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 conversa-

tion 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*.

```
POST❶ /sessions❷ HTTP/1.1❸
Host: twitter.com❹
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&❺password%5D=NotMyPassword❻%21❼
```

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; P

<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.

Method	Purpose
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 connec-

tion. 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	mysterioushadow

Table 1-4: *A Relational Email Table*

UserID	Email
111	*hapi_hacker@email.com*
112	*scuttleph1sh@email.com*
113	*mysterioushadow@email.com*

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 spe-

cializing 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).