7

ENDPOINT ANALYSIS

Now that you've discovered a few APIs, it's time to begin using and testing the endpoints you've found. This chapter will cover interacting with endpoints, testing them for vulnerabilities, and maybe even scoring some early wins.

By "early wins," I mean critical vulnerabilities or data leaks sometimes present during this stage of testing. APIs are a special sort of target because you may not need advanced skills to bypass firewalls and endpoint security; instead, you may just need to know how to use an endpoint as it was designed.

We'll begin by learning how to discover the format of an API's numerous requests from its documentation, its specification, and reverse engineering, and we'll use these sources to build Postman collections so we can perform analysis across each request. Then we'll walk through a simple process you can use to begin your API testing and discuss how you might find your first vulnerabilities, such as information disclosures, security misconfigurations, excessive data exposures, and business logic flaws.

Finding Request Information

If you're used to attacking web applications, your hunt for API vulnerabilities should be somewhat familiar. The primary difference is that you no longer have obvious GUI cues such as search bars, login fields, and buttons for uploading files. API hacking relies on the backend operations of those items that are found in the GUI—namely, GET requests with query parameters and most POST/PUT/UPDATE/DELETE requests.

Before you craft requests to an API, you'll need an understanding of its endpoints, request parameters, necessary headers, authentication requirements, and administrative functionality. Documentation will often point us to those elements. Therefore, to succeed as an API hacker, you'll need to know how to read and use API documentation, as well as how to find it. Even better, if you can find a specification for an API, you can import it directly into Postman to automatically craft requests.

When you're performing a black box API test and the documentation is truly unavailable, you'll be left to reverse engineer the API on your own. You will need to thoroughly fuzz your way through the API to discover endpoints, parameters, and header requirements in order to map out the API and its functionality.

Finding Information in Documentation

As you know by now, an API's documentation is a set of instructions published by the API provider for the API consumer. Because public and partner APIs are designed with self-service in mind, a public user or a partner should be able to find the documentation, understand how to use the API, and do so without assistance from the provider. It is quite common for the documentation to be located under directories like the following:

https://example.com/docs

https://example.com/api/docs

https://docs.example.com

https://dev.example.com/docs

https://developer.example.com/docs

https://api.example.com/docs

https://example.com/developers/documentation

When the documentation is not publicly available, try creating an account and searching for the documentation while authenticated. If you still cannot find the docs, I have provided a couple API wordlists on GitHub that can help you discover API documentation through the use of a fuzzing technique called *directory brute force* (*https://github.com/hAPI-hacker/Hacking-APIs*). You can use the `subdomains_list` and the `dir_list` to brute-force web application subdomains and domains and potentially find API docs hosted on the site. There is a good chance you'll be able to discover documentation during reconnaissance and web application scanning.

If an organization's documentation really is locked down, you still have a few options. First, try using your Google hacking skills to find it on search engines and in other recon tools. Second, use the Wayback Machine (*https://web.archive.org/*). If your target once posted their API documentation publicly and later retracted it, there may be an archive of their docs available. Archived documentation will likely be outdated, but it should give you an idea of the authentication requirements, naming schemes, and endpoint locations. Third, try social engineering techniques to trick an organization into sharing its documentation. These techniques are beyond the scope of this book, but you can get creative with smishing, vishing, and phishing developers, sales departments, and organization partners for access to the API documentation. Act like a new customer trying to work with the target API.

NOTE *API documentation is only a starting point. Never trust that the docs are accurate and up to date or that they include everything there is to know about the endpoints. Always test for methods, endpoints, and parameters that are not included in documentation. Distrust and verify.*

Although API documentation is straightforward, there are a few elements to look out for. The *overview* is typically the first section of API documentation. Normally found at the beginning of the doc, the overview will provide a high-level introduction of how to connect and use the API. In addition, it could contain information about authentication and rate limiting.

Review the documentation for *functionality*, or the actions that you can take using the given API. These will be represented by a combination of an HTTP method (GET, PUT, POST, DELETE) and an endpoint. Every organization's APIs will be different, but you can expect to find functionality related to user account management, options to upload and download data, different ways to request information, and so on.

When making a request to an endpoint, make sure you note the request *requirements*. Requirements could include some form of authentication, parameters, path variables, headers, and information included in the body of the request. The API documentation should tell you what it requires of you and mention in which part of the request that information belongs. If the documentation provides examples, use them to help you. Typically, you can replace the sample values with the ones you're looking for. Table 7-1 describes some of the conventions often used in these examples.

Table 7-1: API Documentation Conventions

Convention	Example	Meaning
: or {}	*/user/:id* */user/{id}* */user/2727* */account/:username* */account/{username}* */account/scuttleph1sh*	The colon or curly brackets are used by some APIs to indicate a path variable. In other words, ":id" represents the variable for an ID number and "{username}" represents the account username you are trying to access.

(continued)

Table 7-1: API Documentation Conventions *(continued)*

Convention	Example	Meaning
[]	*/api/v1/user?find=[name]*	Square brackets indicate that the input is optional.
||	"blue" || "green" || "red"	Double bars represent different possible values that can be used.
< >	<find-function>	Angle brackets represent a DomString, which is a 16-bit string.

For example, the following is a GET request from the vulnerable Pixi API documentation:

```
❶ GET          ❷/api/picture/{picture_id}/likes    get a list of likes by user

❸ Parameters

Name                                Description

x-access-token *
string                              Users JWT Token
(header)

picture_id *                        in URL string

number
(path)
```

You can see that the method is GET ❶, the endpoint is */api/picture/{picture_id}/likes* ❷, and the only requirements are the `x-access-token` header and the `picture_id` variable to be updated in the path ❸. Now you know that, in order to test this endpoint, you'll need to figure out how to obtain a JSON Web Token (JWT) and what form the `picture_id` should be in.

You can then take these instructions and insert the information into an API browser such as Postman (see Figure 7-1). As you'll see, all of the headers besides `x-access-token` will be automatically generated by Postman.

Here, I authenticated to the web page and found the `picture_id` listed under the pictures. I used the documentation to find the API registration process, which generated a JWT. I then took the JWT and saved it as the variable `hapi_token`; we will be using variables throughout this chapter. Once the token is saved as a variable, you can call it by using the variable name surrounded by curly brackets: `{{hapi_token}}`. (Note that if you are working with several collections, you'll want to use environmental variables instead.) Put together, it forms a successful API request. You can see that the provider responded with a "200 OK," along with the requested information.

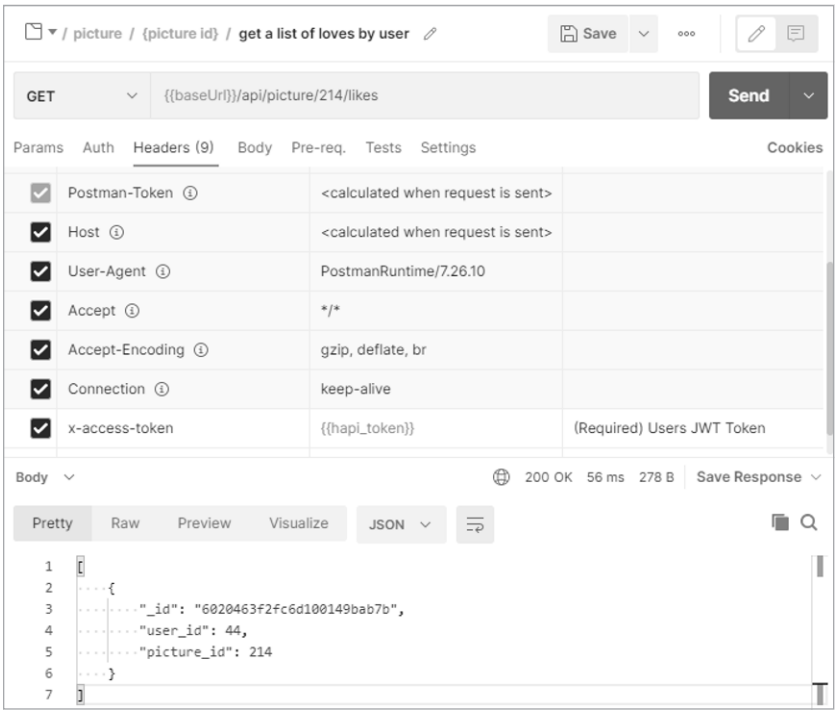

Figure 7-1: The fully crafted request to the Pixi endpoint `/api/{picture_id}/likes`

In situations where your request is improperly formed, the provider will usually let you know what you've done wrong. For instance, if you make a request to the same endpoint without the `x-access-token`, Pixi will respond with the following:

```
{
    "success": false,
    "message": "No token provided."
}
```

You should be able to understand the response and make any necessary adjustments. If you had attempted to copy and paste the endpoint without replacing the `{picture_id}` variable, the provider would respond with a status code of 200 OK and a body with square brackets (`[]`). If you are stumped by a response, return to the documentation and compare your request with the requirements.

Importing API Specifications

If your target has a specification, in a format like OpenAPI (Swagger), RAML, or API Blueprint, or in a Postman collection, finding these will be even more useful than finding the documentation. When provided with a

specification, you can simply import it into Postman and review the requests that make up the collection, as well as their endpoints, headers, parameters, and some required variables.

Specifications should be as easy or as hard to find as their API documentation counterparts. They'll often look like the page in Figure 7-2. The specification will contain plaintext and typically be in JSON format, but could also be in YAML, RAML, or XML format. If the URL path doesn't give away the type of specification, scan the beginning of the file for a descriptor, such as `"swagger":"2.0"`, to find the specification and version.

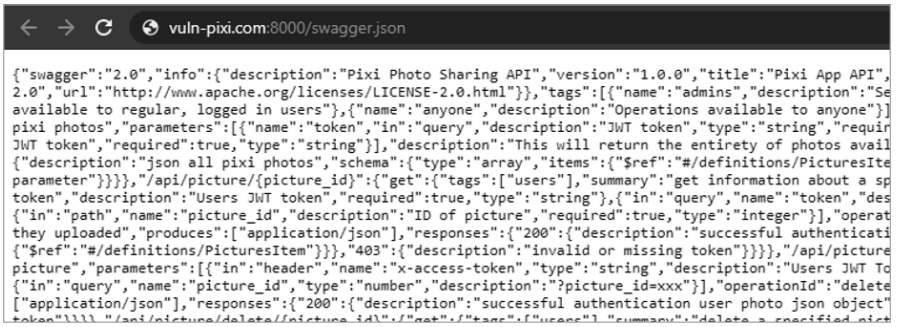

Figure 7-2: The Pixi swagger definition page

To import the specification, begin by launching Postman. Under the Workspace Collection section, click **Import**, select **Link**, and then add the location of the specification (see Figure 7-3).

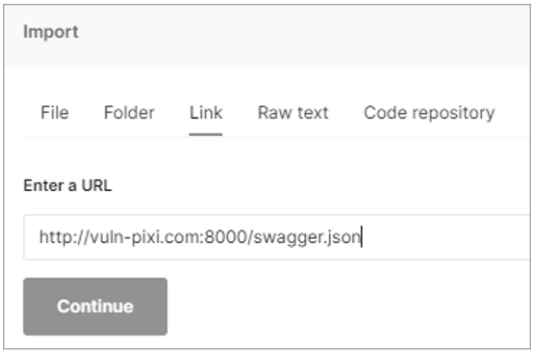

Figure 7-3: The Import Link functionality within Postman

Click **Continue**, and on the final window, select **Import**. Postman will detect the specification and import the file as a collection. Once the collection has been imported into Postman, you can review the functionality here (see Figure 7-4).

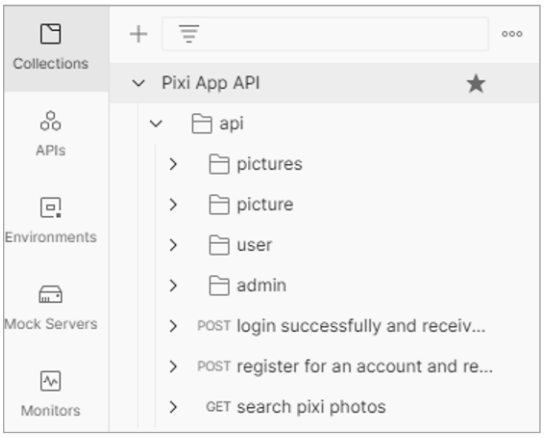

Figure 7-4: The imported Pixi App collection

After you've imported a new collection, make sure to check the collection variables. You can display the collection editor by selecting the three horizontal circles at the top level of a collection and choosing **Edit**. Here, you can select the Variables tab within the collection editor to see the variables. You can adjust the variables to fit your needs and add any new variables you would like to this collection. In Figure 7-5, you can see where I have added the `hapi_token` JWT variable to my Pixi App collection.

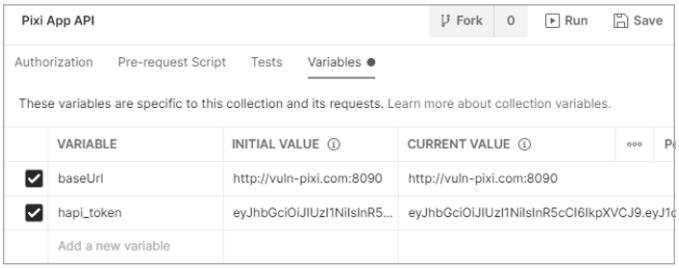

Figure 7-5: The Postman collection variables editor

Once you've finished making updates, save your changes using the **Save** button at the top-right corner. Importing API specifications to Postman like this could save you hours of manually adding all endpoints, request methods, headers, and requirements.

Reverse Engineering APIs

In the instance where there is no documentation and no specification, you will have to reverse engineer the API based on your interactions with it. We will touch on this process in more detail in Chapter 9. Mapping an API with several endpoints and a few methods can quickly grow into quite a beast to attack. To manage this process, build the requests under a collection in order to thoroughly hack the API. Postman can help you keep track of all these requests.

There are two ways to reverse engineer an API with Postman. One way is by manually constructing each request. While this can be a bit cumbersome, it allows you to capture the precise requests you care about. The other way is to proxy web traffic through Postman and then use it to capture a stream of requests. This process makes it much easier to construct requests within Postman, but you'll have to remove or ignore unrelated requests. Finally, if you obtain a valid authentication header, such as a token, API key, or other authentication value, add that to Kiterunner to help map out API endpoints.

Manually Building a Postman Collection

To manually build your own collection in Postman, select **New** under My Workspace, as seen at the top right of Figure 7-6.

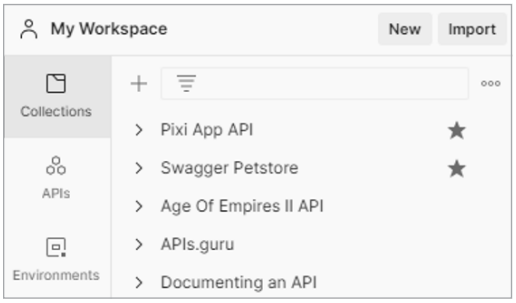

Figure 7-6: The workspace section of Postman

In the Create New window, create a new collection and then set up a `baseURL` variable containing your target's URL. Creating a `baseURL` variable (or using one that is already present) will help you quickly make alterations to the URL across an entire collection. APIs can be quite large, and making small changes to many requests can be time-consuming. For example, suppose you want to test out different API path versions (such as *v1/v2/v3*) across an API with hundreds of unique requests. Replacing the URL with a variable means you would only need to update the variable in order to change the path for all requests using the variable.

Now, any time you discover an API request, you can add it to the collection (see Figure 7-7).

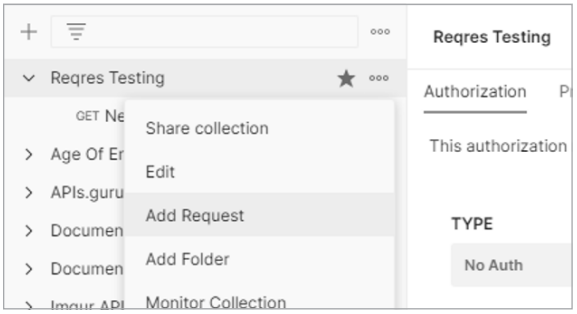

Figure 7-7: The Add Request option within a new Postman collection

Select the collection options button (the three horizontal circles) and select **Add Request**. If you want to further organize the requests, you can create folders to group the requests together. Once you have built a collection, you can use it as though it were documentation.

Building a Postman Collection by Proxy

The second way to reverse engineer an API is to proxy web browser traffic through Postman and clean up the requests so that only the API-related ones remain. Let's reverse engineer the crAPI API by proxying our browser traffic to Postman.

First, open Postman and create a collection for crAPI. At the top right of Postman is a signal button that you can select to open the Capture requests and cookies window (see Figure 7-8).

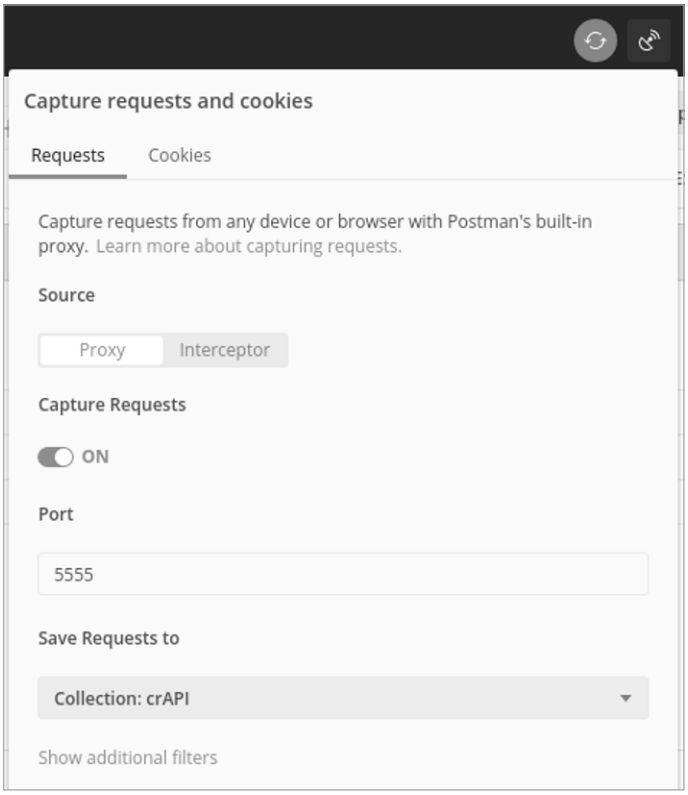

Figure 7-8: The Postman Capture requests and cookies window

Make sure the port number matches the one you've configured in FoxyProxy. Back in Chapter 4, we set this to port 5555. Save requests to your crAPI collection. Finally, set Capture Requests to On. Now navigate to the crAPI web application and set FoxyProxy to forward traffic to Postman.

As you start using the web application, every request will be sent through Postman and added to the selected collection. Use every feature

of the web application, including registering a new account, authenticating, performing a password reset, clicking every link, updating your profile, using the community forum, and navigating to the shop. Once you've finished thoroughly using the web application, stop your proxy and review the crAPI collection made within Postman.

One downside of building a collection this way is that you'll have captured several requests that aren't API related. You will need to delete these requests and organize the collection. Postman allows you to create folders to group similar requests, and you can rename as many requests as you'd like. In Figure 7-9, you can see that I grouped requests by the different endpoints.

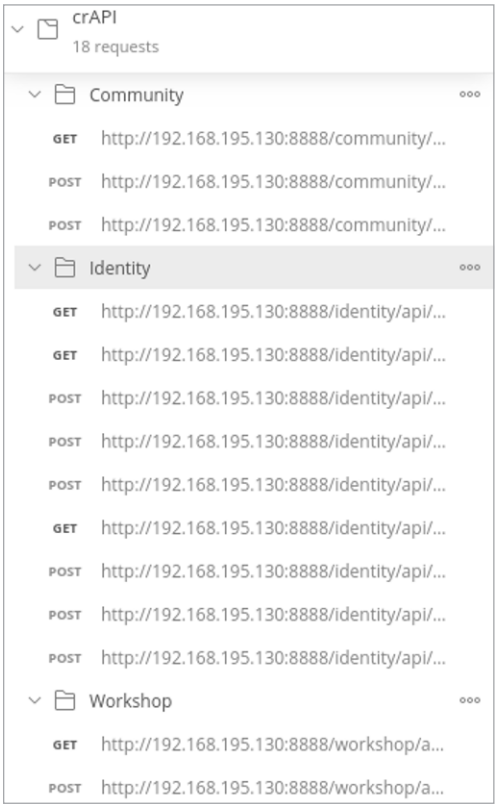

Figure 7-9: An organized crAPI collection

Adding API Authentication Requirements to Postman

Once you've compiled the basic request information in Postman, look for the API's authentication requirements. Most APIs with authentication requirements will have a process for obtaining access, typically by sending credentials over a POST request or OAuth, or else by using a method

separate from the API, such as email, to obtain a token. Decent documentation should make the authentication process clear. In the next chapter, we will dedicate time to testing the API authentication processes. For now, we will use the API authentication requirements to start using the API as it was intended.

As an example of a somewhat typical authentication process, let's register and authenticate to the Pixi API. Pixi's Swagger documentation tells us that we need to make a request with both `user` and `pass` parameters to the */api/register* endpoint to receive a JWT. If you've imported the collection, you should be able to find and select the "Create Authentication Token" request in Postman (see Figure 7-10).

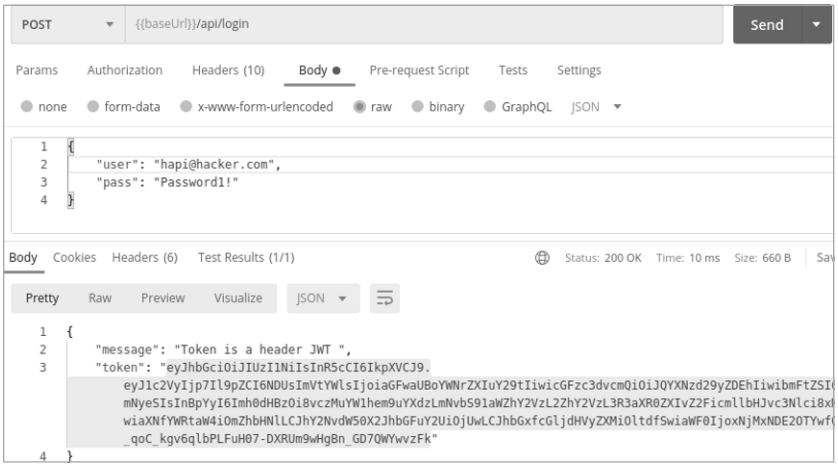

Figure 7-10: A successful registration request to the Pixi API

The preconfigured request contains parameters you may not be aware of and are not required for authentication. Instead of using the preconfigured information, I crafted the response by selecting the `x-www-form-urlencoded` option with the only parameters necessary (`user` and `pass`). I then added the keys `user` and `pass` and filled in the values shown in Figure 7-10. This process resulted in successful registration, as indicated by the 200 OK status code and the response of a token.

It's a good idea to save successful authentication requests so you can repeat them when needed, as tokens could be set to expire quickly. Additionally, API security controls could detect malicious activity and revoke your token. As long as your account isn't blocked, you should be able to generate another token and continue your testing. Also, be sure to save your token as a collection or environmental variable. That way, you'll be able to quickly reference it in subsequent requests instead of having to continuously copy in the giant string.

The next thing you should do when you get an authentication token or API key is to add it to Kiterunner. We used Kiterunner in Chapter 6 to map out a target's attack surface as an unauthenticated user, but adding an authentication header to the tool will greatly improve your results. Not only

will Kiterunner provide you with a list of valid endpoints, but it will also hand you interesting HTTP methods and parameters.

In the following example, we use the `x-access-token` provided to us during the Pixi registration process. Take the full authorization header and add it to your Kiterunner scan with the `-H` option:

```
$ kr scan http://192.168.50.35:8090 -w ~/api/wordlists/data/kiterunner/routes-large.kite -H
'x-access-token: eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJ1c2VyIjp7Il9pZCI6NDUsImVtYWlsIjoiaGF
waUBoYWNrZXIuY29tIiwicGFzc3dvcmQiOiJQYXNzd29yZDEhIiwibmFtZSI6Im15c2VsZmNyeSIsInBpYyI6ImhOdHBzO
i8vczMuYW1hem9uYXdzLmNvbS91aWZhY2VzL2ZhY2VzL3R3aXROZXIvZ2FicmllbHJvc3Nlci8xMjguanBnIiwiaXNfYWRt
aW4iOmZhbHNlLCJhY2NvdW50X2JhbGFuY2UiOjUwLCJhbGxfcGljdHVyZXMiOltdfSwiaWF0IjoxNjMxNDE2OTYwfQ._qoC
_kgv6qlbPLFuHO7-DXRUm9wHgBn_GD7QWYwvzFk'
This scan will result in identifying the following endpoints:
GET     200 [    217,    1,   1] http://192.168.50.35:8090/api/user/info
GET     200 [ 101471, 1871,   1] http://192.168.50.35:8090/api/pictures/
GET     200 [    217,    1,   1] http://192.168.50.35:8090/api/user/info/
GET     200 [ 101471, 1871,   1] http://192.168.50.35:8090/api/pictures
```

Adding authorization headers to your Kiterunner requests should improve your scan results, as it will allow the scanner to access endpoints it otherwise wouldn't have access to.

Analyzing Functionality

Once you have the API's information loaded into Postman, you should begin to look for issues. This section covers a method for initially testing the functionality of API endpoints. You'll begin by using the API as it was intended. In the process, you'll pay attention to the responses and their status codes and error messages. In particular, you'll seek out functionality that interests you as an attacker, especially if there are indications of information disclosure, excessive data exposure, and other low-hanging vulnerabilities. Look for endpoints that could provide you with sensitive information, requests that allow you to interact with resources, areas of the API that allow you to inject a payload, and administrative actions. Beyond that, look for any endpoint that allows you to upload your own payload and interact with resources.

To streamline this process, I recommend proxying Kiterunner's results through Burp Suite so you can replay interesting requests. In past chapters, I showed you the replay feature of Kiterunner, which lets you review individual API requests and responses. To proxy a replay through another tool, you will need to specify the address of the proxy receiver:

```
$ kr kb replay -w ~/api/wordlists/data/kiterunner/routes-large.kite
--proxy=http://127.0.0.1:8080 "GET     403 [     48,    3,   1] http://192.168.50.35:8090/api/
picture/detail.php 0cf6889d2fba4be08930547f145649ffead29edb"
```

This request uses Kiterunner's replay option, as specified by `kb replay`. The `-w` option specifies the wordlist used, and `proxy` specifies the Burp Suite proxy. The remainder of the command is the original Kiterunner output.

In Figure 7-11, you can see that the Kiterunner replay was successfully captured in Burp Suite.

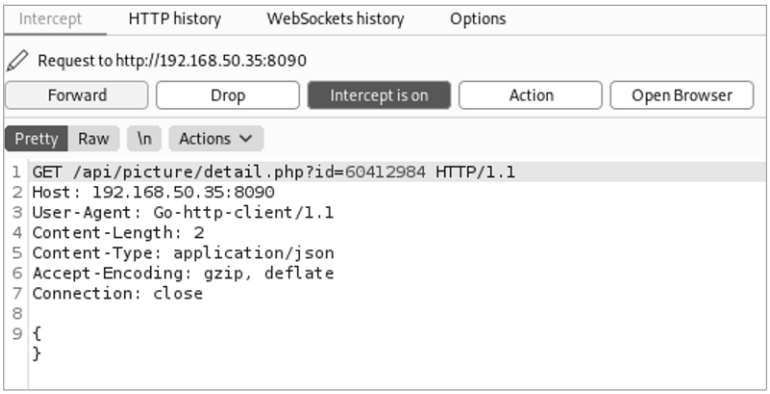

Figure 7-11: A Kiterunner request intercepted with Burp Suite

Now you can analyze the requests and use Burp Suite to repeat all interesting results captured in Kiterunner.

Testing Intended Use

Start by using the API endpoints as intended. You could begin this process with a web browser, but web browsers were not meant to interact with APIs, so you might want to switch to Postman. Use the API documentation to see how you should structure your requests, what headers to include, what parameters to add, and what to supply for authentication, and then send the requests. Adjust your requests until you receive successful responses from the provider.

As you proceed, ask yourself these questions:

- What sorts of actions can I take?
- Can I interact with other user accounts?
- What kinds of resources are available?
- When I create a new resource, how is that resource identified?
- Can I upload a file? Can I edit a file?

There is no need to make every possible request if you are manually working with the API, but make a few. Of course, if you have built a collection in Postman, you can easily make every possible request and see what response you get from the provider.

For example, send a request to Pixi's */api/user/info* endpoint to see what sort of response you receive from the application (see Figure 7-12).

In order to make a request to this endpoint, you must use the GET method. Add the *{{baseUrl}}/api/user/info* endpoint to the URL field. Then add the `x-access-token` to the request header. As you can see, I have set the JWT as the variable {{`hapi_token`}}. If you are successful, you should receive a 200 OK status code, seen just above the response.

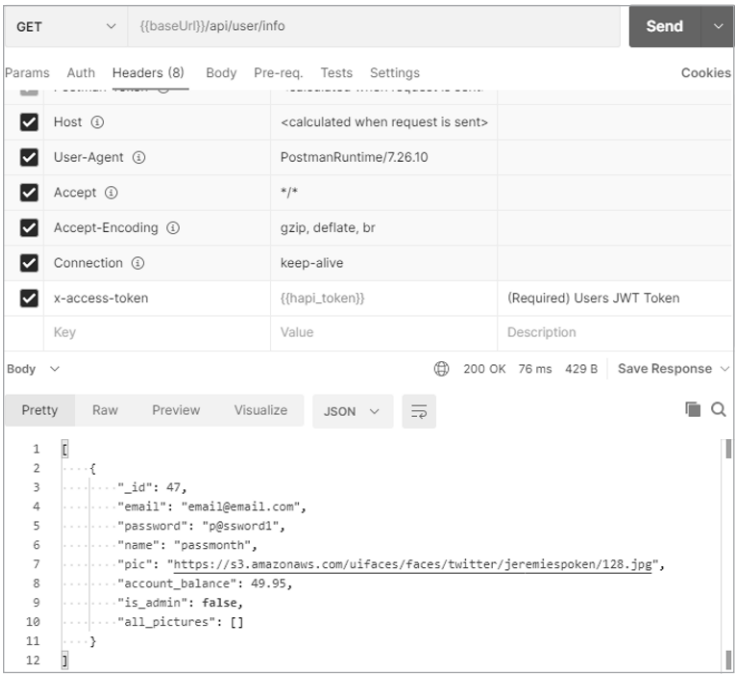

Figure 7-12: Setting the x-access-token as the variable for the JWT

Performing Privileged Actions

If you've gained access to an API's documentation, any sort of administrative actions listed there should grab your attention. Privileged actions will often lead to additional functionality, information, and control. For example, admin requests could give you the ability to create and delete users, search for sensitive user information, enable and disable accounts, add users to groups, manage tokens, access logs, and more. Luckily for us, admin API documentation information is often available for all to see due to the self-service nature of APIs.

If security controls are in place, administrative actions should have authorization requirements, but never assume that they actually do. My recommendation is to test these actions in several phases: first as an unauthenticated user, then as a low-privileged user, and finally as an administrative user. When you make the administrative requests as documented but without any authorization requirements, you should receive some sort of unauthorized response if any security controls are in place.

You'll likely have to find a way to gain access to the administrative requirements. In the case of the Pixi, the documentation in Figure 7-13 clearly shows us that we need an `x-access-token` to perform the GET request to the */api/admin/users/search* endpoint. When you test this administrative endpoint, you'll see that Pixi has basic security controls in place to prevent unauthorized users from using administrative endpoints.

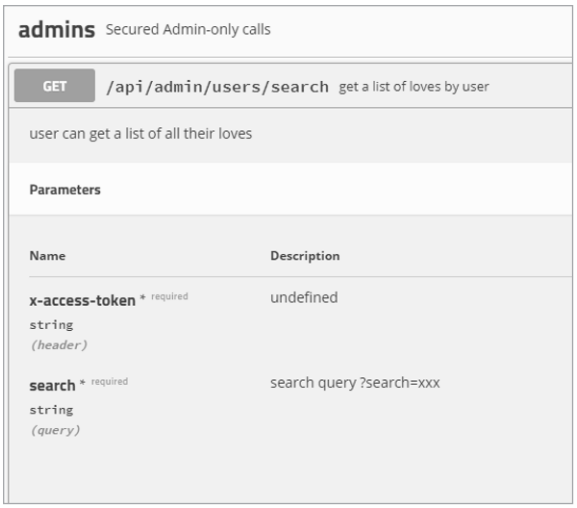

Figure 7-13: The requirements for a Pixi administrative endpoint

Making sure that the most basic security controls are in place is a useful practice. More importantly, protected administrative endpoints establish a goal for us for the next steps in our testing; we now know that in order to use this functionality, we need to obtain an admin JWT.

Analyzing API Responses

As most APIs are meant to be self-service, developers will often leave some hint in the API responses when things don't go as planned. One of the most basic skills you'll need as an API hacker is the ability to analyze the responses you receive. This is initially done by issuing a request and reviewing the response status code, headers, and content included in the body.

First check that you are receiving the responses you expect. API documentation can sometimes provide examples of what you could receive as a response. However, once you begin using the API in unintended ways, you will no longer know what you'll get as a response, which is why it helps to first use the API as it was intended before moving into attack mode. Developing a sense of regular and irregular behavior will make vulnerabilities obvious.

At this point, your search for vulnerabilities begins. Now that you're interacting with the API, you should be able to find information disclosures, security misconfigurations, excessive data exposures, and business logic flaws, all without too much technical finesse. It's time to introduce the most important ingredient of hacking: the adversarial mindset. In the following sections, I will show you what to look for.

Finding Information Disclosures

Information disclosure will often be the fuel for our testing. Anything that helps our exploitation of an API can be considered an information disclosure, whether it's interesting status codes, headers, or user data. When

making requests, you should review responses for software information, usernames, email addresses, phone numbers, password requirements, account numbers, partner company names, and any information that your target claims is useful.

Headers can inadvertently reveal more information about the application than necessary. Some, like `X-powered-by`, do not serve much of a purpose and often disclose information about the backend. Of course, this alone won't lead to exploitation, but it can help us know what sort of payload to craft and reveal potential application weaknesses.

Status codes can also disclose useful information. If you were to brute-force the paths of different endpoints and receive responses with the status codes 404 Not Found and 401 Unauthorized, you could map out the API's endpoints as an unauthorized user. This simple information disclosure can get much worse if these status codes were returned for requests with different query parameters. Say you were able to use a query parameter for a customer's phone number, account number, and email address. Then you could brute-force these items, treating the 404s as nonexistent values and the 401s as existing ones. Now, it probably shouldn't take too much imagination to see how this sort of information could assist you. You could perform password spraying, test password resent mechanisms, or conduct phishing, vishing, and smishing. There is also a chance you could pair query parameters together and extract personally identifiable information from the unique status codes.

API documentation can itself be an information disclosure risk. For instance, it is often an excellent source of information about business logic vulnerabilities, as discussed in Chapter 3. Moreover, administrative API documentation will often tell you the admin endpoints, the parameters required, and the method to obtain the specified parameters. This information can be used to aid you in Improper Assets Management attacks and authorization attacks (such as BOLA and BFLA), which are covered in later chapters.

When you start exploiting API vulnerabilities, be sure to track which headers, unique status codes, documentation, or other hints were handed to you by the API provider.

Finding Security Misconfigurations

Security misconfigurations represent a large variety of items. At this stage of your testing, look for verbose error messaging, poor transit encryption, and other problematic configurations. Each of these issues can be useful later for exploiting the API.

Verbose Errors

Error messages exist to help the developers on both the provider and consumer sides understand what has gone wrong. For example, if the API requires you to POST a username and password in order to obtain an API

token, check how the provider responds to both existing and nonexistent usernames. A common way to respond to nonexistent usernames is with the error "User does not exist, please provide a valid username." When a user does exist but you've used the wrong password, you may get the error "Invalid password." This small difference in error response is an information disclosure that you can use to brute-force usernames, which can then be leveraged in later attacks.

Poor Transit Encryption

Finding an API in the wild without transit encryption is rare. I've only come across this in instances when the provider believes its API contains only nonsensitive public information. In situations like this, the challenge is to see whether you can discover any sensitive information by using the API. In all other situations, make sure to check that the API has valid transit encryption. If the API is transmitting any sensitive information, HTTPS should be in use.

In order to attack an API with transit insecurities, you would need to perform a *man-in-the-middle (MITM)* attack in which you somehow intercept the traffic between a provider and a consumer. Because HTTP sends unencrypted traffic, you'll be able to read the intercept requests and responses. Even if HTTPS is in use on the provider's end, check whether a consumer can initiate HTTP requests and share their tokens in the clear.

Use a tool like Wireshark to capture network traffic and spot plaintext API requests passing across the network you're connected to. In Figure 7-14, a consumer has made an HTTP request to the HTTPS-protected *reqres.in*. As you can see, the API token within the path is clear as day.

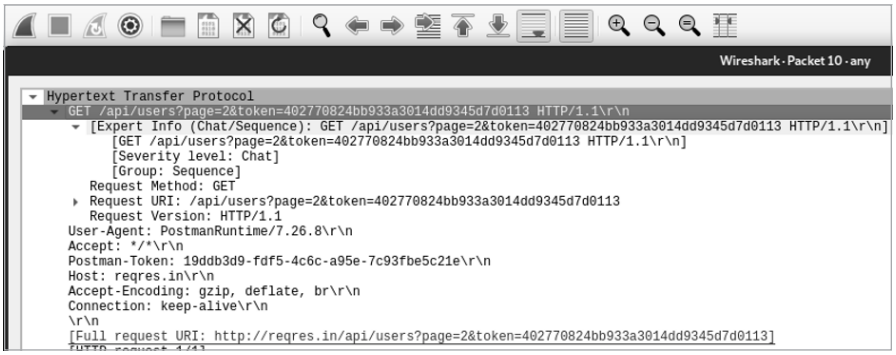

Figure 7-14: A Wireshark capture of a user's token in an HTTP request

Problematic Configurations

Debugging pages are a form of security misconfiguration that can expose plenty of useful information. I have come across many APIs that had debugging enabled. You have a better chance of finding this sort of misconfiguration in newly developed APIs and in testing environments. For example, in Figure 7-15, not only can you see the default landing page for 404 errors and

all of this provider's endpoints, but you can also see that the application is powered by Django.

Figure 7-15: The debug page of Tiredful API

This finding could trigger you to research what sorts of malicious things can be done when the Django debug mode is enabled.

Finding Excessive Data Exposures

As discussed in Chapter 3, excessive data exposure is a vulnerability that takes place when the API provider sends more information than the API consumer requests. This happens because the developers designed the API to depend on the consumer to filter results.

When testing for excessive data exposure on a large scale, it's best to use a tool like Postman's Collection Runner, which helps you make many requests quickly and provides you with an easy way to review the results. If the provider responds with more information than you needed, you could have found a vulnerability.

Of course, not every excess byte of data should be considered a vulnerability; watch for excess information that can be useful in an attack. True excessive data exposure vulnerabilities are often fairly obvious because of the sheer quantity of data provided. Imagine an endpoint with the ability to search for usernames. If you queried for a username and received the username plus a timestamp of the user's last login, this is excess data, but it's hardly useful. Now, if you queried for the username and were provided with a username plus the user's full name, email, and birthday, you have a

finding. For example, say a GET request to *https://secure.example.com/api/users/hapi_hacker* was supposed to give you information about our hapi_hacker account, but it responded with the following:

```
{
 "user": {
"id": 1124,
"admin": false,
"username": hapi_hacker,
"multifactor": false
}
"sales_assoc": {
       "email": "admin@example.com",
        "admin": true,
        "username": super_sales_admin,
        "multifactor": false
}
```

As you can see, a request was made for the hapi_hacker account, but the administrator's account and security settings were included in the response. Not only does the response provide you with an administrator's email address and username, but it also lets you know whether they are an administrator without multifactor authentication enabled. This vulnerability is fairly common and can be extremely useful for gaining access to admin accounts. Also, if there is an excessive data exposure vulnerability on one endpoint and method, you can bet there are others.

Finding Business Logic Flaws

OWASP provides the following advice about testing for business logic flaws (*https://owasp.org/www-community/vulnerabilities/Business_logic_vulnerability*):

> You'll need to evaluate the threat agents who could possibly exploit the problem and whether it would be detected. Again, this will take a strong understanding of the business. The vulnerabilities themselves are often quite easy to discover and exploit without any special tools or techniques, as they are a supported part of the application.

In other words, because business logic flaws are unique to each business and its logic, it is difficult to anticipate the specifics of the flaws you will find. Finding and exploiting these flaws is usually a matter of turning the features of an API against the API provider.

Business logic flaws could be discovered as early as when you review the API documentation and find directions for how not to use the application. (Chapter 3 lists the kinds of descriptions that should instantly make your vulnerability sensors go off.) When you find these, your next step should

be obvious: do the opposite of what the documentation recommends! Consider the following examples:

- *If the documentation tells you not to perform action X*, perform action X.
- *If the documentation tells you that data sent in a certain format isn't validated*, upload a reverse shell payload and try to find ways to execute it. Test the size of file that can be uploaded. If rate limiting is lacking and file size is not validated, you've discovered a serious business logic flaw that will lead to a denial of service.
- *If the documentation tells you that all file formats are accepted*, upload files and test all file extensions. You can find a list of file extensions for this purpose called *file-ext* (*https://github.com/hAPI-hacker/Hacking-APIs/tree/main/Wordlists*). If you can upload these sorts of files, the next step would be to see if you can execute them.

In addition to relying on clues in the documentation, consider the features of a given endpoint to determine how a nefarious person could use them to their advantage. The challenging part about business logic flaws is that they are unique to each business. Identifying features as vulnerabilities will require putting on your evil genius cap and using your imagination.

Summary

In this chapter, you learned how to find information about API requests so you can load it into Postman and begin your testing. Then you learned to use an API as it was intended and analyze responses for common vulnerabilities. You can use the described techniques to begin testing APIs for vulnerabilities. Sometimes all it takes is using the API with an adversarial mindset to make critical findings. In the next chapter, we will attack the API's authentication mechanisms.

Lab #4: Building a crAPI Collection and Discovering Excessive Data Exposure

In Chapter 6, we discovered the existence of the crAPI API. Now we will use what we've learned from this chapter to begin analyzing crAPI endpoints. In this lab, we will register an account, authenticate to crAPI, and analyze various features of the application. In Chapter 9, we'll attack the API's authentication process. For now, I will guide you through the natural progression from browsing a web application to analyzing API endpoints. We'll start by building a request collection from scratch and then work our way toward finding an excessive data exposure vulnerability with serious implications.

In the web browser of your Kali machine, navigate to the crAPI web application. In my case, the vulnerable app is located at 192.168.195.130, but yours might be different. Register an account with the crAPI web

application. The crAPI registration page requires all fields to be filled out with password complexity requirements (see Figure 7-16).

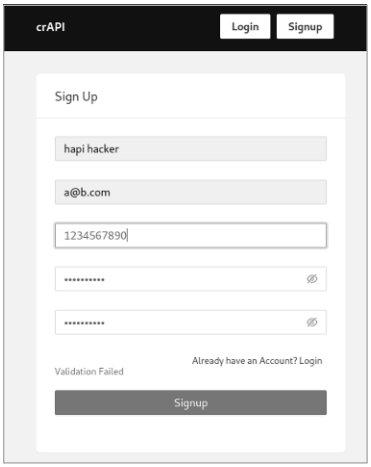

Figure 7-16: The crAPI account registration page

Since we know nothing about the APIs used in this application, we'll want to proxy the requests through Burp Suite to see what's going on below the GUI. Set up your proxy and click **Signup** to initiate the request. You should see that the application submits a POST request to the */identity/api/auth/signup* endpoint (see Figure 7-17).

Notice that the request includes a JSON payload with all of the answers you provided in the registration form.

```
Pretty  Raw  \n  Actions
 1 POST /identity/api/auth/signup HTTP/1.1
 2 Host: 192.168.195.130:8888
 3 Content-Length: 98
 4 User-Agent: Mozilla/5.0 (X11; Linux x86_64) AppleWebKit/537.36
 5 Content-Type: application/json
 6 Accept: */*
 7 Origin: http://192.168.195.130:8888
 8 Referer: http://192.168.195.130:8888/signup
 9 Accept-Encoding: gzip, deflate
10 Accept-Language: en-US,en;q=0.9
11 Connection: close
12
13 {
     "name":"hapi hacker one",
     "email":"email@email.com",
     "number":"0123456789",
     "password":"Password!1"
   }
```

Figure 7-17: An intercepted crAPI authentication request

Now that we've discovered our first crAPI API request, we'll start building a Postman collection. Click the **Options** button under the collection and then add a new request. Make sure that the request you build in Postman matches the request you intercepted: a POST request to the */identity/api/auth/signup* endpoint with a JSON object as the body (see Figure 7-18).

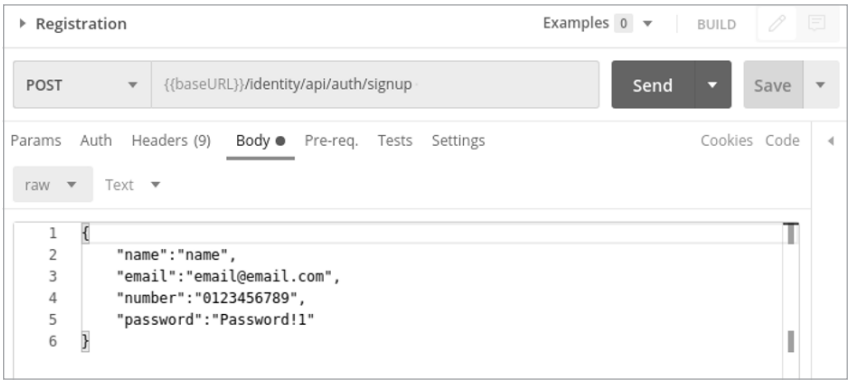

Figure 7-18: The crAPI registration request in Postman

Test the request to make sure you've crafted it correctly, as there is actually a lot that you could get wrong at this point. For example, your endpoint or body could contain a typo, you could forget to change the request method from GET to POST, or maybe you didn't match the headers of the original request. The only way to find out if you copied it correctly is to send a request, see how the provider responds, and troubleshoot if needed. Here are a couple hints for troubleshooting this first request:

- If you receive the status code 415 Unsupported Media Type, you need to update the `Content-Type` header so that the value is *application/json*.
- The crAPI application won't allow you to create two accounts using the same number or email, so you may need to alter those values in the body of your request if you already registered in the GUI.

You'll know your request is ready when you receive a status 200 OK as a response. Once you receive a successful response, make sure to save your request!

Now that we've saved the registration request to our crAPI collection, log in to the web app to see what other API artifacts there are to discover. Proxy the login request using the email and password you registered. When you submit a successful login request, you should receive a Bearer token from the application (see Figure 7-19). You'll need to include this Bearer token in all of your authenticated requests moving forward.

```
GET /identity/api/v2/user/dashboard HTTP/1.1
Host: 192.168.195.130:8888
Authorization: Bearer
eyJhbGciOiJIUzUxMiJ9.eyJzdWIiOiJlbWFpbEBlbWFpbC5jb20iLCJpYXQiOjE
2MTMzNjA3ODgsImV4cCI6MTYxMzQ0NzE4OH0.lm9tWUBf5k8v-4jFCFKFdZWOI5d
oAHoJTJhZGUBCbFY_5dr3WtWGBwOelSYLv22CUwGLmtj8yF19m-uZSzEdyw
User-Agent: Mozilla/5.0 (X11; Linux x86_64) AppleWebKit/537.36
(KHTML, like Gecko) Chrome/87.0.4280.88 Safari/537.36
Content-Type: application/json
Accept: */*
Referer: http://192.168.195.130:8888/login
Accept-Encoding: gzip, deflate
Accept-Language: en-US,en;q=0.9
Connection: close


```

Figure 7-19: An intercepted request after a successful login to crAPI

Add this Bearer token to your collection, either as an authorization method or a variable. I saved mine as an authorization method with the Type set to Bearer Token, as seen in Figure 7-20.

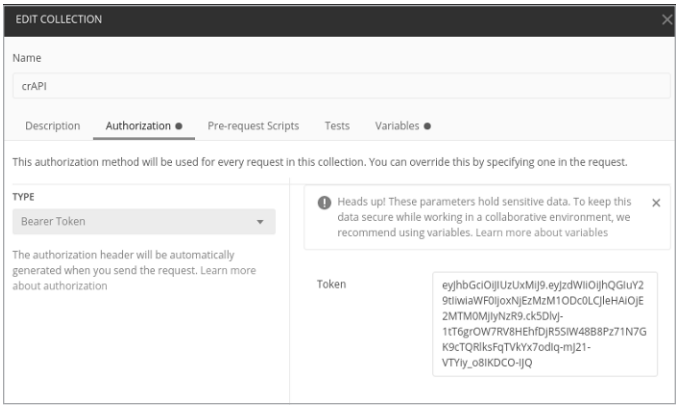

Figure 7-20: The Postman collection editor

Continue using the application in the browser, proxying its traffic, and saving the requests you discover to your collection. Try using different parts of the application, such as the dashboard, shop, and community, to name a few. Be sure to look for the kind of interesting functionality we discussed in this chapter.

One endpoint in particular should catch your attention simply based on the fact that it involves other crAPI users: the forum. Use the crAPI forum as it was intended in your browser and intercept the request. Submitting a comment to the forum will generate a POST request. Save the POST request to the collection. Now send the request used to populate the community forum to the */community/api/v2/community/posts/recent* endpoint. Notice anything significant in the JSON response body in Listing 7-1?

```
        "id": "fyRGJWyeEjKexxyYpQcRdZ",
        "title": "test",
        "content": "test",
        "author": {
            "nickname": "hapi hacker",
            "email": "a@b.com",
            "vehicleid": "493f426c-a820-402e-8be8-bbfc52999e7c",
            "profile_pic_url": "",
            "created_at": "2021-02-14T21:38:07.126Z"
        },
        "comments": [],
        "authorid": 6,
        "CreatedAt": "2021-02-14T21:38:07.126Z"
    },
    {
        "id": "CLnAGQPR4qDCwLPgTSTAQU",
        "title": "Title 3",
        "content": "Hello world 3",
        "author": {
            "nickname": "Robot",
            "email": "robot001@example.com",
            "vehicleid": "76442a32-f32f-4d7d-ae05-3e8c995f68ce",
            "profile_pic_url": "",
            "created_at": "2021-02-14T19:02:42.907Z"
        },
        "comments": [],
        "authorid": 3,
        "CreatedAt": "2021-02-14T19:02:42.907Z"
    }
```


Listing 7-1: A sample of the JSON response received from the /community/api/v2/community/posts/recent *endpoint*

Not only do you receive the JSON object for your post, you also receive the information about every post on the forum. Those objects contain much more information than is necessary, including sensitive information such as user IDs, email addresses, and vehicle IDs. If you've made it this far, congratulations; this means you've discovered an excessive data exposure vulnerability. Great job! There are many more vulnerabilities affecting crAPI, and we'll definitely use our findings here to help locate even more severe vulnerabilities in the upcoming chapters.