9
FUZZING

In this chapter, you'll explore using fuzzing techniques to discover several of the top API vulnerabilities discussed in Chapter 3. The secret to successfully discovering most API vulnerabilities is knowing where to fuzz and what to fuzz with. In fact, you'll likely discover many API vulnerabilities by fuzzing input sent to API endpoints.

Using Wfuzz, Burp Suite Intruder, and Postman's Collection Runner, we'll cover two strategies to increase your success: fuzzing wide and fuzzing deep. We'll also discuss how to fuzz for improper assets management vulnerabilities, find the accepted HTTP methods for a request, and bypass input sanitization.

Effective Fuzzing

In earlier chapters, we defined API fuzzing as the process of sending requests with various types of input to an endpoint in order to provoke an unintended result. While "various types of input" and "unintended result" might sound vague, that's only because there are so many possibilities. Your input could include symbols, numbers, emojis, decimals, hexadecimal, system commands, SQL input, and NoSQL input, for instance. If the API has not implemented validation checks to handle harmful input, you could end up with a verbose error, a unique response, or (in the worst case) some sort of internal server error indicating that your fuzz caused a denial of service, killing the app.

Fuzzing successfully requires a careful consideration of the app's likely expectations. For example, take a banking API call intended to allow users to transfer money from one account to another. The request could look something like this:

```
POST /account/balance/transfer
Host: bank.com
x-access-token: hapi_token

{
"userid": 12345,
"account": 224466,
"transfer-amount": 1337.25,
}
```

To fuzz this request, you could easily set up Burp Suite or Wfuzz to submit huge payloads as the `userid`, `account`, and `transfer-amount` values. However, this could set off defensive mechanisms, resulting in stronger rate limiting or your token being blocked. If the API lacks these security controls, by all means release the krakens. Otherwise, your best bet is to send a few targeted requests to only one of the values at a time.

Consider the fact that the `transfer-amount` value likely expects a relatively small number. Bank.com isn't anticipating an individual user to transfer an amount larger than the global GDP. It also likely expects a decimal value. Thus, you might want to evaluate what happens when you send the following:

- A value in the quadrillions
- String of letters instead of numbers
- A large decimal number or a negative number
- Null values like `null`, `(null)`, `%00`, and `0x00`
- Symbols like the following: `!@#$%^&*();':''|,./?>`

These requests could easily lead to verbose errors that reveal more about the application. A value in the quadrillions could additionally cause an unhandled SQL database error to be sent back as a response. This one piece of information could help you target values across the API for SQL injection vulnerabilities.

Thus, the success of your fuzzing will depend on where you are fuzzing and what you are fuzzing with. The trick is to look for API inputs that are leveraged for a consumer to interact with the application and send input that is likely to result in errors. If these inputs do not have sufficient input handling and error handling, they can often lead to exploitation. Examples of this sort of API input include the fields involved in requests used for authentication forms, account registration,

uploading files, editing web application content, editing user profile information, editing account information, managing users, searching for content, and so on.

The types of input to send really depend on the type of input you are attacking. Generically, you can send all sorts of symbols, strings, and numbers that could cause errors, and then you could pivot your attack based on the errors received. All of the following could result in interesting responses:

- Sending an exceptionally large number when a small number is expected
- Sending database queries, system commands, and other code
- Sending a string of letters when a number is expected
- Sending a large string of letters when a small string is expected
- Sending various symbols (`-_\!@#$%^&*();':''|,./?>`)
- Sending characters from unexpected languages (漢, さ, Ж, Ѫ, Ⱘ, Ⱥ, Ⱨ, Ƶ)

If you are blocked or banned while fuzzing, you might want to deploy evasion techniques discussed in Chapter 13 or else further limit the number of fuzzing requests you send.

Choosing Fuzzing Payloads

Different fuzzing payloads can incite various types of responses. You can use either generic fuzzing payloads or more targeted ones. *Generic payloads* are those we've discussed so far and contain symbols, null bytes, directory traversal strings, encoded characters, large numbers, long strings, and so on.

Targeted fuzzing payloads are aimed at provoking a response from specific technologies and types of vulnerabilities. Targeted fuzzing payload types might include API object or variable names, cross-site scripting (XSS) payloads, directories, file extensions, HTTP request methods, JSON or XML data, SQL or No SQL commands, or commands for particular operating systems. We'll cover examples of fuzzing with these payloads in this and future chapters.

You'll typically move from generic to targeted fuzzing based on the information received in API responses. Similar to reconnaissance efforts in Chapter 6, you will want to adapt your fuzzing and focus your efforts based on the results of generic testing. Targeted fuzzing payloads are more useful once you know the technologies being used. If you're sending SQL fuzzing payloads to an API that leverages only NoSQL databases, your testing won't be as effective.

One of the best sources for fuzzing payloads is SecLists (*https://github.com/danielmiessler/SecLists*). SecLists has a whole section dedicated to fuzzing, and its *big-list-of-naughty-strings.txt* wordlist is excellent at causing useful responses. The fuzzdb project is another good source for fuzzing payloads (*https://github.com/fuzzdb-project/fuzzdb*). Also, Wfuzz has many useful payloads (*https://github.com/xmendez/wfuzz*), including a great list that combines several targeted payloads in their injection directory, called *All_attack.txt*.

Additionally, you can always quickly and easily create your own generic fuzzing payload list. In a text file, combine symbols, numbers, and characters to create each payload as line-separated entries, like this:

```
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
9999999999999999999999999999999999999999
~'!@#$%^&*()-_+
{}[]|\:''; '<>?,./
%00
0x00
$ne
%24ne
$gt
%24gt
|whoami
-- -
' ''
' OR 1=1-- -
'' ''''''
漢, さ, ㅈ, ㅍ, ㅱ, ㅅ, ㅶ, ᶎ
😀 😃 😄 😁 😆
```

Note that instead of 40 instances of `A` or `9`, you could write payloads consisting of hundreds them. Using a small list like this as a fuzzing payload can cause all sorts of useful and interesting responses from an API.

Detecting Anomalies

When fuzzing, you're attempting to cause the API or its supporting technologies to send you information that you can leverage in additional attacks. When an API request payload is handled properly, you should receive some sort of HTTP response code and message indicating that your fuzzing did not work. For example,

sending a request with a string of letters when numbers are expected could result in a simple response like the following:

```
HTTP/1.1 400 Bad Request
{
        "error": "number required"
}
```

From this response, you can deduce that the developers configured the API to properly handle requests like yours and prepared a tailored response.

When input is not handled properly and causes an error, the server will often return that error in the response. For example, if you sent input like `~'!@#$%^&*()-_+` to an endpoint that improperly handles it, you could receive an error like this:

```
HTTP/1.1 200 OK
--snip--

SQL Error: There is an error in your SQL syntax.
```

This response immediately reveals that you're interacting with an API request that does not handle input properly and that the backend of the application is utilizing a SQL database.

You'll typically be analyzing hundreds or thousands of responses, not just two or three. Therefore, you need to filter your responses in order to detect anomalies. One way to do this is to understand what ordinary responses look like. You can establish this baseline by sending a set of expected requests or, as you'll see later in the lab, by sending requests that you expect to fail. Then you can review the results to see if a majority of them are identical. For example, if you issue 100 API requests and 98 of those result in an HTTP 200 response code with a similar response size, you can consider those requests to be your baseline. Also examine a few of the baseline responses to get a sense of their content. Once you know that the baseline responses have been properly handled, review the two anomalous responses. Figure out what input caused the difference, paying particular attention to the HTTP response code, response size, and the content of the response.

In some cases, the differences between baseline and anomalous requests will be miniscule. For example, the HTTP response codes might all be identical, but a few

requests might result in a response size that is a few bytes larger than the baseline responses. When small differences like this come up, use Burp Suite's Comparer to get a side-by-side comparison of the differences within the responses. Right-click the result you're interested in and choose **Send to Comparer (Response)**. You can send as many responses as you'd like to Comparer, but you'll at least need to send two. Then migrate to the Comparer tab, as shown in *Figure 9-1*.

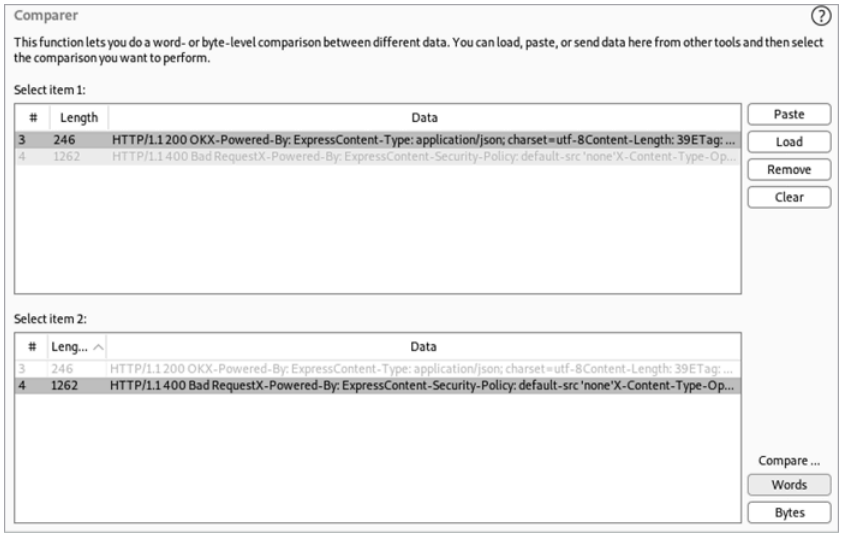

Figure 9-1: Burp Suite's Comparer

Select the two results you would like to compare and use the **Compare Words** button (located at the bottom right of the window) to pull up a side-by-side comparison of the responses (see *Figure 9-2*).

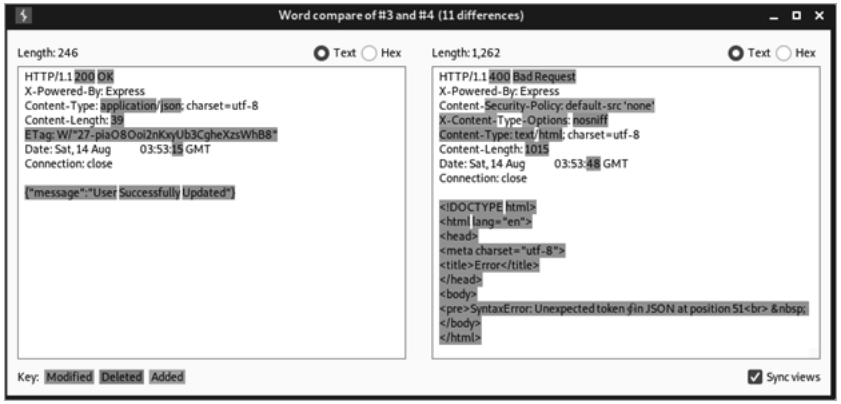

Figure 9-2: Comparing two API responses with Comparer

A useful option located at the bottom-right corner, called Sync Views, will help you synchronize the two responses. Sync Views is especially useful when you're looking for small differences in large responses, as it will automatically highlight differences between the two responses. The highlights signify whether the difference has been modified, deleted, or added.

Fuzzing Wide and Deep

This section will introduce you to two fuzzing techniques: fuzzing wide and fuzzing deep. *Fuzzing wide* is the act of sending an input across all of an API's unique requests in an attempt to discover a vulnerability. *Fuzzing deep* is the act of thoroughly testing an individual request with a variety of inputs, replacing headers, parameters, query strings, endpoint paths, and the body of the request with your payloads. You can think of fuzzing wide as testing a mile wide but an inch deep and fuzzing deep as testing an inch wide but a mile deep.

Wide and deep fuzzing can help you adequately evaluate every feature of larger APIs. When you're hacking, you'll quickly discover that APIs can greatly vary in size. Certain APIs could have only a few endpoints and a handful of unique requests, so you may be able to easily test them by sending a few requests. An API can have many endpoints and unique requests, however. Alternatively, a single request could be filled with many headers and parameters.

This is where the two fuzzing techniques come into play. Fuzzing wide is best used to test for issues across all unique requests. Typically, you can fuzz wide to test for improper assets management (more on this later in this chapter), finding all valid request methods, token-handling issues, and other information disclo-

sure vulnerabilities. Fuzzing deep is best used for testing many aspects of individual requests. Most other vulnerability discovery will be done by fuzzing deep. In later chapters, we will use the fuzzing-deep technique to discover different types of vulnerabilities, including BOLA, BFLA, injection, and mass assignment.

Fuzzing Wide with Postman

I recommend using Postman to fuzz wide for vulnerabilities across an API, as the tool's Collection Runner makes it easy to run tests against all API requests. If an API includes 150 unique requests across all the endpoints, you can set a variable to a fuzzing payload entry and test it across all 150 requests. This is particularly easy to do when you've built a collection or imported API requests into Postman. For example, you might use this strategy to test whether any of the requests fail to handle various "bad" characters. Send a single payload across the API and check for anomalies.

Create a Postman environment in which to save a set of fuzzing variables. This lets you seamlessly use the environmental variables from one collection to the next. Once the fuzzing variables are set, just as they are in *Figure 9-3*, you can save or update the environment.

At the top right, select the fuzzing environment and then use the variable shortcut `{{variable name}}` wherever you would like to test a value in a given collection. In *Figure 9-4*, I've replaced the `x-access-token` header with the first fuzzing variable.

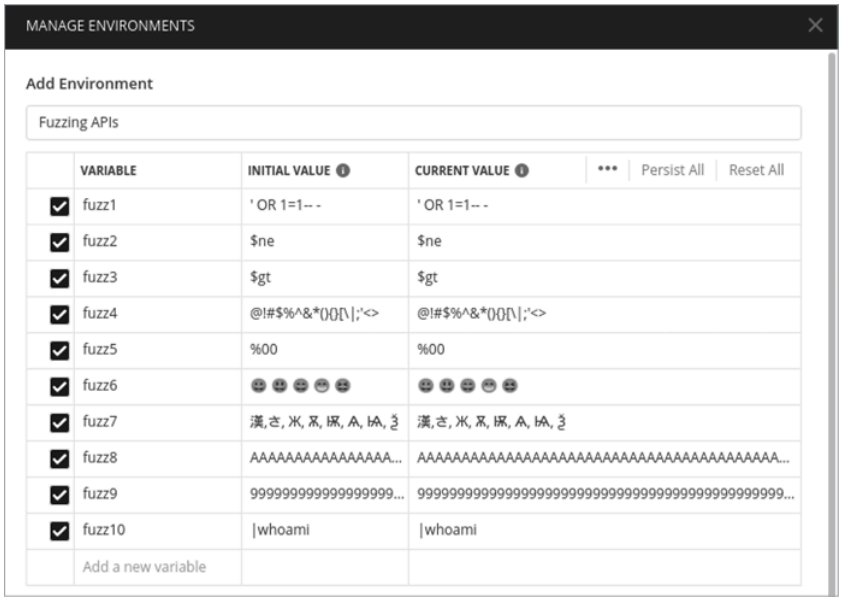

Figure 9-3: Creating fuzzing variables in the Postman environment editor

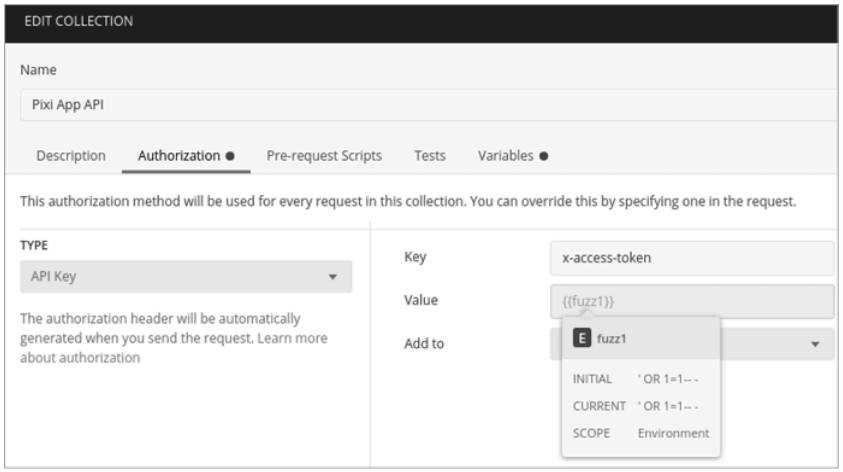

Figure 9-4: Fuzzing a collection token header

Additionally, you could replace parts of the URL, the other headers, or any custom variables you've set in the collection. Then you use the Collection Runner to test every request within the collection.

Another useful Postman feature when fuzzing wide is Find and Replace, found at the bottom left of Postman. Find and Replace lets you search a collection (or all

collections) and replace certain terms with a replacement of your choice. If you were attacking the Pixi API, for example, you might notice that many placeholder parameters use tags like `<email>`, `<number>`, `<string>`, and `<boolean>`. This makes it easy to search for these values and replace them with either legitimate ones or one of your fuzzing variables, like `{{fuzz1}}`.

Next, try creating a simple test in the Tests panel to help you detect anomalies. For instance, you could set up the test covered in Chapter 4 for a status code of 200 across a collection:

```
pm.test("Status code is 200", function () {
    pm.response.to.have.status(200);
});
```

With this test, Postman will check that responses have a status code of 200, and when a response is 200, it will pass the test. You can easily customize this test by replacing 200 with your preferred status code.

There are several ways to launch the Collection Runner. You can click the **Runner Overview** button, the arrow next to a collection, or the **Run** button. As mentioned earlier, you'll need to develop a baseline of normal responses by sending requests with no values or expected values to the targeted field. An easy way to get such a baseline is to unselect the checkbox **Keep Variable Values**. With this option turned off, your variables won't be used in the first collection run.

When we run this sample collection with the original request values, 13 requests pass our status code test and 5 fail. There is nothing extraordinary about this. The 5 failed attempts may be missing parameters or other input values, or they may just have response codes that are not 200. Without us making additional changes, this test result could function as a baseline.

Now let's try fuzzing the collection. Make sure your environment is set up correctly, responses are saved for our review, that **Keep Variable Values** is checked, and that any responses that generate new tokens are disabled (we can test those requests with deep fuzzing techniques). In *Figure 9-5*, you can see these settings applied.

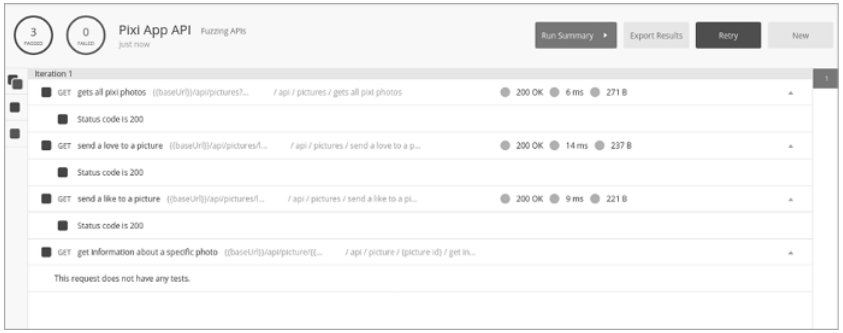

Figure 9-5: Postman Collection Runner results

Run the collection and then look for deviations from the baseline responses. Also watch for changes in the request behavior. For example, when we ran the requests using the value `Fuzz1('OR 1=1-- -)`, the Collection Runner passed three tests and then failed to process any additional requests. This is an indication that the web application took issue with the fuzzing attempt involved in the fourth request. Although we did not receive an interesting response, the behavior itself is an indication that you may have discovered a vulnerability.

Once you've cycled through a collection run, update the fuzzing value to the next variable you would like to test, perform another collection run, and compare results. You could detect several vulnerabilities by fuzzing wide with Postman, such as improper assets management, injection weaknesses, and other information disclosures that could lead to more interesting findings. When you've exhausted your fuzzing-wide attempts or found an interesting response, it is time to pivot your testing to fuzzing deep.

Fuzzing Deep with Burp Suite

You should fuzz deep whenever you want to drill down into specific requests. The technique is especially useful for thoroughly testing each individual API request. For this task, I recommend using Burp Suite or Wfuzz.

In Burp Suite, you can use Intruder to fuzz every header, parameter, query string, and endpoint path, along with any item included in the body of the request. For example, in a request like the one in *Figure 9-6*, shown in Postman, with many fields in the request body, you can perform a deep fuzz that passes hundreds or even thousands of fuzzing inputs into each value to see how the API responds.

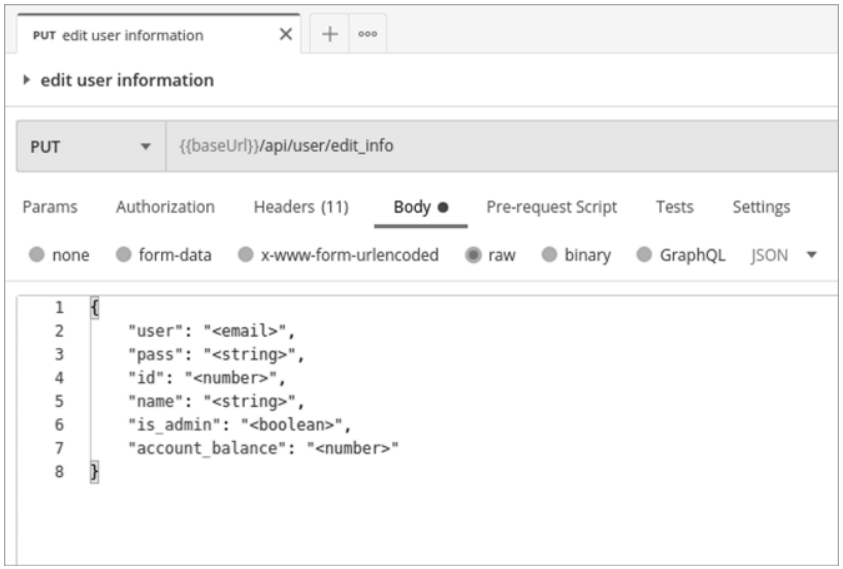

Figure 9-6: A PUT request in Postman

While you might initially craft your requests in Postman, make sure to proxy the traffic to Burp Suite. Start Burp Suite, configure the Postman proxy settings, send the request, and make sure it was intercepted. Then forward it to Intruder. Using the payload position markers, select every field's value to send a payload list as each of those values. A sniper attack will cycle a single wordlist through each attack position. The payload for an initial fuzzing attack could be similar to the list described in the "Choosing Fuzzing Payloads" section of this chapter.

Before you begin, consider whether a request's field expects any particular value. For example, take a look at the following PUT request, where the tags (`< >`) suggest that the API is configured to expect certain values:

```
PUT /api/user/edit_info HTTP/1.1
Host: 192.168.195.132:8090
Content-Type: application/json
x-access-token: eyJhbGciOiJIUzI1NiIsInR5cCI...
--snip--

{
    "user": "§<email>§",
    "pass": "§<string>§",
    "id": "§<number>§",
    "name": "§<string>§",
```

```
    "is_admin": "§<boolean>§",
    "account_balance": "§<number>§"
}
```

When you're fuzzing, it is always worthwhile to request the unexpected. If a field expects an email, send numbers. If it expects numbers, send a string. If it expects a small string, send a huge string. If it expects a Boolean value (true/false), send anything else. Another useful tip is to send the expected value and include a fuzzing attempt following that value. For example, email fields are fairly predictable, and developers often nail down the input validation to make sure that you are sending a valid-looking email. Since this is the case, when you fuzz an email field, you may receive the same response for all your attempts: "not a valid email." In this case, check to see what happens if you send a valid-looking email followed by a fuzzing payload. That would look something like this:

```
"user": "hapi@hacker.com§test§"
```

If you receive the same response ("not a valid email"), it is likely time to try a different payload or move on to a different field.

When fuzzing deep, be aware of how many requests you'll be sending. A sniper attack containing a list of 12 payloads across 6 payload positions will result in 72 total requests. This is a relatively small number of requests.

When you receive your results, Burp Suite has a few tools to help detect anomalies. First, organize the requests by column, such as status code, length of the response, and request number, each of which can yield useful information. Additionally, Burp Suite Pro allows you to filter by search terms.

If you notice an interesting response, select the result and choose the **Response** tab to dissect how the API provider responded. In *Figure 9-7*, fuzzing any field with the payload `{}[]|\:";'<>?,./` resulted in an HTTP 400 response code and the response `SyntaxError: Unexpected token in JSON at position 32`.

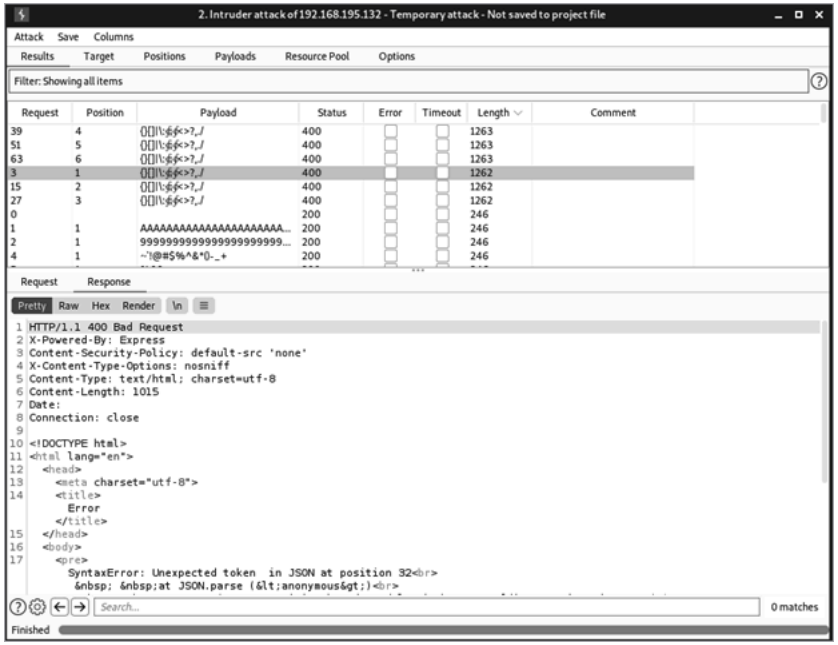

Figure 9-7: Burp Suite attack results

Once you have an interesting error like this one, you could improve your payloads to narrow down exactly what is causing the error. If you figure out the exact symbol or combination of symbols causing the issue, attempt to pair other payloads with it to see if you can get additional interesting responses. For instance, if the resulting responses indicate a database error, you could use payloads that target those databases. If the error indicates an operating system or specific programming language, use a payload targeting it. In this situation, the error is related to an unexpected JSON token, so it would be interesting to see how this endpoint handles JSON fuzzing payloads and what happens when additional payloads are added.

Fuzzing Deep with Wfuzz

If you're using Burp Suite CE, Intruder will limit the rate you can send requests, so you should use Wfuzz when sending a larger number of payloads. Using Wfuzz to send a large POST or PUT request can be intimidating at first due to the amount of information you'll need to correctly add to the command line. However, with a few tips, you should be able to migrate back and forth between Burp Suite CE and Wfuzz without too many challenges.

One advantage of Wfuzz is that it's considerably faster than Burp Suite, so we can increase our payload size. The following example uses a SecLists payload called *big-list-of-naughty-strings.txt*, which contains over 500 values:

```
$ wfuzz -z file,/home/hapihacker/big-list-of-naughty-strings.txt
```

Let's build our Wfuzz command step-by-step. First, to match the Burp Suite example covered in the previous section, we will need to include the `Content-Type` and `x-access-token` headers in order to receive authenticated results from the API. Each header is specified with the option `-H` and surrounded by quotes.

```
$ wfuzz -z file,/home/hapihacker/big-list-of-naughty-strings.txt -H "Content-Type: application/json" -H "x-access-token: [..
```

Next, note that the request method is PUT. You can specify it with the `-X` option. Also, to filter out responses with a status code of 400, use the `--hc 400` option:

```
$ wfuzz -z file,/home/hapihacker/big-list-of-naughty-strings.txt -H "Content-Type: application/json" -H "x-access-token: [..
```

Now, to fuzz a request body using Wfuzz, specify the request body with the `-d` option and paste the body into the command, surrounded by quotes. Note that Wfuzz will normally remove quotes, so use backslashes to keep them in the request body. As usual, we replace the parameters we would like to fuzz with the term `FUZZ`. Finally, we use `-u` to specify the URL we're attacking:

```
$ wfuzz -z file,/home/hapihacker/big-list-of-naughty-strings.txt -H "Content-Type: application/json" -H "x-access-token: [..
    \"user\": \"FUZZ\",
    \"pass\": \"FUZZ\",
    \"id\": \"FUZZ\",
    \"name\": \"FUZZ\",
    \"is_admin\": \"FUZZ\",
    \"account_balance\": \"FUZZ\"
}" -u http://192.168.195.132:8090/api/user/edit_info
```

This is a decent-sized command with plenty of room to make mistakes. If you need to troubleshoot it, I recommend proxying the requests to Burp Suite, which should help you visualize the requests you're sending. To proxy traffic back to Burp, use the `-p` proxy option with your IP address and the port on which Burp Suite is running:

```
$ wfuzz -z file,/home/hapihacker/big-list-of-naughty-strings.txt -H "Content-Type: application/json" -H "x-access-token: [..
    \"user\": \"FUZZ\",
    \"pass\": \"FUZZ\",
    \"id\": \"FUZZ\",
    \"name\": \"FUZZ\",
    \"is_admin\": \"FUZZ\",
    \"account_balance\": \"FUZZ\"
}" -u http://192.168.195.132:8090/api/user/edit_info
```

In Burp Suite, inspect the intercepted request and send it to Repeater to see if there are any typos or mistakes. If your Wfuzz command is operating properly, run it and review the results, which should look like this:

```
********************************************************
* Wfuzz - The Web Fuzzer                               *
********************************************************

Target: http://192.168.195.132:8090/api/user/edit_info
Total requests: 502

=====================================================================
ID           Response   Lines    Word       Chars       Payload
=====================================================================

000000001:   200        0 L      3 W        39 Ch       "undefined - undefined - undefined - undefined - undefined - undefir
000000012:   200        0 L      3 W        39 Ch       "TRUE - TRUE - TRUE - TRUE - TRUE - TRUE"
000000017:   200        0 L      3 W        39 Ch       "\\ - \\ - \\ - \\ - \\ - \\"
000000010:   302        10 L    63 W        1014 Ch       "<a href='\xE2\x80..."
```

Now you can seek out the anomalies and conduct additional requests to analyze what you've found. In this case, it would be worth seeing how the API provider responds to the payload that caused a 302 response code. Use this payload in Burp Suite's Repeater or Postman.

Fuzzing Wide for Improper Assets Management

Improper assets management vulnerabilities arise when an organization exposes APIs that are either retired, in a test environment, or still in development. In any of these cases, there is a good chance the API has fewer protections than its supported production counterparts. Improper assets management might affect only a

single endpoint or request, so it's often useful to fuzz wide to test if improper assets management exists for any request across an API.

NOTE

In order to fuzz wide for this problem, it helps to have a specification of the API or a collection file that will make the requests available in Postman. This section assumes you have an API collection available.

As discussed in Chapter 3, you can find improper assets management vulnerabilities by paying close attention to outdated API documentation. If an organization's API documentation has not been updated along with the organization's API endpoints, it could contain references to portions of the API that are no longer supported. Also, check any sort of changelog or GitHub repository. A changelog that says something along the lines of "resolved broken object level authorization vulnerability in v3" will make finding an endpoint still using v1 or v2 all the sweeter.

Other than using documentation, you can discover improper assets vulnerabilities through the use of fuzzing. One of the best ways to do this is to watch for patterns in the business logic and test your assumptions. For example, in *Figure 9-8*, you can see that the `baseURL` variable used within all requests for this collection is *https://petstore.swagger.io/v2*. Try replacing *v2* with *v1* and using Postman's Collection Runner.

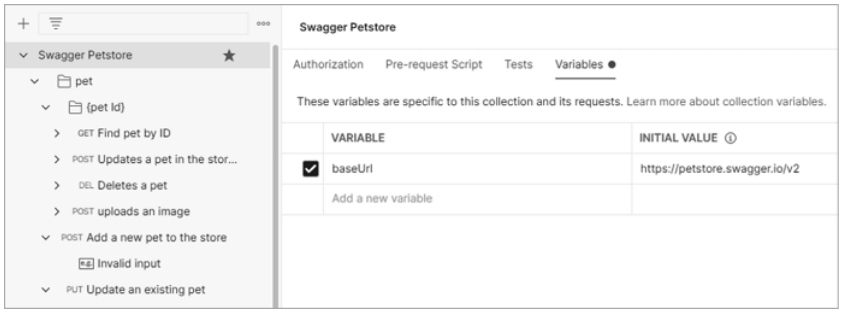

Figure 9-8: Editing the collection variables within Postman

The production version of the sample API is *v2*, so it would be a good idea to test a few keywords, like *v1*, *v3*, *test*, *mobile*, *uat*, *dev*, and *old*, as well as any interesting paths discovered during analysis or reconnaissance testing. Additionally, some API providers will allow access to administrative functionality by adding */internal/* to the path before or after the versioning, which would look like this:

*/api/v2/**internal**/users*
*/api/**internal**/v2/users*

As discussed earlier in the section, begin by developing a baseline for how the API responds to typical requests using the Collection Runner with the API's expected version path. Figure out how an API responds to a successful request and how it responds to bad ones (or requests for resources that do not exist).

To make our testing easier, we'll set up the same test for status codes of 200 we used earlier in this chapter. If the API provider typically responds with status code 404 for nonexistent resources, a 200 response for those resources would likely indicate that the API is vulnerable. Make sure to insert this test at the collection level so that it will be run on every request when you use the Collection Runner.

Now save and run your collection. Inspect the results for any requests that pass this test. Once you've reviewed the results, rinse and repeat with a new keyword. If you discover an improper asset management vulnerability, your next step will be to test the non-production endpoint for additional weaknesses. This is where your information-gathering skills will be put to good use. On the target's GitHub or in a changelog, you might discover that the older version of the API was vulnerable to a BOLA attack, so you could attempt such an attack on the vulnerable

endpoint. If you don't find any leads during reconnaissance, combine the other techniques found in this book to leverage the vulnerability.

Testing Request Methods with Wfuzz

One practical way to use fuzzing is to determine all the HTTP request methods available for a given API request. You can use several of the tools we've introduced to perform this task, but this section will demonstrate it with Wfuzz.

First, capture or craft the API request whose acceptable HTTP methods you would like to test. In this example, we'll use the following:

```
GET /api/v2/account HTTP/1.1
HOST: restfuldev.com
User-Agent: Mozilla/5.0
Accept: application/json
```

Next, create your request with Wfuzz, using `-X FUZZ` to specifically fuzz the HTTP method. Run Wfuzz and review the results:

```
$ wfuzz -z list,GET-HEAD-POST-PUT-PATCH-TRACE-OPTIONS-CONNECT- -X FUZZ http://testsite.com/api/v2/account

********************************************************
* Wfuzz 3.1.0 - The Web Fuzzer                         *
********************************************************

Target: http://testsite.com/api/v2/account
Total requests: 8

=====================================================================
ID           Response   Lines    Word       Chars       Payload
=====================================================================

000000008:   405        7 L      11 W       163 Ch      "CONNECT"
000000004:   405        7 L      11 W       163 Ch      "PUT"
000000005:   405        7 L      11 W       163 Ch      "PATCH"
000000007:   405        7 L      11 W       163 Ch      "OPTIONS"
000000006:   405        7 L      11 W       163 Ch      "TRACE"
000000002:   200        0 L      0 W        0 Ch        "HEAD"
000000001:   200        0 L      107 W      2610 Ch     "GET"
000000003:   405        0 L      84 W       1503 Ch     "POST"
```

Based on these results, you can see that the baseline response tends to include a 405 status code (Method Not Allowed) and a response length of 163 characters. The anomalous responses include the two request methods with 200 response codes. This confirms that GET and HEAD requests both work, which doesn't reveal much of anything new. However, this test also reveals that you can use a POST request to the *api/v2/account* endpoint. If you were testing an API that did not include this request method in its documentation, there is a chance you may have discovered functionality that was not intended for end users. Undocumented functionality is a good find that should be tested for additional vulnerabilities.

Fuzzing "Deeper" to Bypass Input Sanitization

When fuzzing deep, you'll want to be strategic about setting payload positions. For example, for an email field in a PUT request, an API provider may do a pretty decent job at requiring that the contents of the request body match the format of an email address. In other words, anything sent as a value that isn't an email address might result in the same 400 Bad Request error. Similar restrictions likely apply to integer and Boolean values. If you've thoroughly tested a field and it doesn't yield any interesting results, you may want to leave it out of additional tests or save it for more thorough testing in a separate attack.

Alternatively, to fuzz even deeper into a specific field, you could try to escape whatever restrictions are in place. By *escaping*, I mean tricking the server's input sanitization code into processing a payload it should normally restrict. There are a few tricks you could use against restricted fields.

First, try sending something that takes the form of the restricted field (if it's an email field, include a valid-looking email), add a null byte, and then add another payload position for fuzzing payloads to be inserted. Here's an example:

```
"user": "a@b.com%00§test§"
```

Instead of a null byte, try sending a pipe (`|`), quotes, spaces, and other escape symbols. Better yet, there are enough possible symbols to send that you could add a second payload position for typical escape characters, like this:

```
"user": "a@b.com§escape§§test§"
```

Use a set of potential escape symbols for the `§escape§` payload and the payload you want to execute as the `§test§`. To perform this test, use Burp Suite's cluster bomb attack, which will cycle through multiple payload lists and attempt every other payload against it:

```
Escape1
Escape1
Escape1
Escape2
Escape2
Escape2
Payload1
Payload2
Payload3
Payload1
Payload2
Payload3
```

The cluster bomb fuzzing attack is excellent at exhausting certain combinations of payloads, but be aware that the request quantity will grow exponentially. We will spend more time with the style of fuzzing when we are attempting injection attacks in Chapter 12.

Fuzzing for Directory Traversal

Another weakness you can fuzz for is directory traversal. Also known as path traversal, *directory traversal* is a vulnerability that allows an attacker to direct the web application to move to a parent directory using some form of the expression `../` and then read arbitrary files. You could leverage a series of path traversal dots and slashes in place of the escape symbols described in the previous section, like the following ones:

```
..
..\
../
\..\
\..\.\
```

This weakness has been around for many years, and all sorts of security controls, including user input sanitization, are normally in place to prevent it, but with the

right payload, you might be able to avoid these controls and web application firewalls. If you're able to exit the API path, you may be able to access sensitive information such as application logic, usernames, passwords, and additional personally identifiable information (like names, phone numbers, emails, and addresses).

Directory traversal can be conducted using both wide and deep fuzzing techniques. Ideally, you would fuzz deeply across all of an API's requests, but since this can be an enormous task, try fuzzing wide and then focusing in on specific request values. Make sure to enrich your payloads with information collected from reconnaissance, endpoint analysis, and API responses containing errors or other information disclosures.

Summary

This chapter covered the art of fuzzing APIs, one of the most important attack techniques you'll need to master. By sending the right inputs to the right parts of an API request, you can discover a variety of API weaknesses. We covered two strategies, fuzzing wide and deep, useful for testing the entire attack surface of large APIs. In the following chapters, we'll return to the fuzzing deep technique to discover and attack many API vulnerabilities.

Lab #6: Fuzzing for Improper Assets Management Vulnerabilities

In this lab, you'll put your fuzzing skills to the test against crAPI. If you haven't done so already, build a crAPI Postman collection, as we did in Chapter 7, and obtain a valid token. Now we can start by fuzzing wide and then pivot to fuzzing deep based on our findings.

Let's begin by fuzzing for improper assets management vulnerabilities. First, we'll use Postman to fuzz wide for various API versions. Open Postman and navigate to the environmental variables (use the eye icon located at the top right of Postman as a shortcut). Add a variable named `path` to your Postman environment and set the value to *v3*. Now you can update to test for various versioning-related paths (such as *v1*, *v2*, *internal*, and so on).

To get better results from the Postman Collection Runner, we'll configure a test using the Collection Editor. Select the crAPI collection options, choose **Edit**, and select the **Tests** tab. Add a test that will detect when a status code 404 is returned so

that anything that does not result in a 404 Not Found response will stick out as anomalous. You can use the following test:

```
pm.test("Status code is 404", function () {
    pm.response.to.have.status(404);
});
```

Run a baseline scan of the crAPI collection with the Collection Runner. First, make sure that your environment is up-to-date and **Save Responses** is checked (see *Figure 9-9*).

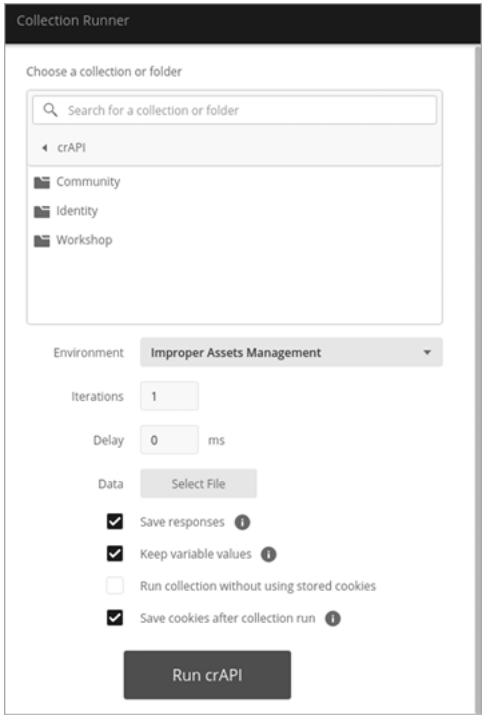

Figure 9-9: Postman Collection Runner

Since we're on the hunt for improper assets management vulnerabilities, we'll only test API requests that contain versioning information in the path. Using Postman's Find and Replace feature, replace the values *v2* and *v3* across the collection with the `path` variable (see *Figure 9-10*).

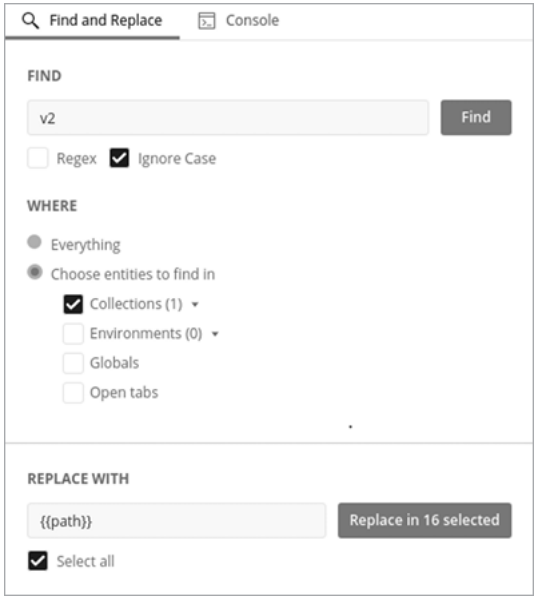

Figure 9-10: Replacing version information in the path with a Postman variable

You may have noticed a matter of interest regarding our collection: all of the endpoints have *v2* in their paths except for the password reset endpoint, */identity/api/auth/v3/check-otp*, which is using *v3*.

Now that the variable is set, run a baseline scan with a path that we expect to fail across the board. As shown in *Figure 9-11*, the `path` variable is set to a current value of `fail12345`, which is not likely to be a valid value in any endpoint. Knowing how the API reacts when it fails will help us understand how the API responds to requests for nonexistent paths. This baseline will aid our attempts to fuzz wide with the Collection Runner (see *Figure 9-12*). If requests to paths that do not exist result in Success 200 responses, we'll have to look out for other indicators to use to detect anomalies.

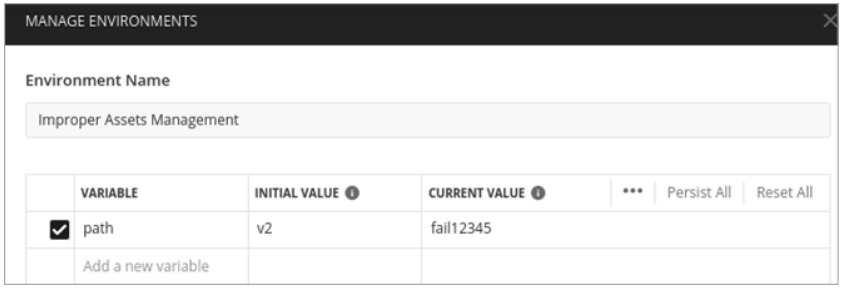

Figure 9-11: The improper assets management variable

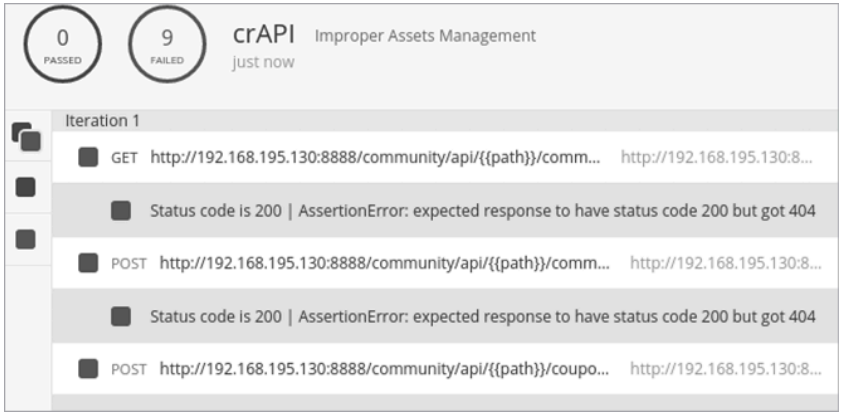

Figure 9-12: A baseline Postman Collection Runner test

As expected, *Figure 9-12* shows that all nine requests failed the test, as the API provider returned a status code 404. Now we can easily spot anomalies when testing for paths such as *test*, *mobile*, *uat*, *v1*, *v2*, and *v3*. Update the current value of the `path` variable to these other potentially unsupported paths and run the Collection Runner again. To quickly update a variable, click the eye icon found at the top right of Postman.

Things should start to get interesting when you return to the path values */v2* and */v3*. When the `path` variable is set to */v3*, all requests fail the test. This is slightly odd, because we noted earlier that the password reset request was using */v3*. Why is that request failing now? Well, based on the Collection Runner, the password reset request is actually receiving a 500 Internal Server Error, while all other requests are receiving a 404 Not Found status code. Anomaly!

Investigating the password reset request further will show that an HTTP 500 error is issued using the */v3* path because the application has a control that limits the

number of times you can attempt to send the one-time passcode (OTP). Sending the same request to */v2* also results in an HTTP 500 error, but the response is slightly larger. It may be worth retrying the two requests back in Burp Suite and using Comparer to see the small differences. The */v3* password reset request responds with `{"message":"ERROR..","status":500}`. The */v2* password reset request responds with `{"message":"Invalid OTP! Please try again..","status":500}`.

The password reset request does not align with the baseline we have developed by responding with a 404 status code when a URL path is not in use. Instead, we have discovered an improper assets management vulnerability! The impact of this vulnerability is that */v2* does not have a limitation on the number of times we can guess the OTP. With a four-digit OTP, we should be able to fuzz deep and discover any OTP within 10,000 requests. Eventually, you'll receive a message indicating your victory: `{"message":"OTP verified","status":200}`.