10
EXPLOITING AUTHORIZATION

In this chapter, we will cover two authorization vulnerabilities: BOLA and BFLA. These vulnerabilities reveal weaknesses in the authorization checks that ensure authenticated users are only able to access their own resources or use functionality that aligns with their permission level. In the process, we'll discuss how to identify resource IDs, use A-B and A-B-A testing, and speed up your testing with Postman and Burp Suite.

Finding BOLAs

BOLA continues to be one of the most prominent API-related vulnerabilities, but it can also be one of the easiest to test for. If you see that the API lists resources following a certain pattern, you can test other instances using that pattern. For instance, say you notice that after making a purchase, the app uses an API to provide you with a receipt at the following location: */api/v1/receipt/135*. Knowing this, you could then check for other numbers by using 135 as the payload position in Burp Suite or Wfuzz and changing 135 to numbers between 0 and 200. This was exactly what we did in the Chapter 4 lab when testing *reqres.in* for the total number of user accounts.

This section will cover additional considerations and techniques pertinent to hunting for BOLA. When you're on the hunt for BOLA vulnerabilities, remember that they aren't only found using GET requests. Attempt to use all possible methods to interact with resources you shouldn't be authorized to access. Likewise, vulnerable resource IDs aren't limited to the URL path. Make sure to consider other possible locations to check for BOLA weaknesses, including the body of the request and headers.

Locating Resource IDs

So far, this book has illustrated BOLA vulnerabilities using examples like performing sequential requests for resources:

```
GET /api/v1/user/account/ 1111
GET /api/v1/user/account/ 1112
```

To test for this vulnerability, you could simply brute-force all account numbers within a certain range and check whether requests result in a successful response.

Sometimes, finding instances of BOLA will actually be this straightforward. However, to perform thorough BOLA testing, you'll need to pay close attention to the information the API provider is using to retrieve resources, as it may not be so obvious. Look for user ID names or numbers, resource ID names or numbers, organization ID names or numbers, emails, phone numbers, addresses, tokens, or encoded payloads used in requests to retrieve resources.

Keep in mind that predictable request values don't make an API vulnerable to BOLA; the API is considered vulnerable only when it provides an unauthorized user access to the requested resources. Often, insecure APIs will make the mistake of validating that the user is authenticated but fail to check whether that user is authorized to access the requested resources.

As you can see in *Table 10-1*, there are plenty of ways you can attempt to obtain resources you shouldn't be authorized to access. These examples are based on actual successful BOLA findings. In each of these requests, the requester used the same UserA token.

Table 10-1: Valid Requests for Resources and the Equivalent BOLA Test

Type	Valid request	BOLA test
Predictable ID	`GET /api/v1/account/ 2222` `Token: UserA_token`	`GET /api/v1/account/ 3333` `Token: UserA_token`
ID combo	`GET /api/v1/ UserA /data/ 2222` `Token: UserA_token`	`GET /api/v1/ UserB /data/ 3333` `Token: UserA_token`
Integer as ID	`POST /api/v1/account/` `Token: UserA_token` `{"Account": 2222 }`	`POST /api/v1/account/` `Token: UserA_token` `{"Account": [ 3333 ]}`
Email as user ID	`POST /api/v1/user/account` `Token: UserA_token` `{"email": " UserA@email.com "}`	`POST /api/v1/user/account` `Token: UserA_token` `{"email": " UserB@email.com "}`
Group ID	`GET /api/v1/group/ CompanyA` `Token: UserA_token`	`GET /api/v1/group/ CompanyB` `Token: UserA_token`

Type	Valid request	BOLA test
Group and user combo	`POST /api/v1/group/ CompanyA Token: UserA_token {"email": " userA@CompanyA.com "}`	`POST /api/v1/group/ CompanyB Token: UserA_token {"email": " userB@CompanyB.com "}`
Nested object	`POST /api/v1/user/checking Token: UserA_token {"Account": 2222 }`	`POST /api/v1/user/checking Token: UserA_token {"Account": {"Account" :3333}}`
Multiple objects	`POST /api/v1/user/checking Token: UserA_token {"Account": 2222 }`	`POST /api/v1/user/checking Token: UserA_token {"Account": 2222, "Account": 3333, "Account": 5555 }`
Predictable token	`POST /api/v1/user/account Token: UserA_token`	`POST /api/v1/user/account Token: UserA_token {"data": "DflK1df7jSdfa` **2df** `aa"}`

Type	Valid request	BOLA test
	`{"data": "DflK1df7jSdfa 1ac aa"}`	

Sometimes, just requesting the resource won't be enough; instead, you'll need to request the resource as it was meant to be requested, often by supplying both the resource ID and the user's ID. Thus, due to the nature of how APIs are organized, a proper request for resources may require the *ID combo* format shown in *Table 10-1*. Similarly, you may need to know the group ID along with the resource ID, as in the *group and user combo* format.

Nested objects are a typical structure found in JSON data. These are simply additional objects created within an object. Since nested objects are a valid JSON format, the request will be processed if user input validation does not prevent it. Using a nested object, you could escape or bypass security measures applied to the outer key/value pair by including a separate key/value pair within the nested object that may not have the same security controls applied to it. If the application processes these nested objects, they are an excellent vector for an authorization weakness.

A-B Testing for BOLA

What we call *A-B testing* is the process of creating resources using one account and attempting to retrieve those resources as a different account. This is one of the best ways to identify how resources are identified and what requests are used to obtain them. The A-B testing process looks like this:

- **Create resources as UserA.** Note how the resources are identified and how the resources are requested.
- **Swap out your UserA token for another user's token.** In many instances, if there is an account registration process, you will be able to create a second account (UserB).
- **Using UserB's token, make the request for UserA's resources.** Focus on resources for private information. Test for any resources that UserB should not have access to, such as full name, email, phone number, Social Security number, bank account information, legal information, and transaction data.

The scale of this testing is small, but if you can access one user's resources, you could likely access all user resources of the same privilege level.

A variation on A-B testing is to create three accounts for testing. That way, you can create resources in each of the three different accounts, detect any patterns in the resource identifiers, and check which requests are used to request those resources, as follows:

- **Create multiple accounts at each privilege level to which you have access.** Keep in mind that your goal is to test and validate security controls, not destroy someone's business. When performing BFLA attacks, there is a chance you could successfully delete the resources of other users, so it helps to limit a dangerous attack like this to a test account you create.

- **Using your accounts, create a resource with UserA's account and attempt to interact with it using UserB's.** Use all the methods at your disposal.

Side-Channel BOLA

One of my favorite methods of obtaining sensitive information from an API is through side-channel disclosure. Essentially, this is any information gleaned from unexpected sources, such as timing data. In past chapters, we discussed how APIs can reveal the existence of resources through middleware like `X-Response-Time`. Side-channel discoveries are another reason why it is important to use an API as it was intended and develop a baseline of normal responses.

In addition to timing, you could use response codes and lengths to determine if resources exist. For example, if an API responds to nonexistent resources with a 404 Not Found but has a different response for existing resources, such as 405 Unauthorized, you'll be able to perform a BOLA side-channel attack to discover existing resources such as usernames, account IDs, and phone numbers.

Table 10-2 gives a few examples of requests and responses that could be useful for side-channel BOLA disclosures. If 404 Not Found is a standard response for nonexistent resources, the other status codes could be used to enumerate usernames, user ID numbers, and phone numbers. These requests provide just a few examples of information that could be gathered when the API has different responses for nonexistent resources and existing resources that you are not authorized to view. If these requests successful, they can result in a serious disclosure of sensitive data.

Table 10-2: *Examples of Side-Channel BOLA Disclosures*

Request	Response
`GET /api/user/test987123`	`404 Not Found HTTP/1.1`
`GET /api/user/hapihacker`	`405 Unauthorized HTTP/1.1` `{` `}`
`GET /api/user/1337`	`405 Unauthorized HTTP/1.1` `{` `}`
`GET /api/user/phone/2018675309`	`405 Unauthorized HTTP/1.1` `{` `}`

On its own, this BOLA finding may seem minimal, but information like this can prove to be valuable in other attacks. For example, you could leverage information gathered through a side-channel disclosure to perform brute-force attacks to gain entry to valid accounts. You could also use information gathered in a disclosure like this to perform other BOLA tests, such as the ID combo BOLA test shown in *Table 10-1*.

Finding BFLAs

Hunting for BFLA involves searching for functionality to which you should not have access. A BFLA vulnerability might allow you to update object values, delete data, and perform actions as other users. To check for it, try to alter or delete resources or gain access to functionality that belongs to another user or privilege level.

Note that if you successfully send a DELETE request, you'll no longer have access to the given resource . . . because you'll have deleted it. For that reason, avoid testing for DELETE while fuzzing, unless you're targeting a test environment. Imagine that you send DELETE requests to 1,000 resource identifiers; if the requests succeed, you'll have deleted potentially valuable information, and your client won't be happy. Instead, start your BFLA testing on a small scale to avoid causing huge interruptions.

A-B-A Testing for BFLA

Like A-B testing for BOLA, A-B-A testing is the process of creating and accessing resources with one account and then attempting to alter the resources with another account. Finally, you should validate any changes with the original account. The A-B-A process should look something like this:

- **Create, read, update, or delete resources as UserA.** Note how the resources are identified and how the resources are requested.
- **Swap out your UserA token for UserB's.** In instances where there is an account registration process, create a second test account.

- **Send GET, PUT, POST, and DELETE requests for UserA's resources using UserB's token.** If possible, alter resources by updating the properties of an object.
- **Check UserA's resources to validate changes have been made by using UserB's token.** Either by using the corresponding web application or by making API requests using UserA's token, check the relevant resources. If, for example, the BFLA attack was an attempt to delete UserA's profile picture, load UserA's profile to see if the picture is missing.

In addition to testing authorization weaknesses at a single privilege level, ensure that you check for weaknesses at other privilege levels. As previously discussed, APIs could have all sorts of different privilege levels, such as basic user, merchant, partner, and admin. If you have access to accounts at the various privilege levels, your A-B-A testing can take on a new layer. Try making UserA an administrator and UserB a basic user. If you're able to exploit BLFA in that situation, it will have become a privilege escalation attack.

Testing for BFLA in Postman

Begin your BFLA testing with authorized requests for UserA's resources. If you were testing whether you could modify another user's pictures in a social media app, a simple request like the one shown in *Listing 10-1* would do:

```
GET /api/picture/2
Token: UserA_token
```

Listing 10-1: Sample request for BFLA testing

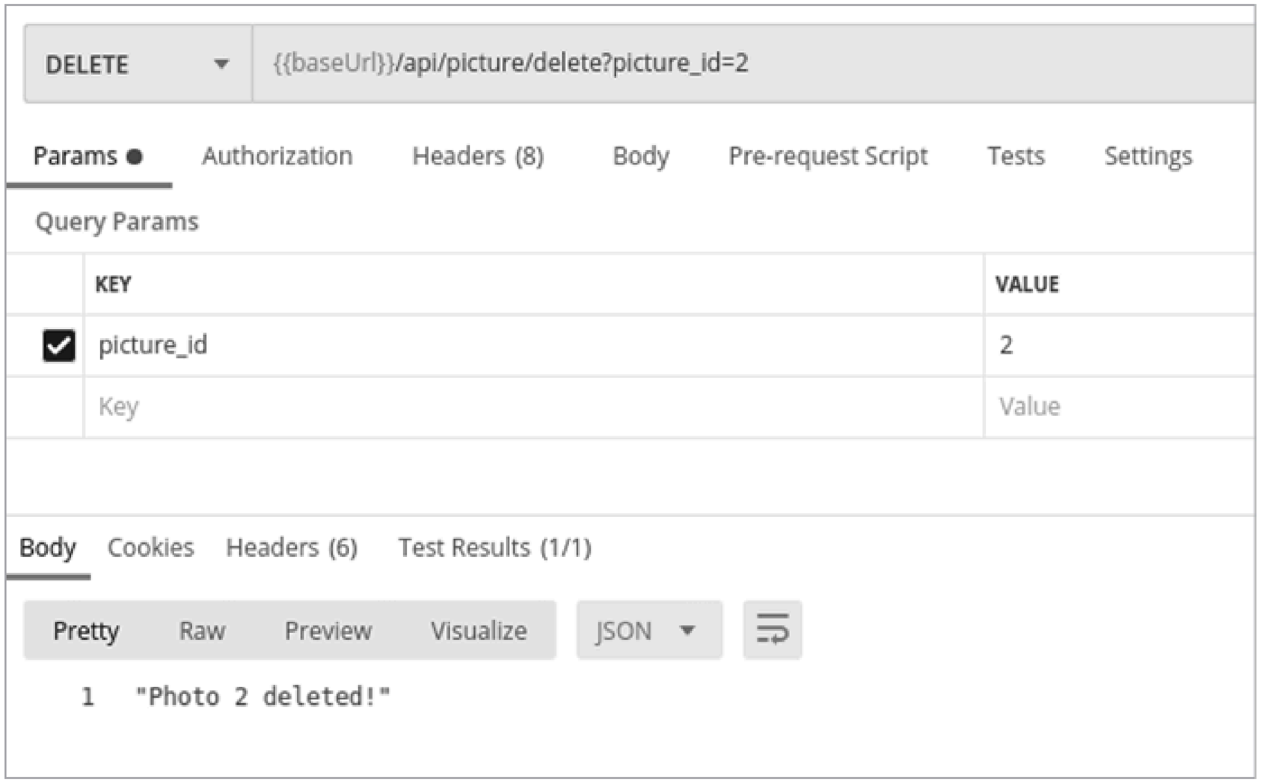
This request tells us that resources are identified by numeric values in the path. Moreover, the response, shown in *Listing 10-2*, indicates that the username of the resource (`"UserA"`) matches the request token.

```
200 OK
{
    "_id": 2,
    "name": "development flower",
    "creator_id": 2,
    "username": "UserA",
    "money_made": 0.35,
    "likes": 0
}
```

Listing 10-2: Sample response from a BFLA test

Now, given that this is a social media platform where users can share pictures, it wouldn't be too surprising if another user had the ability to send a successful GET request for picture 2. This isn't an instance of BOLA but rather a feature. However, UserB shouldn't be able to delete pictures that belong to UserA. That is where we cross into a BFLA vulnerability.

In Postman, try sending a DELETE request for UserA's resource containing UserB's token. As you see in *Figure 10-1*, a DELETE request using UserB's token was able to successfully delete UserA's picture. To validate that the picture was deleted, send a follow-up GET request for `picture_id=2`, and you will confirm that UserA's picture with the ID of 2 no longer exists. This is a very important finding, since a single malicious user could easily delete all other users' resources.



```
DELETE  {{baseUrl}}/api/picture/delete?picture_id=2
```

Query Params

	KEY	VALUE
✓	picture_id	2
	Key	Value

```
"Photo 2 deleted!"
```

Figure 10-1: Successful BFLA attack with Postman

You can simplify the process of finding privilege escalation–related BFLA vulnerabilities if you have access to documentation. Alternatively, you might find administrative actions clearly labeled in a collection, or you might have reverse engineered administrative functionality. If this isn't the case, you'll need to fuzz for admin paths.

One of the simplest ways to test for BFLA is to make administrative requests as a low-privileged user. If an API allows administrators to search for users with a POST request, try making that exact admin request to see if any security controls are in place to prevent you from succeeding. Look at the request in Listing 10-3. In the response (Listing 10-4), we see that the API did not have any such restrictions.

```
POST /api/admin/find/user
Token: LowPriv-Token

{"email": "hapi@hacker.com"}
```

Listing 10-3: Request for user information

```
200 OK HTTP/1.1

{
"fname": "hAPI",
"lname": "Hacker",
"is_admin": false,
"balance": "3737.50"
"pin": 8675
}
```

Listing 10-4: Response with user information

The ability to search for users and gain access to another user's sensitive information was meant to be restricted to only those with an administrative token.

However, by making a request to the */admin/find/user* endpoint, you can test to see if there is any technical enforcement. Since this is an administrative request, a successful response could also provide sensitive information, such as a user's full name, balance, and personal identification number (PIN).

If restrictions are in place, try changing the request method. Use a POST request instead of a PUT request, or vice versa. Sometimes an API provider has secured one request method from unauthorized requests but has overlooked another.

Authorization Hacking Tips

Attacking a large-scale API with hundreds of endpoints and thousands of unique requests can be fairly time-consuming. The following tactics should help you test for authorization weaknesses across an entire API: using Collection variables in Postman and using the Burp Suite Match and Replace feature.

Postman's Collection Variables

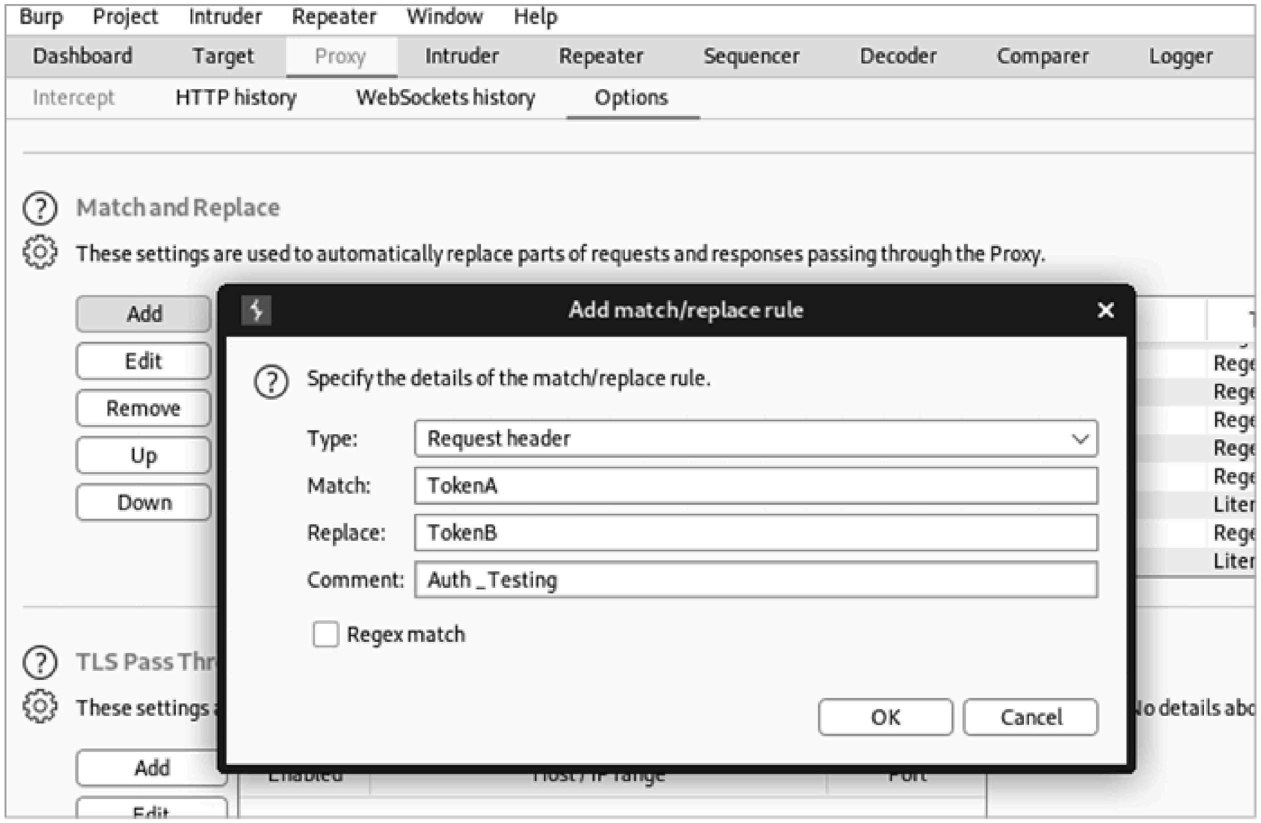
As you would when fuzzing wide, you can use Postman to perform variable changes across a collection, setting the authorization token for your collection as a variable. Begin by testing various requests for your resources to make sure they work properly as UserA. Then replace the token variable with the UserB token. To help you find anomalous responses, use a Collection test to locate 200 response codes or the equivalent for your API.

In Collection Runner, select only the requests that are likely to contain authorization vulnerabilities. Good candidate requests include those that contain private information belonging to UserA. Launch the Collection Runner and review the results. When checking results, look for instances in which the UserB token results in a successful response. These successful responses will likely indicate either BOLA or BFLA vulnerabilities and should be investigated further.

Burp Suite Match and Replace

When you're attacking an API, your Burp Suite history will populate with unique requests. Instead of sifting through each request and testing it for authorization vulnerabilities, use the Match and Replace option to perform a large-scale replacement of a variable like an authorization token.

Begin by collecting several requests in your history as UserA, focusing on actions that should require authorization. For instance, focus on requests that involve a user's account and resources. Next, match and replace the authorization headers with UserB's and repeat the requests (see *Figure 10-2*).


Burp Project Intruder Repeater Window Help
Dashboard Target Proxy Intruder Repeater Sequencer Decoder Comparer Logger
Intercept HTTP history WebSockets history Options
Match and Replace
These settings are used to automatically replace parts of requests and responses passing through the Proxy.
Add
Edit
Remove
Up
Down
TLS Pass Thr
These settings
Add
Edit

Add match/replace rule

Specify the details of the match/replace rule.

Type: Request header
Match: TokenA
Replace: TokenB
Comment: Auth _Testing
Regex match

OK Cancel

Figure 10-2: Burp Suite's Match and Replace feature

Once you find an instance of BOLA or BFLA, try to exploit it for all users and related resources.

Summary

In this chapter, we took a close look at techniques for attacking common weaknesses in API authorization. Since each API is unique, it's important not only to figure out how resources are identified but also to make requests for resources that don't belong to the account you're using.

Authorization can lead to some of the most severe consequences. A BOLA vulnerability could allow an attacker to compromise an organization's most sensitive information, whereas a BFLA vulnerability could allow you to escalate privileges or perform unauthorized actions that could compromise an API provider.

Lab #7: Finding Another User's Vehicle Location

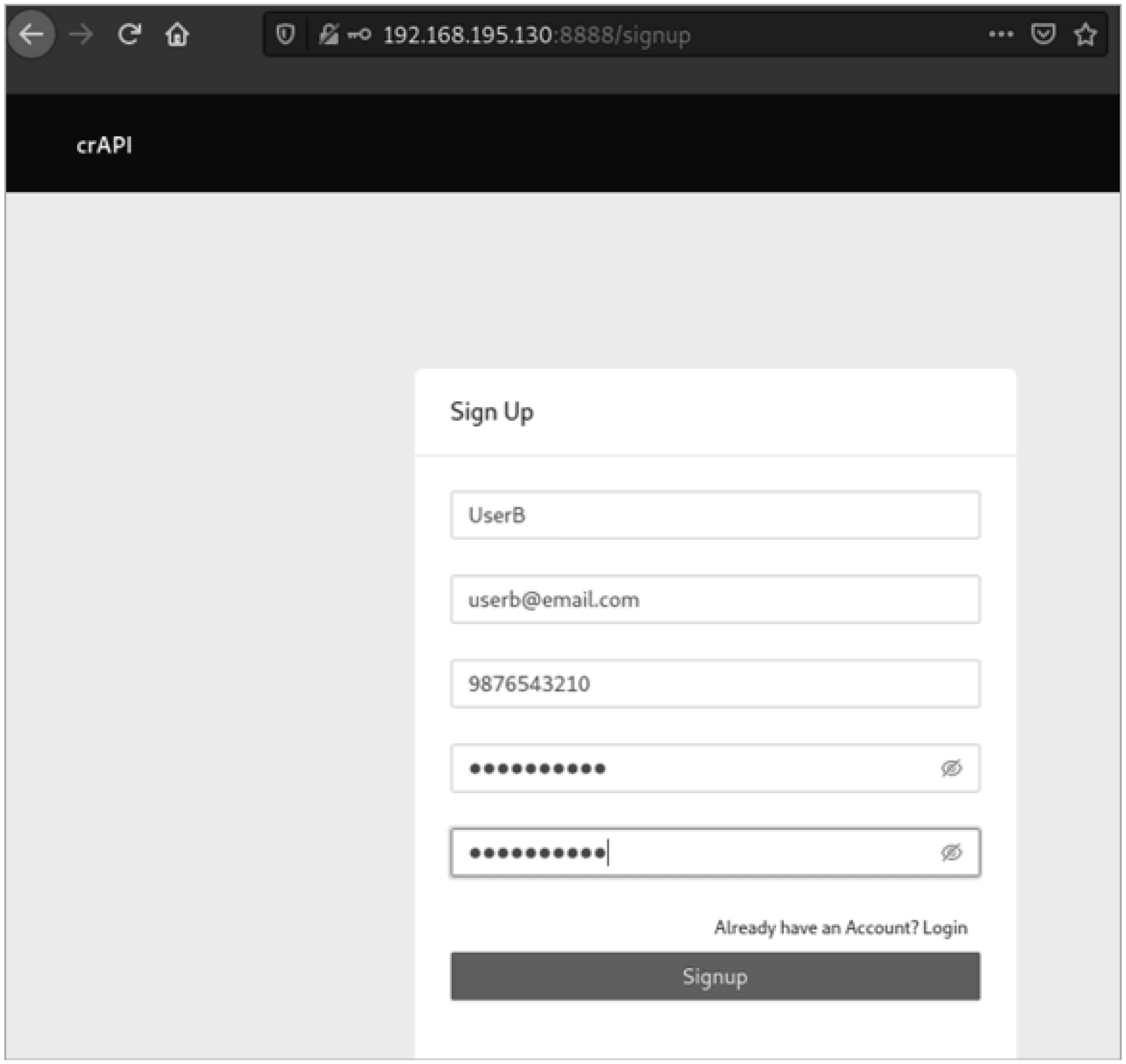
In this lab, we'll search crAPI to discover the resource identifiers in use and test whether we can gain unauthorized access to another user's data. In doing so, we'll see the value of combining multiple vulnerabilities to increase the impact of an attack. If you've followed along in the other labs, you should have a crAPI Postman collection containing all sorts of requests.

You may notice that the use of resource IDs is fairly light. However, one request does include a unique resource identifier. The "refresh location" button at the bottom of the crAPI dashboard issues the following request:

```
GET /identity/api/v2/vehicle/fd5a4781-5cb5-42e2-8524-d3e67f5cb3a6/location.
```

This request takes the user's GUID and requests the current location of the user's vehicle. The location of another user's vehicle sounds like sensitive information worth collecting. We should see if the crAPI developers depend on the complexity of the GUID for authorization or if there are technical controls making sure users can only check the GUID of their own vehicle.

So the question is, how should you perform this test? You might want to put your fuzzing skills from Chapter 9 to use, but an alphanumeric GUID of this length would take an impossible amount of time to brute-force. Instead, you can obtain another existing GUID and use it to perform A-B testing. To do this, you will need to register for a second account, as shown in *Figure 10-3*.


192.168.195.130:8888/signup

crAPI

Sign Up

UserB

userb@email.com

9876543210

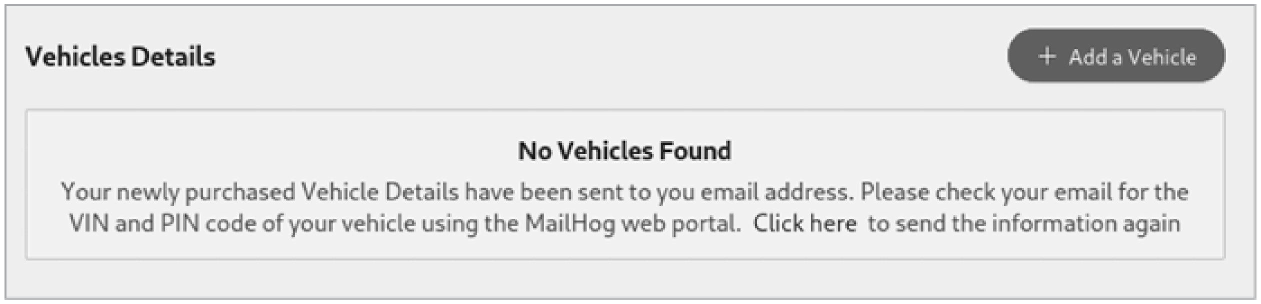
●●●●●●●●●●●

●●●●●●●●●●●

Figure 10-3: Registering UserB with crAPI

In *Figure 10-3*, you can see that we've created a second account, called UserB. With this account, go through the steps to register a vehicle using MailHog. As you may remember, back in the Chapter 6 lab we performed reconnaissance and discovered some other open ports associated with crAPI. One of these was port 8025, which is where MailHog is located.

As an authenticated user, click the **Click Here** link on the dashboard, as seen in Figure 10-4. This will generate an email with your vehicle's information and send it to your MailHog account.

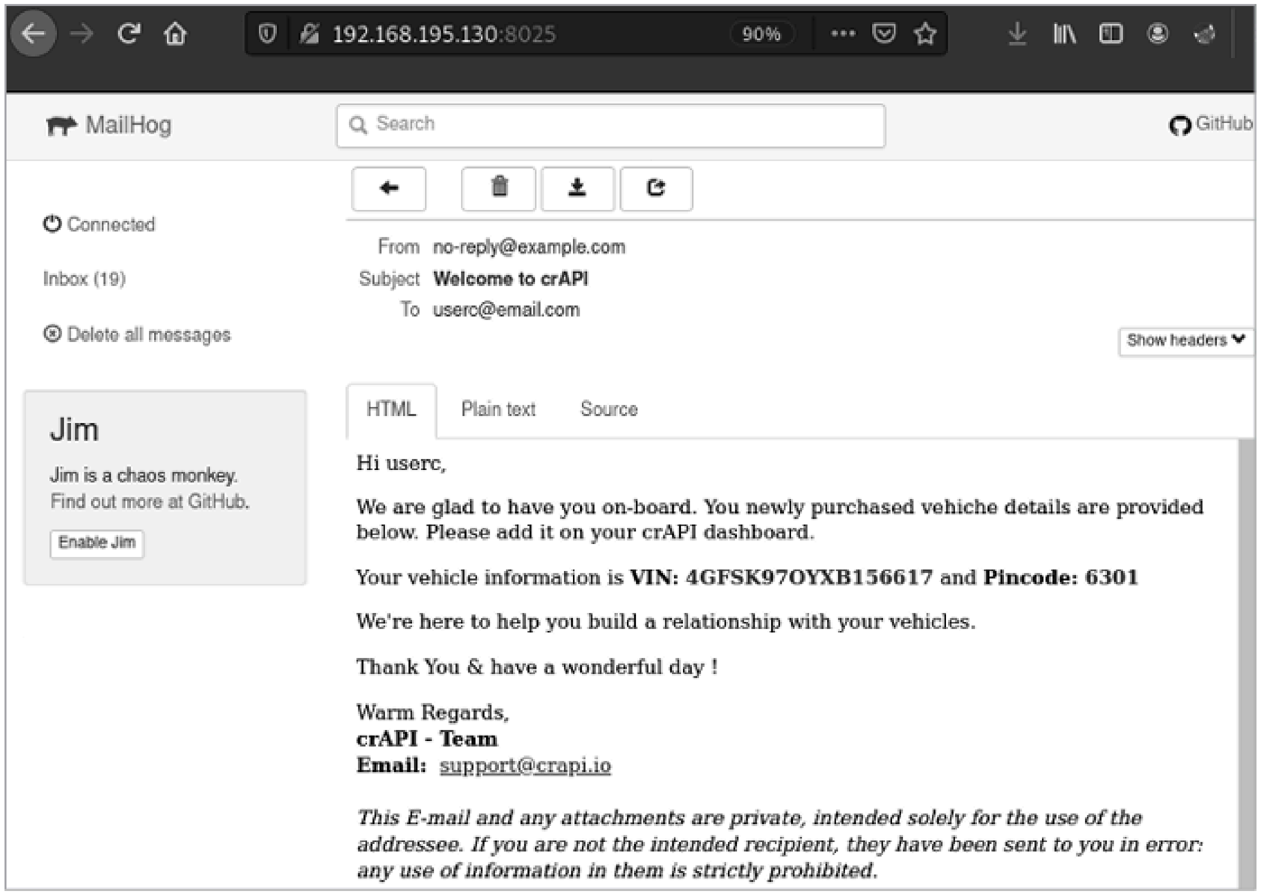

Vehicles Details

No Vehicles Found

Your newly purchased Vehicle Details have been sent to you email address. Please check your email for the VIN and PIN code of your vehicle using the MailHog web portal. Click here to send the information again

Figure 10-4: A crAPI new user dashboard

Update the URL in the address bar to visit port 8025 using the following format: *http://yourIPaddress:8025*. Once in MailHog, open the "Welcome to crAPI" email (see Figure 10-5).



From no-reply@example.com
Subject **Welcome to crAPI**
To userc@email.com

Hi userc,

We are glad to have you on-board. You newly purchased vehiche details are provided below. Please add it on your crAPI dashboard.

Your vehicle information is **VIN: 4GFSK97OYXB156617** and **Pincode: 6301**

We're here to help you build a relationship with your vehicles.

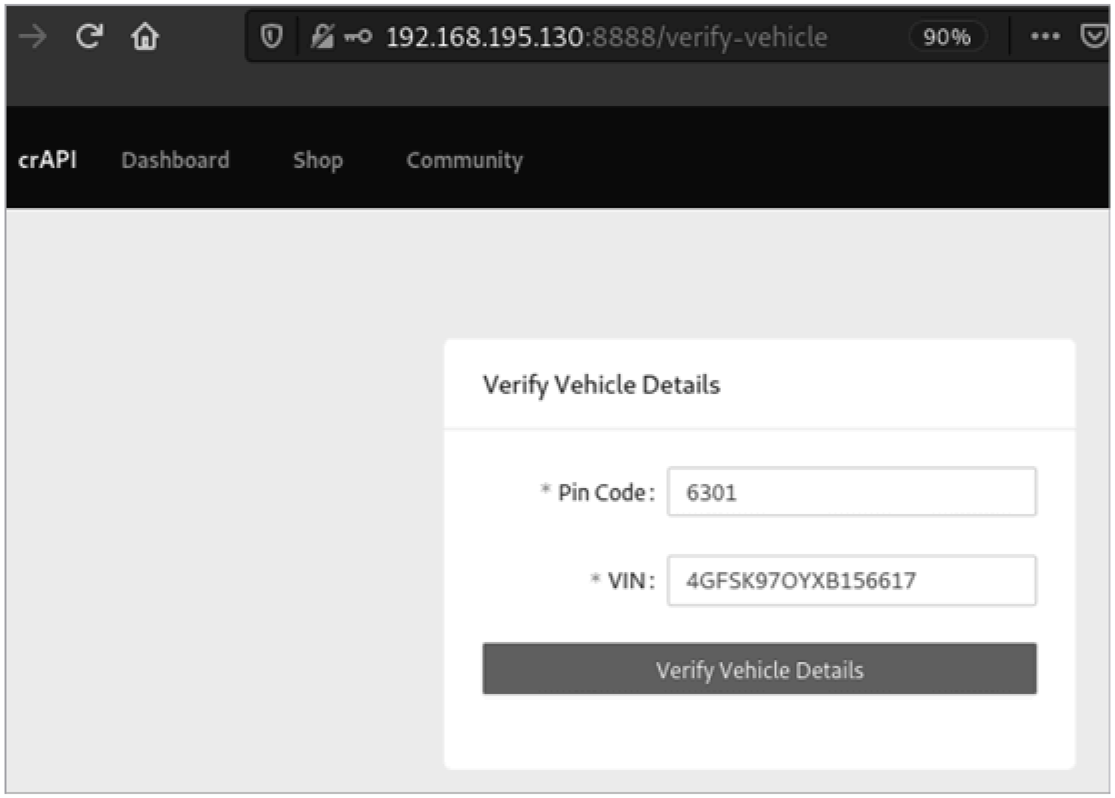
Thank You & have a wonderful day !

Warm Regards,
crAPI - Team
Email: support@crapi.io

This E-mail and any attachments are private, intended solely for the use of the addressee. If you are not the intended recipient, they have been sent to you in error: any use of information in them is strictly prohibited.

Figure 10-5: The crAPI MailHog email service

Take the VIN and pincode information provided in the email and use that to register your vehicle back on the crAPI dashboard by clicking the **Add a Vehicle** button. This results in the window shown in *Figure 10-6*.



* Pin Code: 6301

* VIN: 4GFSK97OYXB156617

Figure 10-6: The crAPI Vehicle Verification screen

Once you've registered the UserB vehicle, capture a request using the **Refresh Location** button. It should look like this:

```
GET /identity/api/v2/vehicle/d3b4b4b8-6df6-4134-8d32-1be402caf45c/location HTTP/1.1
Host: 192.168.195.130:8888
User-Agent: Mozilla/5.0 (X11; Linux x86_64; rv:78.0) Gecko/20100101 Firefox/78.0
Accept: */*
Content-Type: application/json
Authorization: Bearer UserB-Token
Content-Length: 376
```

Now that you have UserB's GUID, you can swap out the UserB Bearer token and send the request with UserA's bearer token. Listing 10-5 shows the request, and Listing 10-6 shows the response.

```
GET /identity/api/v2/vehicle/d3b4b4b8-6df6-4134-8d32-1be402caf45c/location HTTP/1.1
Host: 192.168.195.130:8888
Content-Type: application/json
Authorization: Bearer UserA-Token
```

Listing 10-5: A BOLA attempt

```
HTTP/1.1 200

{
"carId":"d3b4b4b8-6df6-4134-8d32-1be402caf45c",
"vehicleLocation":
     {
     "id":2,
     "latitude":"39.0247621",
     "longitude":"-77.1402267"
     },
"fullName":"UserB"
}
```

Listing 10-6: Response to the BOLA attempt

Congratulations, you've discovered a BOLA vulnerability. Perhaps there is a way to discover the GUIDs of other valid users to take this finding to the next level. Well, remember that, in Chapter 7, an intercepted GET request to */community/api/v2/community/posts/recent* resulted in an excessive data exposure. At first glance, this vulnerability did not seem to have severe consequences. However, we now have plenty of use for the exposed data. Take a look at the following object from that excessive data exposure:

```
{
"id":"sEcaWGHf5d63T2E7asChJc",
"title":"Title 1",
"content":"Hello world 1",
"author":{
"nickname":"Adam",
"email":"adam007@example.com",
"vehicleid":"2e88a86c-8b3b-4bd1-8117-85f3c8b52ed2",
"profile_pic_url":"",
}
```

This data reveals a `vehicleid` that closely resembles the GUID used in the Refresh Location request. Substitute these GUIDs using UserA's token. Listing 10-7 shows the request, and Listing 10-8 shows the response.

```
GET /identity/api/v2/vehicle/2e88a86c-8b3b-4bd1-8117-85f3c8b52ed2/location HTTP/1.1
Host: 192.168.195.130:8888
Content-Type: application/json
```

```
Authorization: Bearer UserA-Token
Connection: close
```

Listing 10-7: A request for another user's GUID

```
HTTP/1.1 200
{
"carId":"2e88a86c-8b3b-4bd1-8117-85f3c8b52ed2",
"vehicleLocation":{
     "id":7,
     "latitude":"37.233333",
     "longitude":"-115.808333"},
"fullName":"Adam"
}
```

Listing 10-8: The response

Sure enough, you can exploit the BOLA vulnerability to discover the location of the user's vehicle. Now you're one Google Maps search away from discovering the user's exact location and gaining the ability to track any user's vehicle location over time. Combining vulnerability findings, as you do in this lab, will make you a master API hacker.