API Pentesting

API Reconnaissance

Passive Reconnaissance

- Google Dorking
  inurl:"/wp-json/wp/v2/users"
  intitle:"index.of" intext:"api.txt"
  inurl:"/api/v1" intext:"index of /"
- Github Dorking
  filename:swagger.json
  extension: .json
  “api key,” "api keys", "apikey", "authorization: Bearer", "access_token", "secret", or “token.”
  search for API key exposed
- Shodan
  hostname:"targetname.com"
  "content-type: application/json"
  "content-type: application/xml"
  "wp-json"
- The Wayback Machine
  See changes to existing API documentation. If the API has not been managed well over time, then there is a chance that you could find retired endpoints that still exist even though the API provider believes them to be retired. These are known as Zombie APIs. Zombie APIs fall under the Improper Assets Management vulnerability on the OWASP API Security Top 10 list. Finding and comparing historical snapshots of API documentation can simplify testing for Improper Assets Management.
  https://www.reddit.com/dev/api/oauth

Active Reconnaissance

- nmap
  nmap -sC -sV [target]
  nmap -p- [target] all ports
  nmap -sV --script=http-enum [target] -p 80,443,8000,8080
- amass
  amass enum -active/passive -d [target] | grep api

- Directory brute-force with Gobuster
  gobuster dir -u target-name.com:8000 -w
  /home/hapihacker/api/wordlists/common_apis_160
- kiterunner
  best tool available for discovering API endpoints and resources.
  kr scan HTTP://127.0.0.1 -w ~/api/wordlists/data/kiterunner/routes-large.kite
- DevTools
  open DevTools with F12 or ctr-shift-I or Inspect. Adjust the DevTools window until you have enough space to work with. Select the Network tab and then refresh the page (CTRL+r). Allow URLs through right-click on file then copy it as cURL and use postman to look at detailed request elements.

End-point Analysis

Reverse Engineering an API

First, let's launch Postman.
Next, create a Workspace to save your collections in. For this course, we will use the ACE workspace. To build your own collection in Postman with the Proxy, use the Capture Requests button, found at the bottom right of the Postman window.
In the Capture requests window, select Enable proxy. The port should match with the number that is set up in FoxyProxy (5555). Next, enable the Postman Proxy, add your target URL to the "URL must contain" field, and click the Start Capture button. then go to your application to make the flow of requests on the application to make postman capture all requests.
Once you have captured all of the features you can find with manual exploration then you will want to Stop the Proxy. Next, it is time to build the crAPI collection. First, create a new collection by selecting the new button (top left side of Postman) and then choose Collection. Go ahead and rename the collection to **crAPI Proxy Collection**. Navigate back to the Proxy debug session and open up the Requests tab.

Automatic Documentation

First, we will begin by proxying all web application traffic using mitmweb.
Simply use a terminal and run: mitmweb
This will create a proxy listener using port 8080. You can then open a browser and use FoxyProxy to proxy your browser to port 8080 using our Burp Suite option. Every request that is created from your actions will be captured by the mitmweb proxy. see the captured traffic by using a browser to visit the mitmweb web server located at http://127.0.0.1:8081.
Continue to explore the target web application until there is nothing left to do. Once you have exhausted what can be done with your target web app, return to the mitmweb web server and click **File > Save** to save the captured requests.

Selecting **Save** will create a file called flows. We can use the "flows" file to create our own API documentation. Using a great tool called mitmproxy2swagger, we will be able to transform our captured traffic into an Open API 3.0 YAML file that can be viewed in a browser and imported as a collection into Postman.
First, run the following:
sudo mitmproxy2swagger -i /Downloads/flows -o spec.yml -p http://crapi.apisec.ai -f flow
After running this you will need to edit the spec.yml file to see if mitmproxy2swagger has ignored too many endpoints. Checking out spec.yml reveals that there are several endpoints that were ignored and the title of the file can be updated.
Save the updated spec.yml file and run the mitmproxy2swagger again. This time around add the "--examples" flag to enhance your API documentation.
After running mitmproxy2swagger successfully a second time through, your reverse-engineered documentation should be ready. You can validate the documentation
by visiting https://editor.swagger.io/ and by importing your spec file into the Swagger Editor.
Use **File>Import file** and select your spec.yml file. If everything has gone as planned then you should see something like the image below. This is a pretty good indication of success, but to be sure we can also import this file as a Postman Collection that way we can prepare to attack the target API.

Using APIs and Excessive Data Exposure

Using the Swagger Editor allows us to have a visual representation of our target's API endpoints. By browsing through and expanding the requests you can see the endpoint, parameters, request body, and example responses.

Editing Postman Collection Variables

You can get to the collection editor by using your crAPI Swagger collection, selecting the three circles on the right side of a collection, and choosing "Edit". Selecting the Variables tab "baseUrl" is used. Make sure that the baseUrl Current Value matches up with the URL to your target. If your target is localhost then it should match the image above. If your target is the ACE lab then the current value should be http://crapi.apisec.ai. Once you have updated a value in the editor, you will need to use the Save button found at the top right of Postman.

Updating Postman Collection Authorization

In order to use Postman to make authorized API requests, we will need to add a valid token to our requests. This can be done for all of the requests within a collection by adding an authorization method to the collection. Using the Authorization tab, within the collection editor, we will need to select the right type for authorization. For crAPI, this will be a Bearer Token. Tokens are usually provided after a successful authentication attempt. For crAPI, we will be able to obtain a Bearer Token once we successfully authenticate with the POST request to

/identity/api/auth/login. Copy the token found within the quotes and paste that value into the collection editor's authorization tab. Make sure to save the update to the collection. Now you should now be able to use the crAPI API as an authorized user and make successful requests using Postman.

Excessive Data Exposure

Excessive Data Exposure occurs when an API provider sends back a full data object, typically depending on the client to filter out the information that they need. From an attacker's perspective, the security issue here isn't that too much information is sent, instead, it is more about the sensitivity of the sent data. This vulnerability can be discovered as soon as you are able to start making requests. API requests of interest include user accounts, forum posts, social media posts, and information about groups (like company profiles).

- A response that includes more information than what was requested
- Sensitive Information that can be leveraged in more complex attacks

Scanning APIs

Finding Security Misconfigurations

- Scan with Nikto, a web application vulnerability scanner. Open a terminal and run:
  nikto -h http://crapi.apisec.ai
- Scan using zaproxy, Automated or manual scan. Open a terminal and run:
  sudo zaproxy
  then put the path of the web application to run the scanner.

API Authentication Attacks

Classic Authentication Attacks

Classic authentication attacks include techniques that have been around for a while like brute forcing and password spraying. Both of these attacks are different methods of guessing username and password combinations.

Password Brute-Force Attacks

you can use burp intruder to do brute-force attack on specific e-mail
or use wfuzz tool including rockyou passwords file.
gzip -d /usr/share/wordlists/rockyou.txt.gz
wfuzz -d '{"email":"a@email.com","password":"FUZZ"}' -H 'Content-Type: application/json' -z file,/usr/share/wordlists/rockyou.txt -u http://127.0.0.1:8888/identity/api/auth/login --hc 500

Password Spraying

Many security controls could prevent you from successfully brute-forcing an API's authentication. A technique called password spraying can evade many of these controls by combining a long list of users with a short list of targeted passwords. Let's say you know that an API authentication process has a lockout policy in place and will only allow 10 login attempts. You could craft a list of the nine most likely passwords (one less password than the limit) and use these to attempt to log in to many user accounts.
Most password policies likely require a minimum character length, upper- and lowercase letters, and perhaps a number or special character. Use passwords that are simple enough to guess but complex enough to meet basic password requirements (generally a minimum of eight characters, a symbol, upper- and lowercase letters, and a number). The first type includes obvious passwords like QWER!@#
, Password1!, and the formula Season+Year+Symbol (such as Winter2021!, Spring2021?, Fall2021!, and Autur
March212006! July152006! Twitter@2022 JPD1976! Dorsey@2022
The real key to password spraying is to maximize your user list. The more usernames you include, the higher your odds of compromising a user account with a bad password. Build a user list during your reconnaissance efforts or by discovering excessive data exposure vulnerabilities.

- to filter the accounts from file use this regex
  grep -oe "[a-zA-Z0-9._]+@[a-zA-Z]+.[a-zA-Z]+" response.json > output1.txt
- to remove duplicate use this
  cat output.txt | sort -u > output2.txt
- then go to burp suite intruder to start password spray attack

API Token Attacks

When implemented correctly, tokens can be an excellent tool that can be used to authenticate and authorize users. However, if anything goes wrong when generating, processing, or handling tokens, they can become our keys to the kingdom.
Next, you will need to right-click on the request and forward it over to Sequencer. In Sequencer, we will be able to have Burp Suite send thousands of requests to the provider and perform an analysis of the tokens received in response. This analysis could demonstrate that a weak token creation process is in use.
Navigate to the Sequencer tab and select the request that you forwarded. Here we can use the Live Capture to interact with the target and get live tokens back in a response to be analyzed.
To make this process work, you will need to define the custom location of the token within the response. Select the Configure button to the right of Custom Location. Highlight the token found within quotations and click OK.
Once the token has been defined then you can Start live capture. At this point, you can either

wait for the capture to process thousands of requests or use the Analyze now button to see results sooner.
Using Sequencer against crAPI shows that the tokens generated seem to have enough randomness and complexity to not be predictable. Just because your target sends you a seemingly complex token, does not mean that it is safe from token forgery. Sequencer is great at showing that some complex tokens are actually very predictable. If an API provider is generating tokens sequentially then even if the token were 20 plus characters long, it could be the case that many of the characters in the token do not actually change. Making it easy to predict and create our own valid tokens.

JWT Attacks

JSON Web Tokens (JWTs) are one of the most prevalent API token types because they operate across a wide variety of programming languages, including Python, Java, Node.js, and Ruby. These tokens are susceptible to all sorts of misconfiguration mistakes that can leave the tokens vulnerable to several additional attacks. These attacks could provide you with sensitive information, grant you basic unauthorized access, or even administrative access to an API. This module will guide you through a few attacks you can use to test and break poorly implemented JWTs.
JWTs consist of three parts, all of which are base64 encoded and separated by periods: the header, payload, and signature. JWT.io is a free web JWT debugger that you can use to check out these tokens. You can spot a JWT because they consist of three periods and begin with "ey". They begin with "ey" because that is what happens when you base64 encode a curly bracket followed by a quote, which is the way that a decoded JWT always begins.

Automating JWT attacks with JWT_Tool

A great command line tool that we can use for analyzing and attacking JWTs.
$jwt_tool

- -h to show more verbose help options
- -t to specify the target URL
- -M to specify the scan mode
- pb to perform a playbook audit (default tests)
- at to perform all tests
- -rc to add request cookies
- -rh to add request headers
- -rc to add request cookies
- -pd to add POST data

To perform a baseline analysis of a JWT simply use jwt_tool along with your captured JWT to see information similar to the JWT Debugger.
$ jwt_tool [token]

Additionally, jwt_tool has a “Playbook Scan” that can be used to target a web application and scan for common JWT vulnerabilities. You can run this scan by using the following:

```
$ jwt_tool -t http://target-name.com/ -rh "Authorization: Bearer JWT_Token" -M pb
```

The None Attack

If you ever come across a JWT using "none" as its algorithm, you’ve found an easy win. After decoding the token, you should be able to clearly see the header, payload, and signature. From here, you can alter the information contained in the payload to be whatever you’d like. For example, you could change the username to something likely used by the provider’s admin account (like root, admin, administrator, test, or adm), as shown here: { "username": "root", "iat": 1516239022 } Once you’ve edited the payload, use Burp Suite’s Decoder to encode the payload with base64; then insert it into the JWT. Importantly, since the algorithm is set to "none", any signature that was present can be removed. In other words, you can remove everything following the third period in the JWT. Send the JWT to the provider in a request and check whether you’ve gained unauthorized access to the API.

The Algorithm Switch Attack

There is a chance the API provider isn’t checking the JWTs properly. If this is the case, we may be able to trick a provider into accepting a JWT with an altered algorithm. One of the first things you should attempt is sending a JWT without including the signature. This can be done by erasing the signature altogether and leaving the last period in place, like this: eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJpc3MiOiJoYWNrYXBpcy5pbyIsImV4cCI6IDE1ODM2Mzc0ODgsInVzZ XJuYW1lIjoiU2N1dHRsZXBoMXNoIiwic3VwZXJhZG1pbiI6dHJ1ZX0. If this isn’t successful, attempt to alter the algorithm header field to "none". Decode the JWT, update the "alg" value to "none", base64-encode the header, and send it to the provider. To simplify this process, you can also use the jwt_tool to quickly create a token with the algorithm switched to none.
$ jwt_tool
eyJhbGciOiJIUzUxMiJ9.eyJzdWIiOiJ1c2VyYWFhQGVtYWlsLmNvbSIsImlhdCI6MTY1ODg1NTc0MCwiZXhwIjoxNjU4OTQyMTQwfQ._EcnSozcUnL5y9SFOgOVBMabx_UAr6Kg0Zym-LH_zyjReHrxU_ASrrR6OysLa6k7wpoBxN9vauhkYNHepOcrlA -X a

JWT Crack Attack

The JWT Crack attack attempts to crack the secret used for the JWT signature hash, giving us full control over the process of creating our own valid JWTs. Hash-cracking attacks like this take

place offline and do not interact with the provider. Therefore, we do not need to worry about causing havoc by sending millions of requests to an API provider. You can use JWT_Tool or a tool like Hashcat to crack JWT secrets. You'll feed your hash cracker a list of words. The hash cracker will then hash those words and compare the values to the original hashed signature to determine if one of those words was used as the hash secret. If you're performing a long-term brute-force attack of every character possibility, you may want to use the dedicated GPUs that power Hashcat instead of JWT_Tool. That being said, JWT_Tool can still test 12 million passwords in under a minute. First, let's use Crunch, a password-generating tool, to create a list of all possible character combinations.
$crunch 5 5 -o brute_force.txt

We can use this password file that contains all possible character combinations created for 5 character passwords. To perform a JWT Crack attack using JWT_Tool, use the following command: $ jwt_tool TOKEN -C -d brute_force.txt
The -C option indicates that you'll be conducting a hash crack attack, and the -d option specifies the dictionary or wordlist you'll be using against the hash. JWT_Tool will either return "CORRECT key!" for each value in the dictionary or indicate an unsuccessful attempt with "key not found in dictionary."

Now that we have the correct secret key, we should be able can generate our own trusted tokens. To test out our new abilities, you can either create a second user account or use an email that you have already discovered. I have created an account named "superadmin".

hit to JWT.io, put your discovered secret and make your trusted token.

Exploiting API Authorization

Broken Object Level Authorization

When authorization controls are lacking or missing, UserA will be able to request UserB's (along with many other) resources. APIs use values, such as names or numbers, to identify various objects. When we discover these object IDs, we should test to see if we can interact with the resources of other users when unauthenticated or authenticated as a different user. The first step toward exploiting BOLA is to seek out the requests that are the most likely candidates for authorization weaknesses.

When hunting for BOLA there are three ingredients needed for successful exploitation.

1. Resource ID: a resource identifier will be the value used to specify a unique resource. This could be as simple as a number, but will often be more complicated.
2. Requests that access resources. In order to test if you can access another user's resource, you will need to know the requests that are necessary to obtain resources that your

account should not be authorized to access.
3. Missing or flawed access controls. In order to exploit this weakness, the API provider must not have access controls in place. This may seem obvious, but just because resource IDs are predictable, does not mean there is an authorization vulnerability present.

Finding Resource IDs and Requests

You can test for authorization weaknesses by understanding how an API's resources are structured and then attempting to access resources you shouldn't be able to access. By detecting patterns within API paths and parameters, you might be able to predict other potential resources. The bold resource IDs in the following API requests should catch your attention:

- GET /api/resource/**1**
- GET /user/account/find?**user_id=15**
- POST /company/account/**Apple**/balance
- POST /admin/pwreset/account/**90**

In these instances, you can probably guess other potential resources, like the following, by altering the bold values:

- GET /api/resource/**3**
- GET /user/account/find?user_id=**23**
- POST /company/account/**Google**/balance
- POST /admin/pwreset/account/**111**

Searching for BOLA

- Parts of the app that are specific to a user's account include
  - The user dashboard with the user's vehicle added
  - The user's profile
  - The user's past orders in the shop
  - The user's posts in the community forum

Authorization Testing Strategy

1. Create a UserA account.
2. Use the API and discover requests that involve resource IDs as UserA.
3. Document requests that include resource IDs and should require authorization.
4. Create a UserB account.
5. Obtaining a valid UserB token and attempt to access UserA's resources.

Broken Function Level Authorization

Where BOLA is all about accessing resources that do not belong to you, BFLA is all about performing unauthorized actions. BFLA vulnerabilities are common for requests that perform actions of other users. These requests could be lateral actions or escalated actions. Lateral actions are requests that perform actions of users that are the same role or privilege level. Escalated actions are requests that perform actions that are of an escalated role like an administrator. The main difference between hunting for BFLA is that you are looking for functional requests. This means that you will be testing for various HTTP methods, seeking out actions of other users that you should not be able to perform.

If you think of this in terms of a social media platform, an API consumer should be able to delete their own profile picture, but they should not be able to delete other users' profile pictures. The average user should be able to create or delete their own account, but they likely shouldn't be able to perform administrative actions for other user accounts. For BFLA we will be hunting for very similar requests to BOLA.

See which requests are worth testing for BFLA. The first three requests I found in our collection were these:

- **POST /workshop/api/shop/orders/return_order?order_id=5893280.0688146055**
- **POST /community/api/v2/community/posts/w4ErxCddX4TcKXbJoBbRMf/comment**
- **PUT /identity/api/v2/user/videos/:id**

Please take note: When successful, BFLA attacks can alter the data of other users. This means that accounts and documents that are important to the organization you are testing could be on the line. DO NOT brute force BFLA attacks, instead, use your secondary account to safely attack your own resources. Deleting other users' resources in a production environment will likely be a violation of most rules of engagement for bug bounty programs and penetration tests.

Try to UPDATE resources data by using [id] parameter and check the response, try to DELETE resources by using [id] parameter, if it needs admin privilege, try to replace /user to /admin.

Improper Assets Management

In the Analyzing API Endpoints module, we created a Postman collection for crAPI. In this module, we will use this collection to test for Improper Assets Management.

Testing for Improper Assets Management is all about discovering unsupported and non-production versions of an API. Often times an API provider will update services and the newer version of the API will be available over a new path like the following:

- api.target.com/v3
- /api/v2/accounts
- /api/v3/accounts
- /v2/accounts

API versioning could also be maintained as a header:

- *Accept: version=2.0*
- *Accept api-version=3*

Improper Assets Management Testing

1. Understand the baseline versioning information of the API you are testing. Make sure to check out the path, parameters, and headers for any versioning information.Understand the baseline versioning information of the API you are testing. Make sure to check out the path, parameters, and headers for any versioning information.
2. 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 200 is returned so that anything that does not result in a 200 Success response may stick out as anomalous. You can use the following test:
   `pm.test("Status code is 200", function () { pm.response.to.have.status(200); })`
3. Run an unauthenticated baseline scan of the crAPI collection with the Collection Runner. Make sure that "Save Responses" is checked as seen below.
4. Review the results from your unauthenticated baseline scan to have an idea of how the API provider responds to requests using supported production versioning.
5. Next, use "Find and Replace" to turn the collection's current versions into a variable. Make sure to do this for all versions, in the case of crAPI that means **v2** and **v3**. Type the current version into "Find", update "Where" to the targeted collection, and update "Replace With" to a variable.
6. Open Postman and navigate to the environmental variables (use the eye icon located at the top right of Postman as a shortcut). *Note, we are using environmental variables so that this test can be accessed and reused for other API collections.* Add a variable named "ver" to your Postman environment and set the initial value to "v1". Now you can update to test for various versioning-related paths such as v1, v2, v3, mobile, internal, test, and uat. As you come across different API versions expand this list of variables.
7. Now that the environmental variable is set to **v1** use the collection runner again and investigate the results. You can drill down into any of the requests by clicking on them. The "check-otp" request was getting a 500 response before and now it is 404. It is worth noting

the difference, but when a resource does not exist, then this would actually be expected behaviour.

8. 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. Update the environmental variable to v2. Although most of the requests were already set to v2, it is worth testing because check-otp was previously set to v3.
   Once again, run the collection runner with the new value set and review the results.
   The **/v2** request for **check-otp** is now receiving the same response as the original baseline request (to /v3). Since the request for **/v1** received a *404 Not Found*, this response is really interesting. Since the request to /v2 is not a 404 and instead mirrors the response to /v3, this is a good indication that we have discovered an Improper Assets Management vulnerability. This is an interesting finding, but what is the full impact of this vulnerability?
9. 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 too many requests to **/v3** will result in a different 500 response.
   Sending the same request to /v2 also results in an HTTP 500 error, but the response is slightly larger. It may be worth viewing the two responses back in Burp Suite Comparer to see the spot differences. Notice how the response on the left has the message that indicates we guessed wrong but can try again. The request on the right indicates a new status that comes up after too many attempts have been made.
   The /v2 password reset request responds with the body (left):
   {"message":"Invalid OTP! Please try again..","status":500}

The /v3 password reset request responds with the body (right):
{"message":"ERROR..","status":500}

10. To test this it is recommended that you use WFuzz, since Burp Suite CE will be throttled. First, make sure to issue a password reset request to your target email address. On the crAPI landing page select "Forgot Password?". Then enter a valid target email address and click "Send OTP".
11. Now an OTP is issued and we should be able to brute force the code using WFuzz. By brute forcing this request, you should see the successful code that was used to change the target's password to whatever you would like. In the attack below, I update the password to "NewPassword1". Once you receive a successful response, you should be able to login with the target's email address and the password that you choose.

```
$ wfuzz -d '{"email":"hapihacker@email.com", "otp":"FUZZ","password":"NewPassword1"}' -H 'Content-Type: application/json' -z file,/usr/share/wordlists/SecLists-master/Fuzzing/4-digits-0000-9999.txt -u http://crapi.apisec.ai/identity/api/auth/v2/check-otp --hc 500
```

Within 10,000 requests, you'll receive a 200 response indicating your victory. Congrats, on

taking this Improper Assets Management vulnerability to the next level! Since we got sidetracked with this interesting finding during unauthenticated testing.

Mass Assignment Attacks

Mass Assignment vulnerabilities are present when an attacker is able to overwrite object properties that they should not be able to. A few things need to be in play for this to happen. An API must have requests that accept user input, these requests must be able to alter values not available to the user, and the API must be missing security controls that would otherwise prevent the user input from altering data objects. The classic example of a mass assignment is when an attacker is able to add parameters to the user registration process that escalate their account from a basic user to an administrator. The user registration request may contain key-values for username, email address, and password. An attacker could intercept this request and add parameters like "isadmin": "true". If the data object has a corresponding value and the API provider does not sanitize the attacker's input then there is a chance that the attacker could register their own admin account.

Testing Account Registration for Mass Assignment

Test the registration process for mass assignment. The simplest form of this attack is to upgrade an account to an administrator role by adding a variable that the API provider likely uses to identify admins. If you have access to admin documentation then there is a good chance that the parameters will be included in the registration requests. You can then use the discovered parameters to see if the API has any security controls preventing you from escalating a user account to an admin account. If you do not have admin docus, then you can do a simple test by including other key-values to the JSON POST body, such as:
"isadmin": true,
"isadmin":"true",
"admin": 1,
"admin": true,
Any of these may cause the API to respond in a unique way indicating success or failure.

Once you attempt to a mass assignment attack on your target, you will need to analyze how the API responds. In the case of crAPI, there is no unique response when additional parameters are added to the request. There are no indications that the user account was changed in any way.

Fuzzing for Mass Assignment with Param Miner

Right-click on a request that you would like to mine for parameters. Select Extensions > Param Miner > Guess params > Guess JSON parameter. Feel free to experiment with the other options!

Set the Param Miner options that you would like and click OK when you are done.
Navigate back to Extender-Extensions and select Param Miner. Next, select the Output tab and wait for results to populate this area.
If any new parameters are detected, insert them back into the original request and fuzz for results.

Hunting for Mass Assignment

As with many other API attacks, we will start hunting for this vulnerability by analyzing the target API collection. Remember, mass assignment is all about binding user input to data objects. So, when you analyze a collection that you are targeting you will need to find requests that:

- Accept user input
- Have the potential to modify objects

if you could add a product try to test for negative values, for example, try to POST a product with -1000 funds, then go to buy it and watch out your funds increased with 1000!

Checking the request out reveals the full catalog of store products along with the product id, name, price, and image URL. If we could submit user data to products there would be a great opportunity to leverage a mass assignment attack. If we were able to submit data here we would be able to create our own products with our own prices. However, this request uses the GET method and is only for requesting data not altering it. Well, how do the crAPI administrators manage the products page? Perhaps they use PUT or POST to submit products to this endpoint and it wouldn't be the first time that we have discovered a BFLA vulnerability with this target. Always try to leverage vulnerability findings in other requests when testing a target organization. Chances are if the secure development practices of an organization fall short in one aspect of the application, they likely fall short in other areas.

Evasion and Combining Techniques

Many APIs in the wild will not be exploited as easily as the deliberately vulnerable applications that we've been up against in this course. Often times security controls like web applications firewalls (WAFs) and rate-limiting can block your attacks. Security controls may differ from one API provider to the next, but at a high level, they will have some threshold for malicious activity that will trigger a response.

String Terminators

Null bytes and other combinations of symbols are often interpreted as string terminators used to end a string. If these symbols are not filtered out, they could terminate the API security control filters that may be in place. When you are able to successfully send a null byte it is interpreted by many back-end programming languages as a signifier to stop processing. If the null byte is

processed by a back-end program that validates user input, then the security control could be bypassed because it stops processing the following input. Here is a list of potential string terminators you can use:

%00
0x00
//
;
%
!
?
[]
%5B%5D
%09
%0a
%0b
%0c
%0e

String terminators can be placed in different parts of the request, like the path or POST body, to attempt to bypass any restrictions in place. For example, in the following injection attack to the user profile endpoint, the null bytes entered into the payload could bypass filtering

POST /api/v1/user/profile/update
[…]
{
“uname”: “hapihacker”
“pass”: "%00'OR 1=1"
}

Case Switching

Case switching is the literal switching of the case of the letters within the URL path or payload. For example, take the following POST requests. Let’s say you were targeting a social media site by attempting an IDOR attack against a uid parameter in the following POST request:
POST /api/myprofile

URL path by switching upper- and lower-case letters in the path:

POST /api/myProfile
POST /api/MyProfile
POST /aPi/MypRoFiLe

Encoding Payloads

To take your WAF-bypassing attempts to the next level, try encoding payloads. Encoded payloads can often trick WAFs while still being processed by the target application or database. Even if the WAF or an input-validation measure blocks certain characters or strings, they might miss encoded versions of those characters. Alternatively, you could also try double encoding your attacks.

URL Encoded Payload: %27%20%4f%52%20%31%3d%31%3b
API Provider URL Decoder: ' OR 1=1;
WAF Rules detect a fairly obvious SQL Injection attack and block the payload.
Double URL Encoded Payload:
%25%32%37%25%32%30%25%34%66%25%35%32%25%32%30%25%33%31%25%33%64%25%33%31%25%33%62

Suggestions for Combining Vulnerabilities and Techniques

- Excessive data exposure is a vulnerability that can be leveraged in many other attacks. Excessive Data Exposure can be a great source of information about users and administrators. Make sure that you take note of all parameters leaked in exposure, such as username, email, privilege level, and unique user identifiers. All of this information could be challenging to come across otherwise but can be leveraged in authentication and authorization attacks.
- Improper Assets Management vulnerability is a weakness that should be combined with all other attack techniques. Improper Assets Management implies that a version of the API may not be as supported as the current version and thus it may not be as protected. Perform all of the other attacks against the unmanaged endpoint. You may find that the security of a api/test/login, api/mobile/login, api/v1/login, api/v1/internal/users may have fewer protections than the supported counterparts. With Improper Assets Management findings there could be lax rate-limiting, missing protections for brute force attempts, missing input validation, unlimited multifactor authentication requests, authorization vulnerabilities, etc. With APIs affected by Improper Assets Management vulnerabilities you never know what you’re going to get. So, make sure to perform the full gambit of attack techniques against unsupported versions of the API.
- If you are testing a file upload function, attempt to use the directory traversal fuzzing attack along with the file upload to manipulate where the file is stored. Upload files that lead to a shell and attempt to execute the uploaded file with the corresponding web application.
- Perhaps you have discovered a Broken Function Level Authorization (BFLA) or Mass Assignment weakness that has given you access to administrative functionality. There is a chance that an API provider trusts its administrators and does not have as many technical protections in place. You could then use this new level of access to search for Improper

Assets Management, injection weaknesses, or Broken Object Level Authorization weaknesses.

- If you are able to perform a mass assignment, then there is a chance that the parameter or parameters that you can alter may not be protected from injection attacks and/or Server-side Request Forgery. When you have discovered mass assignment fuzz the parameter for other weaknesses.
- If you are able to update to perform a BFLA attack and update another user's profile information perhaps you could combine this with an XSS attack.
- Combine Injection attacks with most other vulnerabilities. For example, if you are able to manipulate a JSON Web Token. Consider including various different types of injection attacks into the JSON Web Token.
- Consider ways to fully leverage BOLA findings. Use the information that you are able to access to gain access elsewhere. Always test to see if your BOLA discovery can be used in a BFLA attack. In other words, if you find a BOLA vulnerability that provides you with other users' bank account information check to see if you are also able to exploit a BFLA vulnerability to transfer funds.
- If you are up against API security controls like a WAF or rate-limiting, make sure to apply evasion techniques to your attacks. Encode your attacks and attempt to find ways to bypass these security measures. If you have exhausted your testing efforts, return to fuzzing wide with encoded and obfuscated attacks.