

PART I

HOW WEB API SECURITY WORKS

0

PREPARING FOR YOUR SECURITY TESTS



API security testing does not quite fit into the mold of a general penetration test, nor does it fit into that of a web application penetration test. Due to the size and complexity of many organizations' API attack surfaces, API penetration testing is its own unique service. In this chapter I will discuss the features of APIs that you should include in your test and document prior to your attack. The content in this chapter will help you gauge the amount of activity required for an engagement, ensure that you plan to test all features of the target APIs, and help you avoid trouble.

API penetration testing requires a well-developed *scope*, or an account of the targets and features of what you are allowed to test, that ensures the client and tester have a mutual understanding of the work being done. Scoping an API security testing engagement comes down to a few factors: your methodology, the magnitude of the testing, the target features, any restrictions on testing, your reporting requirements, and whether you plan to conduct remediation testing.

Receiving Authorization

Before you attack APIs, it is supremely important that you receive a signed contract that includes the scope of the engagement and grants you authorization to attack the client's resources within a specific time frame.

For an API penetration test, this contract can take the form of a signed statement of work (SOW) that lists the approved targets, ensuring that you and your client agree on the service they want you to provide. This includes coming to an agreement over which aspects of an API will be tested, determining any exclusions, and setting up an agreed-upon time to perform testing.

Double-check that the person signing the contract is a representative of the target client who is in a position to authorize testing. Also make sure the assets to be tested are owned by the client; otherwise, you will need to rinse and repeat these instructions with the proper owner. Remember to take into consideration the location where the client is hosting their APIs and whether they are truly in a position to authorize testing against both the software and the hardware.

Some organizations can be too restrictive with their scoping documentation. If you have the opportunity to develop the scope, I recommend that, in your own calm words, you kindly explain to your clients that the criminals have no scope or limitations. Real criminals do not consider other projects that are consuming IT resources; they do not avoid the subnet with sensitive production servers or care about hacking at inconvenient times of day. Make an effort to convince your client of the value of having a less-restrictive engagement and then work with them to document the particulars.

Meet with the client, spell out exactly what is going to happen, and then document it exactly in the contract, reminder emails, or notes. If you stick to the documented agreement for the services requested, you should be operating legally and ethically. However, it is probably worth reducing your risk by consulting with a lawyer or your legal department.

Threat Modeling an API Test

Threat modeling is the process used to map out the threats to an API provider. If you model an API penetration test based on a relevant threat, you'll be able to choose tools and techniques directed at that attack. The best tests of an API will be those that align with actual threats to the API provider.

A *threat actor* is the adversary or attacker of the API. The adversary can be anyone, from a member of the public who stumbles upon the API with little to no knowledge of the application to a customer using the application, a rogue business partner, or an insider who knows quite a bit about the application. To perform a test that provides the most value to the security of the API, it is ideal to map out the probable adversary as well as their hacking techniques.

Your testing method should follow directly from the threat actor's perspective, as this perspective should determine the information you are

given about your target. If the threat actor knows nothing about the API, they will need to perform research to determine the ways in which they might target the application. However, a rogue business partner or insider threat may know quite a bit about the application already without any reconnaissance. To address these distinctions, there are three basic penetration testing approaches: black box, gray box, and white box.

Black box testing models the threat of an opportunistic attacker—someone who may have stumbled across the target organization or its API. In a truly black box API engagement, the client would not disclose any information about their attack surface to the tester. You will likely start your engagement with nothing more than the name of the company that signed the SOW. From there, the testing effort will involve conducting reconnaissance using open-source intelligence (OSINT) to learn as much about the target organization as possible. You might uncover the target's attack surface by using a combination of search engine research, social media, public financial records, and DNS information to learn as much as you can about the organization's domain. The tools and techniques for this approach are covered in much more detail in Chapter 6. Once you've conducted OSINT, you should have compiled a list of target IP addresses, URLs, and API endpoints that you can present to the client for review. The client should look at your target list and then authorize testing.

A gray box test is a more informed engagement that seeks to reallocate time spent on reconnaissance and instead invest it in active testing. When performing a gray box test, you'll mimic a better-informed attacker. You will be provided information such as which targets are in and out of scope as well as access to API documentation and perhaps a basic user account. You might also be allowed to bypass certain network perimeter security controls.

Bug bounty programs often fall somewhere on the spectrum between black box and gray box testing. A bug bounty program is an engagement where a company allows hackers to test its web applications for vulnerabilities, and successful findings result in the host company providing a bounty payment to the finder. Bug bounties aren't entirely "black box" because the bounty hunter is provided with approved targets, targets that are out of scope, types of vulnerabilities that are rewarded, and allowed types of attacks. With these restrictions in place, bug bounty hunters are only limited by their own resources, so they decide how much time is spent on reconnaissance in comparison to other techniques. If you are interested in learning more about bug bounty hunting, I highly recommend Vickie Li's *Bug Bounty Bootcamp* (<https://nostarch.com/bug-bounty-bootcamp>).

In a white box approach, the client discloses as much information as possible about the inner workings of their environment. In addition to the information provided for gray box testing, this might include access to application source code, design information, the software development kit (SDK) used to develop the application, and more. White box testing models the threat of an inside attacker—someone who knows the inner workings of the organization and has access to the actual source code. The more information you are provided in a white box engagement, the more thoroughly the target will be tested.

The customer's decision to make the engagement white box, black box, or somewhere in between should be based on a threat model and threat intelligence. Using threat modeling, work with your customer to profile the organization's likeliest attacker. For example, say you're working with a small business that is politically inconsequential; it isn't part of a supply chain for a more important company and doesn't provide an essential service. In that case, it would be absurd to assume that the organization's adversary is a well-funded advanced persistent threat (APT) like a nation-state. Using the techniques of an APT against this small business would be like using a drone strike on a petty thief. Instead, to provide the client with the most value, you should use threat modeling to craft a realistic threat. In this case, the likeliest attacker might be an opportunistic, medium-skilled individual who has stumbled upon the organization's website and is likely to run only published exploits against known vulnerabilities. The testing method that fits the opportunistic attacker would be a limited black box test.

The most effective way to model a threat for a client is to conduct a survey with them. The survey will need to reveal the client's scope of exposure to attacks, their economic significance, their political involvement, whether they are involved in any supply chains, whether they offer essential services, and whether there are other potential motives for a criminal to want to attack them. You can develop your own survey or put one together from existing professional resources like MITRE ATT&CK (<https://attack.mitre.org>) or OWASP (https://cheatsheetseries.owasp.org/cheatsheets/Threat_Modeling_Cheat_Sheet.html).

The testing method you select will determine much of the remaining scoping effort. Since black box testers are provided with very little information about scoping, the remaining scoping items are relevant for gray box and white box testing.

Which API Features You Should Test

One of the main goals of scoping an API security engagement is to discover the quantity of work you'll have to do as part of your test. As such, you must find out how many unique API endpoints, methods, versions, features, authentication and authorization mechanisms, and privilege levels you'll need to test. The magnitude of the testing can be determined through interviews with the client, a review of the relevant API documentation, and access to API collections. Once you have the requested information, you should be able to gauge how many hours it will take to effectively test the client's APIs.

API Authenticated Testing

Determine how the client wants to handle the testing of authenticated and unauthenticated users. The client may want to have you test different API users and roles to see if there are vulnerabilities present in any of the different privilege levels. The client may also want you to test a process they use for authentication and the authorization of users. When it comes to

API weaknesses, many of the detrimental vulnerabilities are discovered in authentication and authorization. In a black box situation, you would need to figure out the target's authentication process and seek to become authenticated.

Web Application Firewalls

In a white box engagement, you will want to be aware of any web application firewalls (WAFs) that may be in use. A WAF is a common defense mechanism for web applications and APIs. A WAF is a device that controls the network traffic that reaches the API. If a WAF has been set up properly, you will find out quickly during testing when access to the API is lost after performing a simple scan. WAFs can be great at limiting unexpected requests and stopping an API security test in its tracks. An effective WAF will detect the frequency of requests or request failures and ban your testing device.

In gray box and white box engagements, the client will likely reveal the WAF to you, at which point you will have some decisions to make. While opinions diverge on whether organizations should relax security for the sake of making testing more effective, a layered cybersecurity defense is key to effectively protecting organizations. In other words, no one should put all their eggs into the WAF basket. Given enough time, a persistent attacker could learn the boundaries of the WAF, figure out how to bypass it, or use a zero-day vulnerability that renders it irrelevant.

Ideally, the client would allow your attacking IP address to bypass the WAF or adjust their typical level of boundary security so that you can test the security controls that will be exposed to their API consumers. As discussed earlier, making plans and decisions like this is really about threat modeling. The best tests of an API will be those that align with actual threats to the API provider. To get a test that provides the most value to the security of the API, it is ideal to map out the probable adversary and their hacking techniques. Otherwise, you'll find yourself testing the effectiveness of the API provider's WAF rather than the effectiveness of their API security controls.

Mobile Application Testing

Many organizations have mobile applications that expand the attack surface. Moreover, mobile apps often rely on APIs to transmit data within the application and to supporting servers. You can test these APIs through manual code review, automated source code analysis, and dynamic analysis. *Manual* code review involves accessing the mobile application's source code and searching for potential vulnerabilities. *Automated* source code analysis is similar, except it uses automated tools to assist in the search for vulnerabilities and interesting artifacts. Finally, *dynamic* analysis is the testing of the application while it is running. Dynamic analysis includes intercepting the mobile app's client API requests and the server API responses and then attempting to find weaknesses that can be exploited.

Auditing API Documentation

An API's *documentation* is a manual that describes how to use the API and includes authentication requirements, user roles, usage examples, and API endpoint information. Good documentation is essential to the commercial success of any self-sufficient API. Without effective API documentation, businesses would have to rely on training to support their consumers. For these reasons, you can bet that your target APIs have documentation.

Yet, this documentation can be riddled with inaccuracies, outdated information, and information disclosure vulnerabilities. As an API hacker, you should search for your target's API documentation and use it to your advantage. In gray box and white box testing, an API documentation audit should be included within the scope. A review of the documentation will improve the security of the target APIs by exposing weaknesses, including business logic flaws.

Rate Limit Testing

Rate limiting is a restriction on the number of requests an API consumer can make within a given time frame. It is enforced by an API provider's web servers, firewall, or web application firewall and serves two important purposes for API providers: it allows for the monetization of APIs and prevents the overconsumption of the provider's resources. Because rate limiting is an essential factor that allows organizations to monetize their APIs, you should include it in your scope during API engagements.

For example, a business might allow a free-tier API user to make one request per hour. Once that request is made, the consumer would be kept from making any other request for an hour. However, if the user pays this business a fee, they could make hundreds of requests per hour. Without adequate controls in place, these non-paying API consumers could find ways to skip the toll and consume as much data as often as they please.

Rate limit testing is not the same as denial of service (DoS) testing. DoS testing consists of attacks that are intended to disrupt services and make the systems and applications unavailable to users. Whereas DoS testing is meant to assess how resilient an organization's computing resources are, rate limit testing seeks to bypass restrictions that limit the quantity of requests sent within a given time frame. Attempting to bypass rate limiting will not necessarily cause a disruption to services. Instead, bypassing rate limiting could aid in other attacks and demonstrate a weakness in an organization's method of monetizing its API.

Typically, an organization publishes its API's request limits in the API documentation. It will read something like the following:

You may make *X* requests within a *Y* time frame. If you exceed this limit, you will get a *Z* response from our web server.

Twitter, for example, limits requests based on your authorization once you're authenticated. The first tier can make 15 requests every 15 minutes, and the next tier can make 180 requests every 15 minutes. If you exceed your request limit, you will be sent an HTTP Error 420, as shown in Figure 0-1.



Figure 0-1: Twitter HTTP status code from <https://developer.twitter.com/en/docs>

If insufficient security controls are in place to limit access to an API, the API provider will lose money from consumers cheating the system, incur additional costs due to the use of additional host resources, and find themselves vulnerable to DoS attacks.

Restrictions and Exclusions

Unless otherwise specified in penetration testing authorization documentation, you should assume that you won't be performing DoS and distributed DoS (DDoS) attacks. In my experience, being authorized to do so is pretty rare. When DoS testing is authorized, it is clearly spelled out in formal documentation. Also, with the exception of certain adversary emulation engagements, penetration testing and social engineering are typically kept as separate exercises. That being said, always check whether you can use social engineering attacks (such as phishing, vishing, and smishing) when penetration testing.

By default, no bug bounty program accepts attempts at social engineering, DoS or DDoS attacks, attacks of customers, and access of customer data. In situations where you could perform an attack against a user, programs normally suggest creating multiple accounts and, when the relevant opportunity arises, attacking your own test accounts.

Additionally, particular programs or clients may spell out known issues. Certain aspects of an API might be considered a security finding but may also be an intended convenience feature. For example, a forgot-your-password function could display a message that lets the end user know whether their email or password is incorrect; this same function could grant an attacker the ability to brute-force valid usernames and emails. The organization may have already decided to accept this risk and does not wish for you to test it.

Pay close attention to any exclusions or restrictions in the contract. When it comes to APIs, the program may allow for testing of specific sections of a given API and may restrict certain paths within an approved API. For example, a banking API provider may share resources with a third party and may not have authorization to allow testing. Thus, they may spell out that you can attack the `/api/accounts` endpoint but not `/api/shared/accounts`. Alternatively, the target's authentication process may be through a third party that you are not authorized to attack. You will need to pay close attention to the scope in order to perform legal authorized testing.

Security Testing Cloud APIs

Modern web applications are often hosted in the cloud. When you attack a cloud-hosted web application, you're actually attacking the physical servers of cloud providers (likely Amazon, Google, or Microsoft). Each cloud provider has its own set of penetration testing terms and services that you'll want to become familiar with. As of 2021, cloud providers have generally become friendlier toward penetration testers, and far fewer of them require authorization submissions. Still, some cloud-hosted web applications and APIs will require you to obtain penetration testing authorization, such as for an organization's Salesforce APIs.

You should always know the current requirements of the target cloud provider before attacking. The following list describes the policies of the most common providers.

Amazon Web Services (AWS) AWS has greatly improved its stance on penetration testing. As of this writing, AWS allows its customers to perform all sorts of security testing, with the exception of DNS zone walking, DoS or DDoS attacks, simulated DoS or DDoS attacks, port flooding, protocol flooding, and request flooding. For any exceptions to this, you must email AWS and request permission to conduct testing. If you are requesting an exception, make sure to include your testing dates, any accounts and assets involved, your phone number, and a description of your proposed attack.

Google Cloud Platform (GCP) Google simply states that you do not need to request permission or notify the company to perform penetration testing. However, Google also states that you must remain compliant with its acceptable use policy (AUP) and terms of service (TOS) and stay within your authorized scope. The AUP and TOS prohibit illegal actions, phishing, spam, distributing malicious or destructive files (such as viruses, worms, and Trojan horses), and interruption to GCP services.

Microsoft Azure Microsoft takes the hacker-friendly approach and does not require you to notify the company before testing. In addition, it has a "Penetration Testing Rules of Engagement" page that spells out exactly what sort of penetration testing is permitted (<https://www.microsoft.com/en-us/msrc/pentest-rules-of-engagement>).

At least for now, cloud providers are taking a favorable stance toward penetration testing activities. As long as you stay up-to-date with the provider's terms, you should be operating legally if you only test targets you are authorized to hack and avoid attacks that could cause an interruption to services.

DoS Testing

I mentioned that DoS attacks are often off the table. Work with the client to understand their risk appetite for the given engagement. You should

treat DOS testing as an opt-in service for clients who want to test the performance and reliability of their infrastructure. Otherwise, work with the customer to see what they're willing to allow.

DoS attacks represent a huge threat against the security of APIs. An intentional or accidental DoS attack will disrupt the services provided by the target organization, making the API or web application inaccessible. An unplanned business interruption like this is usually a triggering factor for an organization to pursue legal recourse. Therefore, be careful to perform only the testing that you are authorized to perform!

Ultimately, whether a client accepts DoS testing as part of the scope depends on the organization's *risk appetite*, or the amount of risk an organization is willing to take on to achieve its purpose. Understanding an organization's risk appetite can help you tailor your testing. If an organization is cutting-edge and has a lot of confidence in its security, it may have a big appetite for risk. An engagement tailored to a large appetite for risk would involve connecting to every feature and running all the exploits you want. On the opposite side of the spectrum are the very risk-averse organizations. Engagements for these organizations will be like walking on eggshells. This sort of engagement will have many details in the scope: any machine you are able to attack will be spelled out, and you may need to ask permission before running certain exploits.

Reporting and Remediation Testing

To your client, the most valuable aspect of your testing is the report you submit to communicate your findings about the effectiveness of their API security controls. The report should spell out the vulnerabilities you discovered during your testing and explain to the client how they can perform remediation to improve the security of their APIs.

The final thing to check when scoping is whether the API provider would like remediation testing. Once the client has their report, they should attempt to fix their API vulnerabilities. Performing a retest of the previous findings will validate that the vulnerabilities were successfully remediated. Retesting could probe exclusively the weak spots, or it could be a full retest to see if any changes applied to the API introduced new weaknesses.

A Note on Bug Bounty Scope

If you hope to hack professionally, a great way to get your foot in the door is to become a bug bounty hunter. Organizations like BugCrowd and HackerOne have created platforms that make it easy for anyone to make an account and start hunting. In addition, many organizations run their own bug bounty programs, including Google, Microsoft, Apple, Twitter, and GitHub. These programs include plenty of API bug bounties, many of which have additional incentives. For example, the Files.com bug bounty program hosted on BugCrowd includes API-specific bounties, as shown in Figure 0-2.

Considering the higher business impact of issues affecting the following targets, we are offering a 10% bonus on valid submissions (severity P2-P4) for them:

- app.files.com
- your-assigned-subdomain.files.com
- REST API

Target	P1	P2	P3	P4
your-assigned-subdomain.files.com	up to \$10,000	\$2,500	\$500	\$100
Files.com Desktop Application for Windows or Mac	up to \$2,000	\$1,000	\$200	\$100
app.files.com	up to \$10,000	\$2,500	\$500	\$100
www.files.com	up to \$2,000	\$1,000	\$200	\$100
Files.com REST API	up to \$10,000	\$2,500	\$500	\$100

Figure 0-2: The Files.com bug bounty program on BugCrowd, one of many to incentivize API-related findings

In bug bounty programs, you should pay attention to two contracts: the terms of service for the bug bounty provider and the scope of the program. Violating either of these contracts could result not only in getting banned from the bug bounty provider but legal trouble as well. The bounty provider's terms of service will contain important information about earning bounties, reporting findings, and the relationship between the bounty provider, testers, researchers, and hackers who participate and the target.

The scope will equip you with the target APIs, descriptions, reward amounts, rules of engagement, reporting requirements, and restrictions. For API bug bounties, the scope will often include the API documentation or a link to the docs. Table 0-1 lists some of the primary bug bounty considerations you should understand before testing.

Table 0-1: Bug Bounty Testing Considerations

Targets	URLs that are approved for testing and rewards. Pay attention to the subdomains listed, as some may be out of scope.
Disclosure terms	The rules regarding your ability to publish your findings.
Exclusions	URLs that are excluded from testing and rewards.
Testing restrictions	Restrictions on the types of vulnerabilities the organization will reward. Often, you must be able to prove that your finding can be leveraged in a real-world attack by providing evidence of exploitation.
Legal	Additional government regulations and laws that apply due to the organization's, customers', and data center's locations.

If you are new to bug hunting, I recommend checking out BugCrowd University, which has an introduction video and page dedicated to API security testing by Sadako (<https://www.bugcrowd.com/resources/webinars/api-security-testing-for-hackers>). Also, check out *Bug Bounty Bootcamp* (No

Starch Press, 2021), which is one of the best resources out there to get you started in bug bounties. It even has a chapter on API hacking!

Make sure you understand the potential rewards, if any, of each type of vulnerability before you spend time and effort on it. For example, I've seen bug bounties claimed for a valid exploitation of rate limiting that the bug bounty host considered spam. Review past disclosure submissions to see if the organization was combative or unwilling to pay out for what seemed like valid submissions. In addition, focus on the successful submissions that received bounties. What type of evidence did the bug hunter provide, and how did they report their finding in a way that made it easy for the organization to see the bug as valid?

Summary

In this chapter, I reviewed the components of the API security testing scope. Developing the scope of an API engagement should help you understand the method of testing to deploy as well as the magnitude of the engagement. You should also reach an understanding of what can and can't be tested as well as what tools and techniques will be used in the engagement. If the testing aspects have been clearly spelled out and you test within those specifications, you'll be set up for a successful API security testing engagement.

In the next chapter, I will cover the web application functionality you will need to understand in order to know how web APIs work. If you already understand web application basics, move on to Chapter 2, where I cover the technical anatomy of APIs.

1

HOW WEB APPLICATIONS WORK



Before you can hack APIs, you must understand the technologies that support them. In this chapter, I will cover everything you need to know about web applications, including the fundamental aspects of HyperText Transfer Protocol (HTTP), authentication and authorization, and common web server databases. Because web APIs are powered by these technologies, understanding these basics will prepare you for using and hacking APIs.

Web App Basics

Web applications function based on the client/server model: your web browser, the client, generates requests for resources and sends these to computers called web servers. In turn, these web servers send resources to

the clients over a network. The term *web application* refers to software that is running on a web server, such as Wikipedia, LinkedIn, Twitter, Gmail, GitHub, and Reddit.

In particular, web applications are designed for end-user interactivity. Whereas websites are typically read-only and provide one-way communication from the web server to the client, web applications allow communications to flow in both directions, from server to client and from client to server. Reddit, for example, is a web app that acts as a newsfeed of information flowing around the internet. If it were merely a website, visitors would be spoon-fed whatever content the organization behind the site provided. Instead, Reddit allows users to interact with the information on the site by posting, upvoting, downvoting, commenting, sharing, reporting bad posts, and customizing their newsfeeds with subreddits they want to see. These features differentiate Reddit from a static website.

For an end user to begin using a web application, a conversation must take place between the web browser and a web server. The end user initiates this conversation by entering a URL into their browser address bar. In this section, we'll discuss what happens next.

The URL

You probably already know that the *uniform resource locator (URL)* is the address used to locate unique resources on the internet. This URL consists of several components that you'll find helpful to understand when crafting API requests in later chapters. All URLs include the protocol used, the hostname, the port, the path, and any query parameters:

Protocol://hostname[:port number]/[path]/[?query][parameters]

Protocols are the sets of rules computers use to communicate. The primary protocols used within the URL are HTTP/HTTPS for web pages and FTP for file transfers.

The *port*, a number that specifies a communication channel, is only included if the host does not automatically resolve the request to the proper port. Typically, HTTP communications take place over port 80. HTTPS, the encrypted version of HTTP, uses port 443, and FTP uses port 21. To access a web app that is hosted on a nonstandard port, you can include the port number in the URL, like so: *https://www.example.com:8443*. (Ports 8080 and 8443 are common alternatives for HTTP and HTTPS, respectively.)

The file directory *path* on the web server points to the location of the web pages and files specified in the URL. The path used in a URL is the same as a filepath used to locate files on a computer.

The *query* is an optional part of the URL used to perform functionality such as searching, filtering, and translating the language of the requested information. The web application provider may also use the query strings to track certain information such as the URL that referred you to the web page, your session ID, or your email. It starts with a question mark and contains a string that the server is programmed to process. Finally, the *query parameters* are the values that describe what should be done with the given query. For example, the query parameter *lang=en* following the query page?

might indicate to the web server that it should provide the requested page in English. These parameters consist of another string to be processed by the web server. A query can contain multiple parameters separated by an ampersand (&).

To make this information more concrete, consider the URL `https://twitter.com/search?q=hacking&src=typed_query`. In this example, the protocol is `https`, the hostname is `twitter.com`, the path is `search`, the query is `?q` (which stands for query), the query parameter is `hacking`, and `src=typed_query` is a tracking parameter. This URL is automatically built whenever you click the search bar in the Twitter web app, type in the search term “hacking,” and press ENTER. The browser is programmed to form the URL in a way that will be understood by the Twitter web server, and it collects some tracking information in the form of the `src` parameter. The web server will receive the request for hacking content and respond with hacking-related information.

HTTP Requests

When an end user navigates to a URL using a web browser, the browser automatically generates an HTTP *request* for a resource. This resource is the information being requested—typically the files that make up a web page. The request is routed across the internet or network to the web server, where it is initially processed. If the request is properly formed, the web server passes the request to the web application.

Listing 1-1 shows the components of an HTTP request sent when authenticating to `twitter.com`.

```
POST ❶ /sessions ❷ HTTP/1.1 ❸
Host: twitter.com ❹
User-Agent: Mozilla/5.0 (X11; Linux x86_64; rv:78.0) Gecko/20100101 Firefox/78.0
Accept: text/html,application/xhtml+xml,application/xml;q=0.9,image/webp,*/*;q=0.8
Accept-Language: en-US,en;q=0.5
Accept-Encoding: gzip, deflate
Content-Type: application/x-www-form-urlencoded
Content-Length: 444
Cookie: _personalization_id=GA1.2.1451399206.1606701545; dnt=1;

username_or_email%5D=hAPI_hacker& ❺ password%5D=NotMyPassword ❻ %21 ❼
```

Listing 1-1: An HTTP request to authenticate with `twitter.com`

HTTP requests start with the method ❶, the path of the requested resource ❷, and the protocol version ❸. The method, described in the “HTTP Methods” section later in this chapter, tells the server what you want to do. In this case, you use the POST method to send your login credentials to the server. The path may contain either the entire URL, the absolute path, or the relative path of a resource. In this request, the path, `/sessions`, specifies the page that handles Twitter authentication requests.

Requests include several *headers*, which are key-value pairs that communicate specific information between the client and the web server. Headers begin with the header’s name, followed by a colon (:) and then the value

of the header. The Host header ❹ designates the domain host, *twitter.com*. The User-Agent header describes the client's browser and operating system. The Accept headers describe which types of content the browser can accept from the web application in a response. Not all headers are required, and the client and server may include others not shown here, depending on the request. For example, this request includes a Cookie header, which is used between the client and server to establish a stateful connection (more on this later in the chapter). If you'd like to learn more about all the different headers, check out Mozilla's developer page on headers (<https://developer.mozilla.org/en-US/docs/Web/HTTP/Headers>).

Anything below the headers is the *message body*, which is the information that the requestor is attempting to have processed by the web application. In this case, the body consists of the username ❺ and password ❻ used to authenticate to a Twitter account. Certain characters in the body are automatically encoded. For example, exclamation marks (!) are encoded as %21 ❼. Encoding characters is one way that a web application may securely handle characters that could cause problems.

HTTP Responses

After a web server receives an HTTP request, it will process and respond to the request. The type of response depends on the availability of the resource, the user's authorization to access the resource, the health of the web server, and other factors. For example, Listing 1-2 shows the response to the request in Listing 1-1.

```
HTTP/1.1❶ 302 Found❷
content-security-policy: default-src 'none'; connect-src 'self'
location: https://twitter.com/
pragma: no-cache
server: tsa_a
set-cookie: auth_token=8ff3f2424f8ac1c4ec635b4adb52cddf28ec18b8; Max-Age=157680000;
Expires=Mon, 01 Dec 2025 16:42:40 GMT; Path=/; Domain=.twitter.com; Secure; HTTPOnly;
SameSite=None

<html><body>You are being <a href="https://twitter.com/">redirected</a>.</body></html>
```

Listing 1-2: An example of an HTTP response when authenticating to twitter.com

The web server first responds with the protocol version in use (in this case, HTTP/1.1 ❶). HTTP 1.1 is currently the standard version of HTTP used. The status code and status message ❷, discussed in more detail in the next section, are 302 Found. The 302 response code indicates that the client successfully authenticated and will be redirected to a landing page the client is authorized to access.

Notice that, like HTTP request headers, there are HTTP response headers. HTTP response headers often provide the browser with instructions for handling the response and security requirements. The set-cookie header is another indication that the authentication request was successful, because the web server has issued a cookie that includes an `auth_token`,

which the client can use to access certain resources. The response message body will follow the empty line after the response headers. In this case, the web server has sent an HTML message indicating that the client is being redirected to a new web page.

The request and response I've shown here illustrates a common way in which a web application restricts access to its resources through the use of authentication and authorization. Web *authentication* is the process of proving your identity to a web server. Common forms of authentication include providing a password, token, or biometric information (such as a fingerprint). If a web server approves an authentication request, it will respond by providing the authenticated user *authorization* to access certain resources. In Listing 1-1, we saw an authentication request to a Twitter web server that sent a username and password using a POST request. The Twitter web server responded to the successful authentication request with 302 Found (in Listing 1-2). The session `auth_token` in the `set-cookie` header authorized access to the resources associated with the `hAPI_hacker` Twitter account.

NOTE

HTTP traffic is sent in cleartext, meaning it's not hidden or encrypted in any way. Anyone who intercepted the authentication request in Listing 1-1 could read the username and password. To protect sensitive information, HTTP protocol requests can be encrypted with Transport Layer Security (TLS) to create the HTTPS protocol.

HTTP Status Codes

When a web server responds to a request, it issues a response status code, along with a response message. The response code signals how the web server has handled the request. At a high level, the response code determines if the client will be allowed or denied access to a resource. It can also indicate that a resource does not exist, there is a problem with the web server, or requesting the given resource has resulted in being redirected to another location.

Listings 1-3 and 1-4 illustrate the difference between a 200 response and a 404 response, respectively.

```
HTTP/1.1 200 OK
Server: tsa_a
Content-length: 6552
```

```
<!DOCTYPE html>
<html dir="ltr" lang="en">
[...]
```

Listing 1-3: An example of a 200 response

```
HTTP/1.1 404 Not Found
Server: tsa_a
Content-length: 0
```

Listing 1-4: An example of a 404 response

The 200 OK response will provide the client with access to the requested resource, whereas the 404 Not Found response will either provide the client with some sort of error page or a blank page, because the requested resource was not found.

Since web APIs primarily function using HTTP, it is important to understand the sorts of response codes you should expect to receive from a web server, as detailed in Table 1-1. For more information about individual response codes or about web technologies in general, check out Mozilla's Web Docs (<https://developer.mozilla.org/en-US/docs/Web/HTTP>). Mozilla has provided a ton of useful information about the anatomy of web applications.

Table 1-1: HTTP Response Code Ranges

Response code	Response type	Description
100s	Information-based responses	Responses in the 100s are typically related to some sort of processing status update regarding the request.
200s	Successful responses	Responses in the 200s indicate a successful and accepted request.
300s	Redirects	Responses in the 300s are notifications of redirection. This is common to see for a request that automatically redirects you to the index/home page or when you request a page from port 80 HTTP to port 443 for HTTPS.
400s	Client errors	Responses in the 400s indicate that something has gone wrong from the client perspective. This is often the type of response you will receive if you have requested a page that does not exist, if there is a timeout in the response, or when you are forbidden from viewing the page.
500s	Server errors	Responses in the 500s are indications that something has gone wrong with the server. These include internal server errors, unavailable services, and unrecognized request methods.

HTTP Methods

HTTP *methods* request information from a web server. Also known as HTTP verbs, the HTTP methods include GET, PUT, POST, HEAD, PATCH, OPTIONS, TRACE, and DELETE.

GET and POST are the two most commonly used request methods. The GET request is used to obtain resources from a web server, and the POST request is used to submit data to a web server. Table 1-2 provides more in-depth information about each of the HTTP request methods.

Table 1-2: HTTP Methods

Method	Purpose
GET	GET requests attempt to gather resources from the web server. This could be any resource, including a web page, user data, a video, an address, and so on. If the request is successful, the server will provide the resource; otherwise, the server will provide a response explaining why it was unable to get the requested resource.
POST	POST requests submit data contained in the body of the request to a web server. This could include client records, requests to transfer money from one account to another, and status updates, for example. If a client submits the same POST request multiple times, the server will create multiple results.
PUT	PUT requests instruct the web server to store submitted data under the requested URL. PUT is primarily used to send a resource to a web server. If a server accepts a PUT request, it will add the resource or completely replace the existing resource. If a PUT request is successful, a new URL should be created. If the same PUT request is submitted again, the results should remain the same.
HEAD	HEAD requests are similar to GET requests, except they request the HTTP headers only, excluding the message body. This request is a quick way to obtain information about server status and to see if a given URL works.
PATCH	PATCH requests are used to partially update resources with the submitted data. PATCH requests are likely only available if an HTTP response includes the Accept-Patch header.
OPTIONS	OPTIONS requests are a way for the client to identify all the request methods allowed from a given web server. If the web server responds to an OPTIONS request, it should respond with all allowed request options.
TRACE	TRACE requests are primarily used for debugging input sent from the client to the server. TRACE asks the server to echo back the client's original request, which could reveal that a mechanism is altering the client's request before it is processed by the server.
CONNECT	CONNECT requests initiate a two-way network connection. When allowed, this request would create a proxy tunnel between the browser and web server.
DELETE	DELETE requests ask that the server remove a given resource.

Some methods are *idempotent*, which means they can be used to send the same request multiple times without changing the state of a resource on a web server. For example, if you perform the operation of turning on a light, then the light turns on. When the switch is already on and you try to flip the switch on again, it remains on—nothing changes. GET, HEAD, PUT, OPTIONS, and DELETE are idempotent.

On the other hand, *non-idempotent* methods can dynamically change the results of a resource on a server. Non-idempotent methods include POST, PATCH, and CONNECT. POST is the most commonly used method for changing web server resources. POST is used to create new resources on a web server, so if a POST request is submitted 10 times, there will be 10 new resources on the web server. By contrast, if an idempotent method like PUT, typically used to update a resource, is requested 10 times, a single resource will be overwritten 10 times.

DELETE is also idempotent, because if the request to delete a resource was sent 10 times, the resource would be deleted only once. The subsequent times, nothing would happen. Web APIs will typically only use POST, GET, PUT, DELETE, with POST as non-idempotent methods.

Stateful and Stateless HTTP

HTTP is a *stateless* protocol, meaning the server doesn't keep track of information between requests. However, for users to have a persistent and consistent experience with a web application, the web server needs to remember something about the HTTP session with that client. For example, if a user is logged in to their account and adds several items to the shopping cart, the web application needs to keep track of the state of the end user's cart. Otherwise, every time the user navigated to a different web page, the cart would empty again.

A *stateful connection* allows the server to track the client's actions, profile, images, preferences, and so on. Stateful connections use small text files, called *cookies*, to store information on the client side. Cookies may store site-specific settings, security settings, and authentication-related information. Meanwhile, the server often stores information on itself, in a cache, or on backend databases. To continue their sessions, browsers include the stored cookies in requests to the server, and when hacking web applications, an attacker can impersonate an end user by stealing or forging their cookies.

Maintaining a stateful connection with a server has scaling limitations. When a state is maintained between a client and a server, that relationship exists only between the specific browser and the server used when the state was created. If a user switches from, say, using a browser on one computer to using the browser on their mobile device, the client would need to reauthenticate and create a new state with the server. Also, stateful connections require the client to continuously send requests to the server. Challenges start to arise when many clients are maintaining state with the same server. The server can only handle as many stateful connections as allowed by its computing resources. This is much more readily solved by stateless applications.

Stateless communications eliminate the need for the server resources required to manage sessions. In stateless communications, the server doesn't store session information, and every stateless request sent must contain all the information necessary for the web server to recognize that the requestor is authorized to access the given resources. These stateless requests can include a key or some form of authorization header to maintain an experience similar to that of a stateful connection. The connections do not store session data on the web app server; instead, they leverage backend databases.

In our shopping cart example, a stateless application could track the contents of a user's cart by updating the database or cache based on requests that contain a certain token. The end-user experience would appear the same, but how the web server handles the request is quite a bit different. Since their appearance of state is maintained and the client issues

everything needed in a given request, stateless apps can scale without the concern of losing information within a stateful connection. Instead, any number of servers can be used to handle requests as long as all the necessary information is included within the request and that information is accessible on the backend databases.

When hacking APIs, an attacker can impersonate an end user by stealing or forging their token. API communications are stateless—a topic I will explore in further detail in the next chapter.

Web Server Databases

Databases allow servers to store and quickly provide resources to clients. For example, any social media platform that allows you to upload status updates, photos, and videos is definitely using databases to save all that content. The social media platform could be maintaining those databases on its own; alternatively, the databases could be provided to the platform as a service.

Typically, a web application will store user resources by passing the resources from frontend code to backend databases. The *frontend* of a web application, which is the part of a web application that a user interacts with, determines its look and feel and includes its buttons, links, videos, and fonts. Frontend code usually includes HTML, CSS, and JavaScript. In addition, the frontend could include web application frameworks like AngularJS, ReactJS, and Bootstrap, to name a few. The *backend* consists of the technologies that the frontend needs to function. It includes the server, the application, and any databases. Backend programming languages include JavaScript, Python, Ruby, Golang, PHP, Java, C#, and Perl, to name a handful.

In a secure web application, there should be no direct interaction between a user and the backend database. Direct access to a database would remove a layer of defense and open up the database to additional attacks. When exposing technologies to end users, a web application provider expands their potential for attack, a metric known as the *attack surface*. Limiting direct access to a database shrinks the size of the attack surface.

Modern web applications use either SQL (relational) databases or NoSQL (nonrelational) databases. Knowing the differences between SQL and NoSQL databases will help you later tailor your API injection attacks.

SQL

Structured Query Language (SQL) databases are *relational databases* in which the data is organized in tables. The table's rows, called *records*, identify the data type, such as username, email address, or privilege level. Its columns are the data's *attributes* and could include all of the different usernames, email addresses, and privilege levels. In Tables 1-3 through 1-5, UserID, Username, Email, and Privilege are the data types. The rows are the data for the given table.

Table 1-3: A Relational User Table

UserID	Username
111	hAPI_hacker
112	Scuttleph1sh
113	mysteriousshadow

Table 1-4: A Relational Email Table

UserID	Email
111	<i>hapi_hacker@email.com</i>
112	<i>scuttleph1sh@email.com</i>
113	<i>mysteriousshadow@email.com</i>

Table 1-5: A Relational Privilege Table

UserID	Privilege
111	admin
112	partner
113	user

To retrieve data from a SQL database, an application must craft a SQL query. A typical SQL query to find the customer with the identification of 111 would look like this:

```
SELECT * FROM Email WHERE UserID = 111;
```

This query requests all records from the Email table that have the value 111 in the UserID column. SELECT is a statement used to obtain information from the database, the asterisk is a wildcard character that will select all of the columns in a table, FROM is used to determine which table to use, and WHERE is a clause that is used to filter specific results.

There are several varieties of SQL databases, but they are queried similarly. SQL databases include MySQL, Microsoft SQL Server, PostgreSQL, Oracle, and MariaDB, among others.

In later chapters, I'll cover how to send API requests to detect injection vulnerabilities, such as SQL injection. SQL injection is a classic web application attack that has been plaguing web apps for over two decades yet remains a possible attack method in APIs.

NoSQL

NoSQL databases, also known as distributed databases, are *nonrelational*, meaning they don't follow the structures of relational databases. NoSQL

databases are typically open-source tools that handle unstructured data and store data as documents. Instead of relationships, NoSQL databases store information as keys and values. Unlike SQL databases, each type of NoSQL database will have its own unique structures, modes of querying, vulnerabilities, and exploits. Here's a sample query using MongoDB, the current market share leader for NoSQL databases:

```
db.collection.find({"UserID": 111})
```

In this example, `db.collection.find()` is a method used to search through a document for information about the `UserID` with 111 as the value. MongoDB uses several operators that might be useful to know:

- \$eq** Matches values that are equal to a specified value
- \$gt** Matches values that are greater than a specified value
- \$lt** Matches values that are less than a specified value
- \$ne** Matches all values that are not equal to a specified value

These operators can be used within NoSQL queries to select and filter certain information in a query. For example, we could use the previous command without knowing the exact `UserID`, like so:

```
db.collection.find({"UserID": {$gt:110}})
```

This statement would find all `UserIDs` greater than 110. Understanding these operators will be useful when conducting NoSQL injection attacks later in this book.

NoSQL databases include MongoDB, Couchbase, Cassandra, IBM Domino, Oracle NoSQL Database, Redis, and Elasticsearch, among others.

How APIs Fit into the Picture

A web application can be made more powerful if it can use the power of other applications. *Application programming interfaces (APIs)* comprise a technology that facilitates communications between separate applications. In particular, *web APIs* allow for machine-to-machine communications based on HTTP, providing a common method of connecting different applications together.

This ability has opened up a world of opportunities for application providers, as developers no longer have to be experts in every facet of the functionality they want to provide to their end users. For example, let's consider a ridesharing app. The app needs a map to help its drivers navigate cities, a method for processing payments, and a way for drivers and customers to communicate. Instead of specializing in each of these different functions, a developer can leverage the Google Maps API for the mapping function, the Stripe API for payment processing, and the Twilio API to access SMS messaging. The developer can combine these APIs to create a whole new application.

The immediate impact of this technology is twofold. First, it streamlines the exchange of information. By using HTTP, web APIs can take advantage of the protocol's standardized methods, status codes, and client/server relationship, allowing developers to write code that can automatically handle the data. Second, APIs allow web application providers to specialize, as they no longer need to create every aspect of their web application.

APIs are an incredible technology with a global impact. Yet, as you'll see in the following chapters, they have greatly expanded the attack surface of every application using them on the internet.

Summary

In this chapter we covered the fundamental aspects of web applications. If you understand the general functions of HTTP requests and responses, authentication/authorization, and databases, you will easily be able to understand web APIs, because the underlying technology of web applications is the underlying technology of web APIs. In the next chapter we will examine the anatomy of APIs.

This chapter is meant to equip you with just enough information to be dangerous as an API hacker, not as a developer or application architect. If you would like additional resources about web applications, I highly suggest *The Web Application Hackers Handbook* (Wiley, 2011), *Web Application Security* (O'Reilly, 2020), *Web Security for Developers* (No Starch Press, 2020), and *The Tangled Web* (No Starch Press, 2011).

2

THE ANATOMY OF WEB APIS



Most of what the average user knows about a web application comes from what they can see and click in the graphical user interface (GUI) of their web browser. Under the hood, APIs perform much of the work. In particular, web APIs provide a way for applications to use the functionality and data of other applications over HTTP to feed a web application GUI with images, text, and videos.

This chapter covers common API terminology, types, data interchange formats, and authentication methods and then ties this information together with an example: observing the requests and responses exchanged during interactions with Twitter's API.

How Web APIs Work

Like web applications, web APIs rely on HTTP to facilitate a client/server relationship between the host of the API (the *provider*) and the system or person making an API request (the *consumer*).

An API consumer can request resources from an *API endpoint*, which is a URL for interacting with part of the API. Each of the following examples is a different API endpoint:

```
https://example.com/api/v3/users/  
https://example.com/api/v3/customers/  
https://example.com/api/updated_on/  
https://example.com/api/state/1/
```

Resources are the data being requested. A *singleton* resource is a unique object, such as `/api/user/{user_id}`. A *collection* is a group of resources, such as `/api/profiles/users`. A *subcollection* refers to a collection within a particular resource. For example, `/api/user/{user_id}/settings` is the endpoint to access the *settings* subcollection of a specific (singleton) user.

When a consumer requests a resource from a provider, the request passes through an *API gateway*, which is an API management component that acts as an entry point to a web application. For example, as shown in Figure 2-1, end users can access an application's services using a plethora of devices, which are all filtered through an API gateway. The API gateway then distributes the requests to whichever microservice is needed to fulfill each request.

The API gateway filters bad requests, monitors incoming traffic, and routes each request to the proper service or microservice. The API gateway can also handle security controls such as authentication, authorization, encryption in transit using SSL, rate limiting, and load balancing.

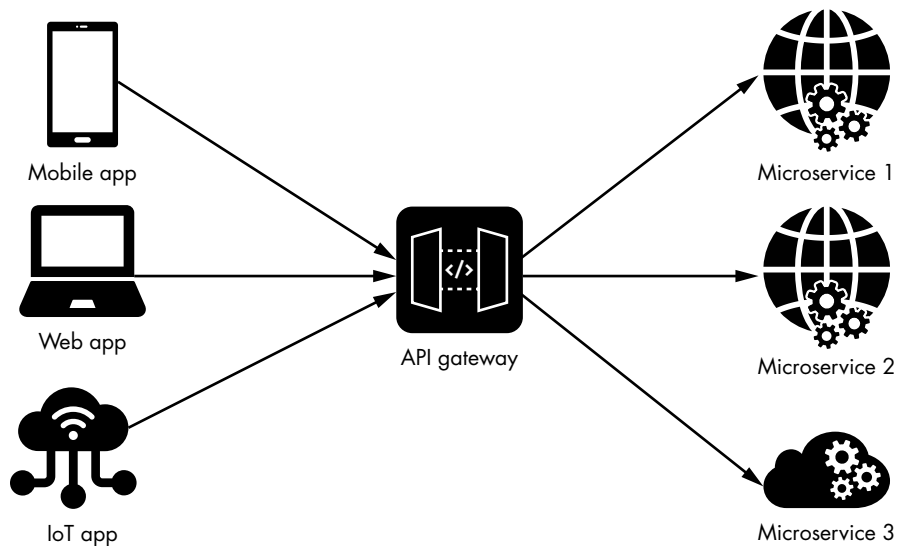


Figure 2-1: A sample microservices architecture and API gateway

A *microservice* is a modular piece of a web app that handles a specific function. Microservices use APIs to transfer data and trigger actions. For example, a web application with a payment gateway may have several different features on a single web page: a billing feature, a feature that logs customer account information, and one that emails receipts upon purchase. The application's backend design could be monolithic, meaning all the services exist within a single application, or it could have a microservice architecture, where each service functions as its own standalone application.

The API consumer does not see the backend design, only the endpoints they can interact with and the resources they can access. These are spelled out in the *API contract*, which is human-readable documentation that describes how to use the API and how you can expect it to behave. API documentation differs from one organization to another but often includes a description of authentication requirements, user permission levels, API endpoints, and the required request parameters. It might also include usage examples. From an API hacker's perspective, the documentation can reveal which endpoints to call for customer data, which API keys you need in order to become an administrator, and even business logic flaws.

In the following box, the GitHub API documentation for the `/applications/{client_id}/grants/{access_token}` endpoint, taken from <https://docs.github.com/en/rest/reference/apps>, is an example of quality documentation.

REVOKE A GRANT FOR AN APPLICATION

OAuth application owners can revoke a grant for their OAuth application and a specific user.

DELETE `/applications/{client_id}/grants/{access_token}`

PARAMETERS

Name	Type	In	Description
accept	string	header	Setting to <code>application/vnd.github.v3+json</code> is recommended.
client_id	string	path	The client ID of your GitHub app.
access_token	string	body	Required. The OAuth access token used to authenticate to the GitHub API.

The documentation for this endpoint includes the description of the purpose of the API request, the HTTP request method to use when interacting with the API endpoint, and the endpoint itself, */applications*, followed by variables.

The acronym *CRUD*, which stands for *Create, Read, Update, Delete*, describes the primary actions and methods used to interact with APIs. *Create* is the process of making new records, accomplished through a POST request. *Read* is data retrieval, done through a GET request. *Update* is how currently existing records are modified without being overwritten and is accomplished with POST or PUT requests. *Delete* is the process of erasing records, which can be done with POST or DELETE, as shown in this example. Note that CRUD is a best practice only, and developers may implement their APIs in other ways. Therefore, when you learn to hack APIs later on, we'll test beyond the CRUD methods.

By convention, curly brackets mean that a given variable is necessary within the path parameters. The *{client_id}* variable must be replaced with an actual client's ID, and the *{access_token}* variable must be replaced with your own access token. Tokens are what API providers use to identify and authorize requests to approved API consumers. Other API documentation might use a colon or square brackets to signify a variable (for example, */api/v2/customers/* or */api/collection/:client_id*).

The "Parameters" section lays out the authentication and authorization requirements to perform the described actions, including the name of each parameter value, the type of data to provide, where to include the data, and a description of the parameter value.

Standard Web API Types

APIs come in standard types, each of which varies in its rules, functions, and purpose. Typically, a given API will use only one type, but you may encounter endpoints that don't match the format and structure of the others or don't match a standard type at all. Being able to recognize typical and atypical APIs will help you know what to expect and test for as an API hacker. Remember, most public APIs are designed to be self-service, so a given API provider will often let you know the type of API you'll be interacting with.

This section describes the two primary API types we'll focus on throughout this book: RESTful APIs and GraphQL. Later parts of the book, as well as the book's labs, cover attacks against RESTful APIs and GraphQL only.

RESTful APIs

Representational State Transfer (REST) is a set of architectural constraints for applications that communicate using HTTP methods. APIs that use REST constraints are called *RESTful* (or just REST) APIs.

REST was designed to improve upon many of the inefficiencies of other older APIs, such as Simple Object Access Protocol (SOAP). For example, it relies entirely on the use of HTTP, which makes it much more approachable to end users. REST APIs primarily use the HTTP methods GET, POST, PUT, and DELETE to accomplish CRUD (as described in the section “How Web APIs Work”).

RESTful design depends on six constraints. These constraints are “shoulds” instead of “musts,” reflecting the fact that REST is essentially a set of guidelines for an HTTP resource-based architecture:

1. **Uniform interface:** REST APIs should have a uniform interface. In other words, the requesting client device should not matter; a mobile device, an IoT (internet of things) device, and a laptop must all be able to access a server in the same way.
2. **Client/server:** REST APIs should have a client/server architecture. Clients are the consumers requesting information, and servers are the providers of that information.
3. **Stateless:** REST APIs should not require stateful communications. REST APIs do not maintain state during communication; it is as though each request is the first one received by the server. The consumer will therefore need to supply everything the provider will need in order to act upon the request. This has the benefit of saving the provider from having to remember the consumer from one request to another. Consumers often provide tokens to create a state-like experience.
4. **Cacheable:** The response from the REST API provider should indicate whether the response is cacheable. *Caching* is a method of increasing request throughput by storing commonly requested data on the client side or in a server cache. When a request is made, the client will first check its local storage for the requested information. If it doesn't find the information, it passes the request to the server, which checks its local storage for the requested information. If the data is not there either, the request could be passed to other servers, such as database servers, where the data can be retrieved.

As you might imagine, if the data is stored on the client, the client can immediately retrieve the requested data at little to no processing cost to the server. This also applies if the server has cached a request. The further down the chain a request has to go to retrieve data, the higher the resource cost and the longer it takes. Making REST APIs cacheable by default is a way to improve overall REST performance and scalability by decreasing response times and server processing power. APIs usually manage caching with the use of headers that explain when the requested information will expire from the cache.

5. **Layered system:** The client should be able to request data from an endpoint without knowing about the underlying server architecture.
6. **Code on demand (optional):** Allows for code to be sent to the client for execution.

REST is a style rather than a protocol, so each RESTful API may be different. It may have methods enabled beyond CRUD, its own sets of authentication requirements, subdomains instead of paths for endpoints, different rate-limit requirements, and so on. Furthermore, developers or an organization may call their API “RESTful” without adhering to the standard, which means you can’t expect every API you come across to meet all the REST constraints.

Listing 2-1 shows a fairly typical REST API GET request used to find out how many pillows are in a store’s inventory. Listing 2-2 shows the provider’s response.

```
GET /api/v3/inventory/item/pillow HTTP/1.1
HOST: rest-shop.com
User-Agent: Mozilla/5.0
Accept: application/json
```

Listing 2-1: A sample RESTful API request

```
HTTP/1.1 200 OK
Server: RESTfulServer/0.1
Cache-Control: no-store
Content-Type: application/json
```

```
{
  "item": {
    "id": "00101",
    "name": "pillow",
    "count": 25
    "price": {
      "currency": "USD",
      "value": "19.99"
    }
  },
}
```

Listing 2-2: A sample RESTful API response

This REST API request is just an HTTP GET request to the specified URL. In this case, the request queries the store’s inventory for pillows. The provider responds with JSON indicating the item’s ID, name, and quantity of items in stock. If there was an error in the request, the provider would respond with an HTTP error code in the 400 range indicating what went wrong.

One thing to note: the *rest-shop.com* store provided all the information it had about the resource “pillow” in its response. If the consumer’s application only needed the name and value of the pillow, the consumer would need to filter out the additional information. The amount of information sent back to a consumer completely depends on how the API provider has programmed its API.

REST APIs have some common headers you should become familiar with. These are identical to HTTP headers but are more commonly seen in REST API requests than in other API types, so they can help you identify REST APIs. (Headers, naming conventions, and the data interchange format used are normally the best indicators of an API's type.) The following subsections detail some of the common REST API headers you will come across.

Authorization

Authorization headers are used to pass a token or credentials to the API provider. The format of these headers is `Authorization: <type> <token/credentials>`. For example, take a look at the following authorization header:

```
Authorization: Bearer Ab4dtok3n
```

There are different authorization types. Basic uses base64-encoded credentials. Bearer uses an API token. Finally, AWS-HMAC-SHA256 is an AWS authorization type that uses an access key and a secret key.

Content Type

Content-Type headers are used to indicate the type of media being transferred. These headers differ from Accept headers, which state the media type you want to receive; Content-Type headers describe the media you're sending.

Here are some common Content-Type headers for REST APIs:

application/json Used to specify JavaScript Object Notation (JSON) as a media type. JSON is the most common media type for REST APIs.

application/xml Used to specify XML as a media type.

application/x-www-form-urlencoded A format in which the values being sent are encoded and separated by an ampersand (&), and an equal sign (=) is used between key/value pairs.

Middleware (X) Headers

X-<anything> headers are known as *middleware headers* and can serve all sorts of purposes. They are fairly common outside of API requests as well. X-Response-Time can be used as an API response to indicate how long a response took to process. X-API-Key can be used as an authorization header for API keys. X-Powered-By can be used to provide additional information about backend services. X-Rate-Limit can be used to tell the consumer how many requests they can make within a given time frame. X-RateLimit-Remaining can tell a consumer how many requests remain before they violate rate-limit enforcement. (There are many more, but you get the idea.) X-<anything> middleware headers can provide a lot of useful information to API consumers and hackers alike.

ENCODING DATA

As we touched upon in Chapter 1, HTTP requests use encoding as a method to ensure that communications are handled properly. Various characters that can be problematic for the technologies used by the server are known as *bad characters*. One way of handling bad characters is to use an encoding scheme that formats the message in such a way as to remove them. Common encoding schemes include Unicode encoding, HTML encoding, URL encoding, and base64 encoding. XML typically uses one of two forms of Unicode encoding: UTF-8 or UTF-16.

When the string “hAPI hacker” is encoded in UTF-8, it becomes the following:

```
\x68\x41\x50\x49\x20\x68\x61\x63\x62\x65\x72
```

Here is the UTF-16 version of the string:

```
\u{68}\u{41}\u{50}\u{49}\u{20}\u{68}\u{61}\u{63}\u{62}\u{65}\u{72}
```

Finally, here is the base64-encoded version:

```
aEFQSSBoYWNIrZXI=
```

Recognizing these encoding schemes will be useful as you begin examining requests and responses and encounter encoded data.

GraphQL

Short for *Graph Query Language*, *GraphQL* is a specification for APIs that allow clients to define the structure of the data they want to request from the server. GraphQL is RESTful, as it follows the six constraints of REST APIs. However, GraphQL also takes the approach of being *query-centric*, because it is structured to function similarly to a database query language like Structured Query Language (SQL).

As you might gather from the specification’s name, GraphQL stores the resources in a graph data structure. To access a GraphQL API, you’ll typically access the URL where it is hosted and submit an authorized request that contains query parameters as the body of a POST request, similar to the following:

```
query {  
  users {  
    username  
    id  
    email  
  }  
}
```

In the right context, this query would provide you with the usernames, IDs, and emails of the requested resources. A GraphQL response to this query would look like the following:

```
{
  "data": {
    "users": {
      "username": "hapi_hacker",
      "id": 1111,
      "email": "hapihacker@email.com"
    }
  }
}
```

GraphQL improves on typical REST APIs in several ways. Since REST APIs are resource based, there will likely be instances when a consumer needs to make several requests in order to get all the data they need. On the other hand, if a consumer only needs a specific value from the API provider, the consumer will need to filter out the excess data. With GraphQL, a consumer can use a single request to get the exact data they want. That's because, unlike REST APIs, where clients receive whatever data the server is programmed to return from an endpoint, including the data they don't need, GraphQL APIs let clients request specific fields from a resource.

GraphQL also uses HTTP, but it typically depends on a single entry point (URL) using the POST method. In a GraphQL request, the body of the POST request is what the provider processes. For example, take a look at the GraphQL request in Listing 2-3 and the response in Listing 2-4, depicting a request to check a store's inventory for graphics cards.

```
POST /graphql HTTP/1.1
HOST: graphql-shop.com
Authorization: Bearer ab4dt0k3n

{query❶ {
  inventory❷ (item:"Graphics Card", id: 00101) {
    name
    fields❸{
      price
      quantity} } }
}
```

Listing 2-3: An example GraphQL request

```
HTTP/1.1 200 OK
Content-Type: application/json
Server: GraphQLServer
```

```

{
  "data": {
    "inventory": { "name": "Graphics Card",
    "fields": ❶ [
      {
        "price": "999.99"
        "quantity": 25 } ] } }
}

```

Listing 2-4: An example GraphQL response

As you can see, a query payload in the body specifies the information needed. The GraphQL request body begins with the query operation ❶, which is the equivalent of a GET request and used to obtain information from the API. The GraphQL node we are querying for, "inventory" ❷, is also known as the root query type. Nodes, similar to objects, are made up of fields ❸, similar to key/value pairs in REST. The main difference here is that we can specify the exact fields we are looking for. In this example, we are looking for the "price" and "quantity" fields. Finally, you can see that the GraphQL response only provided the requested fields for the specified graphics card ❹. Instead of getting the item ID, item name, and other superfluous information, the query resolved with only the fields that were needed.

If this had been a REST API, it might have been necessary to send requests to different endpoints to get the quantity and then the brand of the graphics card, but with GraphQL you can build out a query for the specific information you are looking for from a single endpoint.

GraphQL still functions using CRUD, which may sound confusing at first since it relies on POST requests. However, GraphQL uses three operations within the POST request to interact with GraphQL APIs: query, mutation, and subscription. *Query* is an operation to retrieve data (read). *Mutation* is an operation used to submit and write data (create, update, and delete). *Subscription* is an operation used to send data (read) when an event occurs. Subscription is a way for GraphQL clients to listen to live updates from the server.

GraphQL uses *schemas*, which are collections of the data that can be queried with the given service. Having access to the GraphQL schema is similar to having access to a REST API collection. A GraphQL schema will provide you with the information you'll need in order to query the API.

You can interact with GraphQL using a browser if there is a GraphQL IDE, like GraphiQL, in place (see Figure 2-2).

Otherwise, you'll need a GraphQL client such as Postman, Apollo-Client, GraphQL-Request, GraphQL-CLI, or GraphQL-Compose. In later chapters, we'll use Postman as our GraphQL client.

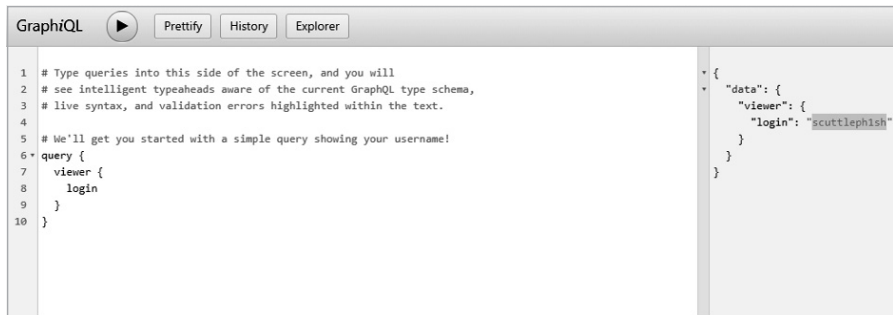


Figure 2-2: The GraphQL interface for GitHub

SOAP: AN ACTION-ORIENTED API FORMAT

Simple Object Access Protocol (SOAP) is a type of action-oriented API that relies on XML. SOAP is one of the older web APIs, originally released as XML-RPC back in the late 1990s, so we won't cover it in this book.

Although SOAP works over HTTP, SMTP, TCP, and UDP, it was primarily designed for use over HTTP. When SOAP is used over HTTP, the requests are all made using HTTP POST. For example, take a look at the following sample SOAP request:

```

POST /Inventory HTTP/1.1
Host: www.soap-shop.com
Content-Type: application/soap+xml; charset=utf-8
Content-Length: nnn

<?xml version="1.0"?>

❶ <soap:Envelope
❷ xmlns:soap="http://www.w3.org/2003/05/soap-envelope/"
  soap:encodingStyle="http://www.w3.org/2003/05/soap-encoding">

❸ <soap:Body xmlns:m="http://www.soap-shop.com/inventory">
  <m:GetInventoryPrice>
    <m:InventoryName>ThebestSOAP</m:InventoryName>
  </m:GetInventoryPrice>
</soap:Body>

</soap:Envelope>
  
```

The corresponding SOAP response looks like this:

```

HTTP/1.1 200 OK
Content-Type: application/soap+xml; charset=utf-8
Content-Length: nnn
  
```

(continued)

```

<?xml version="1.0"?>

<soap:Envelope
  xmlns:soap="http://www.w3.org/2003/05/soap-envelope/"
  soap:encodingStyle="http://www.w3.org/2003/05/soap-encoding">

  <soap:Body xmlns:m="http://www.soap-shop.com/inventory">
    ❶<soap:Fault>
      <faultcode>soap:VersionMismatch</faultcode>
      <faultstring, xml:lang='en">
        Name does not match Inventory record
      </faultstring>
    </soap:Fault>
  </soap:Body>

</soap:Envelope>

```

SOAP API messages are made up of four parts: the envelope ❶ and header ❷, which are necessary, and the body ❸ and fault ❹, which are optional. The *envelope* is an XML tag at the beginning of a message that signifies that the message is a SOAP message. The *header* can be used to process a message; in this example, the Content-Type request header lets the SOAP provider know the type of content being sent in the POST request (application/soap+xml). Since APIs facilitate machine-to-machine communication, headers essentially form an agreement between the consumer and the provider concerning the expectations within the request. Headers are a way to ensure that the consumer and provider understand one another and are speaking the same language. The *body* is the primary payload of the XML message, meaning it contains the data sent to the application. The *fault* is an optional part of a SOAP response that can be used to provide error messaging.

REST API Specifications

The variety of REST APIs has left room for other tools and standardizations to fill in some of the gaps. *API specifications*, or description languages, are frameworks that help organizations design their APIs, automatically create consistent human-readable documentation, and therefore help developers and users know what to expect regarding the API's functionality and results. Without specifications, there would be little to no consistency between APIs. Consumers would have to learn how each API's documentation was formatted and adjust their application to interact with each API.

Instead, a consumer can program their application to ingest different specifications and then easily interact with any API using that given specification. In other words, you can think of specifications as the home electric sockets of APIs. Instead of having a unique electric socket for every home appliance, the use of a single consistent format throughout a home allows you to buy a toaster and plug it into a socket on any wall without any hassle.

OpenAPI Specification 3.0 (OAS), previously known as Swagger, is one of the leading specifications for RESTful APIs. OAS helps organize and manage APIs by allowing developers to describe endpoints, resources, operations, and authentication and authorization requirements. They can then create human- and machine-readable API documentation, formatted as JSON or YAML. Consistent API documentation is good for developers and users.

The *RESTful API Modeling Language (RAML)* is another way to consistently generate API documentation. RAML is an open specification that works exclusively with YAML for document formatting. Similar to OAS, RAML was designed to document, design, build, and test REST APIs. For more information about RAML, check out the raml-spec GitHub repo (<https://github.com/raml-org/raml-spec>).

In later chapters, we will use an API client called Postman to import specifications and get instant access to the capabilities of an organization's APIs.

API Data Interchange Formats

APIs use several formats to facilitate the exchange of data. Additionally, specifications use these formats to document APIs. Some APIs, like SOAP, require a specific format, whereas others allow the client to specify the format to use in the request and response body. This section introduces three common formats: JSON, XML, and YAML. Familiarity with data interchange formats will help you recognize API types, what the APIs are doing, and how they handle data.

JSON

JavaScript Object Notation (JSON) is the primary data interchange format we'll use throughout this book, as it is widely used for APIs. It organizes data in a way that is both human-readable and easily parsable by applications; many programming languages can turn JSON into data types they can use.

JSON represents objects as key/value pairs separated by commas, within a pair of curly brackets, as follows:

```
{
  "firstName": "James",
  "lastName": "Lovell",
  "tripsToTheMoon": 2,
  "isAstronaut": true,
  "walkedOnMoon": false,
  "comment" : "This is a comment",
  "spacecrafts": ["Gemini 7", "Gemini 12", "Apollo 8", "Apollo 13"],
  "book": [
    {
      "title": "Lost Moon",
      "genre": "Non-fiction"
    }
  ]
}
```

Everything between the first curly bracket and the last is considered an object. Within the object are several key/value pairs, such as "firstName": "James", "lastName": "Lovell", and "tripsToTheMoon": 2. The first entry of the key/value pair (on the left) is the *key*, a string that describes the value pair, and the second is the *value* (on the right), which is some sort of data represented by one of the acceptable data types (strings, numbers, Boolean values, null, an array, or another object). For example, notice the Boolean value false for "walkedOnMoon" or the "spacecrafts" array surrounded by square brackets. Finally, the nested object "book" contains its own set of key/value pairs. Table 2-1 describes JSON types in more detail.

JSON does not allow inline comments, so any sort of comment-like communications must take place as a key/value pair like "comment" : "This is a comment". Alternatively, you can find comments in the API documentation or HTTP response.

Table 2-1: JSON Types

Type	Description	Example
Strings	Any combination of characters within double quotes.	{ "Motto": "Hack the planet", "Drink": "Jolt", "User": "Razor" }
Numbers	Basic integers, fractions, negative numbers, and exponents. Notice that the multiple items are comma-separated.	{ "number_1" : 101, "number_2" : -102, "number_3" : 1.03, "number_4" : 1.0E+4 }
Boolean values	Either true or false.	{ "admin" : false, "privesc" : true }
Null	No value.	{ "value" : null }
Arrays	An ordered collection of values. Collections of values are surrounded by brackets ([]) and the values are comma-separated.	{ "uid" : ["1", "2", "3"] }
Objects	An unordered set of value pairs inserted between curly brackets ({}). An object can contain multiple key/value pairs.	{ "admin" : false, "key" : "value", "privesc" : true, "uid" : 101, "vulnerabilities" : "galore" }

To illustrate these types, take a look at the following key/value pairs in the JSON data found in a Twitter API response:

```
{
  "id":1278533978970976256, ❶
  "id_str":"1278533978970976256", ❷
  "full_text":"1984: William Gibson published his debut novel, Neuromancer. It's a cyberpunk
tale about Henry Case, a washed up computer hacker who's offered a chance at redemption by a
mysterious dude named Armitage. Cyberspace. Hacking. Virtual reality. The matrix. Hacktivism. A
must read. https://t.co/R9hm2LOKQi",
  "truncated":false ❸
}
```

In this example, you should be able to identify the number 1278533978970976256 ❶, strings like those for the keys "id_str" and "full_text" ❷, and the Boolean value ❸ for "truncated".

XML

The *Extensible Markup Language (XML)* format has been around for a while, and you'll probably recognize it. XML is characterized by the descriptive tags it uses to wrap data. Although REST APIs can use XML, it is most commonly associated with SOAP APIs. SOAP APIs can only use XML as the data interchange.

The Twitter JSON you just saw would look like the following if converted to XML:

```
<?xml version="1.0" encoding="UTF-8" ?> ❶
<root> ❷
  <id>1278533978970976300</id>
  <id_str>1278533978970976256</id_str>
  <full_text>1984: William Gibson published his debut novel, Neuromancer. It's a cyberpunk
tale about Henry Case, a washed up computer hacker who's offered a chance at redemption by
a mysterious dude named Armitage. Cyberspace. Hacking. Virtual reality. The matrix. Hacktivism.
A must read. https://t.co/R9hm2LOKQi </full_text>
  <truncated>false</truncated>
</root>
```

XML always begins with a *prolog*, which contains information about the XML version and encoding used ❶.

Next, *elements* are the most basic parts of XML. An element is any XML tag or information surrounded by tags. In the previous example, `<id>1278533978970976300</id>`, `<id_str>1278533978</id_str>`, `<full_text>`, `</full_text>`, and `<truncated>false</truncated>` are all elements. XML must have a root element and can contain child elements. In the example, the root element is `<root>` ❷. The child elements are XML attributes. An example of a child element is the `<BookGenre>` element within the following example:

```
<LibraryBooks>
  <BookGenre>SciFi</BookGenre>
</LibraryBooks>
```

Comments in XML are surrounded by two dashes, like this: `<!--XML comment example-->`.

The key differences between XML and JSON are JSON's descriptive tags, character encoding, and length: the XML takes much longer to convey the same information, a whopping 565 bytes.

YAML

Another lightweight form of data exchange used in APIs, *YAML* is a recursive acronym that stands for *YAML Ain't Markup Language*. It was created as a more human- and computer-readable format for data exchange.

Like JSON, YAML documents contain key/value pairs. The value may be any of the YAML data types, which include numbers, strings, Booleans, null values, and sequences. For example, take a look at the following YAML data:

```
---
id: 1278533978970976300
id_str: 1278533978970976256
#Comment about Neuromancer
full_text: "1984: William Gibson published his debut novel, Neuromancer. It's a cyberpunk
tale about Henry Case, a washed up computer hacker who's offered a chance at redemption by a
mysterious dude named Armitage. Cyberspace. Hacking. Virtual reality. The matrix. Hacktivism. A
must read. https://t.co/R9hm2L0KQi"
truncated: false
...
```

You'll notice that YAML is much more readable than JSON. YAML documents begin with

```
---
```

and end with

```
...
```

instead of with curly brackets. Also, quotes around strings are optional. Additionally, URLs don't need to be encoded with backslashes. Finally, YAML uses indentation instead of curly brackets to represent nesting and allows for comments beginning with #.

API specifications will often be formatted as JSON or YAML, because these formats are easy for humans to digest. With only a few basic concepts in mind, we can look at either of these formats and understand what is going on; likewise, machines can easily parse the information.

If you'd like to see more YAML in action, visit <https://yaml.org>. The entire website is presented in YAML format. YAML is recursive all the way down.

API Authentication

APIs may allow public access to consumers without authentication, but when an API allows access to proprietary or sensitive data, it will use some form of authentication and authorization. An API's authentication process should validate that users are who they claim to be, and the authorization

process should grant them the ability to access the data they are allowed to access. This section covers a variety of API authentication and authorization methods. These methods vary in complexity and security, but they all operate on a common principle: the consumer must send some kind of information to the provider when making a request, and the provider must link that information to a user before granting or denying access to a resource.

Before jumping into API authentication, it is important to understand what authentication is. Authentication is the process of proving and verifying an identity. In a web application, authentication is the way you prove to the web server that you are a valid user of said web app. Typically, this is done through the use of credentials, which consist of a unique ID (such as a username or email) and password. After a client sends credentials, the web server compares what was sent to the credentials it has stored. If the credentials provided match the credentials stored, the web server will create a user session and issue a cookie to the client.

When the session ends between the web app and user, the web server will destroy the session and remove the associated client cookies.

As described earlier in this chapter, REST and GraphQL APIs are stateless, so when a consumer authenticates to these APIs, no session is created between the client and server. Instead, the API consumer must prove their identity within every request sent to the API provider's web server.

Basic Authentication

The simplest form of API authentication is *HTTP basic authentication*, in which the consumer includes their username and password in a header or the body of a request. The API could either pass the username and password to the provider in plaintext, like `username:password`, or it could encode the credentials using something like base64 to save space (for example, as `dXNlcm5hbWU6cGFzc3dvcmQK`).

Encoding is not encryption, and if base64-encoded data is captured, it can easily be decoded. For example, you can use the Linux command line to base64-encode `username:password` and then decode the encoded result:

```
$ echo "username:password"|base64
dXNlcm5hbWU6cGFzc3dvcmQK
$ echo "dXNlcm5hbWU6cGFzc3dvcmQK"|base64 -d
username:password
```

As you can see, basic authentication has no inherent security and completely depends on other security controls. An attacker can compromise basic authentication by capturing HTTP traffic, performing a man-in-the-middle attack, tricking the user into providing their credentials through social engineering tactics, or performing a brute-force attack in which they attempt various usernames and passwords until they find some that work.

Since APIs are often stateless, those using only basic authentication require the consumer to provide credentials in every request. It is common for an API provider to instead use basic authentication once, for the first request, and then issue an API key or some other token for all other requests.

API Keys

API keys are unique strings that API providers generate and grant to authorize access for approved consumers. Once an API consumer has a key, they can include it in requests whenever specified by the provider. The provider will typically require that the consumer pass the key in query string parameters, request headers, body data, or as a cookie when they make a request.

API keys typically look like semi-random or random strings of numbers and letters. For example, take a look at the API key included in the query string of the following URL:

```
/api/v1/users?apikey=ju574n3x4mp134p1k3y
```

The following is an API key included as a header:

```
"API-Secret": "17813fg8-46a7-5006-e235-45be7e9f2345"
```

Finally, here is an API key passed in as a cookie:

```
Cookie: API-Key= 4n07h3r4p1k3y
```

The process of acquiring an API key depends on the provider. The NASA API, for example, requires the consumer to register for the API with a name, email address, and optional application URL (if the user is programming an application to use the API), as shown in Figure 2-3.

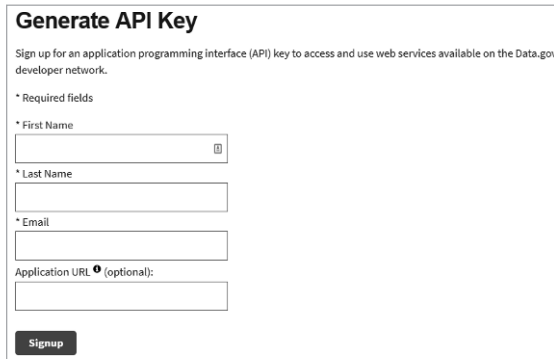


Figure 2-3: NASA's form to generate an API key

The resulting key will look something like this:

```
roS6SmRjLdxZzrNSAkxjCdb6WodSda2G9zc2Q7sK
```

It must be passed as a URL parameter in each API request, as follows:

```
api.nasa.gov/planetary/apod?api_key=roS6SmRjLdxZzrNSAkxjCdb6WodSda2G9zc2Q7sK
```

API keys can be more secure than basic authentication for several reasons. When keys are sufficiently long, complex, and randomly generated, they can be exceedingly difficult for an attacker to guess or brute-force.

Additionally, providers can set expiration dates to limit the length of time for which the keys are valid.

However, API keys have several associated risks that we will take advantage of later in this book. Since each API provider may have their own system for generating API keys, you'll find instances in which the API key is generated based on user data. In these cases, API hackers may guess or forge API keys by learning about the API consumers. API keys may also be exposed to the internet in online repositories, left in code comments, intercepted when transferred over unencrypted connections, or stolen through phishing.

JSON Web Tokens

A *JSON Web Token (JWT)* is a type of token commonly used in API token-based authentication. It's used like this: The API consumer authenticates to the API provider with a username and password. The provider generates a JWT and sends it back to the consumer. The consumer adds the provided JWT to the Authorization header in all API requests.

JWTs consist of three parts, all of which are base64-encoded and separated by periods: the header, the payload, and the signature. The *header* includes information about the algorithm used to sign the payload. The *payload* is the data included within the token, such as a username, timestamp, and issuer. The *signature* is the encoded and encrypted message used to validate the token.

Table 2-2 shows an example of these parts, unencoded for readability, as well as the final token.

NOTE

The signature field is not a literal encoding of HMACSHA512 ...; rather, the signature is created by calling the encryption function HMACSHA512(), specified by "alg": "HS512", on the encoded header and payload, and then encoding the result.

Table 2-2: JWT Components

Component	Content
Header	{ "alg": "HS512", "typ": "JWT" }
Payload	{ "sub": "1234567890", "name": "hAPI Hacker", "iat": 1516239022 }
Signature	HMACSHA512(base64UrlEncode(header) + "." + base64UrlEncode(payload), SuperSecretPassword)
JWT	eyJhbGciOiJIUzUxMiIsInR5cCI6IkpXVCJ9.eyJzdWIiOiIxMjMONTY3ODkwIiwibmFtZSI6ImhBUeKgSGFja2VyIiwiaWF0IjoxNTE2MjM5MDIyfQ.zsUjGDbBjqI-bJbaUmvUdKaGSEvROKfNjy9K6TckK5sd97AMdPDLxUZwsneff40I2WQikhgPm7HH1XYn4jmoQ

JWTs are generally secure but can be implemented in ways that will compromise that security. API providers can implement JWTs that do not use encryption, which means you would be one base64 decode away from being able to see what is inside the token. An API hacker could decode such a token, tamper with the contents, and send it back to the provider to gain access, as you will see in Chapter 10. The JWT secret key may also be stolen or guessed by brute force.

HMAC

A *hash-based message authentication code (HMAC)* is the primary API authentication method used by Amazon Web Services (AWS). When using HMAC, the provider creates a secret key and shares it with consumer. When a consumer interacts with the API, an HMAC hash function is applied to the consumer's API request data and secret key. The resulting hash (also called a *message digest*) is added to the request and sent to the provider. The provider calculates the HMAC, just as the consumer did, by running the message and key through the hash function, and then compares the output hash value to the value provided by the client. If the provider's hash value matches the consumer's hash value, the consumer is authorized to make the request. If the values do not match, either the client's secret key is incorrect or the message has been tampered with.

The security of the message digest depends on the cryptographic strength of the hash function and secret key. Stronger hash mechanisms typically produce longer hashes. Table 2-3 shows the same message and key hashed by different HMAC algorithms.

Table 2-3: HMAC Algorithms

Algorithm	Hash output
HMAC-MD5	f37438341e3d22aa11b4b2e838120dcf
HMAC-SHA1	4c2de361ba8958558de3d049ed1fb5c115656e65
HMAC-SHA256	be8e73ffbd9a953f2ec892f06f9a5e91e6551023d1942ec7994fa1a78a5ae6bc
HMAC-SHA512	6434a354a730f888865bc5755d9f498126d8f67d73f32ccd2b775c47c91ce26b66dfa59c25aed7f4a6bcb4786d3a3c6130f63ae08367822af3f967d3a7469e1b

You may have some red flags regarding the use of SHA1 or MD5. As of the writing of this book, there are currently no known vulnerabilities affecting HMAC-SHA1 and HMAC-MD5, but these functions are cryptographically weaker than SHA-256 and SHA-512. However, the more secure functions are also slower. The choice of which hash function to use comes down to prioritizing either performance or security.

As with the previous authentication methods covered, the security of HMAC depends on the consumer and provider keeping the secret key private. If a secret key is compromised, an attacker could impersonate the victim and gain unauthorized access to the API.

OAuth 2.0

OAuth 2.0, or just *OAuth*, is an authorization standard that allows different services to access each other's data, often using APIs to facilitate the service-to-service communications.

Let's say you want to automatically share your Twitter tweets on LinkedIn. In OAuth's model, we would consider Twitter to be the service provider and LinkedIn to be the application or client. In order to post your tweets, LinkedIn will need authorization to access your Twitter information. Since both Twitter and LinkedIn have implemented OAuth, instead of providing your credentials to the service provider and consumer every time you want to share this information across platforms, you can simply go into your LinkedIn settings and authorize Twitter. Doing so will send you to *api.twitter.com* to authorize LinkedIn to access your Twitter account (see Figure 2-4).

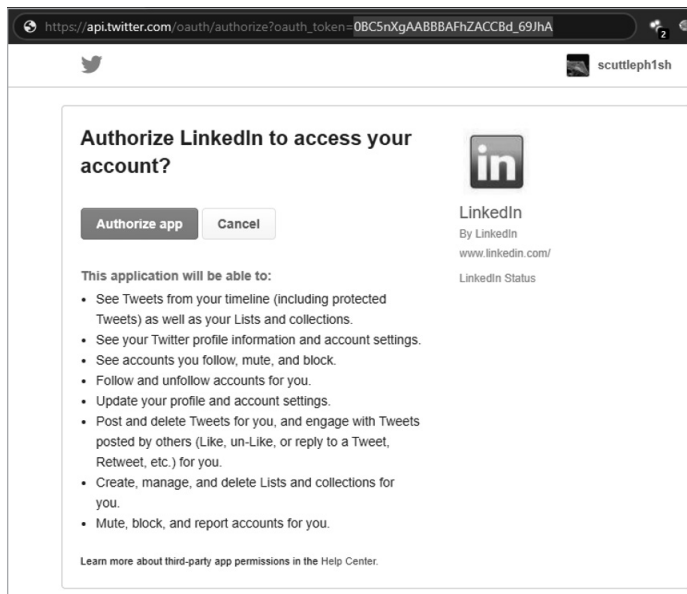


Figure 2-4: LinkedIn–Twitter OAuth authorization request

When you authorize LinkedIn to access your Twitter posts, Twitter generates a limited, time-based access token for LinkedIn. LinkedIn then provides that token to Twitter to post on your behalf, and you don't have to give LinkedIn your Twitter credentials.

Figure 2-5 shows the general OAuth process. The user (*resource owner*) grants an application (the *client*) access to a service (the *authorization server*), the service creates a token, and then the application uses the token to exchange data with the service (also the *resource server*).

In the LinkedIn–Twitter example, you are the resource owner, LinkedIn is the application/client, and Twitter is the authorization server and resource server.

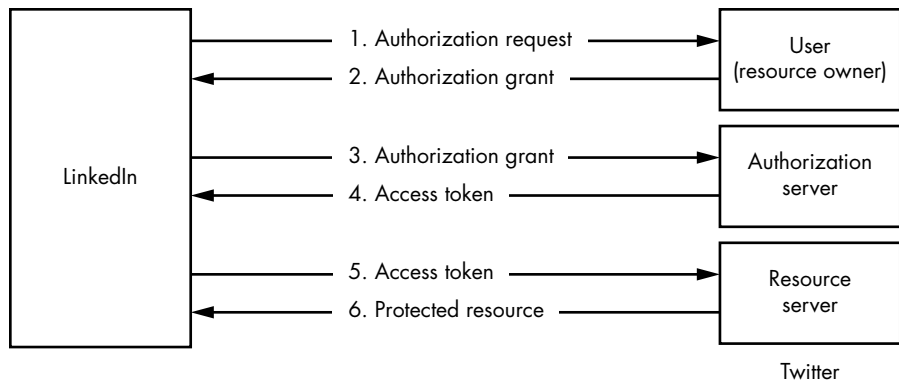


Figure 2-5: An illustration of the OAuth process

OAuth is one of the most trusted forms of API authorization. However, while it adds security to the authorization process, it also expands the potential attack surface—although flaws often have more to do with how the API provider implements OAuth than with OAuth itself. API providers that poorly implement OAuth can expose themselves to a variety of attacks such as token injection, authorization code reuse, cross-site request forgery, invalid redirection, and phishing.

No Authentication

As in web applications generally, there are plenty of instances where it is valid for an API to have no authentication at all. If an API does not handle sensitive data and only provides public information, the provider could make the case that no authentication is necessary.

APIs in Action: Exploring Twitter’s API

After reading this and the previous chapter, you should understand the various components running beneath the GUI of a web application. Let’s now make these concepts more concrete by taking a close look at Twitter’s API. If you open a web browser and visit the URL <https://twitter.com>, the initial request triggers a series of communications between the client and the server. Your browser automatically orchestrates these data transfers, but by using a web proxy like Burp Suite, which we’ll set up in Chapter 4, you can see all the requests and responses in action.

The communications begin with the typical kind of HTTP traffic described in Chapter 1:

1. Once you’ve entered a URL into your browser, the browser automatically submits an HTTP GET request to the web server at *twitter.com*:

```
GET / HTTP/1.1
Host: twitter.com
User-Agent: Mozilla/5.0
Accept: text/html
--snip--
Cookie: [...]
```

2. The Twitter web application server receives the request and responds to the GET request by issuing a successful 200 OK response:

```
HTTP/1.1 200 OK
cache-control: no-cache, no-store, must-revalidate
connection: close
content-security-policy: content-src 'self'
content-type: text/html; charset=utf-8
server: tsa_a
--snip--
x-powered-by: Express
x-response-time: 56

<!DOCTYPE html>
<html dir="ltr" lang="en">
--snip--
```

This response header contains the status of the HTTP connection, client instructions, middleware information, and cookie-related information. *Client instructions* tell the browser how to handle the requested information, such as caching data, the content security policy, and instructions about the type of content that was sent. The actual payload begins just below `x-response-time`; it provides the browser with the HTML needed to render the web page.

Now imagine that the user looks up “hacking” using Twitter’s search bar. This kicks off a POST request to Twitter’s API, as shown next. Twitter is able to leverage APIs to distribute requests and seamlessly provide requested resources to many users.

```
POST /1.1/jot/client_event.json?q=hacking HTTP/1.1
Host: api.twitter.com
User-Agent: Mozilla/5.0
--snip--
Authorization: Bearer AAAAAAAAAAAAAAAAAA...
--snip--
```

This POST request is an example of the Twitter API querying the web service at *api.twitter.com* for the search term “hacking.” The Twitter API

responds with JSON containing the search results, which includes tweets and information about each tweet such as user mentions, hashtags, and post times:

```
"created_at": [...]  
"id":1278533978970976256  
"id_str": "1278533978970976256"  
"full-text": "1984: William Gibson published his debut novel..."  
"truncated":false,  
--snip--
```

The fact that the Twitter API seems to adhere to CRUD, API naming conventions, tokens for authorization, *application/x-www-form-urlencoded*, and JSON as a data interchange makes it pretty clear that this API is a RESTful API.

Although the response body is formatted in a legible way, it's meant to be processed by the browser to be displayed as a human-readable web page. The browser renders the search results using the string from the API request. The provider's response then populates the page with search results, images, and social media-related information such as likes, retweets, comments (see Figure 2-6).

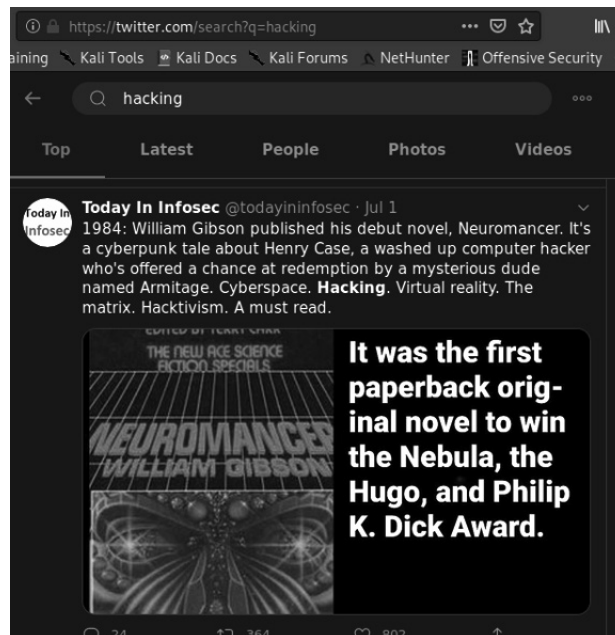


Figure 2-6: The rendered result from the Twitter API search request

From the end user's perspective, the whole interaction appears seamless: you click the search bar, type in a query, and receive the results.

Summary

In this chapter, we covered the terminology, parts, types, and supporting architecture of APIs. You learned that APIs are interfaces for interacting with web applications. Different types of APIs have different rules, functions, and purposes, but they all use some kind of format for exchanging data between applications. They often use authentication and authorization schemes to make sure consumers can access only the resources they're supposed to.

Understanding these concepts will prepare you to confidently strike at the components that make up APIs. As you continue to read, refer to this chapter if you encounter API concepts that confuse you.

3

COMMON API VULNERABILITIES



Understanding common vulnerabilities will help you identify weaknesses when you're testing APIs. In this chapter, I cover most of the vulnerabilities included in the Open Web Application Security Project (OWASP) API Security Top 10 list, plus two other useful weaknesses: information disclosure and business logic flaws. I'll describe each vulnerability, its significance, and the techniques used to exploit it. In later chapters, you'll gain hands-on experience finding and exploiting many of these vulnerabilities.

OWASP API SECURITY TOP 10

OWASP is a nonprofit foundation that creates free content and tools aimed at securing web applications. Due to the increasing prevalence of API vulnerabilities, OWASP released the OWASP API Security Top 10, a list of the 10 most common API vulnerabilities, at the end of 2019. Check out the project, which was led by API security experts Inon Shkedy and Erez Yalon, at <https://owasp.org/www-project-api-security>. In Chapter 15, I will demonstrate how the vulnerabilities described in the OWASP API Security Top 10 have been exploited in major breaches and bug bounty findings. We'll also use several OWASP tools to attack APIs in Parts II and III of the book.

Information Disclosure

When an API and its supporting software share sensitive information with unprivileged users, the API has an *information disclosure* vulnerability. Information may be disclosed in API responses or public sources such as code repositories, search results, news, social media, the target's website, and public API directories.

Sensitive data can include any information that attackers can leverage to their advantage. For example, a site that is using the WordPress API may unknowingly be sharing user information with anyone who navigates to the API path `/wp-json/wp/v2/users`, which returns all the WordPress usernames, or “slugs.” For instance, take a look at the following request:

```
GET https://www.sitename.org/wp-json/wp/v2/users
```

It might return this data:

```
[{"id":1,"name":"Administrator", "slug":"admin"}],  
{ "id":2,"name":"Vincent Valentine", "slug":"Vincent"}]
```

These slugs can then be used in an attempt to log in as the disclosed users with a brute-force, credential-stuffing, or password-spraying attack. (Chapter 8 describes these attacks in detail.)

Another common information disclosure issue involves verbose messaging. Error messaging helps API consumers troubleshoot their interactions with an API and allows API providers to understand issues with their application. However, it can also reveal sensitive information about resources, users, and the API's underlying architecture (such as the version of the web server or database). For example, say you attempt to authenticate to an API and receive an error message such as “the provided user ID does not exist.” Next, say you use another email and the error message changes to “incorrect password.” This lets you know that you've provided a legitimate user ID for the API.

Finding user information is a great way to start gaining access to an API. The following information can also be leveraged in an attack: software packages, operating system information, system logs, and software bugs. Generally, any information that can help us find more severe vulnerabilities or assist in exploitation can be considered an information disclosure vulnerability.

Often, you can gather the most information by interacting with an API endpoint and analyzing the response. API responses can reveal information within headers, parameters, and verbose errors. Other good sources of information are API documentation and resources gathered during reconnaissance. Chapter 6 covers many of the tools and techniques used for discovering API information disclosures.

Broken Object Level Authorization

One of the most prevalent vulnerabilities in APIs is *broken object level authorization* (BOLA). BOLA vulnerabilities occur when an API provider allows an API consumer access to resources they are not authorized to access. If an API endpoint does not have object-level access controls, it won't perform checks to make sure users can only access their own resources. When these controls are missing, User A will be able to successfully request User B's resources.

APIs use some sort of value, such as names or numbers, to identify various objects. When we discover these object IDs, we should test to see if we can interact with the resources of other users when unauthenticated or authenticated as a different user. For instance, imagine that we are authorized to access only the user Cloud Strife. We would send an initial GET request to <https://bestgame.com/api/v3/users?id=5501> and receive the following response:

```
{
  "id": "5501",
  "first_name": "Cloud",
  "last_name": "Strife",
  "link": "https://www.bestgame.com/user/strife.buster.97",
  "name": "Cloud Strife",
  "dob": "1997-01-31",
  "username": "strife.buster.97"
}
```

This poses no problem since we are authorized to access Cloud's information. However, if we are able to access another user's information, there is a major authorization issue.

In this situation, we might check for these problems by using another identification number that is close to Cloud's ID of 5501. Say we are able to obtain information about another user by sending a request for <https://bestgame.com/api/v3/users?id=5502> and receiving the following response:

```
{
  "id": "5502",
  "first_name": "Zack",

```

```
"last_name": "Fair",  
"link": " https://www.bestgame.com/user/shinra-number-1",  
"name": "Zack Fair",  
"dob": "2007-09-13",  
"username": "shinra-number-1"  
}
```

In this case, Cloud has discovered a BOLA. Note that predictable object IDs don't necessarily indicate that you've found a BOLA. For the application to be vulnerable, it must fail to verify that a given user is only able to access their own resources.

In general, you can test for BOLAs by understanding how an API's resources are structured and attempting to access resources you shouldn't be able to access. By detecting patterns within API paths and parameters, you should be able to predict other potential resources. The bolded elements in the following API requests should catch your attention:

```
GET /api/resource/1  
GET /user/account/find?user_id=15  
POST /company/account/Apple/balance  
POST /admin/pwreset/account/90
```

In these instances, you can probably guess other potential resources, like the following, by altering the bolded values:

```
GET /api/resource/3  
GET /user/account/find?user_id=23  
POST /company/account/Google/balance  
POST /admin/pwreset/account/111
```

In these simple examples, you've performed an attack by merely replacing the bolded items with other numbers or words. If you can successfully access information you shouldn't be authorized to access, you have discovered a BOLA vulnerability.

In Chapter 9, I will demonstrate how you can easily fuzz parameters like *user_id*= in the URL path and sort through the results to determine if a BOLA vulnerability exists. In Chapter 10, we will focus on attacking authorization vulnerabilities like BOLA and BFLA (broken function level authorization, discussed later in this chapter). BOLA can be a low-hanging API vulnerability that you can easily discover using pattern recognition and then prodding it with a few requests. Other times, it can be quite complicated to discover due to the complexities of object IDs and the requests used to obtain another user's resources.

Broken User Authentication

Broken user authentication refers to *any* weakness within the API authentication process. These vulnerabilities typically occur when an API provider either doesn't implement an authentication protection mechanism or implements a mechanism incorrectly.

API authentication can be a complex system that includes several processes with a lot of room for failure. A couple decades ago, security expert Bruce Schneier said, “The future of digital systems is complexity, and complexity is the worst enemy of security.” As we know from the six constraints of REST APIs discussed in Chapter 2, RESTful APIs are supposed to be stateless. In order to be stateless, the provider shouldn’t need to remember the consumer from one request to another. For this constraint to work, APIs often require users to undergo a registration process in order to obtain a unique token. Users can then include the token within requests to demonstrate that they’re authorized to make such requests.

As a consequence, the registration process used to obtain an API token, the token handling, and the system that generates the token could all have their own sets of weaknesses. To determine if the *token generation process* is weak, for example, we could collect a sampling of tokens and analyze them for similarities. If the token generation process doesn’t rely on a high level of randomness, or entropy, there is a chance we’ll be able to create our own token or hijack someone else’s.

Token handling could be the storage of tokens, the method of transmitting tokens across a network, the presence of hardcoded tokens, and so on. We might be able to detect hardcoded tokens in JavaScript source files or capture them as we analyze a web application. Once we’ve captured a token, we can use it to gain access to previously hidden endpoints or to bypass detection. If an API provider attributes an identity to a token, we would then take on the identity by hijacking the stolen token.

The other authentication processes that could have their own set of vulnerabilities include aspects of the *registration system*, such as the password reset and multifactor authentication features. For example, imagine a password reset feature requires you to provide an email address and a six-digit code to reset your password. Well, if the API allowed you to make as many requests as you wanted, you’d only have to make one million requests in order to guess the code and reset any user’s password. A four-digit code would require only 10,000 requests.

Also watch for the ability to access sensitive resources without being authenticated; API keys, tokens, and credentials used in URLs; a lack of rate-limit restrictions when authenticating; and verbose error messaging. For example, code committed to a GitHub repository could reveal a hardcoded admin API key:

```
"oauth_client":  
[{"client_id": "12345-abcd",  
  "client_type": "admin",  
  "api_key": "AIzaSyDrbTFCeb5k0yPSfL2heqdf-N19XoLxdw"}]
```

Due to the stateless nature of REST APIs, a publicly exposed API key is the equivalent of discovering a username and password. By using an exposed API key, you’ll assume the role associated with that key. In Chapter 6, we will use our reconnaissance skills to find exposed keys across the internet.

In Chapter 8, we will perform numerous attacks against API authentication, such as authentication bypass, brute-force attacks, credential stuffing, and a variety of attacks against tokens.

Excessive Data Exposure

Excessive data exposure is when an API endpoint responds with more information than is needed to fulfill a request. This often occurs when the provider expects the API consumer to filter results; in other words, when a consumer requests specific information, the provider might respond with all sorts of information, assuming the consumer will then remove any data they don't need from the response. When this vulnerability is present, it can be the equivalent of asking someone for their name and having them respond with their name, date of birth, email address, phone number, and the identification of every other person they know.

For example, if an API consumer requests information for their user account and receives information about other user accounts as well, the API is exposing excessive data. Suppose I requested my own account information with the following request:

```
GET /api/v3/account?name=Cloud+Strife
```

Now say I got the following JSON in the response:

```
{
  "id": "5501",
  "first_name": "Cloud",
  "last_name": "Strife",
  "privilege": "user",
  "representative": [
    {
      "name": "Don Corneo",
      "id": "2203",
      "email": "dcorn@gmail.com",
      "privilege": "super-admin",
      "admin": true,
      "two_factor_auth": false,
    }
  ]
}
```

I requested a single user's account information, and the provider responded with information about the person who created my account, including the administrator's full name, the admin's ID number, and whether the admin had two-factor authentication enabled.

Excessive data exposure is one of those awesome API vulnerabilities that bypasses every security control in place to protect sensitive information and hands it all to an attacker on a silver platter simply because they used the API. All you need to do to detect excessive data exposure is test your target API endpoints and review the information sent in response.

Lack of Resources and Rate Limiting

One of the more important vulnerabilities to test for is *lack of resources and rate limiting*. Rate limiting plays an important role in the monetization and availability of APIs. Without limiting the number of requests consumers can make, an API provider's infrastructure could be overwhelmed by the requests. Too many requests without enough resources will lead to the provider's systems crashing and becoming unavailable—a *denial of service (DoS)* state.

Besides potentially DoS-ing an API, an attacker who bypasses rate limits can cause additional costs for the API provider. Many API providers monetize their APIs by limiting requests and allowing paid customers to request more information. RapidAPI, for example, allows for 500 requests per month for free but 1,000 requests per month for paying customers. Some API providers also have infrastructure that automatically scales with the quantity of requests. In these cases, an unlimited number of requests would lead to a significant and easily preventable increase in infrastructure costs.

When testing an API that is supposed to have rate limiting, the first thing you should check is that rate limiting works, and you can do so by sending a barrage of requests to the API. If rate limiting is functioning, you should receive some sort of response informing you that you're no longer able to make additional requests, usually in the form of an HTTP 429 status code.

Once you are restricted from making additional requests, it's time to attempt to see how rate limiting is enforced. Can you bypass it by adding or removing a parameter, using a different client, or altering your IP address? Chapter 13 includes various measures for attempting to bypass rate limiting.

Broken Function Level Authorization

Broken function level authorization (BFLA) is a vulnerability where a user of one role or group is able to access the API functionality of another role or group. API providers will often have different roles for different types of accounts, such as public users, merchants, partners, administrators, and so on. A BFLA is present if you are able to use the functionality of another privilege level or group. In other words, BFLA can be a lateral move, where you use the functions of a similarly privileged group, or it could be a privilege escalation, where you are able to use the functions of a more privileged group. Particularly interesting API functions to access include those that deal with sensitive information, resources that belong to another group, and administrative functionality such as user account management.

BFLA is similar to BOLA, except instead of an authorization problem involving accessing resources, it is an authorization problem for performing actions. For example, consider a vulnerable banking API. When a BOLA vulnerability is present in the API, you might be able to access the information of other accounts, such as payment histories, usernames, email addresses, and account numbers. If a BFLA vulnerability is present, you might be able to transfer money and actually update the account information. BOLA is about unauthorized access, whereas BFLA is about unauthorized actions.

If an API has different privilege levels or roles, it may use different endpoints to perform privileged actions. For example, a bank may use the `/[user]/account/balance` endpoint for a user wishing to access their account information and the `/admin/account/[user]` endpoint for an administrator wishing to access user account information. If the application does not have access controls implemented correctly, we'll be able to perform administrative actions, such as seeing a user's full account details, by simply making administrative requests.

An API won't always use administrative endpoints for administrative functionality. Instead, the functionality could be based on HTTP request methods such as GET, POST, PUT, and DELETE. If a provider doesn't restrict the HTTP methods a consumer can use, simply making an unauthorized request with a different method could indicate a BFLA vulnerability.

When hunting for BFLA, look for any functionality you could use to your advantage, including altering user accounts, accessing user resources, and gaining access to restricted endpoints. For example, if an API gives partners the ability to add new users to the partner group but does not restrict this functionality to the specific group, any user could add themselves to any group. Moreover, if we're able to add ourselves to a group, there is a good chance we'll be able to access that group's resources.

The easiest way to discover BFLA is to find administrative API documentation and send requests as an unprivileged user that test admin functions and capabilities. Figure 3-1 shows the public Cisco Webex Admin API documentation, which provides a handy list of actions to attempt if you were testing Cisco Webex.

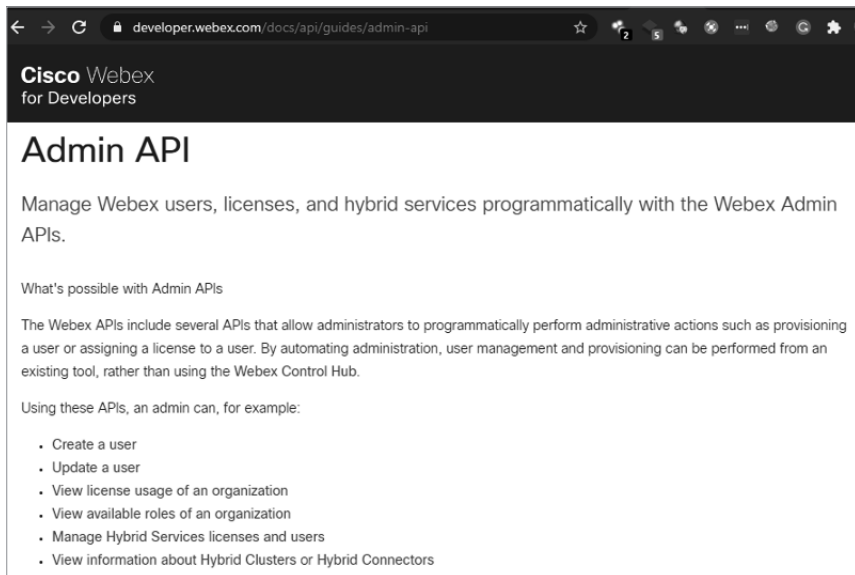


Figure 3-1: The Cisco Webex Admin API documentation

As an unprivileged user, make requests included in the admin section, such as attempting to create users, update user accounts, and so on. If access controls are in place, you'll likely receive an HTTP 401 Unauthorized or 403 Forbidden response. However, if you're able to make successful requests, you have discovered a BFLA vulnerability.

If API documentation for privileged actions is not available, you will need to discover or reverse engineer the endpoints used to perform privileged actions before testing them; more on this in Chapter 7. Once you've found administrative endpoints, you can begin making requests.

Mass Assignment

Mass assignment occurs when an API consumer includes more parameters in their requests than the application intended and the application adds these parameters to code variables or internal objects. In this situation, a consumer may be able to edit object properties or escalate privileges.

For example, an application might have account update functionality that the user should use only to update their username, password, and address. If the consumer can include other parameters in a request related to their account, such as the account privilege level or sensitive information like account balances, and the application accepts those parameters without checking them against a whitelist of permitted actions, the consumer could take advantage of this weakness to change these values.

Imagine an API is called to create an account with parameters for "User" and "Password":

```
{
  "User": "scuttleph1sh",
  "Password": "GreatPassword123"
}
```

While reading the API documentation regarding the account creation process, suppose you discover that there is an additional key, "isAdmin", that consumers can use to become administrators. You could use a tool like Postman or Burp Suite to add the attribute to a request and set the value to true:

```
{
  "User": "scuttleph1sh",
  "Password": "GreatPassword123",
  "isAdmin": true
}
```

If the API does not sanitize the request input, it is vulnerable to mass assignment, and you could use the updated request to create an admin account. On the backend, the vulnerable web app will add the key/value attribute, {"isAdmin": "true"}, to the user object and make the user the equivalent of an administrator.

You can discover mass assignment vulnerabilities by finding interesting parameters in API documentation and then adding those parameters to a request. Look for parameters involved in user account properties, critical functions, and administrative actions. Intercepting API requests and responses could also reveal parameters worthy of testing. Additionally, you can guess parameters or fuzz them in API requests. (Chapter 9 describes the art of fuzzing.)

Security Misconfigurations

Security misconfigurations include all the mistakes developers could make within the supporting security configurations of an API. If a security misconfiguration is severe enough, it can lead to sensitive information exposure or a complete system takeover. For example, if the API's supporting security configuration reveals an unpatched vulnerability, there is a chance that an attacker could leverage a published exploit to easily "pwn" the API and its system.

Security misconfigurations are really a set of weaknesses that includes misconfigured headers, misconfigured transit encryption, the use of default accounts, the acceptance of unnecessary HTTP methods, a lack of input sanitization, and verbose error messaging.

A *lack of input sanitization* can allow attackers to upload malicious payloads to the server. APIs often play a key role in automating processes, so imagine being able to upload payloads that the server automatically processes into a format that could be remotely executed or executed by an unsuspecting end user. For example, if an upload endpoint was used to pass uploaded files to a web directory, it could allow the upload of a script. Navigating to the URL where the file is located could launch the script, resulting in direct shell access to the web server. Additionally, lack of input sanitization can lead to unexpected behavior on the part of the application. In Part III, we will fuzz API inputs in attempts to discover vulnerabilities such as security misconfigurations, improper assets management, and injection weaknesses.

API providers use *headers* to provide the consumer with instructions for handling the response and security requirements. Misconfigured headers can result in sensitive information disclosure, downgrade attacks, and cross-site scripting attacks. Many API providers will use additional services alongside their API to enhance API-related metrics or to improve security. It is fairly common for those additional services to add headers to requests for metrics and perhaps serve as some level of assurance to the consumer. For example, take the following response:

```
HTTP/ 200 OK
--snip--
X-Powered-By: VulnService 1.11
X-XSS-Protection: 0
X-Response-Time: 566.43
```

The X-Powered-By header reveals backend technology. Headers like this one will often advertise the exact supporting service and its version. You could use information like this to search for exploits published for that version of software.

X-XSS-Protection is exactly what it looks like: a header meant to prevent cross-site scripting (XSS) attacks. XSS is a common type of injection vulnerability where an attacker can insert scripts into a web page and trick end users into clicking malicious links. We will cover XSS and cross-API scripting (XAS) in Chapter 12. An X-XSS-Protection value of 0 indicates no protections are in place, and a value of 1 indicates that protection is turned on. This header, and others like it, clearly reveals whether a security control is in place.

The X-Response-Time header is middleware that provides usage metrics. In the previous example, its value represents 566.43 milliseconds. However, if the API isn't configured properly, this header can function as a side channel used to reveal existing resources. If the X-Response-Time header has a consistent response time for nonexistent records, for example, but increases its response time for certain other records, this could be an indication that those records exist. Here's an example:

HTTP/UserA 404 Not Found

--snip--

X-Response-Time: 25.5

HTTP/UserB 404 Not Found

--snip--

X-Response-Time: 25.5

HTTP/UserC 404 Not Found

--snip--

X-Response-Time: 510.00

In this case, UserC has a response time value that is 20 times the response time of the other resources. With this small sample size, it is hard to definitively conclude that UserC exists. However, imagine you have a sample of hundreds or thousands of requests and know the average X-Response-Time values for certain existing and nonexistent resources. Say, for instance, you know that a bogus account like `/user/account/thisdefinitelydoesnotexist876` has an average X-Response-Time of 25.5 ms. You also know that your existing account `/user/account/1021` receives an X-Response-Time of 510.00. If you then sent requests brute-forcing all account numbers from 1000 to 2000, you could review the results and see which account numbers resulted in drastically increased response times.

Any API providing sensitive information to consumers should use Transport Layer Security (TLS) to encrypt the data. Even if the API is only provided internally, privately, or at a partner level, using TLS, the protocol that encrypts HTTPS traffic, is one of the most basic ways to ensure that API requests and responses are protected when being passed across a network. Misconfigured or missing transit encryption can cause API users to pass sensitive API information in cleartext across networks, in which case

an attacker could capture the responses and requests with a man-in-the-middle (MITM) attack and read them plainly. The attacker would need to have access to the same network as the person they were attacking and then intercept the network traffic with a network protocol analyzer such as Wireshark to see the information being communicated between the consumer and the provider.

When a service uses a *default account and credentials* and the defaults are known, an attacker can use those credentials to assume the role of that account. This could allow them to gain access to sensitive information or administrative functionality, potentially leading to a compromise of the supporting systems.

Lastly, if an API provider allows *unnecessary HTTP methods*, there is an increased risk that the application won't handle these methods properly or will result in sensitive information disclosure.

You can detect several of these security misconfigurations with web application vulnerability scanners such as Nessus, Qualys, OWASP ZAP, and Nikto. These scanners will automatically check the web server version information, headers, cookies, transit encryption configuration, and parameters to see if expected security measures are missing. You can also check for these security misconfigurations manually, if you know what you are looking for, by inspecting the headers, SSL certificate, cookies, and parameters.

Injectons

Injection flaws exist when a request is passed to the API's supporting infrastructure and the API provider doesn't filter the input to remove unwanted characters (a process known as *input sanitization*). As a result, the infrastructure might treat data from the request as code and run it. When this sort of flaw is present, you'll be able to conduct injection attacks such as SQL injection, NoSQL injection, and system command injection.

In each of these injection attacks, the API delivers your unsanitized payload directly to the operating system running the application or its database. As a result, if you send a payload containing SQL commands to a vulnerable API that uses a SQL database, the API will pass the commands to the database, which will process and perform the commands. The same will happen with vulnerable NoSQL databases and affected systems.

Verbose error messaging, HTTP response codes, and unexpected API behavior can all be clues that you may have discovered an injection flaw. Say, for example, you were to send `OR 1=0--` as an address in an account registration process. The API may pass that payload directly to the backend SQL database, where the `OR 1=0` statement would fail (because 1 does not equal 0), causing some SQL error:

```
POST /api/v1/register HTTP 1.1
Host: example.com
--snip--
{
  "Fname": "hAPI",
```



```
"lname": "Hacker",  
"Address": "' OR 1=0--",  
}
```

An error in the backend database could show up as a response to the consumer. In this case, you might receive a response like “Error: You have an error in your SQL syntax. . . .” Any response directly from a database or the supporting system is a clear indicator that there is an injection vulnerability.

Injection vulnerabilities are often complemented by other vulnerabilities such as poor input sanitization. In the following example, you can see a code injection attack that uses an API GET request to take advantage of a weak query parameter. In this case, the weak query parameter passes any data in the query portion of the request directly to the underlying system, without sanitizing it first:

```
GET http://10.10.78.181:5000/api/v1/resources/books?show=/etc/passwd
```

The following response body shows that the API endpoint has been manipulated into displaying the host’s `/etc/passwd` file, revealing users on the system:

```
root:x:0:0:root:/root:/bin/bash  
daemon:x:1:1:daemon:/usr/sbin:/usr/sbin/nologin  
bin:x:2:2:bin:/dev:/usr/sbin/nologin  
sync:x:4:65534:sync:/bin:/bin/sync  
games:x:5:60:games:/usr/games:/usr/sbin/nologin  
man:x:6:12:man:/var/cache/man:/usr/sbin/nologin  
lp:x:7:7:lp:/var/spool/lpd:/usr/sbin/nologin  
mail:x:8:8:mail:/var/mail:/usr/sbin/nologin  
news:x:9:9:news:/var/spool/news:/usr/sbin/nologin
```

Finding injection flaws requires diligently testing API endpoints, paying attention to how the API responds, and then crafting requests that attempt to manipulate the backend systems. Like directory traversal attacks, injection attacks have been around for decades, so there are many standard security controls to protect API providers from them. I will demonstrate various methods for performing injection attacks, encoding traffic, and bypassing standard controls in Chapters 12 and 13.

Improper Assets Management

Improper assets management takes place when an organization exposes APIs that are either retired or still in development. As with any software, old API versions are more likely to contain vulnerabilities because they are no longer being patched and upgraded. Likewise, APIs that are still being developed are typically not as secure as their production API counterparts.

Improper assets management can lead to other vulnerabilities, such as excessive data exposure, information disclosure, mass assignment, improper rate limiting, and API injection. For attackers, this means that

discovering an improper assets management vulnerability is only the first step toward further exploitation of an API.

You can discover improper assets management by paying close attention to outdated API documentation, changelogs, and version history on repositories. For example, if an organization's API documentation has not been updated along with the API's endpoints, it could contain references to portions of the API that are no longer supported. Organizations often include versioning information in their endpoint names to distinguish between older and newer versions, such as `/v1/`, `/v2/`, `/v3/`, and so on. APIs still in development often use paths such as `/alpha/`, `/beta/`, `/test/`, `/uat/`, and `/demo/`. If you know that an API is now using `apiv3.org/admin` but part of the API documentation refers to `apiv1.org/admin`, you could try testing different endpoints to see if `apiv1` or `apiv2` is still active. Additionally, the organization's changelog may disclose the reasons why `v1` was updated or retired. If you have access to `v1`, you can test for those weaknesses.

Outside of using documentation, you can discover improper assets management vulnerabilities through the use of guessing, fuzzing, or brute-force requests. Watch for patterns in the API documentation or path-naming scheme, and then make requests based on your assumptions.

Business Logic Vulnerabilities

Business logic vulnerabilities (also known as *business logic flaws*, or *BLFs*) are intended features of an application that attackers can use maliciously. For example, if an API has an upload feature that doesn't validate encoded payloads, a user could upload any file as long as it was encoded. This would allow end users to upload and execute arbitrary code, including malicious payloads.

Vulnerabilities of this sort normally come about from an assumption that API consumers will follow directions, be trustworthy, or only use the API in a certain way. In those cases, the organization essentially depends on trust as a security control by expecting the consumer to act benevolently. Unfortunately, even good-natured API consumers make mistakes that could lead to a compromise of the application.

The Experian partner API leak, in early 2021, was a great example of an API trust failure. A certain Experian partner was authorized to use Experian's API to perform credit checks, but the partner added the API's credit check functionality to their web application and inadvertently exposed all partner-level requests to users. A request could be intercepted when using the partner's web application, and if it included a name and address, the Experian API would respond with the individual's credit score and credit risk factors. One of the leading causes of this business logic vulnerability was that Experian trusted the partner not to expose the API.

Another problem with trust is that credentials, such as API keys, tokens, and passwords, are constantly being stolen and leaked. When a trusted consumer's credentials are stolen, the consumer can become a wolf in sheep's

clothing and wreak havoc. Without strong technical controls in place, business logic vulnerabilities can often have the most significant impact, leading to exploitation and compromise.

You can search API documentation for telltale signs of business logic vulnerabilities. Statements like the following should illuminate the lightbulb above your head:

“Only use feature X to perform function Y.”

“Do not do X with endpoint Y.”

“Only admins should perform request X.”

These statements may indicate that the API provider is trusting that you won’t do any of the discouraged actions, as instructed. When you attack their API, make sure to disobey such requests to test for the presence of security controls.

Another business logic vulnerability comes about when developers assume that consumers will exclusively use a browser to interact with the web application and won’t capture API requests that take place behind the scenes. All it takes to exploit this sort of weakness is to intercept requests with a tool like Burp Suite Proxy or Postman and then alter the API request before it is sent to the provider. This could allow you to capture shared API keys or use parameters that could negatively impact the security of the application.

As an example, consider a web application authentication portal that a user would normally employ to authenticate to their account. Say the web application issued the following API request:

```
POST /api/v1/login HTTP 1.1
Host: example.com
--snip--
UserId=hapihacker&password=arealpassword!&MFA=true
```

There is a chance that we could bypass multifactor authentication by simply altering the parameter MFA to false.

Testing for business logic flaws can be challenging because each business is unique. Automated scanners will have a difficult time detecting these issues, as the flaws are part of the API’s intended use. You must understand how the business and API operate and then consider how you could use these features to your advantage. Study the application’s business logic with an adversarial mindset, and try breaking any assumptions that have been made.

Summary

In this chapter, I covered common API vulnerabilities. It is important to become familiar with these vulnerabilities so that you can easily recognize them, take advantage of them during an engagement, and report them

back to the organization to prevent the criminals from dragging your client into the headlines.

Now that you are familiar with web applications, APIs, and their weaknesses, it is time to prepare your hacking machine and get your hands busy on the keyboard.