

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

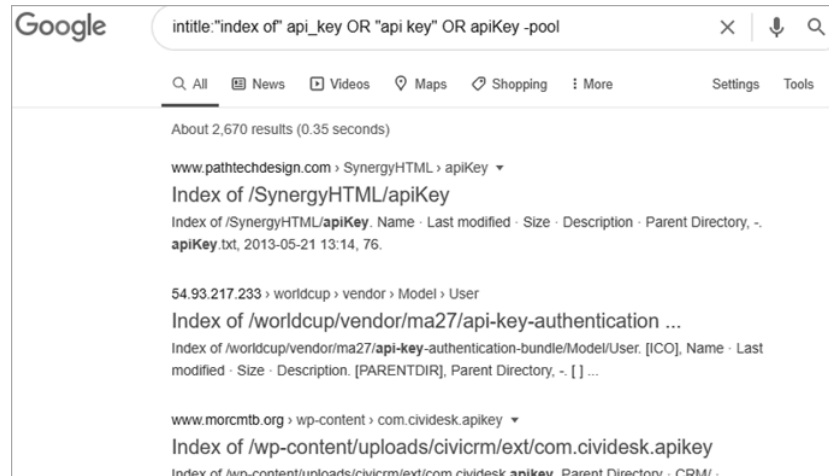


Figure 6-1: The results of a Google hack for APIs, including several web pages with exposed API keys

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.

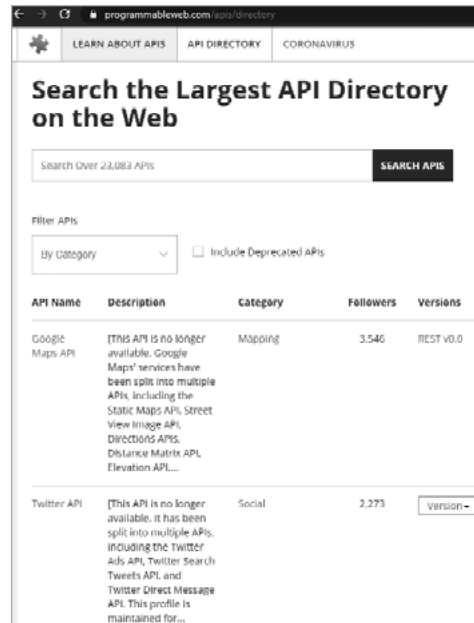
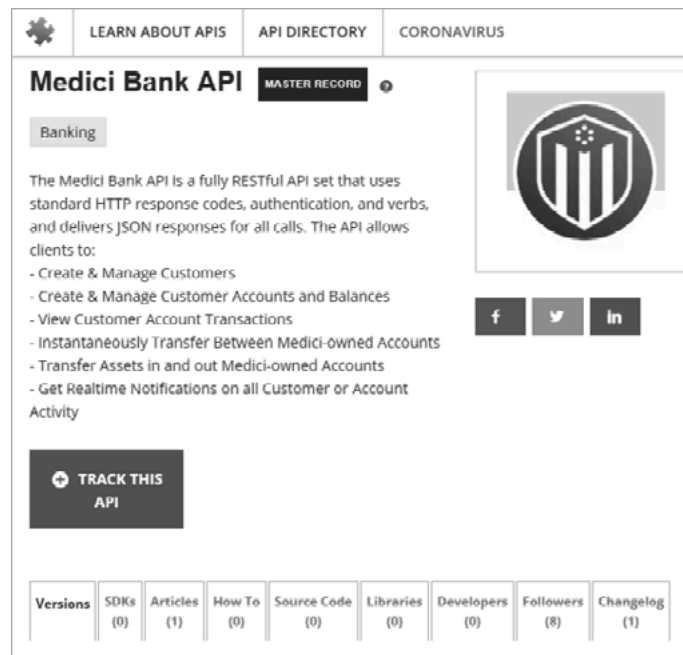


Figure 6-2: The ProgrammableWeb API directory

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](#).



Medici Bank API MASTER RECORD

Banking

The Medici Bank API is a fully RESTful API set that uses standard HTTP response codes, authentication, and verbs, and delivers JSON responses for all calls. The API allows clients to:

- Create & Manage Customers
- Create & Manage Customer Accounts and Balances
- View Customer Account Transactions
- Instantaneously Transfer Between Medici-owned Accounts
- Transfer Assets in and out Medici-owned Accounts
- Get Realtime Notifications on all Customer or Account Activity

TRACK THIS API

Versions	SDKs	Articles	How To	Source Code	Libraries	Developers	Followers	Changelog
	(0)	(1)	(0)	(0)	(0)	(0)	(8)	(1)

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.

Summary	SDKs (0)	Articles (1)	How To (0)	Source Code (0)	Libraries (0)	Developers (0)	Followers (8)	Changelog (0)
SPECS								
API Endpoint https://api.medicbank.io								
API Portal / Home Page https://mbapl.docs.stopligh.io								
Primary Category Banking								
API Provider Medici Bank International								
SSL Support Yes								
Twitter URL https://twitter.com/BankMedici								
Author Information ejboyle								
Authentication Model API Key								

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
<code>hostname:"targetname.com"</code>	Using <code>hostname</code> 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.
<code>"content-type: application/json"</code>	APIs should have their <code>content-type</code> set to JSON or XML. This query will filter results that respond with JSON.
<code>"content-type: application/xml"</code>	This query will filter results that respond with XML.
<code>"200 OK"</code>	You can add <code>"200 OK"</code> 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.
<code>"wp-json"</code>	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"
```

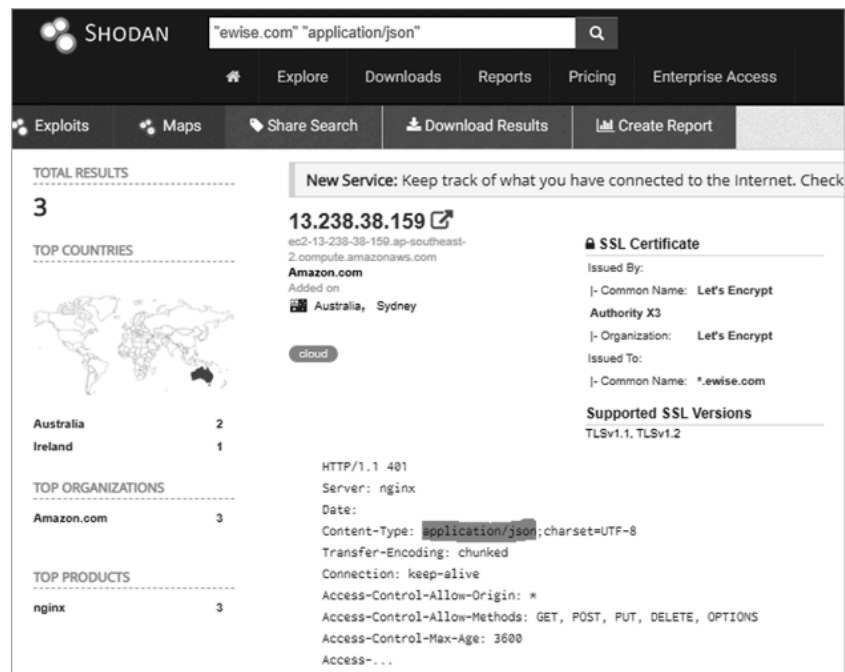


Figure 6-5: Shodan search results

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 certifi-

cate 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>
```

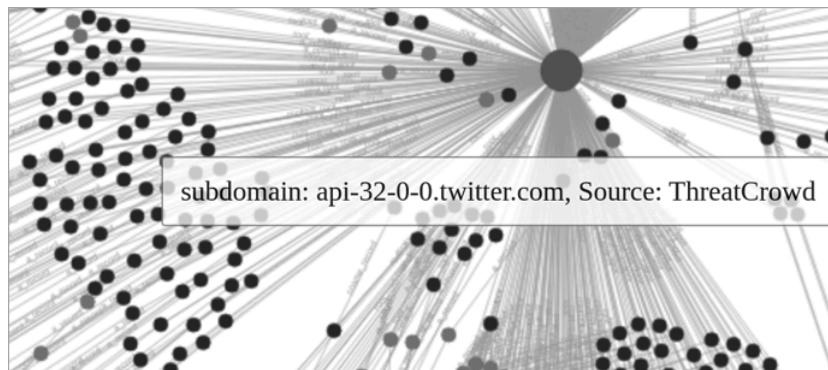


Figure 6-6: OWASP Amass visualization using `-d3` to make an HTML export of Amass findings for twitter.com

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*.

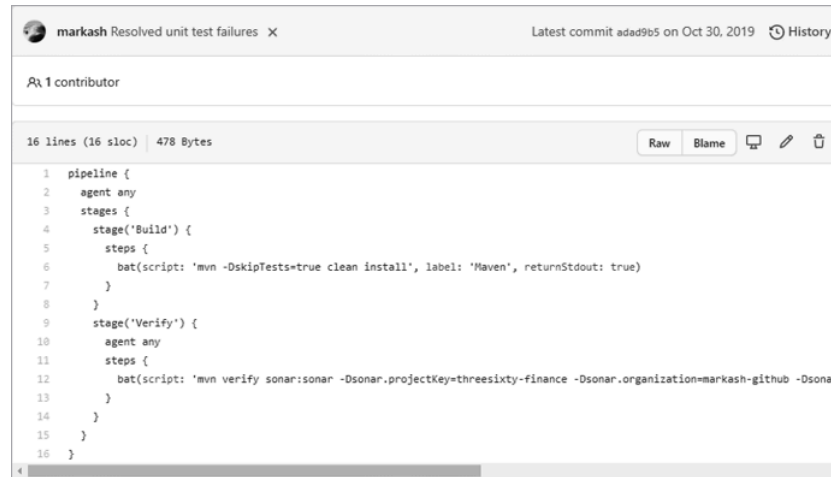
Exposed Information on GitHub

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.



The screenshot shows a GitHub commit page for a repository named 'markash'. The commit message is 'Resolved unit test failures' and the latest commit is 'adad9b5' on Oct 30, 2019. The page shows 1 contributor. The code is displayed in a monospace font, showing a pipeline configuration with stages 'Build' and 'Verify'. The 'Build' stage has a step with a bat script to run 'mvn -DskipTests=true clean install'. The 'Verify' stage has a step with a bat script to run 'mvn verify sonar:sonar' with specific project and organization keys. The code is 16 lines long, 16 sloc, and 478 bytes. There are buttons for 'Raw', 'Blame', and a 'History' link in the top right corner.

```
1 pipeline {
2   agent any
3   stages {
4     stage('Build') {
5       steps {
6         bat(script: 'mvn -DskipTests=true clean install', label: 'Maven', returnStdout: true)
7       }
8     }
9     stage('Verify') {
10      agent any
11      steps {
12        bat(script: 'mvn verify sonar:sonar -Dsonar.projectKey=threesixty-finance -Dsonar.organization=markash-github -Dsonar.'
13      )
14    }
15  }
16 }
```

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](#)).

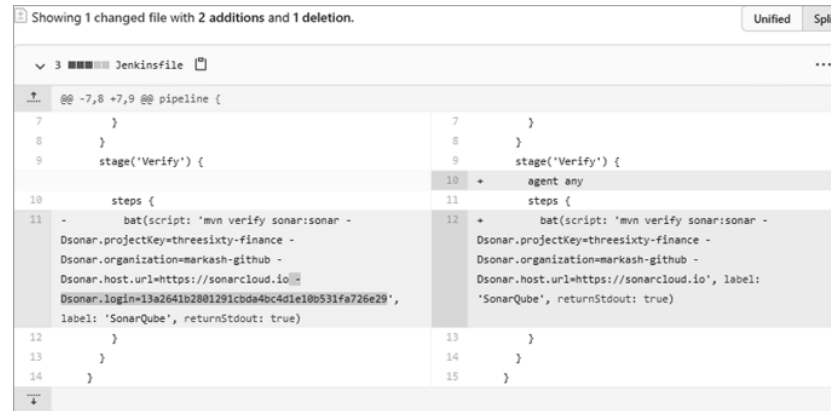


Figure 6-8: The Split button allows you to separate the previous code (left) from the updated code (right).

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**).

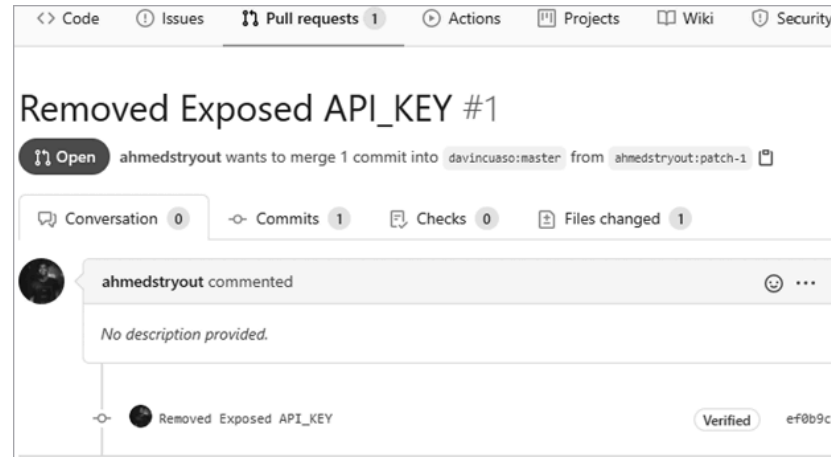


Figure 6-9: An open GitHub issue that provides the exact location of an exposed API key in the code of an application

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](#)).

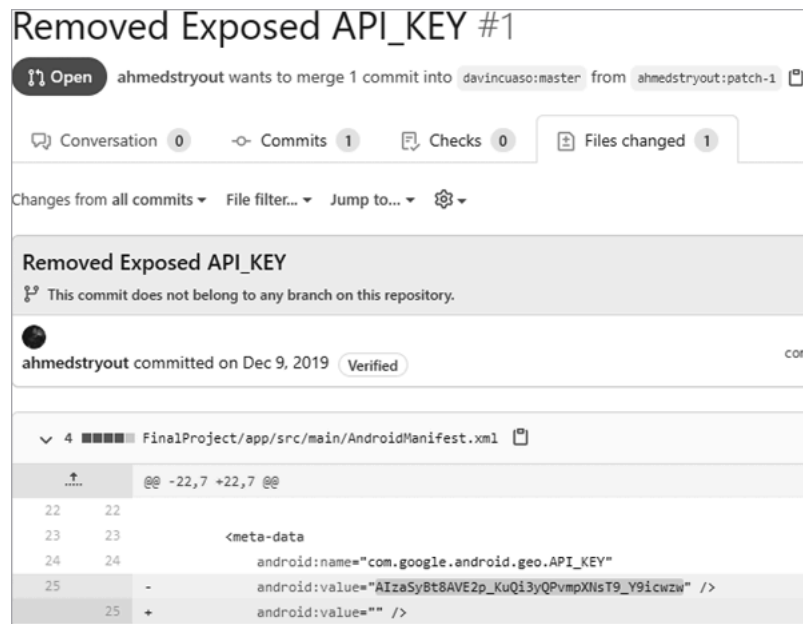


Figure 6-11: The Files Changed tab demonstrates proposed change to the code.

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 6-1 2](#)) to view their source code. Alternatively, click XHR to find see the Ajax requests being made.

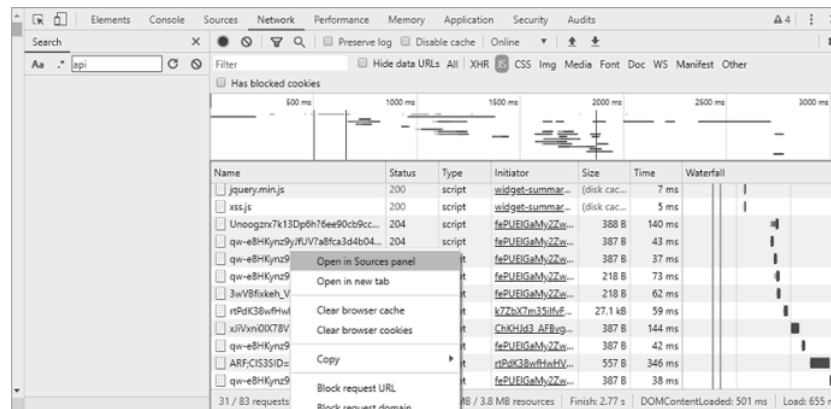


Figure 6-12: The Open in Sources panel option from the DevTools Network tab

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.

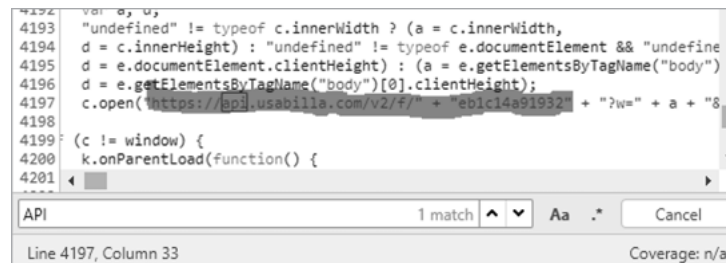


Figure 6-13: On line 4,197 of this page source, an API is in use.

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](#)).

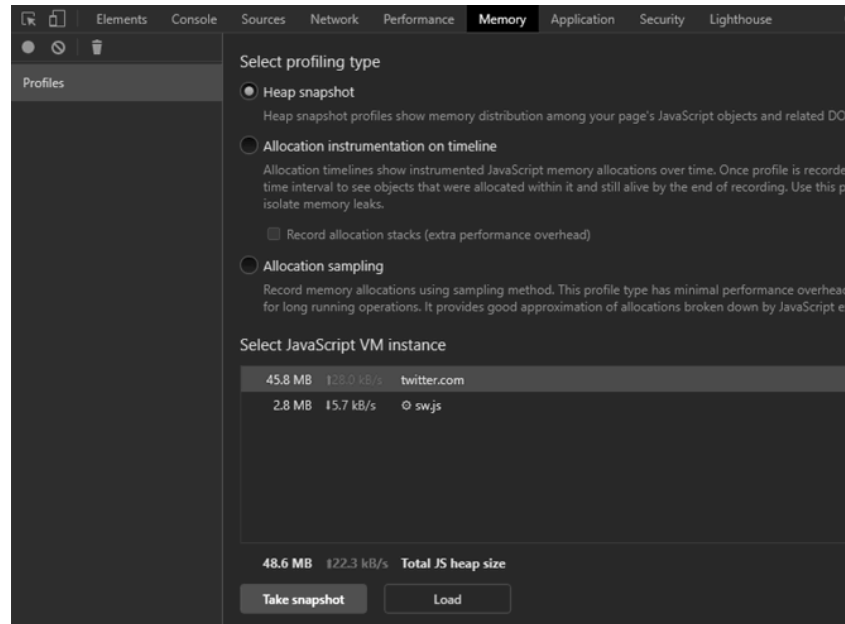


Figure 6-14: The Memory panel within DevTools

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”.

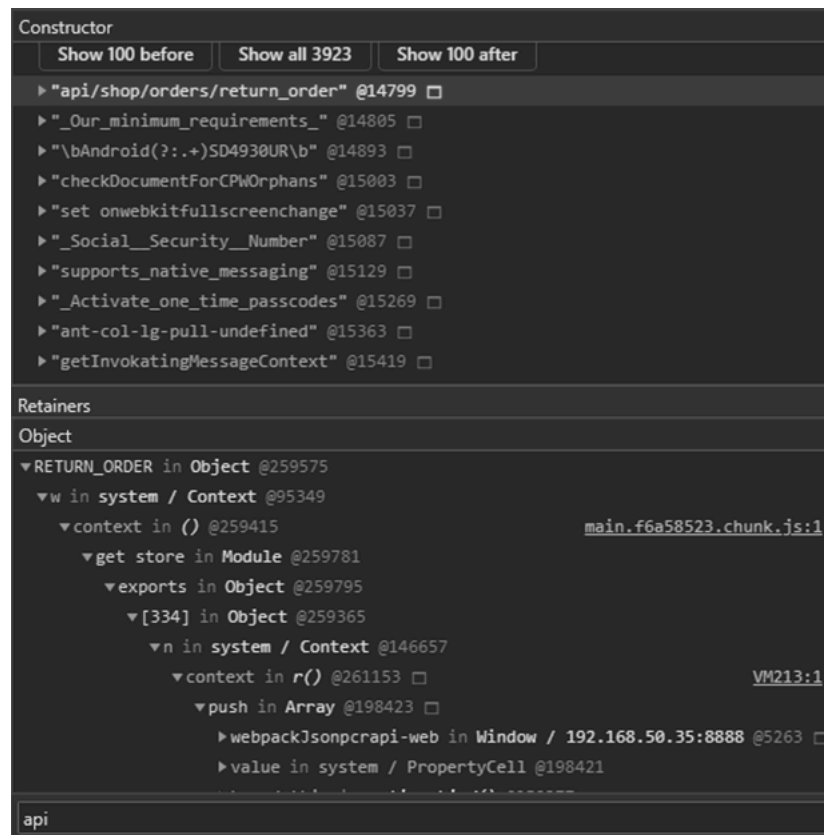


Figure 6-15: The search results from a memory snapshot

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.

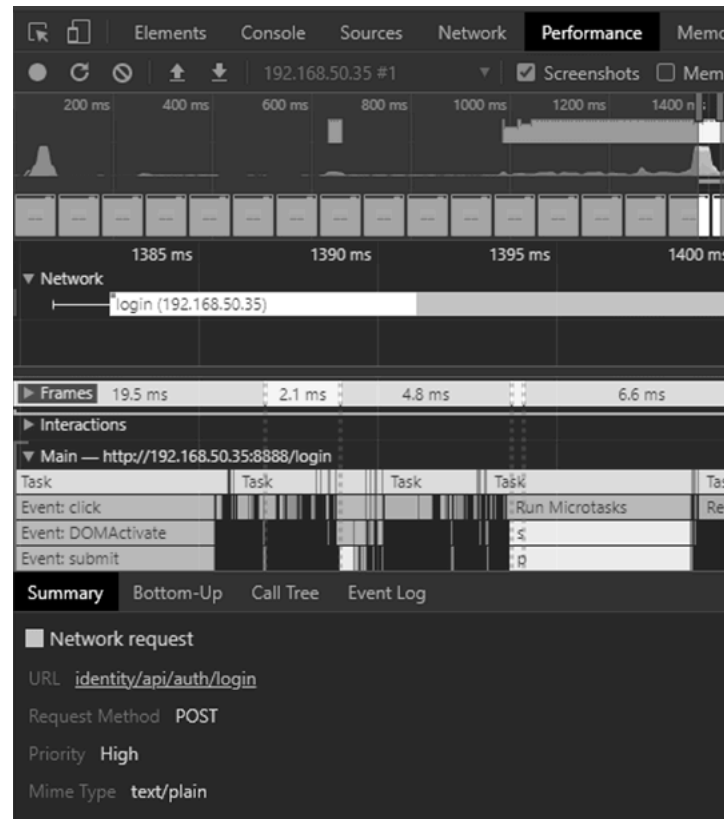


Figure 6-16: A performance recording within DevTools

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.

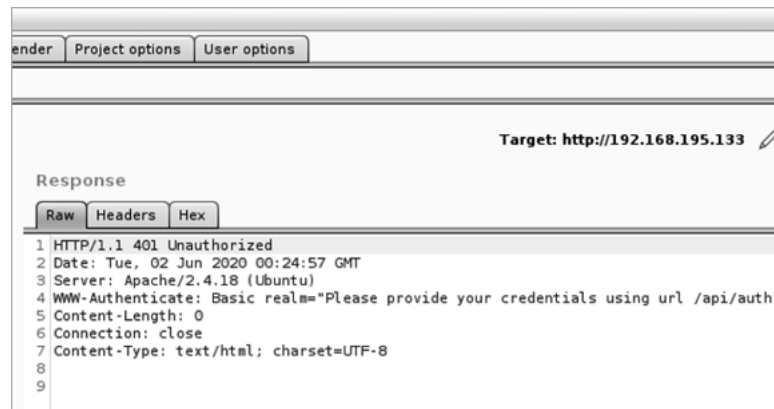
As you can see, DevTools is filled with powerful tools that can help you discover APIs. Do not underestimate the usefulness of its various modules.

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.

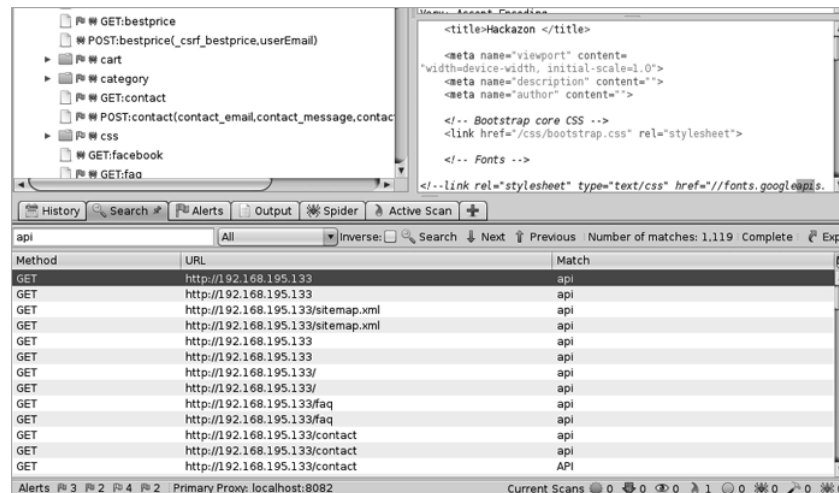


Figure 6-19: The power of searching the ZAP automated scan results for APIs

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) Chrc

```
POST 400 [ 941, 46, 11] http://192.168.195.132:8090/trade/queryTransationRecords 0cf689f783e6dab12b6940616f005ecfct
```



```
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=63578528&route=33616912 0cf681b5cf
```

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 0cf6841b1e7ac8badc6e237a
```

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.”

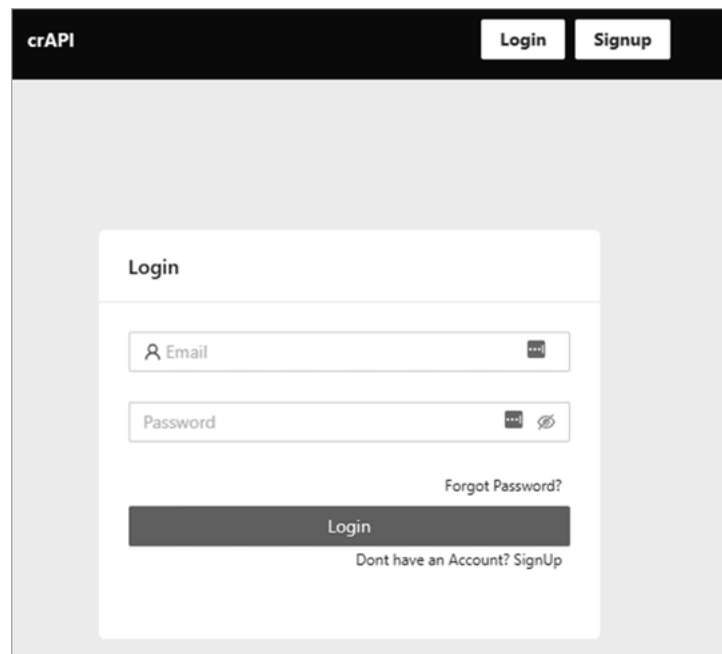
Port 8025 reveals the MailHog web server with a “welcome to crAPI” email. We will return to this later in the labs.

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.



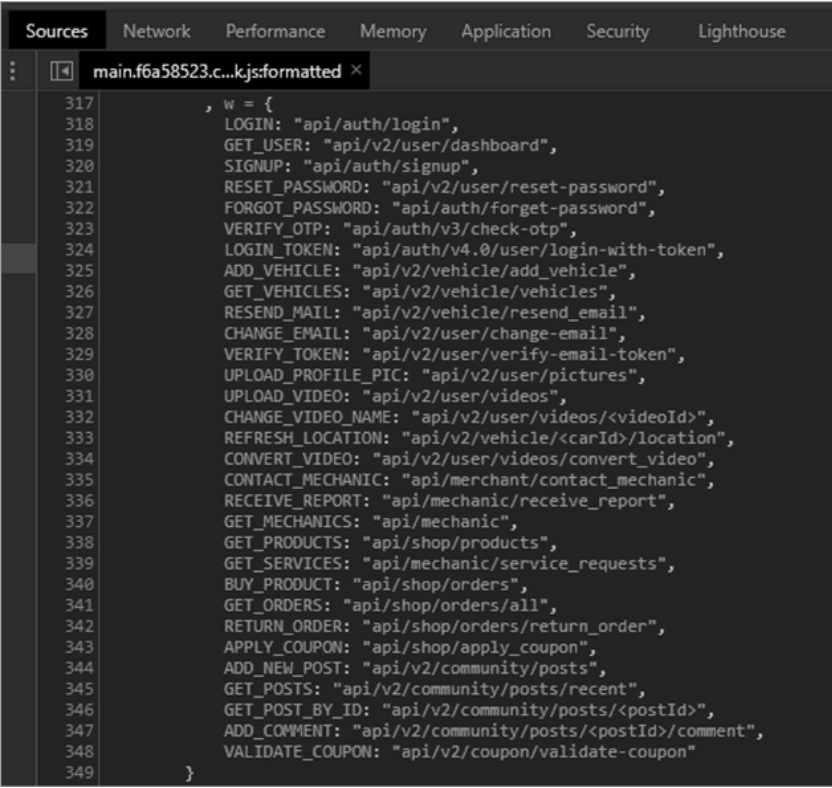
[Figure 6-22](#): The landing page for crAPI

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.



```

317     , w = {
318       LOGIN: "api/auth/login",
319       GET_USER: "api/v2/user/dashboard",
320       SIGNUP: "api/auth/signup",
321       RESET_PASSWORD: "api/v2/user/reset-password",
322       FORGOT_PASSWORD: "api/auth/forget-password",
323       VERIFY_OTP: "api/auth/v3/check-otp",
324       LOGIN_TOKEN: "api/auth/v4.0/user/login-with-token",
325       ADD_VEHICLE: "api/v2/vehicle/add_vehicle",
326       GET_VEHICLES: "api/v2/vehicle/vehicles",
327       RESEND_MAIL: "api/v2/vehicle/resend_email",
328       CHANGE_EMAIL: "api/v2/user/change-email",
329       VERIFY_TOKEN: "api/v2/user/verify-email-token",
330       UPLOAD_PROFILE_PIC: "api/v2/user/pictures",
331       UPLOAD_VIDEO: "api/v2/user/videos",
332       CHANGE_VIDEO_NAME: "api/v2/user/videos/<videoId>",
333       REFRESH_LOCATION: "api/v2/vehicle/<carId>/location",
334       CONVERT_VIDEO: "api/v2/user/videos/convert_video",
335       CONTACT_MECHANIC: "api/merchant/contact_mechanic",
336       RECEIVE_REPORT: "api/mechanic/receive_report",
337       GET_MECHANICS: "api/mechanic",
338       GET_PRODUCTS: "api/shop/products",
339       GET_SERVICES: "api/mechanic/service_requests",
340       BUY_PRODUCT: "api/shop/orders",
341       GET_ORDERS: "api/shop/orders/all",
342       RETURN_ORDER: "api/shop/orders/return_order",
343       APPLY_COUPON: "api/shop/apply_coupon",
344       ADD_NEW_POST: "api/v2/community/posts",
345       GET_POSTS: "api/v2/community/posts/recent",
346       GET_POST_BY_ID: "api/v2/community/posts/<postId>",
347       ADD_COMMENT: "api/v2/community/posts/<postId>/comment",
348       VALIDATE_COUPON: "api/v2/coupon/validate-coupon"
349     }

```

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.

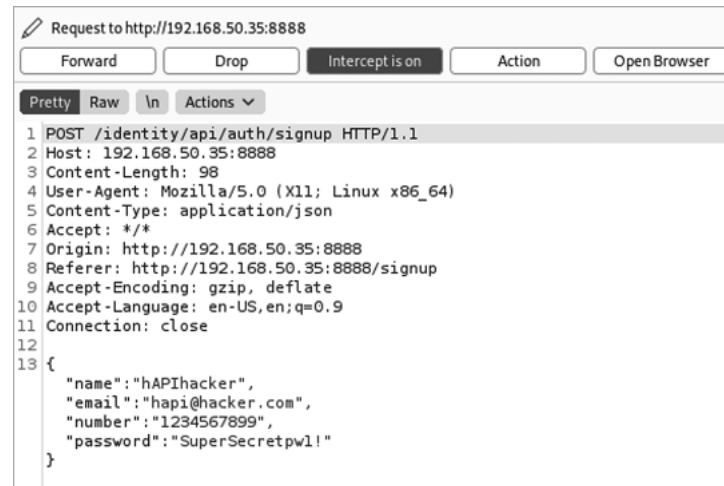


Figure 6-24: The crAPI registration request intercepted using Burp Suite

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?