GraphQL API vulnerabilities

GraphQL vulnerabilities generally arise due to implementation and design flaws.

GraphQL attacks usually take the form of malicious requests that can enable an attacker to obtain data or perform unauthorized actions. These attacks can have a severe impact, especially if the user is able to gain admin privileges by manipulating queries or executing a CSRF exploit

Finding GraphQL endpoints

GraphQL APIs use the same endpoint for all requests

Universal queries

If you send `query{__typename}` to any GraphQL endpoint, it will include the string `{"data": {"__typename": "query"}}` somewhere in its response. This is known as a universal query, and is a useful tool in probing whether a URL corresponds to a GraphQL service.

Common endpoint names

GraphQL services often use similar endpoint suffixes. When testing for GraphQL endpoints, you should look to send universal queries to the following locations

- `/graphql`
- `/api`
- `/api/graphql`
- `/graphql/api`
- `/graphql/graphql`

If these common endpoints don't return a GraphQL response, you could also try appending `/v1` to the path

Request methods

The next step in trying to find GraphQL endpoints is to test using different request methods.

It is best practice for production GraphQL endpoints to only accept POST requests that have a content-type of `application/json`, as this helps to protect against CSRF vulnerabilities. However, some endpoints may accept alternative methods, such as GET requests or POST requests that use a content-type of `x-www-form-urlencoded`.

Exploiting unsanitized arguments

```graphql
#the query below requests a product list for an online shop
query {
    products {
        id
        name
        listed
    }
}
#product list returned contains only listed products.
{
    "data": {
        "products": [
            {
                "id": 1,
                "name": "Product 1",
                "listed": true
            },
            {
                "id": 2,
                "name": "Product 2",
                "listed": true
            },
            {
                "id": 4,
                "name": "Product 4",
                "listed": true
            }
        ]
    }
}
```

Product ID 3 is missing from the list, possibly because it has been delisted, By querying the ID of the missing product, we can get its details, even though it is not listed on the shop and was not returned by the original product query.

```graphql
query {
    product(id: 3) {
        id
        name
        listed
    }
}
#Missing product response
    {
        "data": {
            "product": {
            "id": 3,
            "name": "Product 3",
            "listed": no
```

```
            }
        }
    }
```

Discovering schema information

The best way to do this is to use introspection queries. Introspection is a built-in GraphQL function that enables you to query a server for information about the schema.

Introspection helps you to understand how you can interact with a GraphQL API. It can also disclose potentially sensitive data, such as description fields.

Introspection

To use introspection to discover schema information, query the `__schema` field. This field is available on the root type of all queries.

If introspection is enabled, the response returns the names of all available queries.

```graphql
#Introspection probe request
{
    "query": "{__schema{queryType{name}}}"
}
```

Running a full introspection query

The next step is to run a full introspection query against the endpoint so that you can get as much information on the underlying schema as possible

```graphql
query IntrospectionQuery {
        __schema {
            queryType {
                name
            }
            mutationType {
                name
            }
            subscriptionType {
                name
            }
            types {
             ...FullType
            }
            directives {
                name
                description
                args {
                    ...InputValue
            }
            onOperation  #Often needs to be deleted to run query
            onFragment   #Often needs to be deleted to run query
```

```
            onField      #Often needs to be deleted to run query
            }
        }
    }

    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
                }
            }
        }
    }
```

Visualizing introspection results

Responses to introspection queries can be full of information, but are often very long and hard to process

You can view relationships between schema entities more easily using a GraphQL visualizer. This is an online tool that takes the results of an introspection query and produces a visual representation of the returned data, including the relationships between operations and types.

Disabled Introspection

if introspection is entirely disabled, you can sometimes use suggestions to glean information on an API's structure

Clairvoyance is a tool that uses suggestions to automatically recover all or part of a GraphQL schema, even when introspection is disabled. This makes it significantly less time consuming to piece together information from suggestion responses

Bypassing GraphQL introspection defenses

If you cannot get introspection queries to run for the API you are testing, try inserting a special character after the `__schema` keyword.

When developers disable introspection, they could use a regex to exclude the `__schema` keyword in queries. You should try characters like spaces, new lines and commas, as they are ignored by GraphQL but not by flawed regex

```graphql
{
    "query": "query{__schema
    {queryType{name}}}"
}
```

If this doesn't work, try running the probe over an alternative request method, as introspection may only be disabled over POST. Try a GET request, or a POST request with a content-type of `x-www-form-urlencoded`.

```http
GET /graphql?query=query%7B__schema%0A%7BqueryType%7Bname%7D%7D%7D
GET /api?query=query+IntrospectionQuery+%7B%0A++__schema+%7B%0A++++queryType+%7B%0D%0A+
+++++name%0D%0A++++%7D%0D%0A++++mutationType+%7B%0D%0A++++++name%0D%0A++++%7D%0D%0A++++
subscriptionType+%7B%0D%0A++++++name%0D%0A++++%7D%0D%0A++++types+%7B%0D%0A++++++...Full
Type%0D%0A++++%7D%0D%0A++++directives+%7B%0D%0A++++++name%0D%0A++++++description%0D%0A+
+++++args+%7B%0D%0A++++++++...InputValue%0D%0A++++++%7D%0D%0A++++%7D%0D%0A++%7D%0D%0A%7
D%0D%0A%0D%0Afragment+FullType+on+__Type+%7B%0D%0A++kind%0D%0A++name%0D%0A++descriptio
n%0D%0A++fields%28includeDeprecated%3A+true%29+%7B%0D%0A++++name%0D%0A++++description%0
D%0A++++args+%7B%0D%0A++++++...InputValue%0D%0A++++%7D%0D%0A++++type+%7B%0D%0A++++++...
TypeRef%0D%0A++++%7D%0D%0A++++isDeprecated%0D%0A++++deprecationReason%0D%0A++%7D%0D%0A+
+inputFields+%7B%0D%0A++++...InputValue%0D%0A++%7D%0D%0A++interfaces+%7B%0D%0A++++...Ty
peRef%0D%0A++%7D%0D%0A++enumValues%28includeDeprecated%3A+true%29+%7B%0D%0A++++name%0D%
0A++++description%0D%0A++++isDeprecated%0D%0A++++deprecationReason%0D%0A++%7D%0D%0A++po
ssibleTypes+%7B%0D%0A++++...TypeRef%0D%0A++%7D%0D%0A%7D%0D%0A%0D%0Afragment+InputValue+
on+__InputValue+%7B%0D%0A++name%0D%0A++description%0D%0A++type+%7B%0D%0A++++...TypeRef%
0D%0A++%7D%0D%0A++defaultValue%0D%0A%7D%0D%0A%0D%0Afragment+TypeRef+on+__Type+%7B%0D%0A
++kind%0D%0A++name%0D%0A++ofType+%7B%0D%0A++++kind%0D%0A++++name%0D%0A++++ofType+%7B%0
D%0A++++++kind%0D%0A++++++name%0D%0A++++++ofType+%7B%0D%0A++++++++kind%0D%0A++++++++nam
e%0D%0A++++++%7D%0D%0A++++%7D%0D%0A++%7D%0D%0A%7D%0D%0A
```

Bypassing rate limiting using aliases

Ordinarily, GraphQL objects can't contain multiple properties with the same name. Aliases enable you to bypass this restriction by explicitly naming the properties you want the API to return. You can use aliases to return multiple instances of the same type of object in one request.

While aliases are intended to limit the number of API calls you need to make, they can also be used to brute force a GraphQL endpoint.

Many endpoints will have some sort of rate limiter in place to prevent brute force attacks. Some rate limiters work based on the number of HTTP requests received rather than the number of operations performed on the endpoint. Because aliases effectively enable you to send multiple queries in a single HTTP message, they can bypass this restriction

```graphql
#Request with aliased queries

query isValidDiscount($code: Int) {
    isvalidDiscount(code:$code){
        valid
    }
    isValidDiscount2:isValidDiscount(code:$code){
        valid
    }
    isValidDiscount3:isValidDiscount(code:$code){
        valid
    }
}
```

GraphQL CSRF

GraphQL can be used as a vector for CSRF attacks, whereby an attacker creates an exploit that causes a victim's browser to send a malicious query as the victim user.

How do CSRF over GraphQL vulnerabilities arise?

CSRF vulnerabilities can arise where a GraphQL endpoint does not validate the content type of the requests sent to it and no CSRF tokens are implemented

POST requests that use a content type of `application/json` are secure against forgery as long as the content type is validated. In this case, an attacker wouldn't be able to make the victim's browser send this request even if the victim were to visit a malicious site

However, alternative methods such as GET, or any request that has a content type of `x-www-form-urlencoded`, can be sent by a browser and so may leave users vulnerable to attack if the endpoint accepts these requests. Where this is the case, attackers may be able to craft exploits to send malicious requests to the API.

Preventing GraphQL attacks

- If your API is not intended for use by the general public, disable introspection on it. This makes it harder for an attacker to gain information about how the API works, and reduces the risk of unwanted information disclosure.

  For information on how to disable introspection in the Apollo GraphQL platform, see this blog post.

- If your API is intended for use by the general public then you will likely need to leave introspection enabled. However, you should review the API's schema to make sure that it does not expose unintended fields to the public.

- Make sure that suggestions are disabled. This prevents attackers from being able to use Clairvoyance or similar tools to glean information about the underlying schema.

  You cannot disable suggestions directly in Apollo. See this GitHub thread for a workaround.

- Make sure that your API's schema does not expose any private user fields, such as email addresses or user IDs.

Preventing GraphQL brute force attacks

It is sometimes possible to bypass standard rate limiting when using GraphQL APIs. For an example of this, see the Bypassing rate limiting using aliases section.

With this in mind, there are design steps that you can take to defend your API against brute force attacks. This generally involves restricting the complexity of queries accepted by the API, and reducing the opportunity for attackers to execute denial-of-service (DoS) attacks.

To defend against brute force attacks:

- Limit the query depth of your API's queries. The term "query depth" refers to the number of levels of nesting within a query. Heavily-nested queries can have significant performance implications, and can potentially provide an opportunity for DoS attacks if they are accepted. By limiting the query depth your API accepts, you can reduce the chances of this happening.
- Configure operation limits. Operation limits enable you to configure the maximum number of unique fields, aliases, and root fields that your API can accept.
- Configure the maximum amount of bytes a query can contain.
- Consider implementing cost analysis on your API. Cost analysis is a process whereby a library application identifies the resource cost associated with running queries as they are received. If a query would be too computationally complex to run, the API drops it.

Preventing CSRF over GraphQL

To defend against GraphQL CSRF vulnerabilities specifically, make sure of the following when designing your API:

- Your API only accepts queries over JSON-encoded POST.
- The API validates that content provided matches the supplied content type.
- The API has a secure CSRF token mechanism.