: Command Injection in systemDiagnostics.cmd - http://127.0.0.1:5013/graphql
```

```
HIGH: DoS Vulnerability: Circular Query DoS - http://127.0.0.1:5013/graphql
```

```
HIGH: DoS Vulnerability: Field Duplication DoS - http://127.0.0.1:5013/graphql
```

```
HIGH: DoS Vulnerability: Deeply Nested Query DoS - http://127.0.0.1:5013/graphql
```

```
HIGH: DoS Vulnerability: Fragment Bomb DoS - http://127.0.0.1:5013/graphql
```

```
HIGH: DoS Vulnerability: Array Batching Attack - http://127.0.0.1:5013/graphql
```