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*).

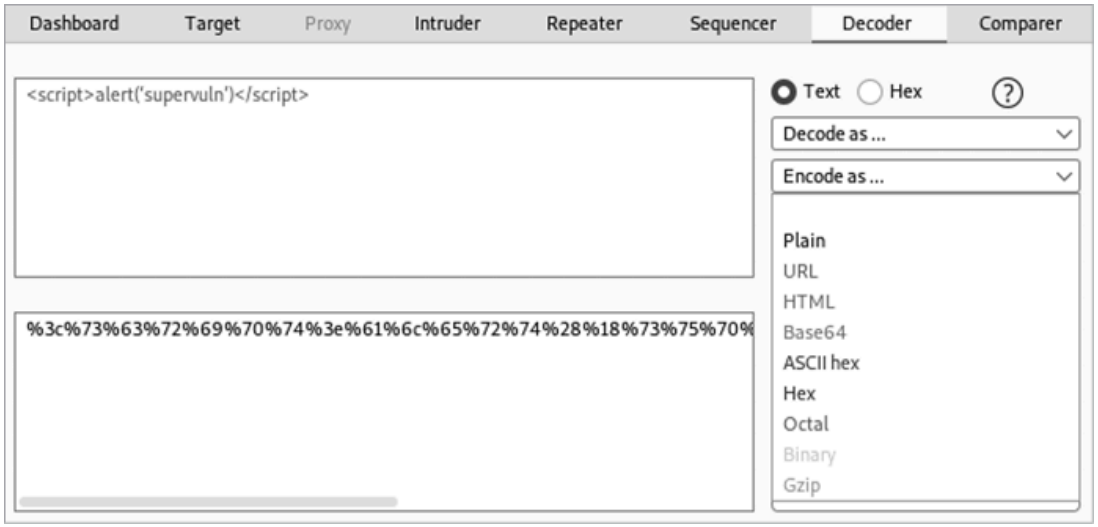

Figure 13-1: Burp Suite Decoder

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.

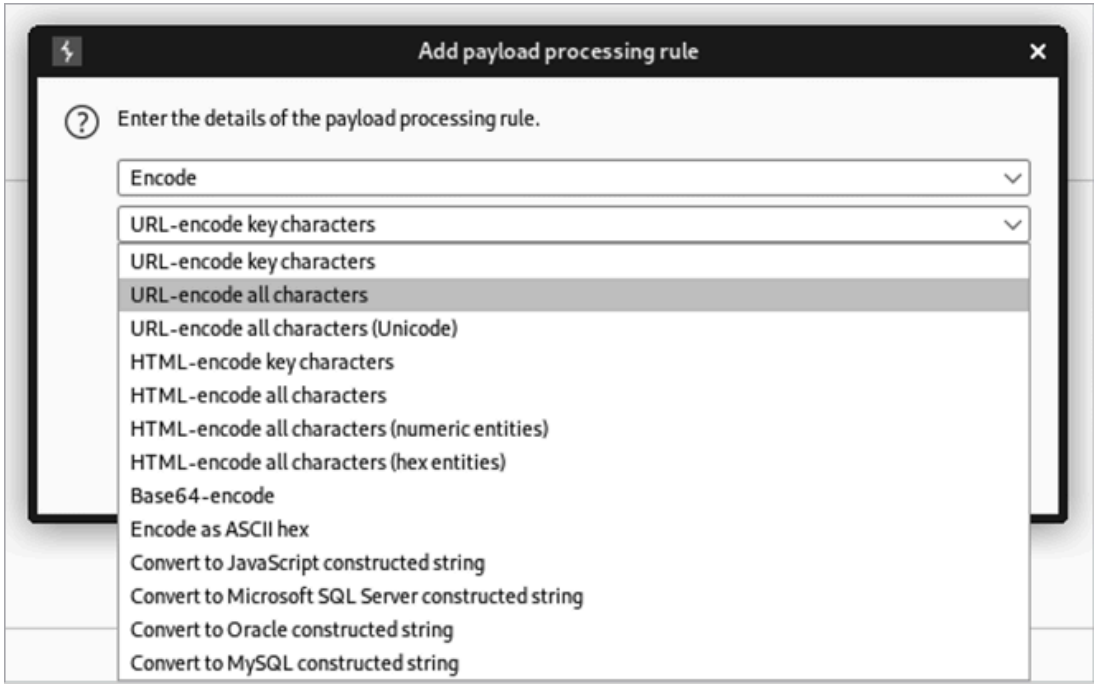

Figure 13-2: The Add Payload Processing Rule screen

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*.

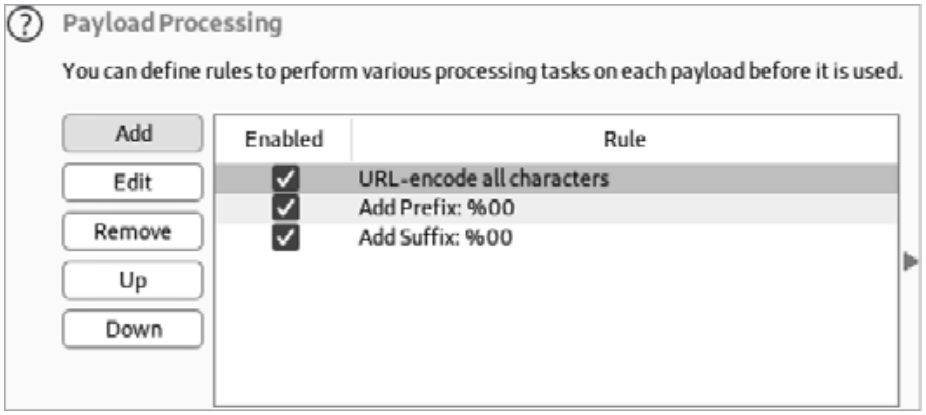

Figure 13-3: Intruder's payload-processing options

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/
000000003:   404        0 L      2 W        131 Ch      "Q0NDQ2M="
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*).

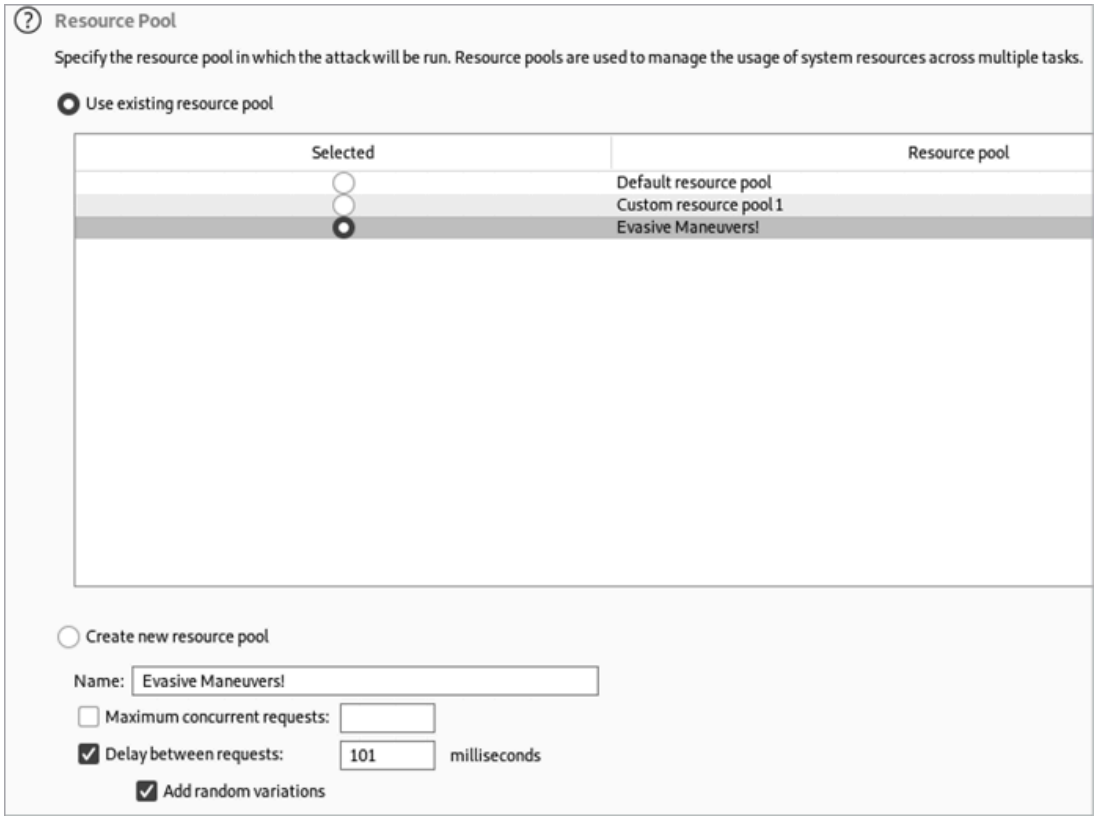

Figure 13-4: Burp Suite Intruder's Resource Pool

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*.

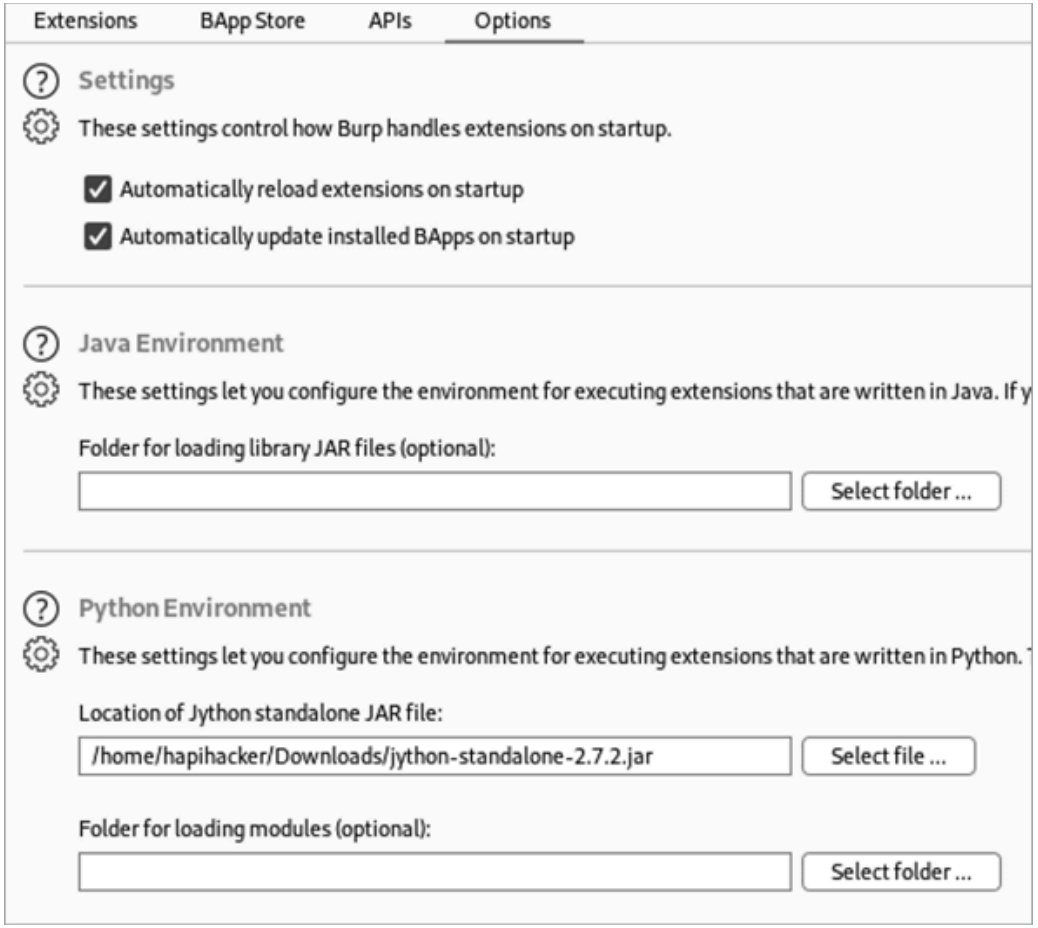

Figure 13-5: Burp Suite Extender options

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*).

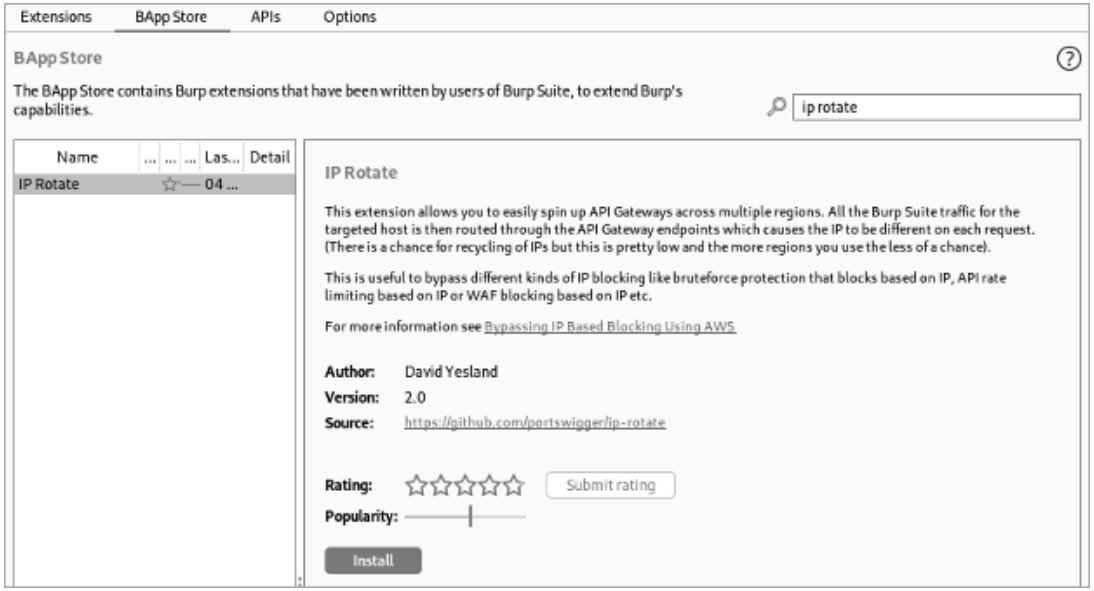

Figure 13-6: IP Rotate in the BApp Store

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*).

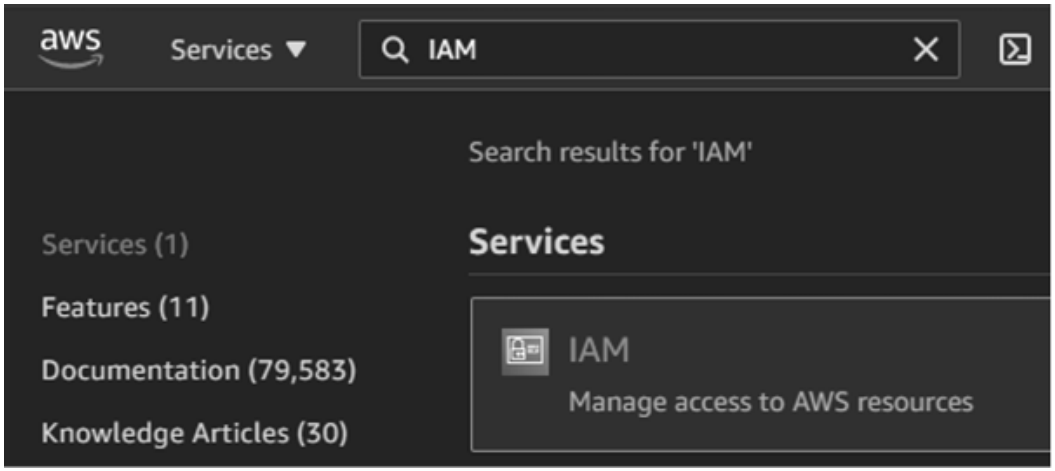

Figure 13-7: Finding the AWS IAM service

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.

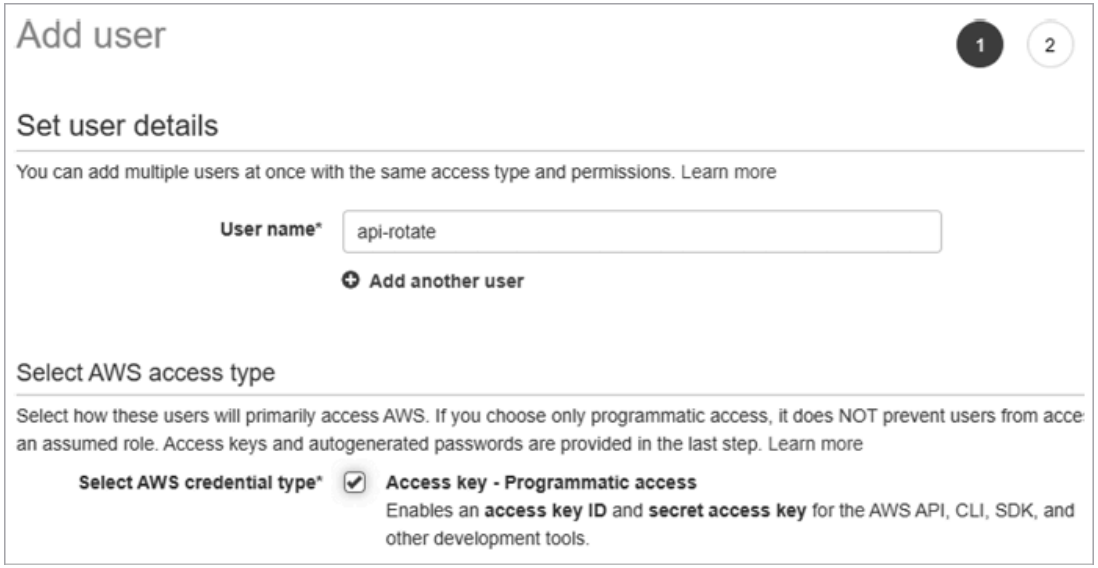

Figure 13-8: AWS Set User Details page

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

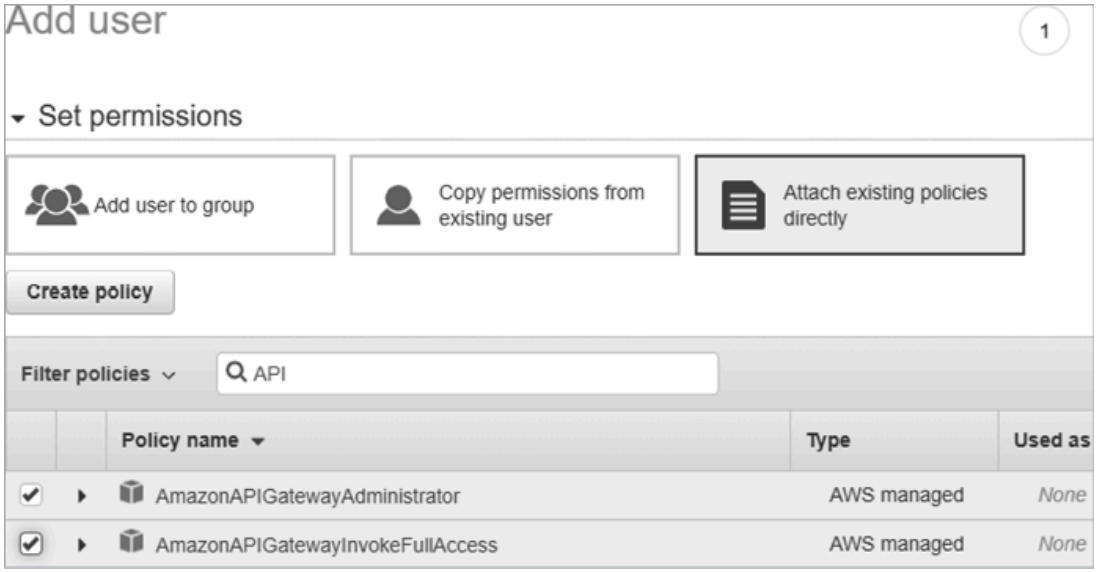

Figure 13-9: AWS Set Permissions page

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*).

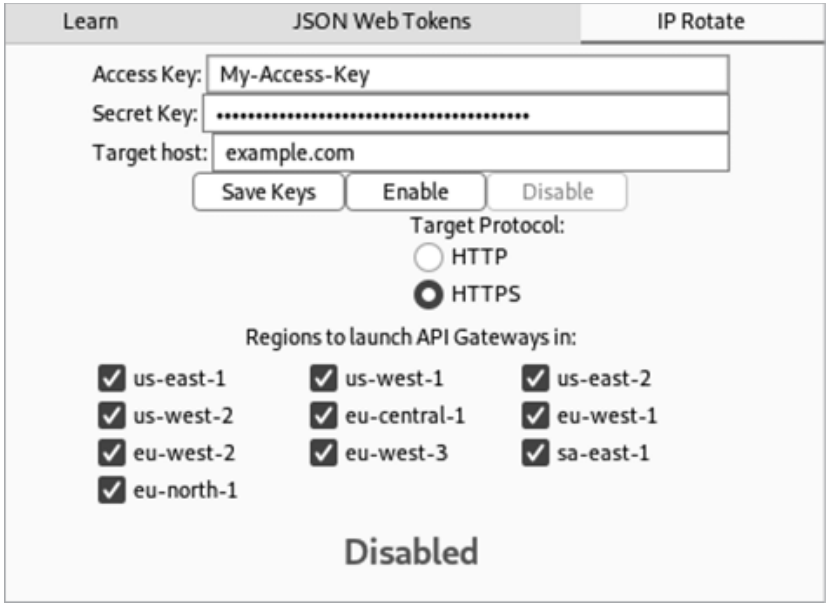

Figure 13-10: The Burp Suite IP Rotate module

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*.

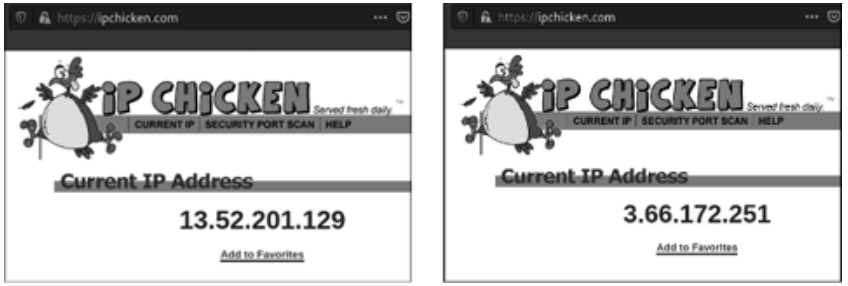

Figure 13-11: IPChicken

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.