4
RECONNAISSANCE

All security tests start with a *reconnaissance phase*. In this phase, we attempt to collect as much information as possible about our target. This information will prepare us to make informed decisions about how to attack the application and increase our chances of success.

You might be asking yourself, what is there to know about GraphQL, seeing as it's just an API layer? You'll learn that we can gather a lot of information, through experimentation and the use of specialized tooling, about the application running behind a GraphQL API, as well as the GraphQL implementation itself. While the GraphQL query structure is consistent across all GraphQL implementations, irrespective of the programming language they are written in, you will likely see differences in the available operations, fields, arguments, directives, security controls, responses to specially crafted queries, and so on.

Here are a few key questions we should strive to answer during our reconnaissance phase: Does the web server even have a GraphQL API? On which endpoint is GraphQL configured to receive queries? What language is the GraphQL implementation written in? What implementation of GraphQL is running on the target server? Is the implementation known to be vulnerable to certain attacks? What types of defenses exist for the specific GraphQL implementation? What are some of the out-of-the-box default configuration settings of this implementation? Does the GraphQL server have any additional security protection layers in place? Being able to answer these questions will allow us to plan a more focused attack against our target server and uncover gaps in its defense.

NOTE *Throughout this chapter, as well as the following ones, we will use the DVGA as our target vulnerable application. You should already have it running as part of the GraphQL security lab we built in Chapter 2.*

Detecting GraphQL

To detect GraphQL in a penetration test, it's important to first familiarize yourself with the GraphQL server implementations that exist in the wild today. GraphQL has many implementations written in a variety of programming languages, each of which could have different default configurations or known weaknesses. Table 4-1 lists several GraphQL implementations and the languages in which they are written.

Table 4-1: GraphQL Server Implementations and Their Programming Languages

Server implementation	Language
Apollo	TypeScript
Graphene	Python
Yoga	TypeScript
Ariadne	Python
graphql-ruby	Ruby
graphql-php	PHP
graphql-go	Go
graphql-java	Java
Sangria	Scala
Juniper	Rust
HyperGraphQL	Java
Strawberry	Python
Tartiflette	Python

These are some of the most popular implementations in use today, as well as more niche implementations, such as Sangria for Scala, Juniper for Rust, and HyperGraphQL for Java. Later in this chapter, we will discuss how to distinguish between them during a penetration test.

Detection of GraphQL APIs can be done in several ways: either manually, which is typically harder to scale if you have more than a few hosts on a network, or automatically, using various web scanners. The advantage of using web-scanning tools is that they are scalable. They are threaded, and often have the ability to read external files as program input, such as text files with a list of hostnames to scan. These tools already have the logic to detect web interfaces built into them, and using scripting languages (such as Bash or Python), you can programmatically run them against hundreds of IP addresses or subdomains. In this chapter, we will use popular scanners such as Nmap, as well as GraphQL-oriented scanning tools, such as Graphw00f, for reconnaissance.

Common Endpoints

In Chapter 1, we highlighted some of the differences between REST and GraphQL APIs. One of these differences, relevant to the reconnaissance phase, is that a GraphQL API endpoint is typically static, and most commonly */graphql*.

However, although */graphql* is often the default GraphQL endpoint, the GraphQL implementation can be reconfigured to use a completely different path. In those cases, what can we do to detect it? One way is to manually attempt a few common alternative paths to the GraphQL API, such as versioned endpoints:

/v1/graphql

/v2/graphql

/v3/graphql

You'll typically see these versioned API endpoints when the application needs to support multiple versions of its API, either for backward compatibility or for the introduction of a new feature in a way that doesn't conflict with the stable API version that customers might still be using.

Another way to find a GraphQL implementation is through IDEs, such as GraphQL Playground or GraphiQL Explorer, which we used in Chapter 1 to experiment with GraphQL queries. When either of these interfaces is enabled, it often uses an additional, dedicated endpoint. This means GraphQL can potentially exist under the following endpoints as well:

/graphiql

/playground

If these endpoints happen to also be versioned, they may have a version number prepended to their path, such as */v1/graphiql*, */v2/graphiql*, */v1/playground*, */v2/playground*, and so on.

Listing 4-1 shows how Graphene, a Python-based implementation of GraphQL, can expose two endpoints, one for GraphQL, and the other for GraphiQL Explorer, which is built into Graphene:

```
app.add_url_rule('/graphql', view_func=GraphQLView.as_view(
  'graphql',
  schema=schema
))

app.add_url_rule('/graphiql', view_func=GraphQLView.as_view(
  'graphiql',
  schema = schema,
  graphiql = True
))
```

Listing 4-1: Graphene's endpoint definition

Graphene defines the */graphql* endpoint as its main GraphQL query endpoint. It then defines */graphiql* as a second endpoint that GraphiQL Explorer will query against. Lastly, it enables the GraphiQL Explorer interface. The GraphQL server will render the IDE to the client when it browses to the */graphiql* endpoint.

Keep in mind that each endpoint could have different security settings. One could be stricter than the other, for example. When you find two endpoints serving GraphQL queries on the same target host, you will want to test them separately.

NOTE *In this book's GitHub repository, you can find a more comprehensive list of common GraphQL endpoints:* https://github.com/dolevf/Black-Hat-GraphQL/blob/master/ch04/common-graphql-endpoints.txt. *You can use this as a wordlist file when you need to scan for GraphQL servers during a penetration test or a bug bounty hunt.*

The most important takeaway here is that, while the GraphQL endpoint is typically located at a predictable path, the developer can still customize it to fit their needs, perhaps in an attempt to hide it from curious eyes or to simply conform to internal application deployment standards.

Common Responses

Now that you have an idea of the endpoints from which GraphQL typically receives queries, the next step is to learn how GraphQL APIs respond to packets. GraphQL is fairly easy to identify on a network. This is particularly helpful whenever you are performing a zero-knowledge penetration test or bug bounty hunt.

The GraphQL specification describes how a query response structure should be formatted. This allows API consumers to expect a predetermined format when they parse the GraphQL response. The following excerpt from the GraphQL specification describes how the response to a query should look:

> If the operation is a query, the result of the operation is the result of executing the operation's top-level selection set with the query root operation type.

An initial value may be provided when executing a query operation:

```
ExecuteQuery(query, schema, variableValues, initialValue)
```

1. Let `queryType` be the root `Query` type in the schema.
2. Assert: `queryType` is an `Object` type.
3. Let `selectionSet` be the top-level selection set in the query.
4. Let `data` be the result of running `ExecuteSelectionSet(selectionSet, queryType, initialValue, variableValues)` normally (allowing parallelization).
5. Let `errors` be any field errors produced while executing the selection set.
6. Return an unordered map containing data and errors.

In practice, this means a GraphQL API will return a `data` JSON field when there is a result to return to a client's query. It will also return an `errors` JSON field whenever errors occur during the execution of a client query.

Knowing these two pieces of information ahead of time is valuable. To put it simply, we now have two conditions that a response must meet before we can say that it came from a GraphQL API:

1. A valid query response should *always* have the `data` field populated with query response information.
2. An invalid query response should *always* have the `errors` field populated with information about what went wrong.

Now we can leverage these as part of scanning and detection logic to automate the discovery of GraphQL servers on a network. All we need to do is send a valid or malformed query and observe the response we receive.

Let's run a simple GraphQL query using the HTTP POST method against the DVGA to see these response structures in action. Open the Altair GraphQL client and ensure that the address bar has the *http://localhost:5013/graphql* address set; then run the following query by entering it in Altair's left pane:

```
query {
  pastes {
    id
  }
}
```

Next, click the play button to send the query to the GraphQL server. This should return the `id` field of the `pastes` object. You should be able to see a response similar to the following output:

```
"data": {
    "pastes": [
     {
        "id": "1"
      }
    ]
  }
```

As you can see, GraphQL returns the query response as part of the `data` JSON field, exactly as described in the GraphQL specification. We get the `pastes` object and the `id` field we specified in the query. Don't worry if you see a different `id` string returned in your lab than the one shown here; this is expected.

Now, let's run another query to explore what happens when an invalid query is sent to GraphQL. This will demonstrate that the `errors` JSON field is returned by the GraphQL server when it encounters issues during query execution. The following query is malformed, and GraphQL won't be able to process it. Run it in Altair and observe the response:

```
query {
  badfield {
    id
  }
}
```

Notice that we specify a top-level field with the name of `badfield`. Because this field does not exist, the GraphQL server can't fulfill the query. The GraphQL response can be seen here:

```
{
   "errors": [
    {
       "message": "Cannot query field \"badfield\" on type \"Query\".",
      "locations": [
        {
          "line": 2,
          "column": 3
        }
      ]
    },
  ]
}
```

As you can see, the GraphQL server isn't able to process our query successfully. It returns a response containing the `errors` JSON field. The `message` JSON field indicates to us that the server couldn't query the field named `badfield`, because it does not exist in the GraphQL schema.

Nmap Scans

Imagine that you need to conduct a penetration test against a network containing thousands of hosts; it would be fairly difficult to manually go through each host to find ones that are potentially serving interesting content, such as an API or a vulnerable commercial application. In these cases, penetration testers often use web application scanners or custom scripts to automatically grab information from the hosts. For example, information such as the `<title>` HyperText Markup Language (HTML) tag, the entire `<body>` tag, and even the `server` HTTP response header could all hint at specific applications that the remote server is running.

It's important to note that web applications may not always have a user interface, meaning they may not serve any HTML content related to the application or even expose HTTP headers by which we can detect them. They will often act as standalone API servers that expose data only through designated APIs. So, how can we detect GraphQL in those cases? Luckily, GraphQL APIs often return predictable responses under certain conditions, such as the HTTP method in use or the payload sent to the server. Listing 4-2 shows a common GraphQL response returned when a client makes a GET request.

```
# curl -X GET http://localhost:5013/graphql

{"errors":[{"message":"Must provide query string."}]}
```

Listing 4-2: A GraphQL response to an HTTP GET request

The string `Must provide query string` is often used in GraphQL implementations, such as Python- and Node.js-based ones. (Keep in mind that GET-based queries are often not supported by GraphQL servers. Rest assured: we have many other ways of detecting GraphQL should we run into such a situation.)

With this information, we now have the ability to automate a scan and pick up any other GraphQL servers that may exist on a network. Listing 4-3 shows how to do this with Nmap, using the *http-grep* NSE script, which uses pattern matching to look for keywords in a given web page.

```
# nmap -p 5013 -sV --script=http-grep
--script-args='match="Must provide query string", ❶ http-grep.url="/graphql"' localhost ❷

PORT     STATE SERVICE VERSION
5013/tcp open  http    Werkzeug httpd
| http-grep:
|   (1) http://localhost:5013/graphql:
|       (1) User Pattern 1:
|       + Must provide query string
```

Listing 4-3: A GraphQL response to word-matching using Nmap's `http-grep`

At ❶ we specify a script argument to *http-grep* called `match` with a value of `Must provide query string` (the message we received in our GraphQL response). At ❷ we define another argument, called `http-grep.url`, with a value of `/graphql`, which instructs Nmap to search a specific page within the web application. Under the hood, Nmap will make an HTTP GET request to `localhost` and use the argument string value we defined as the pattern for its search within the text it extracts from the web server's response. In its output, Nmap shows that a pattern was found on the web page and indicates the string for which it found a match.

You may have noticed that we're passing a specific port to Nmap (`-p`)—namely, port `5013`. Like any web server, GraphQL servers could run on any port, but a few are quite common, such as 80–89, 443, 4000–4005, 8443, 8000, and 8080. We recommend scanning both common and uncommon port ranges when possible.

The __typename Field

So far, we've known exactly which fields to ask for in our queries, such as `pastes` with a selection set of `id`, as we requested earlier. You might be wondering, what if we don't know what fields exist on the GraphQL API? How can we go about identifying GraphQL without this information? Luckily, there is a quick way to query GraphQL and return a valid response without knowing anything about the application's schema.

Meta-fields are built-in fields that GraphQL APIs expose to clients. One example is `__schema` (part of introspection in GraphQL). Another example of a meta-field is `__typename`. When used, it returns the name of the object type being queried. Listing 4-4 shows a query that uses this meta-field.

```
query {
  pastes {
    __typename
  }
}
```

Listing 4-4: A GraphQL query with the `__typename` meta-field

When you run this query with Altair, the response will be the name of the `pastes` object type:

```
  "data": {
    "pastes": [
      {
        "__typename": "PasteObject"
      }
    ]
  }
```

As you can see, GraphQL tells us that the `pastes` object's type name is `PasteObject`. The real hack here is that the `__typename` meta-field can be used against the query root type as well, as shown in Listing 4-5.

```
query {
  __typename
}
```

Listing 4-5: A GraphQL meta-field used with the query root type

This query uses `__typename` to describe the query root type and will work against pretty much any GraphQL implementation, since `__typename` is part of the official specification.

When you're attempting to query GraphQL from the command line, GraphQL servers expect a certain request structure. For HTTP GET-based queries, a request should have the following HTTP query parameters:

- `query` for the GraphQL query (mandatory parameter).
- `operationName` for the operation name, used when multiple queries are sent in a single document. This parameter tells the GraphQL server which specific operation to run when more than one is present (optional parameter).
- `variables` for query variables (optional parameter).

For HTTP POST-based queries, the same parameters should be passed in the HTTP body in JSON.

When GraphQL servers accept queries using GET, you can pass the `query` parameter along with the GraphQL query (in this case, the query `{__typename}`) by using shorthand syntax. With this in mind, we can automate the detection of GraphQL by using Nmap fairly easily. Listing 4-6 shows how to run a `__typename` query with Nmap.

```
# nmap -p 5013 -sV --script=http-grep --script-args='match="__typename",
http-grep.url="/graphql?query=\{__typename\}"' localhost

PORT     STATE SERVICE VERSION
5013/tcp open  http    Werkzeug httpd
| http-grep:
|   (1) http://localhost:5013/graphql?query=\{__typename\}:
|     (1) User Pattern 1:
|_      + __typename
```

Listing 4-6: Detecting GraphQL by using GET-based queries with Nmap

In this example, the Nmap script *http-grep* uses the GET method under the hood to do its work.

If you have more than a handful of hosts to scan, you may want to leverage Nmap's `-iL` flag to point to a file that contains a list of hostnames, as shown in Listing 4-7.

```
# nmap -p 5013 -iL hosts.txt -sV --script=http-grep
--script-args='match="__typename", http-grep.url="/graphql?query=\{__typename\}"'
```

Listing 4-7: Scanning multiple targets defined in a file with Nmap

The *hosts.txt* file in this example would contain IP addresses or Domain Name System (DNS) hostnames listed on separate lines.

If the GraphQL server does not support GET-based queries, we can use cURL and the `__typename` field to make a POST request to detect GraphQL, as shown in Listing 4-8.

```
# curl -X POST http://localhost:5013/graphql -d '{"query":"{__typename }"}'
-H "Content-Type: application/json"
```

Listing 4-8: Sending a POST-based query using cURL

To use this detection method against a list of hosts, you can use Bash scripting, as shown in Listing 4-9.

```
# for host in $(cat hosts.txt); do
     curl -X POST "$host" -d '{"query":"{__typename }"}' -H "Content-Type: application/json"
done
```

Listing 4-9: A Bash script to automate a POST-based GraphQL detection using cURL

The *hosts.txt* file in this example would contain a list of full target URLs on separate lines (including their protocol schemes, domains, ports, and endpoints).

Graphw00f

In Chapter 2, we briefly discussed Graphw00f, a GraphQL tool based on Python for detecting GraphQL and performing implementation-level fingerprinting. In this section, we will use it to detect DVGA in our lab, walking you through how it does its detection magic.

We mentioned earlier in this chapter that GraphQL servers are found at the endpoint */graphql* by default. When this is not the case, we might need an automated way to iterate through known endpoints in order to figure out where queries are served from. Graphw00f allows you to specify a custom list of endpoints when running a scan. If you don't provide a list, Graphw00f will use its hardcoded list of common endpoints whenever it is tasked with detecting GraphQL, as shown in Listing 4-10.

```
def possible_graphql_paths():
    return [
        '/graphql',
        --snip--
        '/console',
        '/playground',
        '/gql',
        '/query',
        --snip--
    ]
```

Listing 4-10: A list of common GraphQL endpoints in Graphw00f's source code

To see Graphw00f in action, open your terminal and execute the command in Listing 4-11. We use command line parameters `-t` (target) and `-d` (detection). The `-t` flag in this case will be the remote URL *http://localhost:5013*, and the `-d` flag will turn on detection mode, which indicates to Graphw00f that it should run a GraphQL detection check against the target URL. If you have questions about Graphw00f's arguments, use the `-h` flag to read more about its options.

```
# cd ~/graphw00f
# python3 main.py -d -t http://localhost:5013

                    graphw00f
          The fingerprinting tool for GraphQL
```

```
 [*] Checking http://localhost:5013/
 [*] Checking http://localhost:5013/graphql
 [!] Found GraphQL at http://localhost:5013/graphql
```

Listing 4-11: A GraphQL detection with Graphw00f

Run in detect mode, Graphw00f iterates through various web paths. It checks for the existence of GraphQL in the main web root folder and the */graphql* folder. Then it signals to us that it found GraphQL under */graphql* based on the HTTP response heuristics we discussed earlier.

To use your own list of endpoints, you can pass the `-w` (wordlist) flag and point it at a file containing your endpoints, as shown in Listing 4-12.

```
# cat wordlist.txt

/app/graphql
/dev/graphql
/v5/graphql

# python3 main.py -d -t http://localhost:5013 -w wordlist.txt

[*] Checking http://localhost:5013/app/graphql
[*] Checking http://localhost:5013/dev/graphql
[*] Checking http://localhost:5013/v5/graphql
```

Listing 4-12: Using a custom endpoint list with Graphw00f

Detecting GraphiQL Explorer and GraphQL Playground

The GraphiQL Explorer and GraphQL Playground IDEs are built using the JavaScript library React. Yet when performing reconnaissance, we will often rely on tools that are incapable of rendering web pages containing JavaScript, such as command line HTTP clients like cURL or web application scanners like Nikto. In the process, we might miss interesting web interfaces.

In general, you'll find it beneficial to look for any signs of web interfaces available on the network, such as administration, debugging, or configuration panels, all of which are great candidates to hack. These panels tend to be data rich and often become a way to pivot to other networks or to escalate privileges. They also tend to be far less hardened than publicly facing applications. Companies assume that the external space (the internet) is riskier than the internal space (the corporate network). As such, they often have guidelines for securing publicly facing servers and applications via aggressive patching policies, configuration reviews, and frequent vulnerability scanning. Unfortunately, internal applications rarely get the same treatment, which often makes them an easier target for hackers.

An interesting and often overlooked technique to scan for graphical web interfaces is through the use of tools such as headless browsers. *Headless browsers* are fully functional command line web browsers that the user can program for a variety of purposes, such as retrieving page contents, submitting forms, or simulating real user behavior on a web page. For example, the headless browsers Selenium and PhantomJS can be handy when you need to render web pages containing JavaScript code.

One security tool in particular has incorporated a headless browser to solve this gap: *EyeWitness*. This web scanner is capable of taking screenshots of web pages by leveraging the Selenium headless browser driver engine behind the scenes. EyeWitness then generates a nice report, along with a screen capture of the page.

Scanning for Graphical Interfaces with EyeWitness

Since the two GraphQL IDEs use JavaScript code, we need a capable scanner to help us identify them when we perform network-wide scans. Let's use EyeWitness to identify these graphical interfaces.

EyeWitness offers many options for customizing its scanner behavior, and you can see them by running the tool with the `-h` option. To detect GraphQL IDE panels, we'll use the `--web` option, which will attempt a screen capture of the scanned site with the headless browser engine, together with the `--single` option, which is suitable when you need to scan only a single target URL. We will then use the `-d` flag to indicate to EyeWitness the folder in which it should dump the report (in this case, the *dvga-report* folder). Listing 4-13 puts everything together.

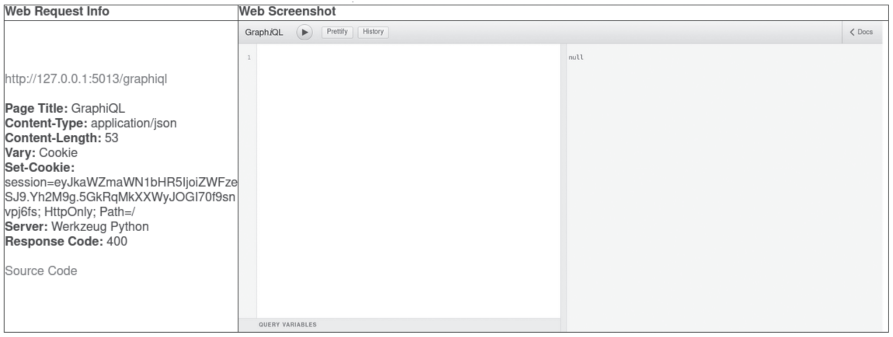
```
# eyewitness --web --single http://localhost:5013/graphiql -d dvga-report

Attempting to screenshot http://localhost:5013/graphiql

 [*] Done! Report written in the dvga-report folder!
 Would you like to open the report now? [Y/n]
```

Listing 4-13: The runtime output of EyeWitness

In the output, EyeWitness indicates that it saved the collected web page source files in the *dvga-report* folder and asks us whether to open the report. Press Y and ENTER to open a web browser displaying the HTML report, including the screenshot it took during the scan. Figure 4-1 shows the report.



http://127.0.0.1:5013/graphiql

Page Title: GraphiQL
Content-Type: application/json
Content-Length: 53
Vary: Cookie
Set-Cookie:
session=eyJkaWZmaWN1bHR5IjoiZWFzeSJ9.Yh2M9g.5GkRqMkXXWyJOGI70f9snvpj6fs; HttpOnly; Path=/
Server: Werkzeug Python
Response Code: 400

Source Code

```
null
```

Figure 4-1: An HTML report produced by EyeWitness

Additionally, the *dvga-report* will contain several folders, as shown here:

```
# ls -l dvga-report/
total 112
-rw-r--r-- 1 kali kali 95957 Dec 15 15:19 jquery.min.js
-rw-r--r-- 1 kali kali  2356 Feb 11 15:10 report.html
drwxr-xr-x 2 kali kali  4096 Feb 11 15:09 screens
drwxr-xr-x 2 kali kali  4096 Feb 11 15:09 source
-rw-r--r-- 1 kali kali   684 Feb 11 15:09 style.css
```

The *report.html* file includes information about the target, such as the HTTP response headers it sent back to the client, a screen capture of the application running on the target, and a link to the web page's source code. While you can visually identify the GraphiQL IDE by using the screen capture taken by EyeWitness, you can also confirm your finding by searching the *source* folder, where the source code files reside. Run the command shown in Listing 4-14 to search for any GraphiQL Explorer or GraphQL Playground strings within the source code.

```
# grep -Hnio "graphiql|graphql-playground" dvga-report/source/*
source/http.localhost.5013.graphiql.txt:18:graphiql
source/http.localhost.5013.graphiql.txt:18:graphiql
source/http.localhost.5013.graphiql.txt:18:graphiql
```

Listing 4-14: Keyword matches in the web page source code

Let's break down the command to explain what's happening here. We run a case-insensitive search using `grep` by passing it the `i` flag to find any instances of the words *graphql* or *graphql-playground* in the *source* folder. Using the `-H` flag, we tell `grep` to print the names of files containing any pattern matches. The `-n` flag indicates the line number at which the match is located (in this case, 18). The `-o` flag prints only the parts of matching lines that yielded positive results. As you can see, the search found multiple instances of the string *graphiql* at line number 18.

EyeWitness can run the same type of scan against a list of URLs, as opposed to a single URL, using the `-f` (file) flag. When you use this flag, EyeWitness will expect a text file containing a list of target URLs to scan. Listing 4-15 shows how to write a single URL (*http://localhost:5013/graphiql*) to a text file (*urls.txt*) and pass it on to EyeWitness as its custom URL list.

```
# echo 'http://localhost:5013/graphiql' > urls.txt
# eyewitness --web -f urls.txt -d dvga-report

Starting Web Requests (1 Hosts)
Attempting to screenshot http://localhost:5013/graphiql
Finished in 8 seconds

[*] Done! Report written in the dvga-report folder!
```

Listing 4-15: Scanning multiple URLs with EyeWitness

EyeWitness iterates over the URLs specified in the file, scans them, and saves its output into the *dvga-report* folder for further inspection.

In this example, we used a file that contains only a single URL. Often, you may want to search for any additional web paths beyond the */graphql* endpoint to check whether GraphQL lives in an alternative location, particularly one that's obscure. You could create a list of URLs to use with EyeWitness in multiple ways. The first option is to use the list of common GraphQL endpoints mentioned in "Common Endpoints" on page 73.

Alternatively, use one of Kali's built-in directory wordlists, located at */usr/share/wordlists*. One such example is the *dirbuster* wordlist. EyeWitness needs full URLs, and this wordlist contains only web paths, so we'd first need to format it using a Bash script, as shown in Listing 4-16.

```
# for i in $(cat /usr/share/wordlists/dirbuster/directory-list-2.3-small.txt);
do echo http://localhost:5013/$i >> urls.txt; done

# cat urls.txt

http://localhost:5013/api
http://localhost:5013/apis
http://localhost:5013/apidocs
http://localhost:5013/apilist
```

Listing 4-16: Using Bash and a directory wordlist to build a list of URLs

This Bash `for` loop ensures that the directories in the wordlist *directory-list-2.3-small.txt* are appended to our target host (*http://localhost:5013*) so EyeWitness can use them in its scan. All that's left is to run EyeWitness with our new wordlist file, *urls.txt*.

Attempting a Query Using Graphical Clients

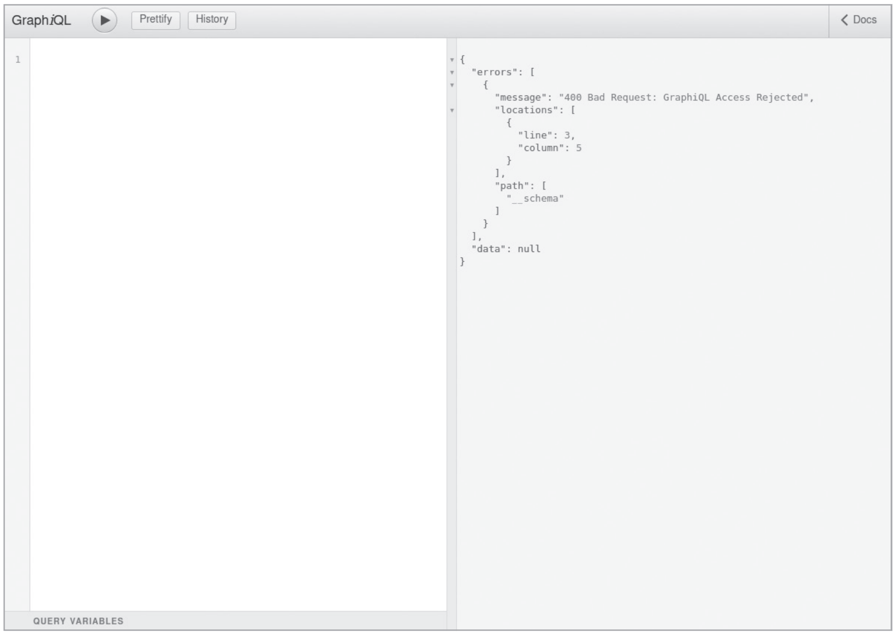
Finding instances of GraphiQL Explorer or GraphQL Playground in a penetration test doesn't guarantee that the GraphQL API itself will allow you to make unauthorized queries. Because both GraphiQL Explorer and GraphQL Playground are simply frontend interfaces to a GraphQL API, they are effectively HTTP clients that interact with a GraphQL server.

In some cases, these graphical interfaces might fail to query the API for multiple reasons. An authentication or authorization layer might be implemented in the GraphQL API that prevents unauthorized queries. The API might also restrict queries based on client properties, such as geographical location or an IP address–based allow list. Client-side mitigations could also prevent clients from running queries through GraphiQL Explorer or GraphQL Playground.

NOTE *The specification doesn't describe how to implement security measures in GraphQL or whether authorization and authentication should exist at the GraphQL layer. Chapter 7 covers how to identify these mechanisms when they are implemented in GraphQL and how to test them in black-box penetration tests.*

To confirm that we can use the interface to query the GraphQL server, we will need to send some form of an unauthenticated GraphQL query. The query must be one that will work on any GraphQL API. Think of this query as a way to confirm that the remote GraphQL API is accepting unauthenticated queries from clients. We might call it a *canary GraphQL query*.

Open the Firefox web browser in your lab machine and navigate to ***http://localhost:5013/*** to access the DVGA. You should see the DVGA's main page. Next, browse to the GraphiQL Explorer panel we discovered earlier at *http://localhost:5013/graphiql*. You will notice that we get an immediate error, indicating that our access was rejected, with the message `400 Bad Request: GraphiQL Access Rejected`, as shown in Figure 4-2.



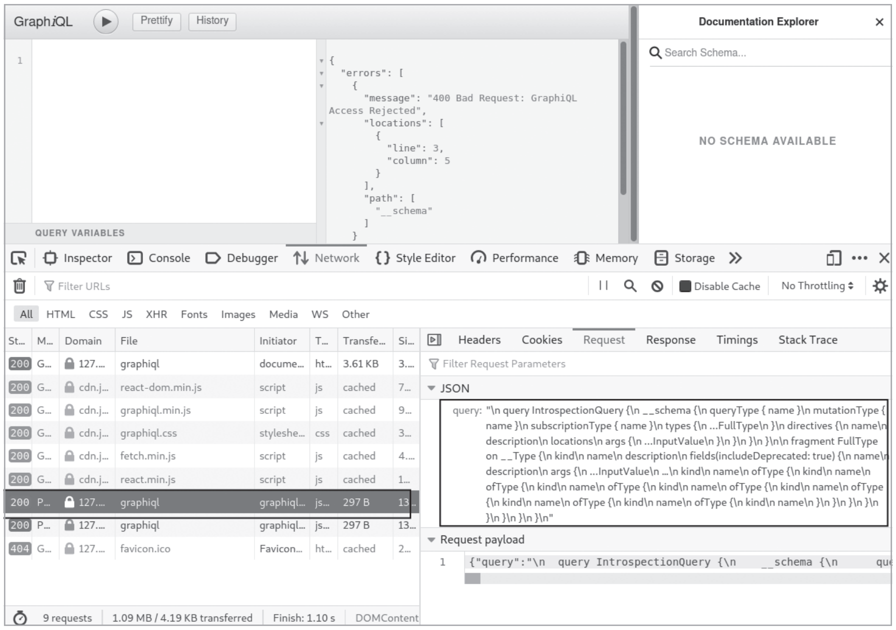
```
{
  "errors": [
    {
      "message": "400 Bad Request: GraphiQL Access Rejected",
      "locations": [
        {
          "line": 3,
          "column": 5
        }
      ],
      "path": [
        "__schema"
      ]
    }
  ],
  "data": null
}
```

Figure 4-2: The GraphiQL Explorer rejecting client access

As hackers, it's important to look at how things work under the hood. Click the **Docs** button located at the top right of the window. You should see an error message, `No Schema Available`. This error means that GraphiQL Explorer wasn't able to retrieve schema information from the API. Because GraphiQL Explorer automatically sends an introspection query to the GraphQL API to populate the documentation section with schema information on every page load, it relies on this documentation being available.

You can see this behavior by using the Developer Tools in Firefox. Press SHIFT-F9 or right-click anywhere in the web page and select **Inspect Element** to open the Developer Tools console. Click the **Network** tab; then reload the page by pressing F5.

You should be able to see a POST request sent to the */graphiql* endpoint. Figure 4-3 shows this introspection query.



```
{
  "errors": [
    {
      "message": "400 Bad Request: GraphiQL
Access Rejected",
      "locations": [
        {
          "line": 3,
          "column": 5
        }
      ],
      "path": [
        "__schema"
      ]
    }
```

St...	M...	Domain	File	Initiator	T...	Transfe...	Si...
200	G...	127....	graphiql	docume...	ht...	3.61 KB	3....
200	G...	cdn.j...	react-dom.min.js	script	js	cached	7...
200	G...	cdn.j...	graphiql.min.js	script	js	cached	9...
200	G...	cdn.j...	graphiql.css	styleshe...	css	cached	3...
200	G...	cdn.j...	fetch.min.js	script	js	cached	4....
200	G...	cdn.j...	react.min.js	script	js	cached	1...
200	P...	127....	graphiql	graphiql...	js...	297 B	13...
200	P...	127....	graphiql	graphiql...	js...	297 B	13...
404	G...	127....	favicon.ico	Favicon...	ht...	cached	2...

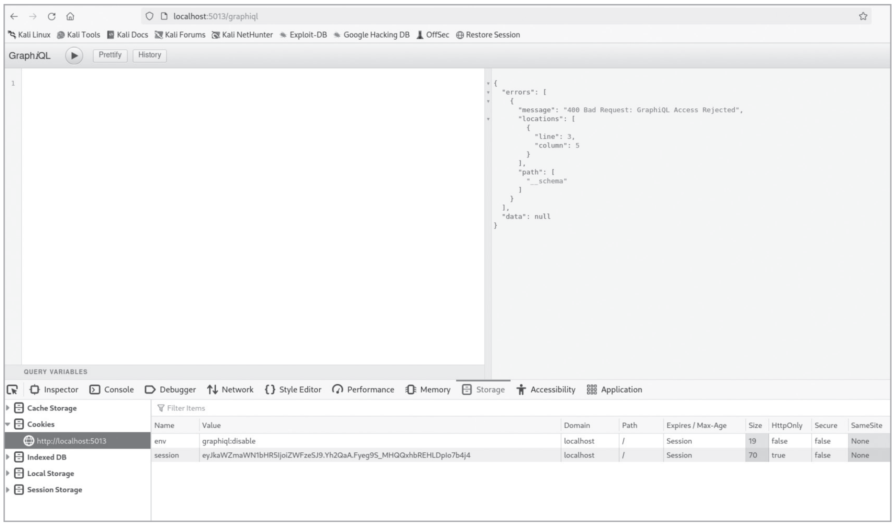
9 requests | 1.09 MB / 4.19 KB transferred | Finish: 1.10 s | DOMContent

```
JSON
query: "\n query IntrospectionQuery {\n __schema {\n queryType { name }\n mutationType {
  name }\n subscriptionType { name }\n types {\n ...FullType\n }\n directives {\n name\n
  description\n locations\n args {\n ...InputValue\n }\n }\n }\n}\n\n fragment FullType
  on __Type {\n kind\n name\n description\n fields(includeDeprecated: true) {\n name\n
  description\n args {\n ...InputValue\n ...\n kind\n name\n ofType {\n kind\n name\n
  ofType {\n kind\n name\n ofType {\n kind\n name\n ofType {\n kind\n name\n ofType
  {\n kind\n name\n ofType {\n kind\n name\n ofType {\n kind\n name\n }\n }\n }\n }\n
  }\n }\n }\n }\n"

Request payload
{"query":"\n  query IntrospectionQuery {\n    __schema {\n     que
```

Figure 4-3: A GraphiQL Explorer introspection query shown in Firefox Developer Tools

If the introspection query was successfully sent, what could possibly be rejecting our access to GraphiQL Explorer? Let's continue to explore the Developer Tools in Firefox for clues. Click the **Storage** tab, shown in Figure 4-4.

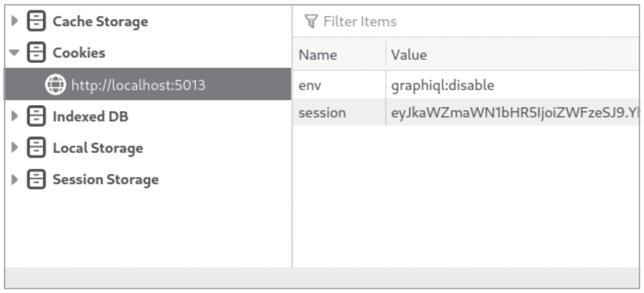


```
{
  "errors": [
    {
      "message": "400 Bad Request: GraphiQL Access Rejected",
      "locations": [
        {
          "line": 3,
          "column": 5
        }
      ],
      "path": [
        "__schema"
      ]
    }
  ],
  "data": null
}
```

Name	Value	Domain	Path	Expires / Max-Age	Size	HttpOnly	Secure	SameSite
env	graphiql:disable	localhost	/	Session	19	false	false	None
session	eyJkaWZmaWN1bHR5IjoiZWFzeSJ9.Yh2QaA.Fyeg9S_MHQQxhbREHLDplo7b4j4	localhost	/	Session	70	true	false	None

Figure 4-4: The Developer Tools Storage tab in Firefox

The Storage tab gives us a view of the HTTP cookies that were set up by the application, as well as access to the browser's local and session storage. On the left pane, click the **Cookies** drop-down menu and select **http://localhost:5013** to see the specific cookies for the domain, as shown in Figure 4-5.



Name	Value
env	graphiql:disable
session	eyJkaWZmaWN1bHR5IjoiZWFzeSJ9.Y

Figure 4-5: HTTP cookies

You'll notice that, in the right pane, we have two keys set in our HTTP cookies: `env` and `session`. The `env` key in particular is interesting, because it appears to have the string `graphiql:disable` set as its value. As hackers, this should ring a bell or two. Is it possible that this cookie value is responsible for GraphiQL Explorer's denying access? We can find out by tampering with its value.

Double-click the text `graphiql:disable`, which will allow you to modify it; then simply remove `disable` and replace it with **`enable`**. Next, refresh the web page. You'll notice that we no longer see the rejection message in GraphiQL Explorer. To confirm that tampering with the cookie actually works, attempt to run a GraphQL query. You should be able to get a response from the GraphQL API! This is an example of a weak client-side security control that can easily be circumvented.

Developers often create web applications with the mindset that clients are to be trusted, but not everyone will play by the rules. Threat actors who are interested in finding loopholes will tamper with applications and attempt to defeat any countermeasures in place. It's important to remember that anything an attacker can directly control can potentially be circumvented. Yet controls implemented on the client are not a rare thing to see; you may find applications implementing input validation or file upload validation only on the client side. These can often be bypassed. In Chapter 7, you'll learn more about defeating GraphQL authorization and authentication mechanisms.

Querying GraphQL by Using Introspection

Introspection is one of GraphQL's key features, as it provides information about the various types and fields the GraphQL schema supports. A self-documenting API is very useful for anyone who needs to consume it, such as third-party businesses or other clients.

As hackers, one of the first things we want to test when we run into a GraphQL application is whether its introspection mechanism is enabled. Many GraphQL implementations enable introspection by default. Some implementations might have an option to disable introspection, but others might not. For example, the Python GraphQL implementation Graphene does not provide the option to disable introspection. To do so, the consumer would have to dig into the code and identify ways to prevent introspection queries from being processed. On the other hand, the GraphQL PHP implementation graphql-php enables introspection by default but also documents how to completely disable this feature. Table 4-2 shows the state of introspection in some of the popular GraphQL server implementations.

Table 4-2: The State of Introspection in GraphQL Implementations

Language	Implementation	Introspection configuration	Disable introspection option
Python	Graphene	Enabled by default	Not available
Python	Ariadne	Enabled by default	Available
PHP	graphql-php	Enabled by default	Available
Go	graphql-go	Enabled by default	Not available
Ruby	graphql-ruby	Enabled by default	Available
Java	graphql-java	Enabled by default	Not available

Any default setting that directly impacts security is always good news for hackers. Application maintainers rarely change these default settings. (Some maintainers may not even be aware of them.) In Table 4-2, you can see that in some cases—such as in graphql-go, graphql-java, and Graphene—introspection can be disabled only if the application maintainers code the solution into the GraphQL API themselves; there is no official, vendor-vetted solution to disable it.

While opinions on this matter vary, especially in security circles, introspection in GraphQL is widely considered a feature and not a vulnerability. Companies that adopt GraphQL may choose to keep it enabled, while others may disable it to avoid disclosing information that could be leveraged in attacks. If no external consumers integrate with a GraphQL API, it's possible that developers could disable introspection altogether without impacting normal application flows.

Depending on your target, the response to an introspection query could be fairly large. Also, if you're attacking a target with a mature security program, these queries may be monitored for any attempts from untrusted clients, such as those in new geographical locations or with new IP addresses.

To experiment with the introspection query by using our vulnerable server, open the Altair client in your lab and ensure that the address bar is set to *http://localhost:5013/graphql*. Next, enter the introspection query shown in Listing 4-17 and execute it in Altair.

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

Listing 4-17: An introspection query in its simplest form

This query uses the meta-field `__schema`, which is the type name of the GraphQL schema introspection system. It then requests the `name` of all `types` available in the GraphQL server. The following output shows the server's response to the query:

```
{
  "data": {
    "__schema": {
      "types": [
--snip--
        {
          "name": "PasteObject"
        },
        {
          "name": "CreatePaste"
        },
        {
          "name": "DeletePaste"
        },
        {
          "name": "UploadPaste"
        },
        {
          "name": "ImportPaste"
        },
--snip--
      ]
    }
  }
}
```

While we receive a valid response, this query in its current form gives us only a partial view of the features available through the API. The response is missing key information, such as query and mutation names, information about which queries allow arguments to be passed by clients, the data types of arguments (such as scalar types like `String` and `Boolean`), and so on. These are important, because queries that accept arguments could be prone to vulnerabilities, such as injections, server-side request forgeries, and so on.

We can craft a more specialized introspection query that would give us more data about the target application's schema. A useful introspection query is one that will give us information on the entry points into the application, such as queries, mutations, subscriptions, and the type of data that can be injected into them. Consider the introspection query shown in Listing 4-18.

```
query IntrospectionQuery {
   __schema {
 ❶ queryType { name }
    mutationType { name }
    subscriptionType { name }
 ❷ types {
      kind
      name
    ❸ fields {
       name
      ❹ args {
          name
        }
      }
    }
  }
}
```

Listing 4-18: A more useful introspection query

The introspection query in Listing 4-18 gives us a bit more insight into the API. At ❶ we get the `name` of all queries (`queryType`), mutations (`mutationType`), and subscriptions (`subscriptionType`) available in the GraphQL API. These names are typically self-explanatory, to make it easier for clients to use the API, so knowing these query names gives us an idea of the information we could receive.

At ❷ we get all the `types` in the schema, along with their `kind` (such as an object) and name (such as `PasteObject`). At ❸ we get the `fields` along with the `name` of each one, which will allow us to know the types of fields we can fetch when we use different GraphQL objects. Next, we get the arguments (`args`) of these fields along with their `name` ❹. Arguments could be any information the API is expecting the client to supply when it queries the API (typically, dynamic data). For example, when a client creates a new paste, it will supply an arbitrary `title` argument and a `content` argument containing the body of the paste, which might be a code snippet or other text.

In penetration tests, you may want to run an introspection query against an entire network, assuming a GraphQL server may be present. In this case, you would either need to write your own script or use the Nmap NSE script *graphql-introspection.nse* we installed in Chapter 2. This script is simple: it queries GraphQL by using the `__schema` meta-field to determine if it's fetchable.

Say you have a list of IP addresses in a text file such as *hosts.txt*. Using Nmap's `-iL` flag, you can tell Nmap to use it as its list of targets. Using the `--script` flag, you can then tell Nmap to run the *graphql-introspection* NSE script against any host that has port `5013` open (`-p` flag). The `-sV` flag performs a service and version scan. The command in Listing 4-19 shows how this is accomplished.

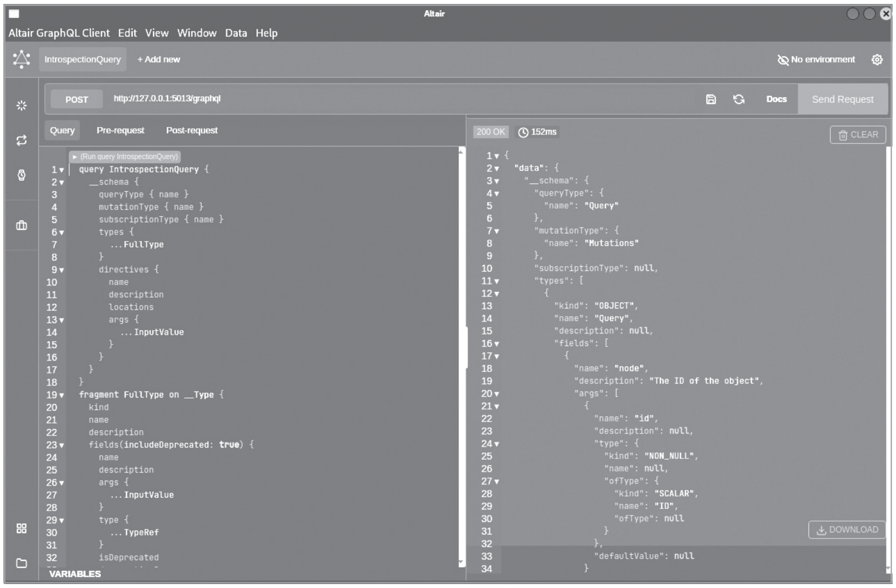
```
# nmap --script=graphql-introspection -iL hosts.txt -sV -p 5013

PORT     STATE SERVICE VERSION
5013/tcp open  http    Ajenti http control panel
| graphql-introspection:
|   VULNERABLE:
|   GraphQL Server allows Introspection queries at endpoint:
|   Endpoint: /graphql is vulnerable to introspection queries!
|     State: VULNERABLE
|       Checks if GraphQL allows Introspection Queries.
|
|     References:
|_      https://graphql.org/learn/introspection/
```

Listing 4-19: A GraphQL introspection detection with the Nmap NSE

Using `nmap` to detect when introspection is enabled is just the first step. The next step is to extract all possible schema information by using a more robust query.

In the book's GitHub repository, you can find a comprehensive introspection query that, when executed, will extract a lot of useful information about the target's schema: *https://github.com/dolevf/Black-Hat-GraphQL/blob/master/queries/introspection_query.txt*. This query will return information such as queries, mutations, and subscriptions names, with the arguments they accept; names of objects and fields, along with their types; names and descriptions of GraphQL directives; and object relationships. If you run that query in Altair, the server should return a fairly large response, as shown in Figure 4-6.



```
query IntrospectionQuery {
  __schema {
    queryType { name }
    mutationType { name }
    subscriptionType { name }
    types {
      ...FullType
    }
    directives {
      name
      description
      locations
      args {
        ...InputValue
      }
    }
  }
}
fragment FullType on __Type {
  kind
  name
  description
  fields(includeDeprecated: true) {
    name
    description
    args {
      ...InputValue
    }
    type {
      ...TypeRef
    }
    isDeprecated
```

```
{
  "data": {
    "__schema": {
      "queryType": {
        "name": "Query"
      },
      "mutationType": {
        "name": "Mutations"
      },
      "subscriptionType": null,
      "types": [
        {
          "kind": "OBJECT",
          "name": "Query",
          "description": null,
          "fields": [
            {
              "name": "node",
              "description": "The ID of the object",
              "args": [
                {
                  "name": "id",
                  "description": null,
                  "type": {
                    "kind": "NON_NULL",
                    "name": null,
                    "ofType": {
                      "kind": "SCALAR",
                      "name": "ID",
                      "ofType": null
                    }
                  },
                  "defaultValue": null
                }
```

Figure 4-6: An introspection in Altair

The response is large enough (containing approximately 2,000 lines) that it would be challenging for any human to go through it manually and make sense of it without investing a significant amount of time. This is where GraphQL visualizers such as *GraphQL Voyager* come in handy.

Visualizing Introspection with GraphQL Voyager

GraphQL Voyager, which can be found at either *https://ivangoncharov.github.io/graphql-voyager* or *http://lab.blackhatgraphql.com:9000*, is an open source tool that processes either introspection query responses or GraphQL SDL files and visualizes them, making it easy to identify the various queries, mutations, and subscriptions and the relationships between them.

The tool's introspection query option is most suitable for scenarios such as black-box penetration tests, in which the application's code base is not accessible to us. The SDL option is useful when we might have direct access to the GraphQL schema files, such as during a white-box penetration test in which the company provides us with full access to the source code.

Try visualizing the introspection query response you just received in Altair and importing it into GraphQL Voyager. Copy the response and then open your browser and navigate to GraphQL Voyager. Click the **Change Schema** button located at the top-left corner. Select the **Introspection** tab, paste in the response, and click the **Display** button. You should see a visualization similar to the one shown in Figure 4-7.

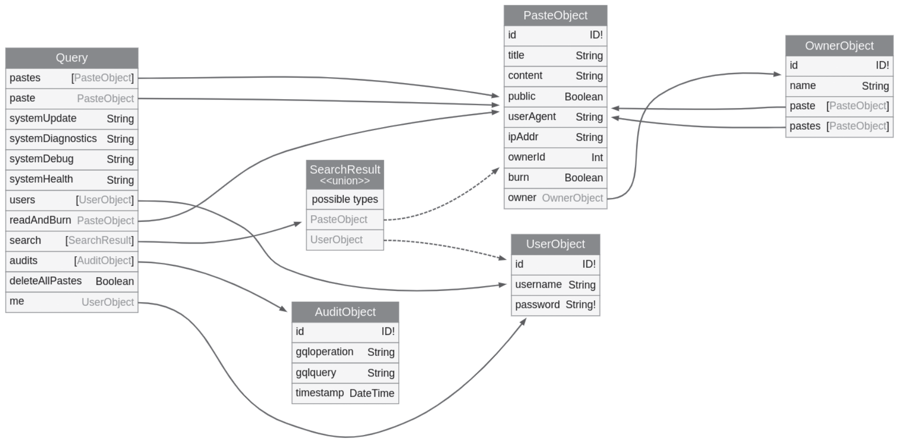


Figure 4-7: The schema view in Voyager

The visualization we receive from Voyager represents the queries, mutations, and subscriptions available in our target application and how they relate to the different objects and fields that exist in the schema.

Under Query, you can see that the application supports 12 queries. The arrows in the view represent the mapping between these queries and the schema objects. For example, when you use the `pastes` query, it will return an array of `[PasteObject]` objects, which is also the reason you're seeing an arrow pointing to the `PasteObject` table. The `system` queries (update, diagnostics, debug, and health) are not tied to any other schema objects; they simply return a string whenever you use them.

You can also see that we have relationships (edges) between fields. For example, the `owner` field in the `PasteObject` object is linked to `OwnerObject`, and the `paste` field in `OwnerObject` is linked back to `PasteObject`. This circular condition could lead to DoS conditions, as you will learn in Chapter 5.

NOTE *You can toggle between the Query view, Mutation view, and Subscription view by using the drop-down menu at the bottom of Voyager.*

Now that we've experimented with visualizing an introspection response in Voyager, let's do the same with SDL files. Voyager accepts SDL files and can process them just as well as it does introspection responses. To see this in action, click the **Change Schema** button located at the top-left corner in Voyager, select the **SDL** tab, and paste in the SDL file located at *https://github.com/dolevf/Black-Hat-GraphQL/blob/master/ch04/sdl.graphql*. Then click the **Display** button. You should see a similar visualization to the one generated in the Introspection tab.

Generating Introspection Documentation with SpectaQL

SpectaQL (*https://github.com/anvilco/spectaql*) is an open source project that allows you to generate static documentation based on an SDL file. The document that gets generated will include information about how to construct queries, mutations, and subscriptions; the different types; and their fields. We've hosted an example SpectaQL-generated schema of DVGA at *http://lab.blackhatgraphql.com:9001* so you can see how SpectaQL looks when it's functional.

Exploring Disabled Introspection

At some point, you'll probably encounter a GraphQL API that has introspection disabled. To see what this looks like, let's use one of the neat features of our vulnerable GraphQL server: turning on its hardened mode.

The DVGA works in two modes, a Beginner mode and an Expert (hardened) mode. Both versions are vulnerable; the only difference is that the Expert mode has a few security mechanisms to protect the application from any dangerous queries.

To change the application's mode, open the Altair client and ensure that the address points to *http://localhost:5013/graphql*. In the left sidebar, click the Set Headers icon, which looks like a small sun symbol. Set **Header Key** to **X-DVGA-MODE** and set **Header Value** to **Expert**. This HTTP header set instructs DVGA to perform security checks on any incoming queries that include the headers as part of the request. Alternatively, you can toggle on Expert mode from within DVGA's web interface by using the drop-down menu located at the top-right corner (the cubes icon).

Now attempt a simple introspection query using Altair:

```
query {
   __schema {
     __typename
   }
}
```

You should see an error response indicating that introspection is disabled, causing the query to fail (Listing 4-20).

```
{
  "errors": [
    {
      "message": "400 Bad Request: Introspection is Disabled",
      "locations": [
        {
          "line": 2,
          "column": 7
        }
      ],
      "path": [
        "__schema"
      ]
    }
  ],
  "data": null
}
```

Listing 4-20: An error returned when introspection is disabled

In cases like this one, you'll need a plan B. In Chapter 6, you'll learn how to discover information about the GraphQL application even if introspection data isn't available.

Fingerprinting GraphQL

Earlier in this chapter, we highlighted the many GraphQL implementations available. How can we tell which one is running on the server we're trying to hack? The answer is *server fingerprinting*, the operation of identifying information about the target's running services and their versions. For example, a common and simple technique for fingerprinting web servers is to make an HTTP HEAD request using a tool like cURL and observe the HTTP response headers that are returned.

Once we know the specific technology and version running an application, we can perform a more accurate vulnerability assessment against the service. For example, we can look for publicly available exploits to run against the target's version or read the software's documentation to identify weaknesses.

Popular web servers such as Apache or Nginx are great examples of services that are easy to fingerprint, since both typically set the `server` HTTP response header when a client makes a request to them. Listing 4-21 shows an example of how the web server behind the Apache Software Foundation website identifies itself by using the `server` header:

```
# curl -I https://apache.org/

HTTP/2 200
server: Apache
vary: Accept-Encoding
content-length: 73190
```

Listing 4-21: The Apache web server fingerprinting using a HEAD request

As expected, the Apache Software Foundation's website is, in fact, running on the Apache web server. (It would have been a little odd if this were not the case!)

Fingerprinting services in a penetration test won't always be this easy; sometimes accurate fingerprinting requires looking closely at the details, as not all software self-identifies, including GraphQL servers. The techniques used to fingerprint GraphQL implementations are relatively new in the security industry. We (the authors of this book) have developed several strategies for doing so, based on our research, and incorporated them into Graphw00f.

GraphQL fingerprinting relies on the observation of various discrepancies between implementations of GraphQL servers. Here are a few examples:

- Inconsistencies in error messages
- Inconsistencies in response outputs to malformed GraphQL queries
- Inconsistencies in response outputs to properly structured queries
- Inconsistencies in response outputs to queries deviating from the GraphQL specification

Using all four of these factors, we can uniquely identify the implementation behind a GraphQL-backed application.

Let's examine how two GraphQL server implementations respond to a malformed query. This query, shown in Listing 4-22, introduces an additional `y` character in the word `queryy`, which is not compliant with the GraphQL specification. We want to see how two GraphQL implementations respond to it. The first implementation is Sangria, a Scala-based GraphQL server.

```
queryy {
   __typename
}
```

Listing 4-22: A malformed GraphQL query

Listing 4-23 shows Sangria's response to the malformed query.

```
{
  "syntaxError": "Syntax error while parsing GraphQL query.
  Invalid input \"queryy\", expected ExecutableDefinition or
  TypeSystemDefinition (line 1, column 1):\nqueryy {\n^",
  "locations": [
    {
      "line": 1,
      "column": 1
    }
  ]
}
```

Listing 4-23: Sangria's response to the malformed query

The second implementation is HyperGraphQL, a Java-based GraphQL server. Listing 4-24 shows how it responds to the malformed query.

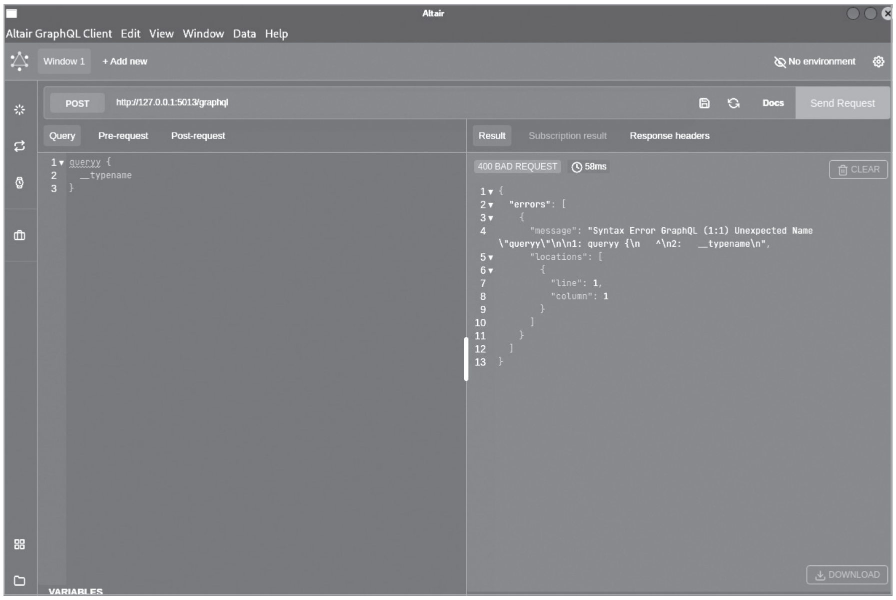
```
{
  "extensions": {},
  "errors": [
    {
      "message": "Validation error of type InvalidSyntax: Invalid query syntax.",
      "locations": [
        {
          "line": 0,
          "column": 0,
          "sourceName": null
        }
      ],
      "description": "Invalid query syntax.",
      "validationErrorType": "InvalidSyntax",
      "queryPath": null,
      "errorType": "ValidationError",
      "extensions": null,
      "path": null
    }
  ]
}
```

Listing 4-24: HyperGraphQL's response to the malformed query

As you can observe, the two responses are different in every possible way, and we can distinguish between these implementations based solely on their responses.

Next, we'll attempt the same malformed query in our lab against the DVGA to see the kind of response we get. Open the Altair client and send the GraphQL query. You should see output similar to Figure 4-8.

As you can see, the output is different from both the Sangria and HyperGraphQL responses. This is because DVGA is based on Graphene, a Python GraphQL implementation.



```
http://127.0.0.1:5013/graphql
```

```
queryy {
  __typename
}
```

```
{
  "errors": [
    {
      "message": "Syntax Error GraphQL (1:1) Unexpected Name
\"queryy\"\n\n1: queryy {\n   ^\n2:   __typename\n",
      "locations": [
        {
          "line": 1,
          "column": 1
        }
      ]
    }
  ]
}
```

Figure 4-8: Sending a malformed query with Altair

Running queries manually and analyzing the discrepancies between implementations doesn't really scale well, which is why we built a server fingerprinting capability into Graphw00f. In the next section, we'll use it for server fingerprinting purposes.

Detecting Servers with Graphw00f

Graphw00f is currently the only tool available for GraphQL server fingerprinting. It can detect many of the popular GraphQL server implementations and provide meaningful information whenever it successfully fingerprints a server.

In your lab, open the terminal emulator. If you enter the *graphw00f* directory and run `python3 main.py -l`, you'll see that Graphw00f is capable of fingerprinting over 24 GraphQL implementations. This list comprises the majority of GraphQL targets currently in use.

Let's use it to fingerprint the DVGA. We'll run Graphw00f with the `-f` flag to enable fingerprint mode and the `-t` flag to specify the target (Listing 4-25). You could combine the `-f` flag with the `-d` flag (covered earlier in this chapter) if you wanted to detect GraphQL and fingerprint at the same time. Here, we'll use the `-f` flag on its own, as we already know the path to GraphQL on the server.

```
# cd ~/graphw00f
# python3 main.py -f -t http://localhost:5013/graphql

  [*] Checking if GraphQL is available at http://localhost:5013/graphql...
  [!] Found GraphQL.
  [*] Attempting to fingerprint...
[*] Discovered GraphQL Engine: (Graphene)
[!] Attack Surface Matrix: https://github.com/nicholasaleks
/graphql-threat-matrix/blob/master/implementations/graphene.md
[!] Technologies: Python
[!] Homepage: https://graphene-python.org
[*] Completed.
```

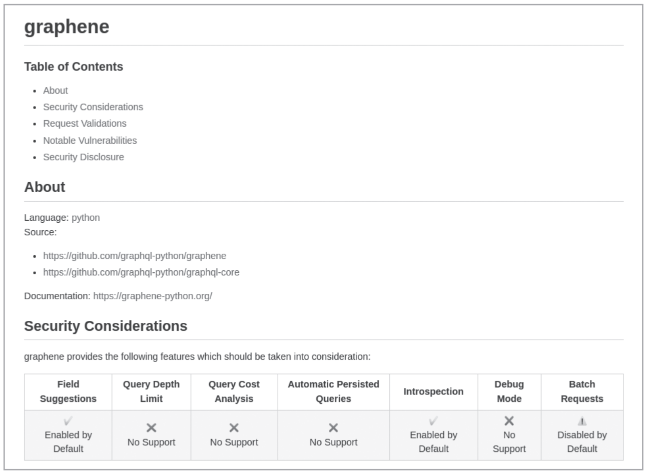
Listing 4-25: The fingerprinting of a GraphQL server

The tool first checks whether the target is, in fact, a GraphQL server. It does so by sending a few queries and inspecting their responses against its own database of signatures. As you can see, it is able to discover a GraphQL server running on Graphene and provides us with an attack surface matrix link. The *attack surface matrix* is essentially knowledge about the security posture of the various GraphQL implementations that Graphw00f can fingerprint. Graphw00f uses the GraphQL Threat Matrix we discussed in Chapter 3 as its implementation security posture database.

Since we now know that DVGA runs Graphene, we need to analyze Graphene's weaknesses to determine which attacks we can run against this specific implementation. Some implementations have been around longer than others. Thus, they are more mature, stable, and offer more security features than others. This is why knowing the backend implementation is an advantage when we hack a GraphQL target.

Analyzing Results

Take a look at the attack surface threat matrix, which provides information about the implementation's default behavior and the security controls available for it (for example, the settings that are enabled by default, the security controls that exist, and other useful features we can leverage for hacking purposes). Figure 4-9 shows the attack surface matrix for Graphene. You can also find it on GitHub at *https://github.com/nicholasaleks/graphql-threat-matrix/blob/master/implementations/graphene.md*.



graphene

Table of Contents

- About
- Security Considerations
- Request Validations
- Notable Vulnerabilities
- Security Disclosure

About

Language: python
Source:

- https://github.com/graphql-python/graphene
- https://github.com/graphql-python/graphql-core

Documentation: https://graphene-python.org/

Security Considerations

graphene provides the following features which should be taken into consideration:

Field Suggestions	Query Depth Limit	Query Cost Analysis	Automatic Persisted Queries	Introspection	Debug Mode	Batch Requests
✓ Enabled by Default	✗ No Support	✗ No Support	✗ No Support	✓ Enabled by Default	✗ No Support	⚠ Disabled by Default

Figure 4-9: Graphene's attack surface matrix

The table under Security Considerations shows various GraphQL features and whether they are available in Graphene. If they do exist, the table lists whether they are enabled or disabled by default. Some of the items in the table are security controls, while others are native GraphQL features:

- *Field Suggestions* informs a client whenever they send a query with a spelling mistake and suggests alternative options. This can be leveraged for information disclosure.
- *Query Depth Limit* is a security control to prevent DoS attacks that may abuse conditions such as cyclical node relationships in schemas.
- *Query Cost Analysis* is a security control to prevent DoS attacks that stem from computationally complex queries.
- *Automatic Persisted Queries* is a caching mechanism. It allows the client to pass a hash representing a query as a way to save bandwidth and can be used as a security control with an allow list of safe queries.
- *Introspection* provides access to information about queries, mutations, subscriptions, fields, objects, and so on through the `__schema` meta-field. This can be abused to disclose information about the application's schema.
- *Debug Mode* is a mode in GraphQL that provides additional information in the response for debugging purposes. This can potentially introduce information disclosure issues.
- *Batch Requests* is a feature that provides clients with the ability to send a sequence of queries in a single HTTP request. Batch queries are a great vector for DoS attacks.

In later chapters, you'll learn how each of these features can make our hacking lives easier (or harder).

Summary

In this chapter, you learned the art of performing reconnaissance against GraphQL servers by using a variety of security tools. We discussed how to detect and fingerprint GraphQL servers deployed in standard and non-standard locations, as well as how to find GraphQL IDE clients by using the EyeWitness security tool. We also visualized an introspection query and SDL files by using GraphQL Voyager to better understand queries, mutations, and object relationships.