

```
# Create buttons for keys
keys = ['Q', 'W', 'E', 'R', 'T', 'Y', 'U', 'I', 'O', 'P']
in the following keys:
    button = tk.Button(root, text=key, width=5,
        command= lambda k=key: key_pressed(k))
    button.pack(side=tk.LEFT)
```

```
def key_pressed(key):
    print(f"{key} pressed")
```

The Art of GraphQL Hacking

- Step-by-Step Methodology for Pentesting

```
# Create the main window
root = tk.Tk()
root.title("Simple Keyboard")
```



Who Am I ?



Ahmed Qaramany

Just a guy who loves spending his time in the security field.

- Penetration Tester at DeepStrike
- Bug Bounty Hunter sometimes

Reach me out via @c0nqr0r

Agenda

```
with open('/path/to/your/flashdrive/  
myfile.txt', 'w') as file:  
    file.write('Hello, Flash Drive!')
```

01 Introduction

02 Methodology

▪ **03** GraphQL
Attack Surface

```
with open('/path/to/your/flashdrive/  
myfile.txt', 'r') as file:  
    content = file.read()  
    print(content)
```



01

```
with open('/path/to/your/flashdrive/  
myfile.txt', 'w') as file:  
    file.write('Hello, Flash Drive!')
```

Introduction

```
with open('/path/to/your/flashdrive/  
myfile.txt', 'r') as file:  
    content = file.read()  
    print(content)
```



Basics

What is GraphQL?

- Query language for APIs
- Alternative to REST API
- Flexible but complex
- No more over-fetching and under-fetching
- JSON based
- It uses only one method (GET , POST) and one endpoint

REST vs GraphQL

GET /users/1



```
{
  "id": 1,
  "name": "John",
  "email": "john@example.com"
}
```

GET /users/1/posts



```
[
  { "id": 101, "title": "First Post" },
  { "id": 102, "title": "Second Post" }
]
```

REST API

One Request

```
{
  user(id: 1) {
    name
    email
    posts {
      title
    }
  }
}
```



```
{
  "user": {
    "name": "John",
    "email": "john@example.com",
    "posts": [
      { "title": "First Post" },
      { "title": "Second Post" }
    ]
  }
}
```

GraphQL API

GraphQL request

- All requests are routed through a single endpoint
 - such as `/graphql` , `/api/graphql` etc.
- Mostly will be through **POST** request
- Sometimes will be in a **GET** request
- HTTP response codes are often 200 OK
 - even when an error is present in the response body.

```
Request
Pretty Raw Hex GraphQL GraphQL (InQL - GraphQL Scanner)
1 POST /graphql/v1 HTTP/2
2 Host: host.net
3 Cookie: session=xxxxxxxxxx
4 User-Agent: Mozilla/5.0 (Windows NT 10.0; Win64; x64; rv:135.0)
  Gecko/20100101 Firefox/135.0
5 Content-Length: 249
6
7 {
  "query":
    "\n  mutation login($input: LoginInput!) {\n      login(i
    nput: $input) {\n          token\n              success\n
      }\n  }",
  "operationName": "login",
  "variables": {
    "input": {
      "username": "test",
      "password": "my-super-secure-password"
    }
  }
}
```

GraphQL request

A GraphQL request consists of two main parts:

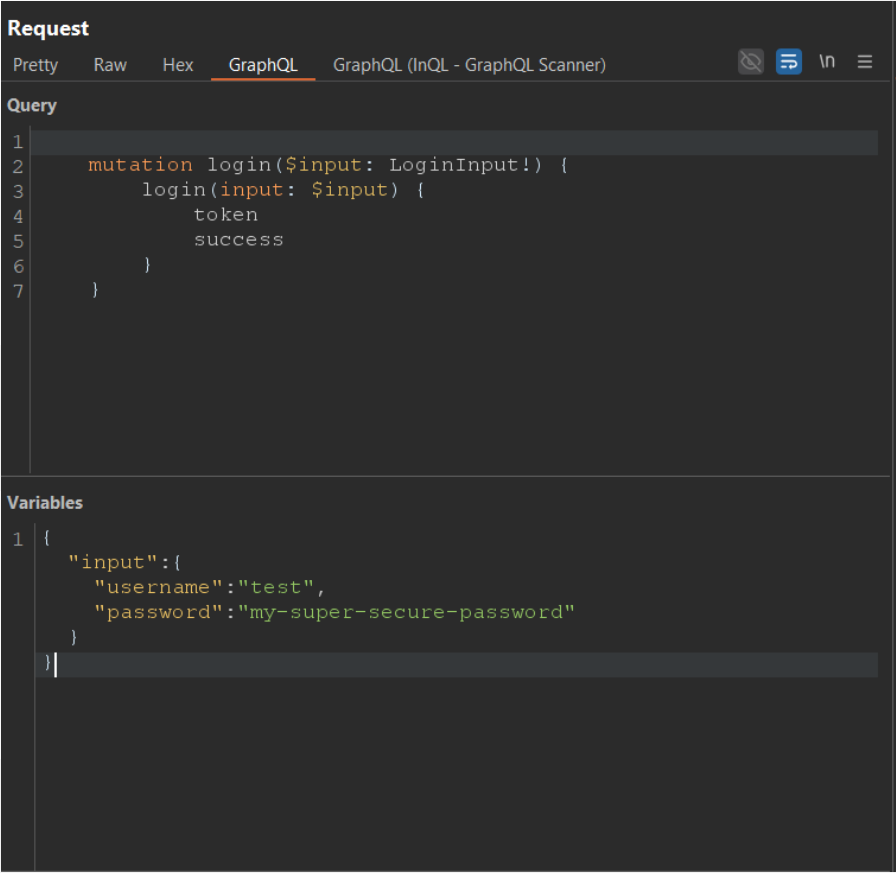
- Input (Query or Mutation)
- Variables (Optional)

No need to type query (optional)

- If you don't specify "query", the GraphQL engine automatically treats it as a query

There is a need to type mutation

- Queries are the default operation type, so query is (optional). Mutations must be declared because they change data.



The screenshot shows a GraphQL client interface with a dark theme. At the top, there's a 'Request' tab with sub-tabs for 'Pretty', 'Raw', 'Hex', 'GraphQL' (which is selected), and 'GraphQL (InQL - GraphQL Scanner)'. Below the tabs, the 'Query' section contains a GraphQL mutation:

```
1  
2  mutation login($input: LoginInput!) {  
3      login(input: $input) {  
4          token  
5          success  
6      }  
7  }
```

 The 'Variables' section below it contains a JSON object:

```
1 {  
  "input": {  
    "username": "test",  
    "password": "my-super-secure-password"  
  }  
}
```

Essential Components

Queries: Fetching Data

- Similar to **GET** in REST API
- Used for reading data
- **Example:** query retrieves a user by id: 1, returning their name, email, and a list of their post titles.

Request:

```
{
  User( id: 1) {
    name
    email
    posts {
      title
    }
  }
}
```



Response:

```
{
  "user": {
    "name": "John",
    "email": "john@example.com",
    "posts": [
      { "title": "First Post" },
      { "title": "Second Post" }
    ]
  }
}
```

Essential Components

Mutations: Modifying Data

- Similar to **POST** in REST API
- Used for creating / writing data

Example: mutation creates a new post with a title, content, and author ID, returning the new post's ID and title.

Request:

```
mutation {  
  CreatePost(  
    title: "New Post",  
    content: "This is a new post",  
    AuthorId: 1  
  )  
  {  
    id  
    title  
  }  
}
```



Response:

```
{  
  "createPost": {  
    "id": "101",  
    "title": "New Post"  
  }  
}
```

Essential Components

Schema

- The **schema** shows that:
- A **User** has multiple posts.
- Each **Post** has an author of type User.
- A query (**getPosts**) to fetch all posts.
- A mutation (**createPost**) to create a new post with a title, content, and authors.

```
type User {  
  id: ID  
  name: String  
  email: String  
  posts: [Post]  
}
```

```
type Post {  
  id: ID  
  title: String  
  content: String  
  author: [User]  
}
```

```
type Query {  
  getPosts: [Post]  
}
```

```
type Mutation {  
  CreatePost(  
    title: String!,  
    content: String!,  
    authorIds: [ID!]!  
  ) :Post  
}
```

Essential Components

Introspection Query

- feature of the GraphQL specification that allows clients to query the schema itself
- some times it will be enabled and sometimes it will be not, we'll cover all cases.

Example of an Introspection Query :

```
{  
  __schema {  
    types {  
      name  
      fields {  
        name  
      }  
    }  
  }  
}
```

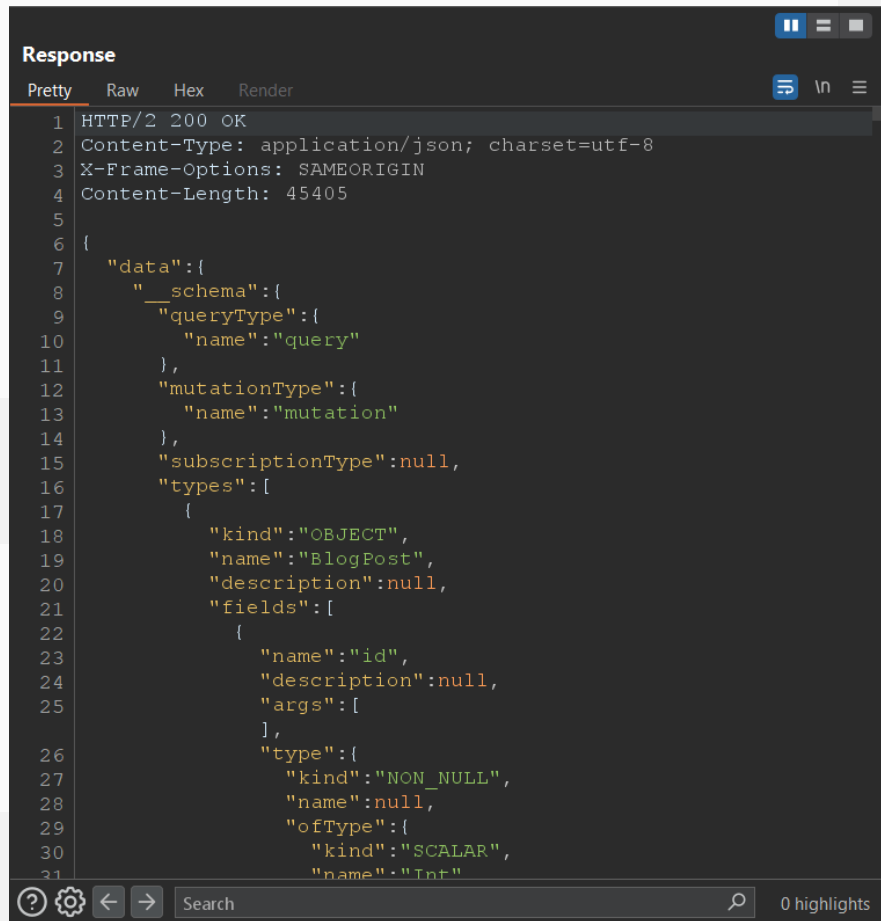


Response will be in the Next Slide

If it is Enabled

You will see the schema like this :

```
{
  "data": {
    "__schema": {
      "types": [
        {
          "name": "User",
          "fields": [
            { "name": "id" },
            { "name": "name" },
            { "name": "email" }
          ]
        },
        {
          "name": "Post",
          "fields": [
            { "name": "id" },
            { "name": "title" },
            { "name": "content" }
          ]
        }
      ]
    }
  }
}
```



```
Response
Pretty Raw Hex Render
1 HTTP/2 200 OK
2 Content-Type: application/json; charset=utf-8
3 X-Frame-Options: SAMEORIGIN
4 Content-Length: 45405
5
6 {
7   "data": {
8     "__schema": {
9       "queryType": {
10         "name": "query"
11       },
12       "mutationType": {
13         "name": "mutation"
14       },
15       "subscriptionType": null,
16       "types": [
17         {
18           "kind": "OBJECT",
19           "name": "BlogPost",
20           "description": null,
21           "fields": [
22             {
23               "name": "id",
24               "description": null,
25               "args": [
26               ],
27               "type": {
28                 "kind": "NON_NULL",
29                 "name": null,
30                 "ofType": {
31                   "kind": "SCALAR",
32                   "name": "Int"
```

Introspection Query

If it is not Enabled

For sure you will not be able to query the schema!
But we can do some other interesting things

```
Preview ▾ Header 21 Cookie Timeline
1 {
2   "errors": [
3     {
4       "message": "GraphQL introspection is not allowed by Apollo Server, but
the query contained __schema or __type. To enable introspection, pass
introspection: true to ApolloServer in production",
5       "extensions": {
6         "code": "GRAPHQL_VALIDATION_FAILED"
7       }
8     }
9   ]
10 }
```

02

```
with open('/path/to/your/flashdrive/  
myfile.txt', 'w') as file:  
    file.write('Hello, Flash Drive!')
```

Methodology

```
with open('/path/to/your/flashdrive/  
myfile.txt', 'r') as file:  
    content = file.read()  
    print(content)
```

positi
Cairo
ha

Basic recon

- We need to find exposed GraphQL instances
- It is a good idea to include certain paths when performing a directory brute force attack.

graphw00f is a tool for GraphQL fingerprinting during recon.
It helps identify the GraphQL engine, and other good things

```
graphw00f -t https://target.com/graphql
```

```
[*] Checking if GraphQL is available at https://target.com/graphql...
```

```
[*] Found GraphQL...
```

```
[*] Attempting to fingerprint...
```

```
[*] Discovered GraphQL Engine: (HyperGraphQL)
```

```
[!] Technologies: Java
```

```
[!] Homepage: https://www.target.com
```

```
[*] Completed.
```

Some common GraphQL paths :

```
/graphql  
/graphql  
/graphql.php  
/graphql/console  
/api  
/api/graphql  
/graphql/api  
/graphql/graphql  
/v1/graphql  
/v2/graphql  
/graphql/v1  
/graphql/v2  
/gql  
/graphql-playground  
/playground  
/altair  
/query
```

Know the Engine

GraphQL-Threat-Matrix

- Comparing security risks in different GraphQL implementations.
- Helping teams make safer choices when using GraphQL.
- Improving security for all GraphQL implementations.

Legend							
✔ - Enabled by Default							
⚠ - Disabled by Default							
✖ - No Support							
Implementation	Validations	Field Suggestions	Query Depth limit	Query Cost Analysis	Automatic Persisted Queries	Introspection	Debug Mode
wp-graphql	38	✔	⚠	✖	✖	⚠	⚠
graphql-php	37	✔	⚠	⚠	✖	✔	⚠
graphql-api-for-wp	37	⚠	✖	✖	✔	✔	⚠
Apollo	34	✔	⚠	⚠	✔	✔	✔
graphql-yoga	34	✔	⚠	✖	✖	⚠	⚠
graphene	34	✔	✖	✖	✖	✔	✖
Ariadne	34	✔	⚠	⚠	✖	✔	⚠
Strawberry	34	✔	⚠	✖	✖	✔	✖
graphql-dotnet	29	✔	⚠	⚠	✖	✔	✖
graphql-ruby	28	✔	✖	⚠	⚠	✔	✖
Sangria	27	✔	⚠	⚠	✖	✔	✖
Tartiflette	26	✖	✖	✖	✖	✔	✖

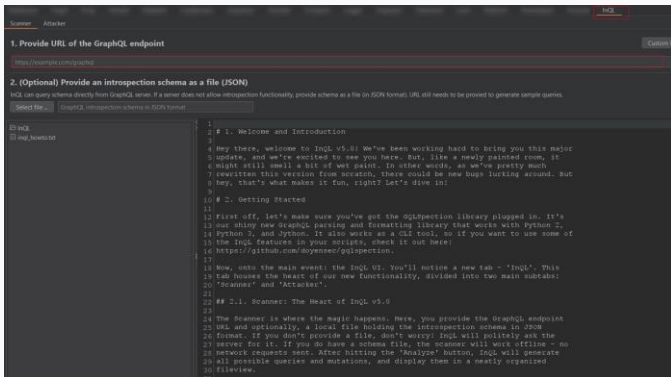
<https://github.com/nicholasaleks/graphql-threat-matrix/>

Testing Introspection Query

Test Case 1: Introspection Query is enabled

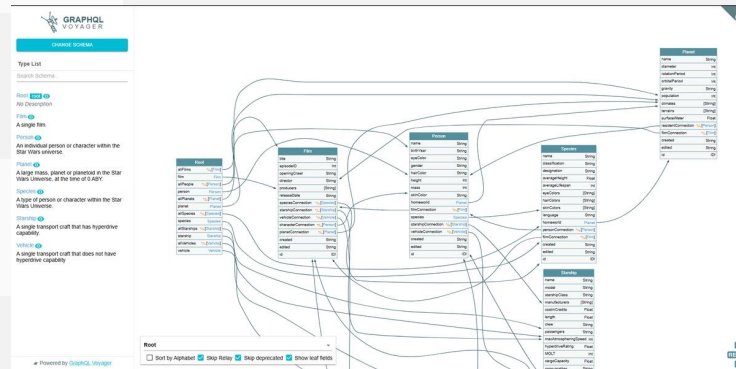
- Not a security issue but may lead to some private things
- we can use some powerful tools to dump the schema and deals with its queries and mutations

InQL



```
1. Provide URL of the GraphQL endpoint
2. (Optional) Provide an introspection schema as a file (JSON)
InQL can query schema directly from GraphQL server. If a server does not allow introspection functionality, provide schema as a file in JSON format. URL still needs to be provided to generate sample queries.
Select file... GraphQL introspection schema in JSON format
InQL
InQL schema
1 # 1. Welcome and Introduction
2
3 Hey there, welcome to InQL v1.0! We've been working hard to bring you this magic
4 update, and we're excited to see you here. But, like a newly painted room, it
5 might still smell a bit of wet paint. In other words, we're pretty much
6 rewritten this version from scratch, there could be new bugs lurking around. But
7 hey, that's what makes it fun, right? Let's dive in!
8
9 # 2. Getting Started
10
11 First off, let's make sure you've got the gql introspection library plugged in. It's
12 our shiny new schema parsing and formatting library that works with Python 2,
13 Python 3, and Jython. It also works as a CLI tool, so if you want to use some of
14 the InQL features in your scripts, check it out here:
15 https://github.com/dywaner/gqlintrospection
16
17 Now, onto the main event! The InQL UI. You'll notice a new tab - "InQL". This
18 tab houses the heart of our new functionality, divided into two main sub-tabs:
19 "Scanner" and "Attacker".
20
21 ## 2.1. Scanner: The Heart of InQL v1.0
22
23 The Scanner is where the magic happens. Here, you provide the GraphQL endpoint
24 URL and optionally, a local file holding the introspection schema in JSON
25 format. If you don't provide a file, don't worry! InQL will magically get the
26 schema for it. If you do have a schema file, the scanner will work offline - no
27 network requests needed. After hitting the "Run" button, InQL will generate
28 all possible queries and mutations, and display them in a neatly organized
29 file view.
```

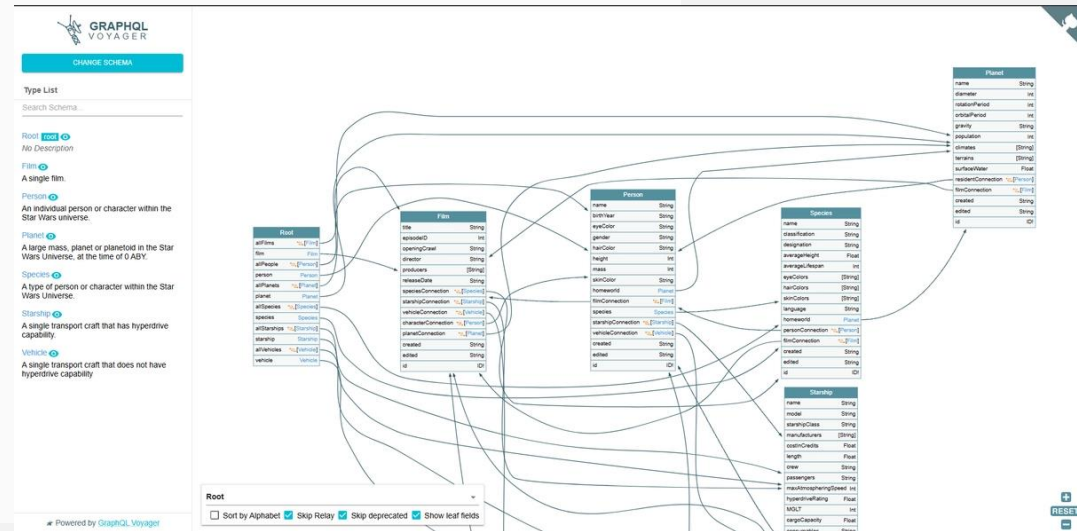
GraphQL
Voyager



Testing Introspection Query

GraphQL Voyager

- visualize the schema
- you will need to make the query and mutation by your self in burp



Testing Introspection Query

- In Burp Suite we use InQL he It's different of the visualization
- it makes separated folder for query and mutation

InQL

1. Provide URL of the GraphQL endpoint

`https://0a33000204b3f282812016b900d20014.web-security-academy.net/graphql/v1`

2. (Optional) Provide an introspection schema as a file (JSON)

InQL can query schema directly from GraphQL server. If a server does not allow introspection functionality, provide schema as a file (in JSON format). URL still needs to be provided to generate sample queries.

Select file ... GraphQL introspection schema in JSON format

File explorer view:

- ✓ InQL
 - ✓ 0a33000204b3f282812016b900d20014.web-security-academy.net
 - ✓ 2025-02-20_193054
 - ✓ mutations
 - changeEmail.graphql
 - login.graphql
 - ✓ queries
 - getAllBlogPosts.graphql
 - getBlogPost.graphql
 - getUser.graphql
 - poi.txt
 - request_template.txt
 - schema.json
 - inql_howto.txt

GraphQL query editor view:

```
1 mutation {  
2   changeEmail(input: ChangeEmailInput) {  
3     email  
4   }  
5 }
```

Testing Introspection Query

Test Case 2: Introspection Query is NOT enabled

- If introspection is disabled, there are other ways to bypass this restriction.

1. Bypass Introspection Using Regex Manipulation

- Some GraphQL security filters block requests that contains "**__schema**" or "**__type**" using strict regex matching
- By adding characters like a **space**, **newline**, or **comma** after "**query{**" to avoid detection .

```
{ "query": "query{\n__schema { queryType{name}}}" }
```

Testing Introspection Query

2. Bypass by Switching from POST to GET

- Many GraphQL APIs are designed to accept POST requests by default. However, some APIs also support GET requests.

`https://target.com/graphql?query={__schema{queryType{name}}}`

What Next?

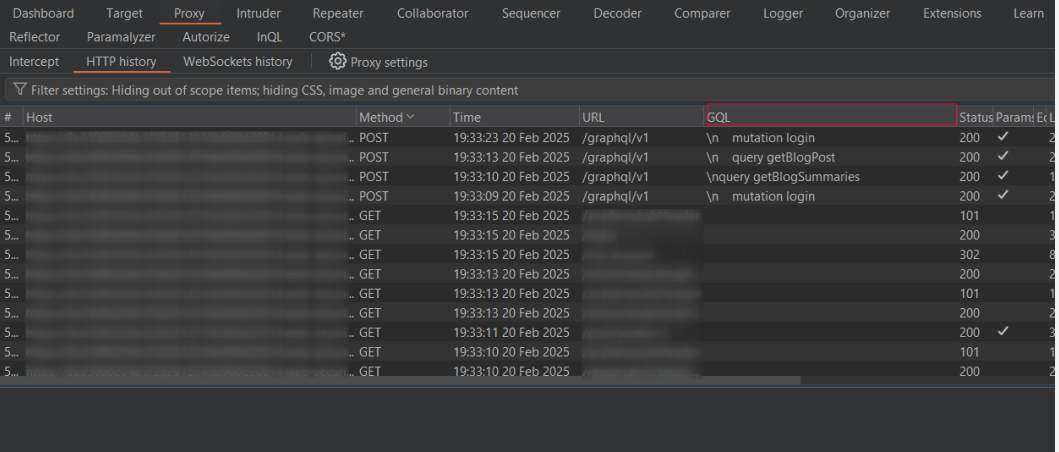
Widen you scope

- Click all the Buttons
- Use the application as a normal user , use all functionalities
- JS analysis to gather them (JS Miner)
- Use tools like (GraphQL Raider) intercepts and extracts GraphQL queries & mutations in HTTP requests.

What Next?

Prepare the Burp Suite

Make a column for getting graphql mutations , queries name fast



#	Host	Method	Time	URL	GQL	Status	Param	Ext
5...	...	POST	19:33:23 20 Feb 2025	/graphql/v1	\n mutation login	200	✓	2
5...	...	POST	19:33:13 20 Feb 2025	/graphql/v1	\n query getBlogPost	200	✓	2
5...	...	POST	19:33:10 20 Feb 2025	/graphql/v1	\nquery getBlogSummaries	200	✓	1
5...	...	POST	19:33:09 20 Feb 2025	/graphql/v1	\n mutation login	200	✓	2
5...	...	GET	19:33:15 20 Feb 2025			101		1
5...	...	GET	19:33:15 20 Feb 2025			200		3
5...	...	GET	19:33:15 20 Feb 2025			302		8
5...	...	GET	19:33:13 20 Feb 2025			200		2
5...	...	GET	19:33:13 20 Feb 2025			101		1
5...	...	GET	19:33:13 20 Feb 2025			200		2
5...	...	GET	19:33:11 20 Feb 2025			200	✓	3
5...	...	GET	19:33:10 20 Feb 2025			101		1
5...	...	GET	19:33:10 20 Feb 2025			200		2

Code :

```
String q = requestResponse.request().parameterValue("query", HttpParameterType.JSON);
if (q != null) {
    return q.split("\\{\\\\"}[0];
}
return "";
```

03

```
with open('/path/to/your/flashdrive/  
myfile.txt', 'w') as file:  
    file.write('Hello, Flash Drive!')
```

GraphQL Attack Surface

```
with open('/path/to/your/flashdrive/  
myfile.txt', 'r') as file:  
    content = file.read()  
    print(content)
```

positi
Cairo
ha

Field Suggestions

- Some GraphQL APIs have auto-suggestions enabled in their responses.
- It's a feature of GraphQL , Not a vulnerability
- It could suggest a private query / mutation
- It is good if the introspection query is not enabled
- **Clairvoyance** to fuzz that behavior

```
type Query {  
  user(id: ID!): User  
}
```

```
type User {  
  id: ID!  
  name: String  
  email: String  
}
```

```
{  
  user(id: "123") {  
    nam  
  }  
}
```

```
{  
  "errors": [  
    {  
      "message": "Cannot query field 'nam' on type 'User'. Did you mean 'name'?",  
      "locations": [{ "line": 3, "column": 5 }]  
    }  
  ]  
}
```

Rate Limiting Bypass

Rate Limiting Bypass using Aliases

- Server blocks OTP verification requests after 5 attempts due to rate limiting.
- the attacker sends multiple aliased queries in a single request

```
type Mutation {  
  verifyOTP(phone: String!, otp: String!): Boolean  
}
```

```
mutation {  
  attempt1: verifyOTP(phone: "+1234567890", otp: "111111")  
  attempt2: verifyOTP(phone: "+1234567890", otp: "222222")  
  attempt3: verifyOTP(phone: "+1234567890", otp: "333333")  
  attempt4: verifyOTP(phone: "+1234567890", otp: "444444")  
  attempt5: verifyOTP(phone: "+1234567890", otp: "555555")  
  attempt6: verifyOTP(phone: "+1234567890", otp: "666666")  
}
```

```
# Bypasses rate limit!  
}
```

Should be

- Instead of limiting requests, limit executions per user.
- Limit Aliases – Restrict the number of queries per request.

Denial of Service (DoS)

Circular Queries

Also known as Nested Query Attack

- If a GraphQL schema allows a type to reference itself (directly or indirectly)
- If a GraphQL doesn't enforce a default depth limit on queries

```
type User {  
  id: ID!  
  name: String!  
  friends: [User]! # Self-referencing field  
}
```



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

```
type User {  
  id: ID!  
  name: String!  
  friends(first: Int = 5, offset: Int = 0): [User]!  
  # Limit the number of friends returned  
}
```



Denial of Service (DoS)

Large Lists (Over-fetching Data)

- Asking for thousands of records at once can overload the server's memory and processing power.



```
type Query {  
  users: [User]!  
  # Returns all users without any limit  
}
```



```
type Query {  
  users(first: Int = 10, offset: Int = 0): [User]!  
  #(default: 10 users per request)  
}
```

```
query {  
  users(first: 10000) {  
    # Fetching 10,000 users in one request  
    id  
    name  
    email  
  }  
}
```

Denial of Service (DoS)

Field Duplication

- Repeat the same field multiple times in a query.
- Even if the fields are simple, excessive repetition can overload the server.

Should be

- Restricting excessive field requests using query cost analysis.

```
type Query {  
  user(id: ID!): User!  
}
```

```
type User {  
  id: ID!  
  name: String!  
  email: String!  
}
```

```
query {  
  user(id: "123") {  
    id  
    id  
    id  
    id  
    id  
    id  
    id  
    id  
    id  
    id  
    id  
    id  
  }  
}  
# Repeated thousands of times...
```

Denial of Service (DoS)

Query Aliasing

- use aliases to request the same field multiple times,
- forcing the server to process the same data repeatedly.

```
query {  
  user1: user(id: "1") { name }  
  user2: user(id: "1") { name }  
  user3: user(id: "1") { name }  
  # Repeated thousands of times...  
}
```

Query Batching

- send multiple queries in a single request,
- making the server process too many operations at once
- server overwhelmed

```
query {  
  first: user(id: "1") { name }  
  second: user(id: "2") { name }  
  third: user(id: "3") { name }  
  # Repeated thousands of times...  
}
```

GraphQL CSRF

- **CSRF hates JSON content/type**

So check if it accepts application/x-www-form-urlencoded or text/plain content types.

- **Cookie-Based Authentication?**

If authentication relies on cookies, CSRF can be exploited if proper protections are missing.

- **SameSite = Lax?**

Check making mutations through the GET request

-

Automation GraphQL testing

- **GraphCrawler**
A GraphQL automated security toolkit. Grab introspection, search for sensitive queries, and then test authorization.
- **CrackQL**
GraphQL password brute-force and fuzzing utility.
- **GraphQLMap**
A scripting engine to interact with a GraphQL endpoint for pentesting purposes.
- **GraphQL.Security**
One-click quick security scan of your GraphQL endpoints. Free, no login required.
- **BatchQL**
GraphQL security auditing script with a focus on performing batch GraphQL queries and mutations.

Thanks