# 24

# API HACKING

*Application programming interfaces (APIs)* are a way for programs to communicate with each other, and they power a wide variety of applications. As applications become more complex, developers are increasingly using APIs to combine components of an application or multiple applications belonging to the same organization. And more and more, APIs have the ability to execute important actions or communicate sensitive information.

In this chapter, we'll talk about what APIs are, how they work, and how you can find and exploit API vulnerabilities.

## What Are APIs?

In simple terms, an API is a set of rules that allow one application to communicate with another. They enable applications to share data in a controlled way. Using APIs, applications on the internet can take advantage of other applications' resources to build more complex features.

For example, consider Twitter's API (*https://developer.twitter.com/en/docs/twitter-api/*). This public API allows outside developers to access Twitter's data and actions. For example, if a developer wants their code to retrieve the contents of a tweet from Twitter's database, they can use a Twitter API endpoint that returns tweet information by sending a GET request to the Twitter API server located at *api.twitter.com*:

```
GET /1.1/statuses/show.json?id=210462857140252672
Host: api.twitter.com
```

This URL indicates that the developer is using Twitter's API version 1.1 and requesting the resource called `statuses` (which is what Twitter calls its tweets) with the ID 210462857140252672. The `id` field in the URL is a request parameter required by the API endpoint. API endpoints often require certain parameters to determine which resource to return.

Twitter's API server would then return the data in JSON format to the requesting application (this example is taken from Twitter's public API documentation):

```
❶ {
❷ "created_at": "Wed Oct 10 20:19:24 +0000 2018",
  "id": 1050118621198921728,
  "id_str": "1050118621198921728",
  "text": "To make room for more expression, we will now count all emojis
as equal—including those with gender... and skin t... https://t.co/
MkGjXf9aXm",
  "truncated": true,
  "entities": {
  ❸ "hashtags": [],
    "symbols": [],
    "user_mentions": [],
    "urls": [
      {
        "url": "https://t.co/MkGjXf9aXm",
        "expanded_url": "https://twitter.com/i/web/
status/1050118621198921728",
        "display_url": "twitter.com/i/web/status/1...",
        "indices": [
          117,
          140
        ]
      }
    ]
  },
  ❹ "user": {
    "id": 6253282,
    "id_str": "6253282",
    "name": "Twitter API",
    "screen_name": "TwitterAPI",
    "location": "San Francisco, CA",
    "description": "The Real Twitter API. Tweets about API changes, service
issues and our Developer Platform.
Don't get an answer? It's on my website.",
```

```
[...]
❶ }
```

APIs usually return data in JSON or XML format. JSON is a way to represent data in plaintext, and it's commonly used to transport data within web messages. You'll often see JSON messages when you're testing applications, so it's helpful to learn how to read them.

JSON objects start and end with a curly bracket ❶. Within these curly brackets, the properties of the represented object are stored in key-value pairs. For example, in the preceding data block representing a tweet, the `created_at` property has the value `Wed Oct 10 20:19:24 +0000 2018`. This indicates that the tweet was created on Wednesday, October 10, 2018 at 8:19 PM ❷.

JSON objects can also contain lists or other objects. Curly brackets denote objects. The preceding tweet contains a `user` object indicating the user who created the tweet ❹. Lists are denoted with square brackets. Twitter returned an empty list of hashtags in the preceding JSON block, which means no hashtags were used in the tweet ❸.

You might be wondering how the API server decides who can access data or execute actions. APIs often require users to authenticate before accessing their services. Typically, users include access tokens in their API requests to prove their identities. Other times, users are required to use special authentication headers or cookies. The server would then use the credentials presented in the request to determine which resources and actions the user should access.

### REST APIs

There are multiple kinds of APIs. The Twitter API discussed here is called a *Representational State Transfer (REST)* API. REST is one of the most commonly used API structures. Most of the time, REST APIs return data in either JSON or plaintext format. REST API users send requests to specific resource endpoints to access that resource. In Twitter's case, you send GET requests to *https://api.twitter.com/1.1/statuses/show/* to retrieve tweet information, and GET requests to *https://api.twitter.com/1.1/users/show/* to retrieve user information.

REST APIs usually have defined structures for queries that make it easy for users to predict the specific endpoints to which they should send their requests. For example, to delete a tweet via the Twitter API, users can send a POST request to *https://api.twitter.com/1.1/statuses/destroy/*, and to retweet a tweet, users can send a POST request to *https://api.twitter.com/1.1/statuses/retweet/*. You can see here that all of Twitter's API endpoints are structured in the same way (*https://api.twitter.com/1.1/RESOURCE/ACTION*):

```
https://api.twitter.com/1.1/users/show
https://api.twitter.com/1.1/statuses/show
https://api.twitter.com/1.1/statuses/destroy
https://api.twitter.com/1.1/statuses/retweet
```

REST APIs can also use various HTTP methods. For example, GET is usually used to retrieve resources, POST is used to update or create resources, PUT is used to update resources, and DELETE is used to delete them.

### SOAP APIs

*SOAP* is an API architecture that is less commonly used in modern applications. But plenty of older apps and IoT apps still use SOAP APIs. SOAP APIs use XML to transport data, and their messages have a header and a body. A simple SOAP request looks like this:

```
DELETE / HTTPS/1.1
Host: example.s3.amazonaws.com

<DeleteBucket xmlns="http://doc.s3.amazonaws.com/2006-03-01">
  <Bucket>quotes</Bucket>
  <AWSAccessKeyId> AKIAIOSFODNN7EXAMPLE</AWSAccessKeyId>
  <Timestamp>2006-03-01T12:00:00.183Z</Timestamp>
  <Signature>Iuyz3d3P0aTou39dzbqaEXAMPLE=</Signature>
  </DeleteBucket>
```

This example request is taken from Amazon S3's SOAP API documentation. It deletes an S3 bucket named *quotes*. As you can see, API request parameters are passed to the server as tags within the XML document.

The SOAP response looks like this:

```
<DeleteBucketResponse xmlns="http://s3.amazonaws.com/doc/2006-03-01">
  <DeleteBucketResponse>
    <Code>204</Code>
    <Description>No Content</Description>
  </DeleteBucketResponse>
</DeleteBucketResponse>
```

This response indicates that the bucket is successfully deleted and no longer found.

SOAP APIs have a service called *Web Services Description Language (WSDL)*, used to describe the structure of the API and how to access it. If you can find the WSDL of a SOAP API, you can use it to understand the API before hacking it. You can often find WSDL files by adding *.wsdl* or *?wsdl* to the end of an API endpoint or searching for URL endpoints containing the term *wsdl*. In the WSDL, you will be able to find a list of API endpoints you can test.

### GraphQL APIs

*GraphQL* is a newer API technology that allows developers to request the precise resource fields they need, and to fetch multiple resources with just a single API call. GraphQL is becoming increasingly common because of these benefits.

GraphQL APIs use a custom query language and a single endpoint for all the API's functionality. These endpoints are commonly located at

*/graphql*, */gql*, or */g*. GraphQL has two main kinds of operations: queries and mutations. *Queries* fetch data, just like the GET requests in REST APIs. *Mutations* create, update, and delete data, just like the POST, PUT, and DELETE requests in REST APIs.

As an example, take a look at the following API requests to Shopify's GraphQL API. Shopify is an e-commerce platform that allows users to interact with their online stores via a GraphQL API. To access Shopify's GraphQL API, developers need to send POST requests to the endpoint *https://SHOPNAME.myshopify.com/admin/api/API_VERSION/graphql.json* with the GraphQL query in the POST request body. To retrieve information about your shop, you can send this request:

```
query {
   shop {
     name
     primaryDomain {
       url
       host
     }
   }
 }
```

This GraphQL query indicates that we want to retrieve the name and `primaryDomain` of the shop, and that we need only the `primaryDomain`'s URL and host properties.

Shopify's server will return the requested information in JSON format:

```
{
   "data": {
     "shop": {
       "name": "example",
       "primaryDomain": {
         "url": "https://example.myshopify.com",
         "host": "example.myshopify.com"
       }
     }
   }
}
```

Notice that the response doesn't contain all the object's fields, but instead the exact fields the user has requested. Depending on your needs, you can request either more or fewer fields of the same data object. Here is an example that requests fewer:

```
query {
   shop {
     name
   }
 }
```

You can also request the precise subfields of a resource's properties and other nested properties. For example, here, you request only the URL of the `primaryDomain` of a shop:

```
query {
   shop {
     primaryDomain {
       url
     }
   }
 }
```

These queries are all used to retrieve data.

Mutations, used to edit data, can have arguments and return values. Let's take a look at an example of a mutation taken from *graphql.org*. This mutation creates a new customer record and takes three input parameters: `firstName`, `lastName`, and `email`. It then returns the ID of the newly created customer:

```
mutation {
  customerCreate(
    input: {
      firstName: "John",
      lastName: "Tate",
      email: "john@johns-apparel.com" })
     {
     customer {
       id
     }
   }
}
```

GraphQL's unique syntax might make testing it hard at first, but once you understand it, you can test these APIs the same way that you test other types of APIs. To learn more about GraphQL's syntax, visit *https://graphql.org/*.

GraphQL APIs also include a great reconnaissance tool for bug hunters: a feature called *introspection* that allows API users to ask a GraphQL system for information about itself. In other words, they're queries that return information about how to use the API. For example, `__schema` is a special field that will return all the types available in the API; the following query will return all the type names in the system. You can use it to find data types you can query for:

```
{
  __schema {
    types {
      name
    }
  }
}
```

You can also use the `__type` query to find the associated fields of a particular type:

```
{
  __type(name: "customer") {
    name
    fields {
      name
    }
  }
}
```

You will get the fields of a type returned like this. You can then use this information to query the API:

```
{
  "data": {
    "__type": {
      "name": "customer",
      "fields": [
        {
          "name": "id",
        },
        {
          "name": "firstName",
        },
        {
          "name": "lastName",
        },
        {
          "name": "email",
        }
      ]
    }
  }
}
```

Introspection makes recon a breeze for the API hacker. To prevent malicious attackers from enumerating their APIs, many organizations disable introspection in their GraphQL APIs.

### API-Centric Applications

Increasingly, APIs aren't used as simply a mechanism to share data with outside developers. You'll also encounter *API-centric applications*, or applications built using APIs. Instead of retrieving complete HTML documents from the server, API-centric apps consist of a client-side component that requests and renders data from the server by using API calls.

For example, when a user views Facebook posts, Facebook's mobile application uses API calls to retrieve data about the posts from the server instead of retrieving entire HTML documents containing embedded data. The application then renders that data on the client side to form web pages.

Many mobile applications are built this way. When a company already has a web app, using an API-centric approach to build mobile apps saves time. APIs allow developers to separate the app's rendering and data-transporting tasks: developers can use API calls to transport data and then build a separate rendering mechanism for mobile, instead of reimplementing the same functionalities.

Yet the rise of API-centric applications means that companies and applications expose more and more of their data and functionalities through APIs. APIs often leak sensitive data and the application logic of the hosting application. As you'll see, this makes API bugs a widespread source of security breaches and a fruitful target for bug hunters.

## Hunting for API Vulnerabilities

Let's explore some of the vulnerabilities that affect APIs and the steps you can take to discover them. API vulnerabilities are similar to the ones that affect non-API web applications, so make sure you have a good understanding of the bugs we've discussed up to this point. That said, when testing APIs, you should focus your testing on the vulnerabilities listed in this section, because they are prevalent in API implementations.

Before we dive in, there are many open source API development and testing tools that you can use to make the API testing process more efficient. Postman (*https://www.postman.com/*) is a handy tool that will help you test APIs. You can use Postman to craft complex API requests from scratch and manage the large number of test requests that you will be sending. GraphQL Playground (*https://github.com/graphql/graphql-playground/*) is an IDE for crafting GraphQL queries that has autocompletion and error highlighting.

ZAP has a GraphQL add-on (*https://www.zaproxy.org/blog/2020-08-28-introducing-the-graphql-add-on-for-zap/*) that automates GraphQL introspection and test query generation. Clairvoyance (*https://github.com/nikitastupin/clairvoyance/*) helps you gain insight into a GraphQL API's structure when introspection is disabled.

### *Performing Recon*

First, hunting for API vulnerabilities is very much like hunting for vulnerabilities in regular web applications in that it requires recon. The most difficult aspect of API testing is knowing what the application expects and then tailoring payloads to manipulate its functionality.

If you're hacking a GraphQL API, you might start by sending introspection queries to figure out the API's structure. If you are testing a SOAP API, start by looking for the WSDL file. If you're attacking a REST or SOAP API, or if introspection is disabled on the GraphQL API you're attacking, start by enumerating the API. *API enumeration* refers to the process of identifying as many of the API's endpoints as you can so you can test as many endpoints as possible.

To enumerate the API, start by reading the API's public documentation if it has one. Companies with public APIs often publish detailed documentation about the API's endpoints and their parameters. You should be able to find public API documentations by searching the internet for *company_name API* or *company_name developer docs*. This documentation provides a good start for enumerating API endpoints, but don't be fooled into thinking that the official documentation contains all the endpoints you can test! APIs often have public and private endpoints, and only the public ones will be found in these developer guides.

Try using Swagger (*https://swagger.io/*), a toolkit developers use for developing APIs. Swagger includes a tool for generating and maintaining API documentation that developers often use to document APIs internally. Sometimes companies don't publicly publish their API documentation but forget to lock down internal documentation hosted on Swagger. In this case, you can find the documentation by searching the internet for *company_name inurl:swagger*. This documentation often includes all API endpoints, their input parameters, and sample responses.

The next thing you can do is go through all the application workflows to capture API calls. You can do this by browsing the company's applications with an intercepting proxy recording HTTP traffic in the background. You might find API calls used in the application's workflow that aren't in public documentation.

Using the endpoints you've found, you can try to deduce other endpoints. For instance, REST APIs often have a predictable structure, so you can deduce new endpoints by studying existing ones. If both */posts/POST_ID/read* and */posts/POST_ID/delete* exist, is there an endpoint called */posts/POST_ID/edit*? Similarly, if you find blog posts located at */posts/1234* and */posts/1236*, does */posts/1235* also exist?

Next, search for other API endpoints by using recon techniques from Chapter 5, such as studying JavaScript source code or the company's public GitHub repositories. You can also try to generate error messages in hopes that the API leaks information about itself. For example, try to provide unexpected data types or malformed JSON code to the API endpoints. Fuzzing techniques can also help you find additional API endpoints by using a wordlist. Many online wordlists are tailored for fuzzing API endpoints; one example wordlist is at *https://gist.github.com/yassineaboukir/8e12adefbd505ef704674ad6ad48743d/*. We will talk more about how to fuzz an endpoint in Chapter 25.

Also note that APIs are often updated. While the application might not actively use older versions of the API, these versions might still elicit a response from the server. For every endpoint you find in a later version of the API, you should test whether an older version of the endpoint works. For example, if the */api/**v2**/user_emails/52603991338963203244* endpoint exists, does this one: */api/**v1**/user_emails/52603991338963203244*? Older versions of an API often contain vulnerabilities that have been fixed in newer versions, so make sure to include finding older API endpoints in your recon strategy.

Finally, take the time to understand each API endpoint's functionality, parameters, and query structure. The more you can learn about how an API works, the more you'll understand how to attack it. Identify all the possible user data input locations for future testing. Look out for any authentication mechanisms, including these:

- What access tokens are needed?
- Which endpoints require tokens and which do not?
- How are access tokens generated?
- Can users use the API to generate a valid token without logging in?
- Do access tokens expire when updating or resetting passwords?

Throughout your recon process, make sure to take lots of notes. Document the endpoints you find and their parameters.

### Testing for Broken Access Control and Info Leaks

After recon, I like to start by testing for access-control issues and info leaks. Most APIs use access tokens to determine the rights of the client; they issue access tokens to each API client, and clients use these to perform actions or retrieve data. If these API tokens aren't properly issued and validated, attackers might bypass authentication and access data illegally.

For example, sometimes API tokens aren't validated after the server receives them. Other times, API tokens are not randomly generated and can be predicted. Finally, some API tokens aren't invalidated regularly, so attackers who've stolen tokens maintain access to the system indefinitely.

Another issue is broken resource or function-level access control. Sometimes API endpoints don't have the same access-control mechanisms as the main application. For example, say a user with a valid API key can retrieve data about themselves. Can they also read data about other users? Or can they perform actions on another's behalf through the API? Finally, can a regular user without admin privileges read data from endpoints restricted to admins? Separately from REST or SOAP APIs, the GraphQL API of an application may have its own authorization mechanisms and configuration. This means that you can test for access-control issues on GraphQL endpoints even though the web or REST API of an application is secure. These issues are similar to the IDOR vulnerabilities discussed in Chapter 10.

Other times still, an API offers multiple ways to perform the same action, and access control isn't implemented across all of them. For example, let's say that a REST API has two ways of deleting a blog post: sending a POST request to */posts/POST_ID/delete* and sending a DELETE request to */posts/POST_ID*. You should ask yourself: are the two endpoints subject to the same access controls?

Another common API vulnerability is information leaks. API endpoints often return more information than they should, or than is needed to render the web page. For example, I once found an API endpoint that populated a user's profile page. When I visited someone else's profile page, an API call was used to return the profile owner's information. At first glance, the profile

page didn't leak any sensitive information, but the API response used to fetch the user's data actually returned the profile owner's private API token as well! After an attacker steals the victim's API token by visiting their profile page, they could impersonate the victim by using this access token.

Make a list of the endpoints that should be restricted by some form of access control. For each of these endpoints, create two user accounts with different levels of privilege: one that should have access to the functionality and one that shouldn't. Test whether you can access the restricted functionality with the lower-privileged account.

If your lower-privileged user can't access the restricted functionality, try removing access tokens, or adding additional parameters like the cookie `admin=1` to the API call. You can also switch out the HTTP request methods, including GET, POST, PUT, PATCH, and DELETE, to see if access control is properly implemented across all methods. For example, if you can't edit another user's blog posts via a POST request to an API endpoint, can you bypass the protection by using a PUT request instead?

Try to view, modify, and delete other users' info by switching out user IDs or other user identification parameters found in the API calls. If IDs used to identify users and resources are unpredictable, try to leak IDs through info leaks from other endpoints. For example, I once found an API endpoint that returned user information; it revealed the user's ID as well as all of the user's friends' IDs. With the ID of both the user and their friend, I was able to access messages sent between the two users. By combining two info leaks and using just the user IDs, I was able to read a user's private messages!

In GraphQL, a common misconfiguration is allowing lower-privileged users to modify a piece of data that they should not via a mutation request. Try to capture GraphQL queries allowed from one user's account, and see if you can send the same query and achieve the same results from another who shouldn't have permission.

While hunting for access control issues, closely study the data being sent back by the server. Don't just look at the resulting HTML page; dive into the raw API response, as APIs often return data that doesn't get displayed on the web page. You might be able to find sensitive information disclosures in the response body. Is the API endpoint returning any private user information, or sensitive information about the organization? Should the returned information be available to the current user? Does the returned information pose a security risk to the company?

### *Testing for Rate-Limiting Issues*

APIs often lack rate limiting; in other words, the API server doesn't restrict the number of requests a client or user account can send within a short time frame. A lack of rate limiting in itself is a low-severity vulnerability unless it's proven to be exploitable by attackers. But on critical endpoints, a lack of rate limiting means that malicious users can send large numbers of requests to the server to harvest database information or brute-force credentials.

Endpoints that can be dangerous when not rate limited include authentication endpoints, endpoints not protected by access control, and endpoints that return large amounts of sensitive data. For example, I once encountered an API endpoint that allows users to retrieve their emails via an email ID, like this:

```
GET /api/v2/user_emails/5260399133896320 3244
```

This endpoint isn't protected by any access control. Since this endpoint isn't rate limited, either, an attacker can essentially guess the email ID field by sending numerous requests. Once they've guessed a valid ID, they can access another user's private email.

To test for rate-limiting issues, make large numbers of requests to the endpoint. You can use the Burp intruder or `curl` to send 100 to 200 requests in a short time. Make sure you repeat the test in different authentication stages, because users with different privilege levels can be subject to different rate limits.

Be really careful when you are testing for rate-limiting issues because it's very possible to accidentally launch a DoS attack on the app by drowning it with requests. You should obtain written permission before conducting rate-limiting tests and time-throttle your requests according to the company's policies.

Also keep in mind that applications could have rate limits that are higher than your testing tools' capabilities. For instance, applications could set a rate limit of 400 requests a second, and your tooling may not be capable of reaching that limit.

### *Testing for Technical Bugs*

Many of the bugs that we've discussed in this book so far—such as SQL injection, deserialization issues, XXEs, template injections, SSRF, and RCEs—are caused by improper input validation. Sometimes developers forget to implement proper input validation mechanisms for APIs.

APIs are therefore susceptible to many of the other vulnerabilities that affect regular web applications too. Since APIs are another way applications accept user input, they become another way for attackers to smuggle malicious input into the application's workflow.

If an API endpoint can access external URLs, it might be vulnerable to SSRF, so you should check whether its access to internal URLs isn't restricted. Race conditions can also happen within APIs. If you can use API endpoints to access application features affected by race conditions, these endpoints can become an alternative way to trigger the race condition.

Other vulnerabilities, like path traversal, file inclusion, insecure deserialization issues, XXE, and XSS can also happen. If an API endpoint returns internal resources via a filepath, attackers might use that endpoint to read sensitive files stored on the server. If an API endpoint used for file uploads

doesn't limit the data type that users can upload, attackers might upload malicious files, such as web shells or other malware, to the server. APIs also commonly accept user input in serialized formats such as XML. In this case, insecure deserialization or XXEs can happen. RCEs via file upload or XXEs are commonly seen in API endpoints. Finally, if an API's URL parameters are reflected in the response, attackers can use that API endpoint to trigger reflected XSS on victims' browsers.

The process of testing for these issues will be similar to testing for them in a regular web app. You'll simply supply the payloads to the application in API form.

For example, for vulnerabilities like path traversals and file-inclusion attacks, look out for absolute and relative filepaths in API endpoints and try to mess with the path parameters. If an API endpoint accepts XML input, try to insert an XXE payload into the request. And if the endpoint's URL parameters are reflected in the response, see if you can trigger a reflected XSS by placing a payload in the URL.

You can also utilize fuzz-testing techniques, which we'll discuss in Chapter 25, to find these vulnerabilities.

Applications are becoming increasingly reliant on APIs, even as APIs aren't always as well protected as their web application counterparts. Pay attention to the APIs used by your targets, and you might find issues not present in the main application. If you are interested in learning more about hacking APIs and web applications in general, the OWASP Web Security Testing Guide (*https://github.com/OWASP/wstg/*) is a great resource to learn from.