Chapter 5. API Analysis

API endpoint analysis is the next logical skill in a recon toolkit after subdomain discovery. What domains does this application make use of? If this application has three domains (*x.domain*, *y.domain*, and *z.domain*, for example), I should be aware that each of them may have their own unique API endpoints.

Generally speaking, we can use very similar techniques to those we used when attempting to find subdomains. Brute force attacks and dictionary attacks work well here, but manual efforts and logical analysis are also often rewarded.

Finding APIs is the second step in learning about the structure of a web application following discovery of subdomains. This step will provide us with the information we need to begin understanding the purpose of an exposed API. When we understand why an API is exposed over the network, we can then begin to see how it fits into an application and what its business purpose is.

Endpoint Discovery

Previously we discussed how most enterprise applications today follow a particular scheme when defining the structure of their APIs. Typically, APIs will either follow a REST format or a SOAP format. REST is becoming much more popular and is considered to be the ideal structure for modern web application APIs today.

We can make use of the developer tools in our browser as we walk through an application and analyze the network requests. If we see a

number of HTTP requests that look like this, then it's pretty safe to assume that this is a REST API:

```
GET api.mega-bank.com/users/1234
GET api.mega-bank.com/users/1234/payments
POST api.mega-bank.com/users/1234/payments
```

Notice that each endpoint specifies a particular resource rather than a function.

Furthermore, we can assume that the nested resource `payments` belongs to user `1234`, which tells us this API is hierarchical. This is another telltale sign of RESTful design.

If we look at the cookies getting sent with each request, and look at the headers of each request, we may also find signs of RESTful architecture:

```
POST /users/1234/payments HTTP/1.1
Host: api.mega-bank.com
Authorization: Bearer abc21323
Content-Type: application/x-www-form-urlencoded
User-Agent: Mozilla/5.0 (X11; Linux x86_64) AppleWebKit/1.0 (KHTML, like Gecko)
```

A token being sent on every request is another sign of RESTful API design. REST APIs are supposed to be stateless, which means the server should not keep track of its requesters.

Once we know this is indeed a REST API, we can start to make logical hypotheses regarding available endpoints. Table 5-1 lists the HTTP verbs that REST architecture supports. Using Table 5-1, we can look at the requests we found in the browser console targeting particular resources and attempt to make requests to those resources using different HTTP verbs to see if the API returns anything interesting.

Table 5-1. HTTP verbs that REST architecture supports

REST HTTP Verb	Usage
POST	Create
GET	Read
PUT	Update/replace
PATCH	Update/modify
DELETE	Delete

The HTTP specification defines a special method that only exists to give information about a particular API's verbs. This method is called `OPTIONS`, which should be our go-to when performing recon against an API. We can easily make a request in curl from the terminal:

```
curl -i -X OPTIONS https://api.mega-bank.com/users/1234
```

If the `OPTIONS` request was successful, we should see the following response:

```
200 OK
Allow: HEAD, GET, PUT, DELETE, OPTIONS
```

Generally speaking, `OPTIONS` will only be available on APIs specifically designated for public use. So while it's an easy first attempt, we will need a more robust discovery solution for most apps we attempt to test. Very few enterprise applications expose `OPTIONS`.

Let's move on to a more likely method of determining accepted HTTP verbs. The first API call we saw in our browser was the following:

```
GET api.mega-bank.com/users/1234
```

We can now expand this to:

```
GET api.mega-bank.com/users/1234
POST api.mega-bank.com/users/1234
PUT api.mega-bank.com/users/1234
PATCH api.mega-bank.com/users/1234
DELETE api.mega-bank.com/users/1234
```

With the preceding list of HTTP verbs in mind, we can generate a script to test the legitimacy of our theory.

WARNING

Brute forcing API endpoint HTTP verbs has the possible side effect of deleting or altering application data. Make sure you have explicit permission from the application owner prior to performing any type of brute force attempt against an application API.

Our script has a simple purpose: using a given endpoint (we know this endpoint already accepts at least one HTTP verb), try each additional HTTP verb. After each additional HTTP verb is tried against the endpoint, record and print the results:

```
/*
 * Given a URL (corresponding to an API endpoint),
 * attempt requests with various HTTP verbs to determine
 * which HTTP verbs map to the given endpoint.
 */
const discoverHTTPVerbs = function(url) {
 const verbs = ['POST', 'GET', 'PUT', 'PATCH', 'DELETE'];
 const promises = [];

 verbs.forEach((verb) => {
  const promise = new Promise((resolve, reject) => {
   const http = new XMLHttpRequest();

   http.open(verb, url, true)
   http.setRequestHeader('Content-type', 'application/x-www-form-urlencoded');

   /*
    * If the request is successful, resolve the promise and
```

```
     * include the status code in the result.
     */
    http.onreadystatechange = function() {
     if (http.readyState === 4) {
      return resolve({ verb: verb, status: http.status });
     }
    }

    /*
     * If the request is not successful, or does not complete in time, mark
     * the request as unsuccessful. The timeout should be tweaked based on
     * average response time.
     */
    setTimeout(() => {
     return resolve({ verb: verb, status: -1 });
    }, 1000);

    // initiate the HTTP request
    http.send({});
   });

   // add the promise object to the promises array
   promises.push(promise);
  });

  /*
   * When all verbs have been attempted, log the results of their
   * respective promises to the console.
   */
  Promise.all(promises).then(function(values) {
   console.log(values);
  });
 }
```

The way this script functions on a technical level is just as simple. HTTP endpoints return a status code alongside any message they send back to the browser. We don't actually care what this status code is. We just want to see a status code.

We make a number of HTTP requests against the API, one for each HTTP verb. Most servers do not respond to requests that do not map to a valid

endpoint, so we have an additional case where we return –1 if a request does not receive a response within 1 second. Generally speaking, 1 second (or 1,000 ms in this case) is plenty of time for an API to respond. You can tweak this up or down depending on your own use case.

After the promises have all resolved, you can look at the log output to determine which HTTP verbs have an associated endpoint.

Authentication Mechanisms

Guessing the payload shape required for an API endpoint is much more difficult than just asserting that an API endpoint exists The easiest way is to analyze the structure of known requests being sent via the browser. Beyond that we must make educated guesses about the shape required for the API endpoint and test them manually. It's possible to automate the discovery of the structure of an API endpoint, but any attempts at doing so that don't involve analyzing existing requests would be very easy to detect and log.

It's usually best to start with common endpoints that can be found on nearly every application: sign in, sign up, password reset, etc. These often take a similarly shaped payload to that of other apps since authentication is usually designed based on a standardized scheme.

Every application with a public web user interface should have a login page. The way they authenticate your session, however, may differ. It's important to know what type of authentication scheme you are working with because many modern applications send authentication tokens with every request. This means if we can reverse engineer the type of authentication used and understand how the token is being attached to requests, it will be easier to analyze other API endpoints that rely on an authenticated user token. There are several major authentication schemes in use today, the most common of which are shown in Table 5-2.

Table 5-2. Major authentication schemes

Authentication scheme	Implementation details	Strengths	Weaknesses
HTTP Basic Auth	Username and password sent on each request	All major browsers support this natively	Session does not expire; easy to intercept
HTTP Digest Authentication	Hashed `username:realm:password` sent on each request	More difficult to intercept; server can reject expired tokens	Encryption strength dependent on hashing algorithm used
OAuth	"Bearer" token-based auth; allows sign in with other websites such as Amazon → Twitch	Tokenized permissions can be shared from one app to another for integrations	Phishing risk; central site can be compromised, compromising all connected apps

If we log in to *https://www.mega-bank.com* and analyze the network response, we might see something like this after the login succeeds:

```
GET /homepage
HOST mega-bank.com
Authorization: Basic am9lOjEyMzQ=
Content Type: application/json
```

We can tell at first glance that this is HTTP basic authentication because of the `Basic` authorization header being sent. Furthermore, the string `am9lOjEyMzQ=` is simply a base64-encoded `username:password` string. This is the most common way to format a username and password combination for delivery over HTTP.

In the browser console, we can use the built-in functions `btoa(str)` and `atob(base64)` to convert strings to base64 and vice versa. If we run the

base64-encoded string through the `atob` function, we will see the username and password being sent over the network:

```
/*
 * Decodes a string that was previously encoded with base64.
 * Result = joe:1234
 */
atob('am9lOjEyMzQ=');
```

Because of how insecure this mechanism is, basic authentication is typically only used on web applications that enforce SSL/TLS traffic encryption. This way, credentials cannot be intercepted midair—for example, at a sketchy mall WiFi hotspot.

The important thing to note from the analysis of this login/redirect to the home page is that our requests are indeed being authenticated, and they are doing so with `Authorization: Basic am9lOjEyMzQ=`. This means that if we ever run into another endpoint that is not returning anything interesting with an empty payload, the first thing we should try is attaching an authorization header and seeing if it does anything different when we request as an authenticated user.

Endpoint Shapes

After locating a number of subdomains and the HTTP APIs contained within those subdomains, begin determining the HTTP verbs used per resource and adding the results of that investigation to your web application map. Once you have a comprehensive list of subdomains, APIs, and shapes, you may begin to wonder how you can actually learn what type of payload any given API expects.

Common Shapes

Sometimes this process is simple—many APIs expect payload shapes that are common in the industry. For example, an authorization endpoint that is set up as part of an OAuth 2.0 flow may expect the following data:

```
{
  "response_type": code,
  "client_id": id,
  "scope": [scopes],
  "state": state,
  "redirect_uri": uri
}
```

Because OAuth 2.0 is a widely implemented public specification, determining the data to include in an OAuth 2.0 authorization endpoint can often be done through a combination of educated guesses combined with the available public documentation. The naming conventions and list of scopes in an OAuth 2.0 authorization endpoint may differ slightly from implementation to implementation, but the overall payload shape should not.

An example of an OAuth 2.0 authorization endpoint can be found in the Discord (instant messaging) public documentation. Discord suggests that a call to the OAuth 2.0 endpoint should be structured as follows:

```
https://discordapp.com/api/oauth2/authorize?response_type=code&client_\
id=157730590492196864&scope=identify%20guilds.\
join&state=15773059ghq9183habn&redirect_uri=https%3A%2F%2Fnicememe.\
website&prompt=consent
```

Here `response_type`, `client_id`, `scope`, `state`, and `redirect_uri` are all part of the official spec.

Facebook's public documentation for OAuth 2.0 is very similar, suggesting the following request for the same functionality:

```
GET https://graph.facebook.com/v4.0/oauth/access_token?
   client_id={app-id}
   &redirect_uri={redirect-uri}
   &client_secret={app-secret}
   &code={code-parameter}
```

So finding the shape of an HTTP API isn't complex when dealing with common endpoint archetypes. However, it is wise to consider that while many APIs implement common specifications like OAuth, they will often not use a common specification for their internal APIs that are responsible for initiating application logic.

Application-Specific Shapes

Application-specific shapes are much harder to determine than those that are based on public specifications. To determine the shape of a payload expected by an API endpoint, you may need to rely on a number of recon techniques and slowly learn about the endpoint by trial and error.

Insecure applications may give you hints in the form of HTTP error messages. For example, imagine you call `POST https://www.mega-bank.com/users/config` with the following body:

```
{
 "user_id": 12345,
 "privacy": {
   "publicProfile": true
 }
}
```

You would likely get an HTTP status code like `401 not authorized` or a `400 internal error`. If the status code comes with a message like `auth_token not supplied`, you may have accidentally stumbled across a missing param.

In an alternative request with a correct `auth_token`, you might get another error message: `publicProfile only accepts "auth" and "noAuth" as params.`

Bingo.

But more secure applications will probably just throw a generic error, and you will have to move on to other techniques.

If you have a privileged account, you can try the same request against your account using the UI before attempting it against another account to determine what the outgoing shape looks like. This can be found in the browser Developer tools → Network tab or with a network monitoring tool like Burp.

Finally, if you know the name of a variable expected in the payload, but not a value, then you may be able to brute force the request by repeating it with variations until one sticks. Obviously, brute forcing values is slow manually, so you want a script to speed up the process. The more rules you can learn about an expected variable, the better. If you know an `auth_token` is always 12 characters, that's great. If you know it is always hexadecimal, that's even better. The more rules you can learn and apply, the more likely you will be able to brute force a successful combination.

The list of possible combinations for a field is known as the *solutions space*. You want to decrease the solutions space to the smallest viable search space.

Rather than searching for valid solutions, you may also want to try searching for invalid solutions. These may help you reduce the solutions space and potentially even uncover bugs in the application code.

Summary

After developing a mental model (ideally also recorded in some form) of the subdomains that power an application, the next step is to find the API endpoints hosted on those subdomains so that you can try to determine their purpose later. Although it sounds like a simple step, it is crucial as a recon technique because without it you may spend time trying to find holes in well-secured endpoints while less-secure endpoints exist with similar functionality or data. Additionally, finding endpoints on an API is one step toward understanding the purpose and function of the API if you are not already aware of its intended use.

Once you have found and documented a number of API endpoints, then determining the shape of the payloads that endpoint takes is the next logi-

cal step. Using a combination of educated guesses, automation, and analysis of common endpoint archetypes like we did in this chapter will eventually lead you to discover the data that these endpoints expect and the data that is sent in response. With this knowledge in mind, you now understand the function of the application, which is the first major step toward breaking or securing the application.