ISSN No:-2456-2165

Vulnerability Analysis of GraphQl Api's after Integrated with React Js

[1]Vinayak Iswalkar
University of Mumbai
Institute of Distance & Open Learning
(IDOL), Mumbai, India

[2]Prashant Singh
University of Mumbai
Institute of Distance & Open Learning
(IDOL), Mumbai, India

[3]Neha Mahesh Raut
Assistant Professor (AI&DS)
Vidyavardhini's College of Engineering and Technology

[4]Dr. Uday Aswalekar
HOD (Mechanical Engineering)
Vidyavardhini's College of Engineering and Technology

Abstract:- GraphQL is a novel query language proposed by Facebook to implement Web-based APIs. In this paper, we will present a practical analysis of graphql api's while integrating with react js. First, we will see how graphql works in web applications and How we can integrate it with react js. After that, we will work on the vulnerability finder tools to check the loopholes in the api's. For the result, we will show the various vulnerability issues in graphql and provide the solution to fix those issues.

Keywords:- *Web Development, Full Stack Development, React.js, Graphql, Vulnerability, Bug Bounty.*

I. INTRODUCTION

A flexible and intuitive query language called GraphQl is intended for use in the development of client applications and systems that need to describe their data requirements and interactions. GraphQL adopts a declarative model for data retrieval, allowing clients to define precisely what information they require from the API. Hence, unlike REST APIs, which offer several endpoints to enable the client to access the data it requires, GraphQL offers a single endpoint. GraphQL clients usually use a single query to obtain all the data required to complete a task; in contrast, REST clients generally require clients to request numerous endpoints. Thus, the number of endpoints accessed by REST clients and the number of endpoints reached by the same client after refactoring to utilize GraphQL are compared in this RQ. Major web services are now supporting GraphQL as it gains popularity.

II. METHODOLOGY

The answer to a more fluid and personalized method for sophisticated API data retrieval turned out to be GraphQL. The abundance of queries and endpoints in classical REST adds to the complexity and duplication of data in API interactions. GraphQL is essentially a server-side runtime and query language for APIs that let clients submit numerous resource requests using different kinds and attributes.

This is the difference between REST and GraphQL requests.

A. REST:

Since it is not restricted to any particular transfer protocol, REST (Representational State Transfer) is an API (Application Programming Interface) that facilitates client-server communications on a web application using the HTTP protocol. RESTful clients retrieve resources and either display or utilise them, while RESTful servers offer a way to access resources (data sources). Using the supplied parameters, GraphQL resolvers can be used as a REST API gateway to create and submit requests to the API. If the parameters are not correctly validated, resolvers may be susceptible to SSRF attacks.

```
{
  "userId": "12345",
  "username": "john_doe",
  "email": "john.doe@example.com",
  "firstName": "John",
  "lastName": "Doe",
  "age": 30,
  "gender": "male",
  "createdAt": "2022-01-01T10:30:00Z",
  "updatedAt": "2023-04-19T09:30:00Z"
}
```

Fig 1 Sample Raw Inputs REST Multiple Endpoints

B. GraphQL:

Facebook Inc. has developed GraphQL, an open-source query language for building new APIs. Only the data specified by a type system in the relevant Web Service is returned by GraphQL after it has completed server-side queries. GraphQL is not directly tied to any database, even though it is a query language; that is, it is not restricted to any particular database, be it SQL or NOSQL. GraphQL is independent of the server-side programming language and database since it is merely a translator (query language) and runtime.

```
query {
  getUser(userId: "12345") {
    userId
    username
    email
    firstName
    lastName
    age
    gender
    createdAt
    updatedAt
  }
}
```

Fig 2 Sample Graphql Query Single Endpoint

GraphQL's fundamental concept is based on mutations and queries that only return and specify the data we actually need. This is an illustration of a mutation and query in GraphQL.

Query

```
{
  movie(id:"1234595830") {
    name
    id
    actor {
      name
      age
      id
    }
  }
}
```

Mutation

```
mutation IncrementViews($incrementViewsId: ID!) {
  incrementViews(id: $incrementViewsId) {
    code
    success
    message
    track {
      Id
      numberOfViews
    }
  }
}
```

Vars

```
{
"incrementViewsId": "2345"
}
```

Without over- or under-fetching, the clients describe the needed data and its format in queries they send to the server.

In contrast to REST, GraphQL has become more popular for the main reasons listed below:

- *Easier to Fetch and Manage Data:*
  A single endpoint is used for both data queries and modifications. This covers making changes or requesting data (e.g., CRUD methods).

- *Schema as API Documentation:*
  Both queries and changes to the data are made using a single endpoint. This includes requesting data or making changes (e.g., CRUD methods).

- *Use in Parallel with REST:*
  It takes gradual adoption to move away from legacy artefacts rather than just deleting and switching. Developers can improve existing REST APIs without disrupting current clients thanks to GraphQL's built-in versioning.

Because it doesn't need to make numerous calls to different endpoints to obtain our data, it is faster than REST APIs.

III. INTEGRATION WITH REACT JS

We must first learn how to call a GraphQL API over HTTP before we can begin testing the various GraphQL components. There are various methods by which we could initiate an API. I've listed a few simple, often used methods below:

Let's review the top five methods for using React to retrieve data from GraphQL APIs.

Although there are several well-known libraries designed to interface with GraphQL APIs from a React application, there are numerous methods for retrieving data using GraphQL.

I've included code samples that demonstrate how to use each of these various methods to connect React with GraphQL and how to fetch or "query" data in the shortest amount of code possible.

- *Apollo Client :*
  Apollo Client is the most widely used and extensive GraphQL library.

We can manage data locally using an internal cache and a comprehensive state management API, in addition to using GraphQL to fetch remote data, as we are doing here.

In order to utilize Apollo Client, we must first install GraphQL and the primary Apollo Client dependency:

```
npm install @apollo/client graphql
```

The Apollo Client is intended to be utilized throughout our whole application. In order to accomplish this, we will pass a created Apollo client down our entire component tree using a unique Apollo Provider component. A GraphQL endpoint must be specified as the URI value when creating an Apollo Client. Furthermore, a cache needs to be specified. Apollo Client has a built-in in-memory cache that it uses to manage and cache our queries and the data they return locally:

```
import React from "react";
import ReactDOM from "react-dom";
import { ApolloProvider, ApolloClient,
InMemoryCache } from "@apollo/client";

import App from "./App";

const client = new ApolloClient({
  uri: "https://api.spacex.land/graphql/",
  cache: new InMemoryCache()
});

const rootElement =
document.getElementById("root");
ReactDOM.render(
  <ApolloProvider client={client}>
    <App />
  </ApolloProvider>,
  rootElement
);
```

We can use all of the various React hooks that Apollo Client provides for all of the various GraphQL operations once we've set up the Provider and client within our App component. With a special hook called useApolloClient, we can even use the created Apollo Client directly. We'll make use of the useQuery hook because this is just a data query.

```
import React from "react";
import { useQuery, gql } from
"@apollo/client";

const FILMS_QUERY = gql`
  {
    launchesPast(limit: 10) {
      id
      mission_name
    }
  }
`;

export default function App() {
  const { data, loading, error } =
useQuery(FILMS_QUERY);
  if (loading) return "Loading...";
  if (error) return <pre>{error.message}</pre>
  return (
    <div>
      <h1>SpaceX Launches</h1>
      <ul>
        {data.launchesPast.map((launch) => (
          <li
key={launch.id}>{launch.mission_name}</li>
        ))}
      </ul>
    </div>
  );
}
```

➢ *Urql :*

Urql is another feature-rich library that links React apps and GraphQL APIs. It is slightly smaller in size and requires less setup code, but it aims to give you many of the features and syntax of Apollo. While it lacks Apollo's integrated state management library, it does offer caching capabilities if desired. Installing the urql and GraphQL packages is required in order to use urql as your GraphQL client-library.

npm install urql graphql

You should use the specific Provider component and create a client with your GraphQL endpoint, just like Apollo did. Keep in mind that you are not required to specify a cache by default.

```
import React from "react";
import ReactDOM from "react-dom";
import App from "./App";
import { createClient, Provider } from 'urql';

const client = createClient({
  url: 'https://api.spacex.land/graphql/',
});

const rootElement =
document.getElementById("root");
ReactDOM.render(
  <Provider value={client}>
    <App />
  </Provider>,
  rootElement
);
```

Urql is very similar to Apollo in that it provides you with custom hooks that manage all of the common GraphQL operations—thus, the names are similar. Once more, the urql package's useQuery hook is available to you. That being said, you can write your query using a template literal and do away with the need for the gql function. You can destructure the array you receive back from useQuery as an array rather than an object. This array's first element is an object called result, which provides you with several destructible properties, including data, fetching, and error.

```
import React from "react";
import { useQuery } from 'urql';

const FILMS_QUERY = `
  {
    launchesPast(limit: 10) {
      id
      mission_name
    }
  }
`;

export default function App() {
  const [result] = useQuery({
    query: FILMS_QUERY,
  });

  const { data, fetching, error } = result;

  if (fetching) return "Loading...";
  if (error) return
<pre>{error.message}</pre>
  return (
    <div>
      <h1>SpaceX Launches</h1>
      <ul>
        {data.launchesPast.map((launch) => (
          <li
key={launch.id}>{launch.mission_name}</li>
        ))}
      </ul>
    </div>
  );
}
```

You can manage both your error and loading states while you're fetching your remote data, just like how Apollo displays the data that you fetch.

➢ *React Query + GraphQL Request :*

As we will see later, it's crucial to note that you can interact with your GraphQL API without a complex, heavy-weight GraphQL client library like urql or Apollo. Libraries such as Apollo and urql were developed to provide you with more tools to manage the server state in your React client in addition to assisting you with all the standard GraphQL operations. In addition, they have unique hooks that simplify handling loading, errors, and other related states, among other repetitive tasks. Let's look at how you can use a very simplified GraphQL library for your data fetching in conjunction with an improved method for handling and caching the server state that you're integrating into your application, keeping that in mind. Data can be easily retrieved with the aid of the graphql-request package.

```
npm install react-query graphql-request
```

Using the Provider component of React Query, you build a query client with the option to configure default data fetching parameters. The useQuery hook can then be used within your app component or any of its offspring.

```
import React from "react";
import ReactDOM from "react-dom";
import App from "./App";
import { QueryClient, QueryClientProvider }
from "react-query";

const client = new QueryClient();

const rootElement =
document.getElementById("root");
ReactDOM.render(
  <QueryClientProvider client={client}>
    <App />
  </QueryClientProvider>,
  rootElement
);
```

You only need to pass it a key value as the first argument to act as an identifier in order for the result of your operation to be stored in the React Query cache. This makes it very easy to refer to and retrieve data from the cache, as well as to invalidate or refetch a specific query in order to retrieve updated data. Once more, the outcome of executing that request will be returned by this hook. You must indicate how to retrieve the data in order for the second argument, useQuery, to function. React Query will handle resolving the promise that the GraphQL request returns.

```
import React from "react";
import { request, gql } from "graphql-
request";
import { useQuery } from "react-query";

const endpoint =
"https://api.spacex.land/graphql/";
const FILMS_QUERY = gql`
  {
    launchesPast(limit: 10) {
      id
      mission_name
    }
  }
`;

export default function App() {
  const { data, isLoading, error } =
useQuery("launches", () => {
    return request(endpoint, FILMS_QUERY);
  });

  if (isLoading) return "Loading...";
  if (error) return
<pre>{error.message}</pre>;

  return (
    <div>
      <h1>SpaceX Launches</h1>
      <ul>
        {data.launchesPast.map((launch) => (
          <li
key={launch.id}>{launch.mission_name}</li>
        ))}
      </ul>
    </div>);}
```

➢ *React Query + Axios :*

To fetch your data, you can use even more straightforward HTTP client libraries that are unrelated to GraphQL. The popular library axios can be applied in this situation. Once more, it can be combined with React Query to obtain all the unique hooks and state management features.

```
npm install react-query axios
```

A POST request must be made to your API endpoint in order to execute a query from a GraphQL API using an HTTP client such as Axios. You will supply an object with a property called query, which will be set to your film's query, for the data that you send along with the request.

```
import React from "react";
import axios from "axios";
import { useQuery } from "react-query";

const endpoint =
"https://api.spacex.land/graphql/";
const FILMS_QUERY = `
  {
    launchesPast(limit: 10) {
      id
      mission_name
    }
  }
`;

export default function App() {
  const { data, isLoading, error } =
useQuery("launches", () => {
    return axios({
      url: endpoint,
      method: "POST",
      data: {
        query: FILMS_QUERY
      }
    }).then(response => response.data.data);
  });

  if (isLoading) return "Loading...";
  if (error) return
<pre>{error.message}</pre>;

  return (
    <div>
      <h1>Xspace Launches</h1>
      <ul>
        {data.launchesPast.map((launch) => (
          <li
key={launch.id}>{launch.mission_name}</li>
        ))}
      </ul>
    </div>
  );
}
```

You'll need to provide a little more details with Axios regarding how to fulfil this commitment and retrieve your data. For the data to appear on the data property that useQuery returns, you must tell React Query where the data is located.

Specifically, you receive the data back on the property of response.data International Journal of Innovative Science and Research Technology SEM VI, 3rd year, October- 2023

- *React Query + Fetch API :*

Using React query in conjunction with the fetch API is the most straightforward method of data retrieval among all of these approaches.

You just need to install react-query within your application; no thirdparty libraries need to be installed because the fetch API is built into all current browsers.

```
npm install react-query
```

You can replace your axios code with fetch once the entire application has the React Query client. The only slight variation is that you must include the content type of the data you want returned from your request in a header that you specify. It's JSON data in this instance. Additionally, you must stringify the object you're sending as payload by setting its query property to your movie's query:

```
import React from "react";
import axios from "axios";
import { useQuery } from "react-query";

const endpoint =
"https://api.spacex.land/graphql/";
const FILMS_QUERY = `
  {
    launchesPast(limit: 10) {
      id
      mission_name
    }
  }
`;

export default function App() {
  const { data, isLoading, error } =
useQuery("launches", () => {
    return fetch(endpoint, {
      method: "POST",
      headers: { "Content-Type":
"application/json" },
      body: JSON.stringify({ query:
FILMS_QUERY })
    })
      .then((response) => {
        if (response.status >= 400) {
          throw new Error("Error fetching
data");
        } else {
          return response.json();
        }
      })
      .then((data) => data.data);
  });

  if (isLoading) return "Loading...";
  if (error) return
<pre>{error.message}</pre>;return(
      <h1>SpaceX Launches</h1>
      <ul>
        {data.launchesPast.map((launch) => (
          <li
key={launch.id}>{launch.mission_name}</li>
        ))}
      </ul>
    </div>
  );}
```

The fact that axios handles errors automatically is one advantage over fetch. As the code above illustrates, in order to use fetch, you must look for a specific status code—more specifically, one that is greater than 400. This indicates that there is a problem with your request. If so, you will need to manually throw an error, which your useQuery hook will handle. Otherwise, just return the JSON data and display it if the response falls between the 200 and 300 range, indicating that the request was successful.

IV. VULNERABILITY FINDERS

➢ *Scanning a GraphQL API for Vulnerabilities :*

GraphQL-based APIs can be scanned using tools such as Invicti, ManageEngine, PortSwigger, graphql.security, etc., which make use of the web application's current security checks.

➢ *GraphQL Security Escape*

Escape builds its products with developers in mind, and the same is true of its security checker for GraphQL. You can be sure that the brand-new vulnerability will be quickly scanned because we are one of the very few suppliers of security services.

➢ *There's More, Though:*

The initial scan takes roughly 60 seconds to begin. Vulnerabilities have been updated in Escape's database. demonstrates actual risks as opposed to potential risks.

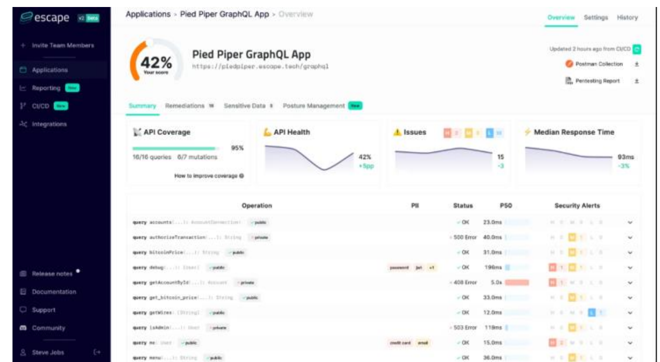

Fig 3 Escape Security Response

➢ *Inviciti GraphQL Scanner*

Inviciti is one of the most trusted and popular names among the scanning APIs. But what a customer wants to know is how many types of attacks it can take care of, so here's a list of severe attacks and vulnerabilities that can be scanned with this product:

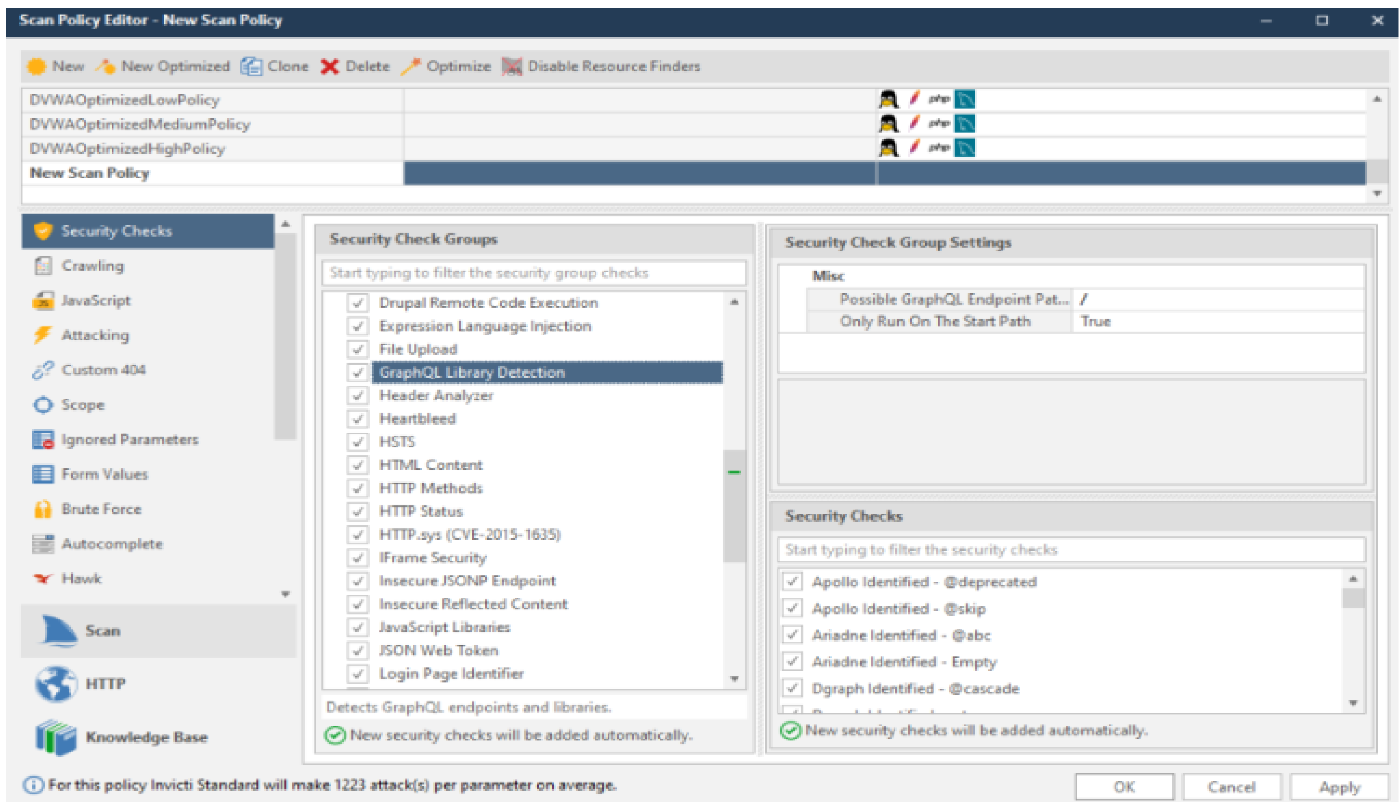

Fig 4 Invincti Tool Response

➢ *StackHawk GraphQL Security Testing*

The best part of using StackHawk's GraphQL testing is it checks for all the GraphQL vulnerabilities at every pull request.

And if that key feature is not enough to win your heart, here are more exciting features from StackHawk:

Automated security testing. Lightning-fast testing and fixing. Easy UI. Magnificent documentation for easy self-fixing

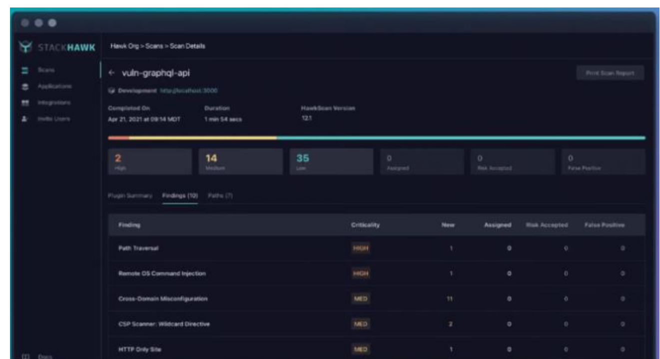

Fig 5 StackHawk Response

➢ *Qualysec GraphQL API Penetration Testing*

Qualysec provides professional GraphQL API Penetration Testing and is a cybersecurity assessment service, so you can uncover vulnerabilities and fix them and be assured of all security issues. And here are some interesting features that they provide:

Product analyzed for the OWASP Top 10 GraphQL API Testing to get protected against the most common threats. Dynamic API testing. Static API testing. Software composition analysis. Apart from security features, their report for vulnerability scan is outstanding as it includes a penetration report, retest report, Letter of attestation, and Security certificate.

➢ *Bright Security API Testing*

Bright security services are designed for modern microservice environments and provide seamless integration with SDLC, CI/CD, and git workflows so the vulnerabilities can be detected as easily as possible. And here are some key features of Bright security: Convenient CLI for developers. 100% SaaS-based. CI/CD Integration. Vulnerabilities mapped to OWASP API Security Top 10.

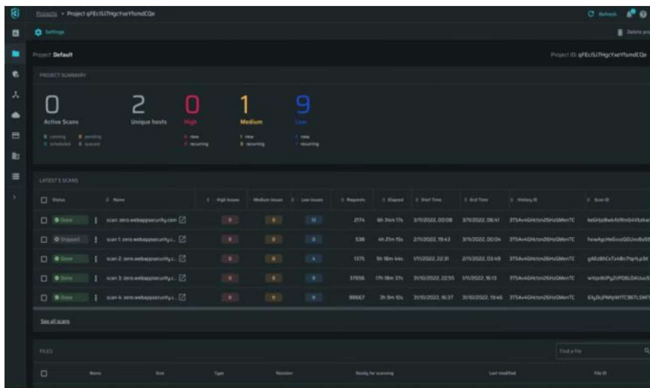

Fig 6 Bright Security Response

There are other tools as well. But these 4 are market leaders.

V. FIXING VULNERABILITY

Because React is so simple to use, scale, and maintain, it has revolutionised the web development ecosystem. Unidentified assets, which can be a part of a library or a third-party integration, can raise the likelihood of vulnerabilities existing, so a stable codebase is not always a secure codebase. Furthermore, every new library and update release may raise the possibility of introducing new vulnerabilities that are not immediately noticeable.

➢ *Cross-Site Scripting (XSS)*

- *CWE-79:*

One of the most prevalent vulnerabilities on the internet, cross-site scripting (XSS) has been listed in the OWASP top 10 for a number of years. When a hacker inserts malicious client-side scripts into online applications, XSS occurs. These scripts will probably run as valid code, giving the attacker complete control over the programme. XSS attacks come in a variety of forms, including DOM-based, stored, and reflective attacks.

Developers should be aware that React packages, like graphqlplayground-react, Semantic-UI, and React-DOM, may be vulnerable to XSS attacks if the code has not been developed securely.

Best Practices for Preventing Cross-Site Scripting (XSS) in React

✓ Recognise how data is auto-escaped by JSX and React createElement before it is rendered.
✓ Recognise why dangerouslySetInnerHTML is named that way, and try to avoid it at all costs. Make sure you are escaping anything you are passing to it if you do need to use it.
✓ Use the "DOMPurify" library to sanitise data before rendering it in DOM
✓ Use blacklists and whitelists for validation testing

Server-Side Rendering Attacks in React

Server-side rendering is a key feature in React that helps developers improve performance.

This is because, in order to speed up the page loading process, HTML is rendered in the back end and sent to the front end rather than being generated by JavaScript on the client-side. Search engine optimisation (SEO) is also aided by it.

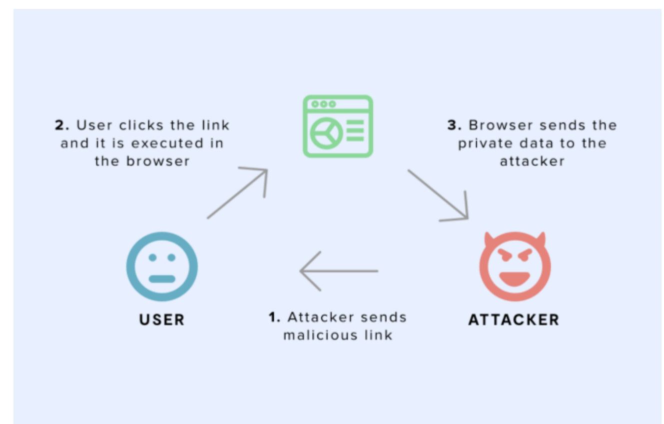

Fig 7 SSR Attacks

Make sure you initialise the state from the server-side if you are using Redux for state management. "Inject Initial Component HTML and State" is one of these.

For instance, the script's JSON.stringify function won't look for potentially harmful inputs. Thus, it is possible to insert malicious JavaScript code that modifies valid data.

window.__STATE__ = ${JSON.stringify({ username: "AYX", Letter: "

✓ JavaScriptConduct data sanitization before rendering in DOM with the use of the “DOMPurify” library -Use a module that will avoid serialization, such as Serialize

➢ *SQL Injections in React*

Many web applications have SQL injection vulnerabilities, and React is no exception. SQL vulnerabilities are a means by which attackers can get around user permission and ultimately compromise databases. This is a serious concern because there could be significant financial losses and compliance issues in the event of a database breach involving sensitive or personal data.

Error-based, logic-based, and time-based injections can be common in React applications. This is primarily because the principles of least privilege are not followed, or there may be a coding bug that prevents user input from being filtered.

Fig 8 SQL Injection Attack

Best Practices for Preventing SQL Injections in React

- ✓ In order to guard against SQL injection, all user inputs must pass through a stringent whitelist filter, which guarantees that all inputs are carefully examined before being processed.
- ✓ Applying the least privilege principle by granting each account the minimal amount of privileges. For example, a website should only be granted the ability to extract content using SELECT statements; it should not be granted access to other privileges such as UPDATE, INSERT, or DELETE.
- ✓ It is possible to find and fix security flaws in your React applications before hackers take advantage of them by routinely scanning them with vulnerability scanners like Acunetix. Also checking that every API function complies with its corresponding API schema, especially to prevent time-based SQL injection.

➢ *Broken Authentication*

One serious security flaw that can impact all web apps, including React apps, is broken authentication. Hackers can easily circumvent or compromise the authentication solutions implemented in the app by taking advantage of poorly implemented session management functions and authentication processes.

Fig 9 Broken Auth

- *How to avoid broken authentication in React?*

- ✓ Whenever feasible, using multi-factor authentication
- ✓ Enforcing password strength checks
- ✓ Employing NIST 800-63 B recommendations for the length and complexity of passwords;
- ✓ Using uniform messages for all outcomes related to authentication
- ✓ Generating a fresh session ID for each user login via a secure, server-side session manager.
- ✓ To keep the app safe, it's also essential to store session IDs safely and invalidate them at the end of the session.

➢ *Zip Slip*

A security flaw in React apps that allows users to upload zip files is called zip slip. This feature can be turned on by web developers to minimise file sizes during uploading. After that, the app decompresses these files in order to extract the original zip files. Hackers can use zip slip, which is essentially a directory traversal, to extract files, usually from archives.

A file system's components may occasionally stay outside of the folder for which they are intended. The assailant might

- ✓ Obtain entry to these file sections
- ✓ Replace them
- ✓ Call these files from a distance or have the system do so.

They are able to execute commands remotely on the user's device in this manner.

You have to be extra cautious about this React security vulnerability. It can lead to overwriting of sensitive resources like configuration files. What’s worse? The attacker can exploit this not only on the client-side but also on the server-side.

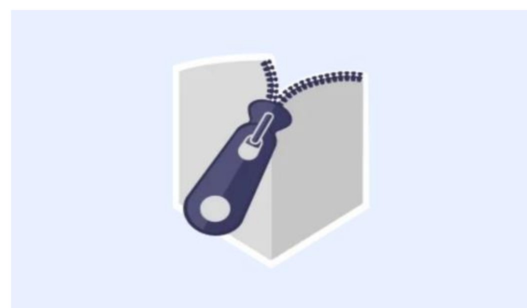

Fig 10 Zip Slip Attack

- *How can I prevent zip slippage in React?*

- ✓ Making sure that no malicious file gets into the application is the only way to avoid this security hazard.
- ✓ Making certain that the file names are standard - Prohibiting the use of special characters in file names
- ✓ Constantly matching and comparing the names with standard, regular expressions

- ✓ Generating new names for each uploaded file and renaming them all before the app uses or stores them in a zip file.

- ➢ *Implementing JWT Authentication in React Applications*

Web applications frequently employ SON Web Tokens (JWT) for authentication. An example of how to incorporate JWT authentication into a React application that communicates with a GraphQL API can be found here:

In the below example, the React component captures the username and password from the user, sends a request to the /api/login endpoint, and receives a JWT token in response. The token is then stored in the local storage for future API requests.

```
// React component handling user login
import { useState } from 'react';
import axios from 'axios';

const Login = () => {
  const [username, setUsername] = useState('');
  const [password, setPassword] = useState('');

  const handleLogin = async () => {
    try {
      const response = await axios.post('/api/login', { username,
password });
      const { token } = response.data;
      // Save token to local storage or cookies for future API requests
      localStorage.setItem('token', token);
    } catch (error) {
      console.error('Login failed:', error);
    }
  };

  return (
    <div>
      <input
        type="text"
        placeholder="Username"
        value={username}
        onChange={(e) => setUsername(e.target.value)}
      />
      <input
        type="password"
        placeholder="Password"
        value={password}
        onChange={(e) => setPassword(e.target.value)}
      />
      <button onClick={handleLogin}>Login</button>
    </div>
  );
};
```

- ➢ *Securing GraphQL Resolvers with Authorization Middleware*

It is imperative to secure GraphQL resolvers in order to manage access to particular information or features. Here's an example of how authorization middleware is implemented using Apollo Server in a GraphQL resolver:

```
// GraphQL resolver with authorization middleware
import { ForbiddenError } from 'apollo-server';

const resolvers = {
  Query: {
    // Publicly accessible query
    publicData: () => {
      // Return public data
    },
    // Restricted query
    privateData: (_, __, context) => {
      // Check if user is authenticated
      if (!context.user) {
        throw new ForbiddenError('Not authorized');
      }
      // Return private data
    },
  },
};
```

The privateData resolver in the aforementioned example accesses the context object to determine whether the user is authenticated. A ForbiddenError is raised, preventing access to the private data, if the user is not authenticated.

- ➢ *Protecting Sensitive Data in GraphQL Queries and Mutations*

Protecting sensitive data in GraphQL queries and mutations involves input validation and sanitization. Here's an example of implementing input validation in a GraphQL resolver using a library like joi:

```
// Input validation using joi
import joi from 'joi';
const schema = joi.object({
  email: joi.string().email().required(),
  password: joi.string().min(8).required(),
});

const resolvers = {
  Mutation: {
    registerUser: async (_, { input }) => {
      const { error, value } = schema.validate(input);
      if (error) {
        throw new Error('Invalid input');
      }
      // Process registration
    },
  },
};
```

In the example above, the registerUser mutation validates the input fields (email and password) using joi. If the input is invalid, an error is thrown, preventing the registration process.

➢ *Role-Based Access Control in GraphQL APIs*

Role-based access control (RBAC) allows different levels of access based on user roles. Here's an example of implementing RBAC in a GraphQL API using middleware:

```
// Express middleware for role-based access control
const checkRole = (requiredRole) => (req, res, next) => {
  if (req.user.role !== requiredRole) {
    return res.status(403).json({ error: 'Not authorized' });
  }
  next();
};
const resolvers = {
  Mutation: {
    deleteUserData: checkRole('admin')(async (_, { userId }) => {
      // Delete user data
    }),
    updateProfile: checkRole('user')(async (_, { name }, context) => {
      // Update user profile
    }),
  },
};
```

VI. CONCLUSION

By implementing authentication and authorization mechanisms, utilizing JWT authentication, XXS Fix, Broken Auth fix, Zip Slip Fix and SQL Injection fix to Securing GraphQL APIs in React applications is essential to protect against security threats., incorporating role-based access control, and securing GraphQL resolvers, you can build secure and robust React applications that handle sensitive data with confidence. By following the best practices outlined in this guide, you can ensure the security of your React application and GraphQL API.

REFERENCES

[1]. F. Inc, "Introduction to GraphQL," 2018. [Online].
[2]. Intro to React, 2022 React Legacy [Online]
[3]. Helgason, Arnar Freyr. "Performance analysis of Web Services: Comparison between RESTful & GraphQL web services." (2017).
[4]. Reddy, Ch Ram Mohan, and RV Raghavendra Rao. "QoS of Web service: Survey on performance and scalability." Computer Science and Information Technology 3.9 (2013): 65-73.
[5]. M. Host, B. Regnell, and C. Wohlin, "Using students as subjects—a ¨comparative study of students and professionals in lead-time impact assessment," Empirical Software Engineering, vol. 5, no. 3, pp. 201–214, 2000.
[6]. B. Ray, D. Posnett, V. Filkov, and P. Devanbu, "A large scale study of programming languages and code quality in GitHub," in 22nd International Symposium on Foundations of Software Engineering (FSE), 2014, pp. 155–165.
[7]. Facebook Inc., "GraphQL specification (draft)," https://facebook.github.io/graphql/draft/, 2015, [accessed 15- October-2018].