

1. If POST request is actioned on a web server 10 times then there would be 10 new resources on the web server.
2. If PUT request is actioned on the web server 10 times it will mean that it changes the same resource 10 times without adding more resources.

## Cookies and cache

1. STATEFUL CONNECTION : Client information is stored via small text files called cookies. This will keep track of the users information on the website for example the items that the user has added to their card.

When we talk about cookies being stored on the client-side, we're referring to the fact that they're stored on the user's web browser (e.g., Google Chrome, Mozilla Firefox). However, the information in those cookies can be accessed and utilized by the server-side as well.

1. SERVER stores the information via caching the information.

Step by step process:

Server-side caching of cookie information:

> Cookie creation: When a user visits a website, the server sets a cookie on the user's browser using the Set-Cookie header.

> Cookie storage: The cookie is stored on the user's browser.

> Subsequent requests: On subsequent requests to the same website, the browser sends the cookie back to the server in the Cookie header.

> Server-side processing: The server can then access and process the information in the cookie.

> Caching: The server can cache the processed information in various ways, such as:

° Memory caching: Storing the information in the server's memory (e.g., RAM) for faster access.

° Database caching: Storing the information in a database, which can be accessed quickly.

° Cache servers: Using dedicated cache servers, like Redis or Memcached, to store and retrieve the information.

## Stateless protocol

not information is cached , each request that is sent out to the server should include every bit of information that is required by the server to process the request including authentication ( which can be performed by adding key to the request).

In a stateless HTTP protocol, where no information is cookied or cached, the shopping cart can be maintained from an end-user perspective using the following methods:

1. URL Rewriting:

The server appends the cart information to the URL as query parameters (e.g., ?cart=item1,item2,item3).

The user's browser stores the URL, allowing the cart to be maintained across requests.

2. Hidden Form Fields:

The server generates a hidden form field containing the cart information (e.g., <input type="hidden" name="cart" value="item1,item2,item3">).

The user's browser stores the form field values, allowing the cart to be maintained across requests.

3. Query Parameters:

The server appends the cart information as query parameters in the URL (e.g., ?cart=item1,item2,item3).

The user's browser stores the URL, allowing the cart to be maintained across requests.

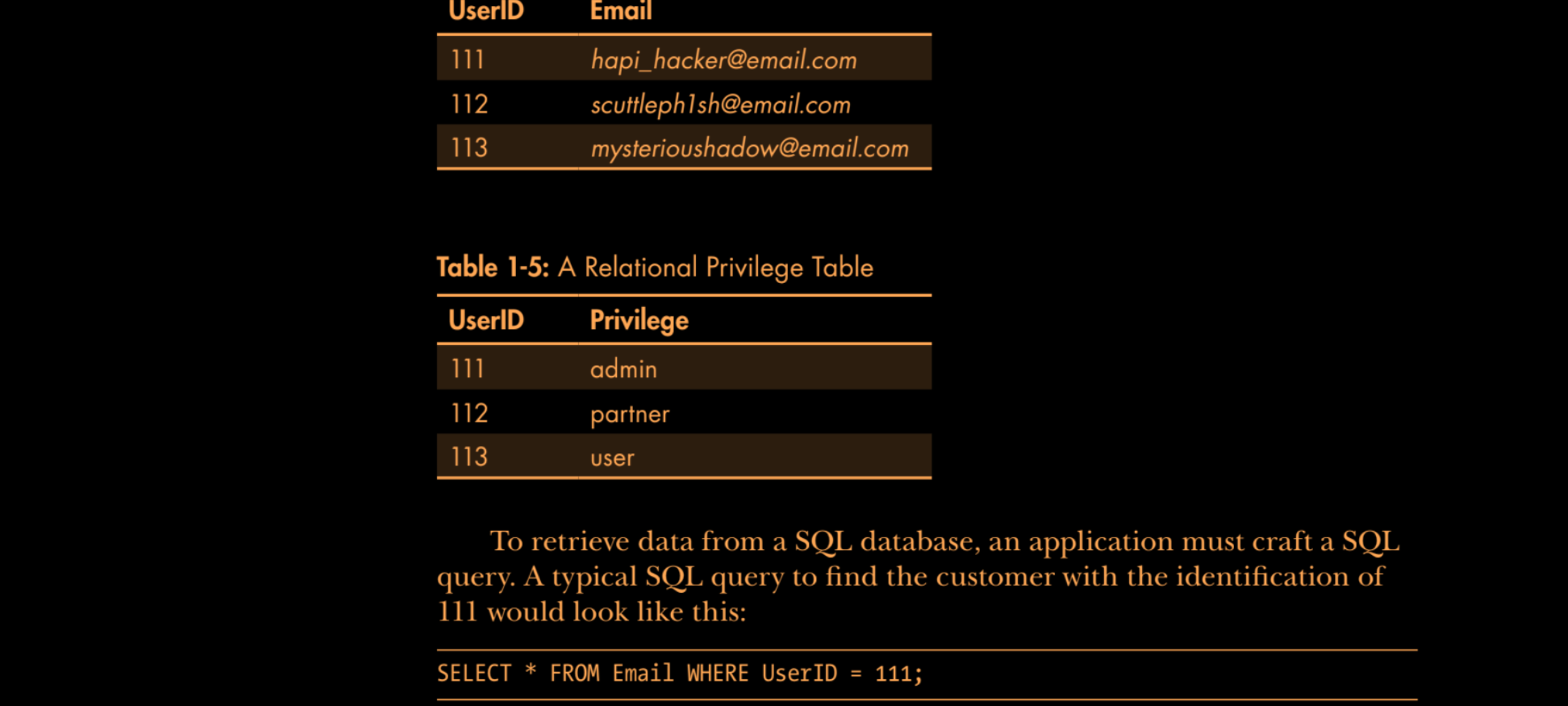
4. Client-side Storage:

The server provides a JavaScript library that stores the cart information in the user's browser using:

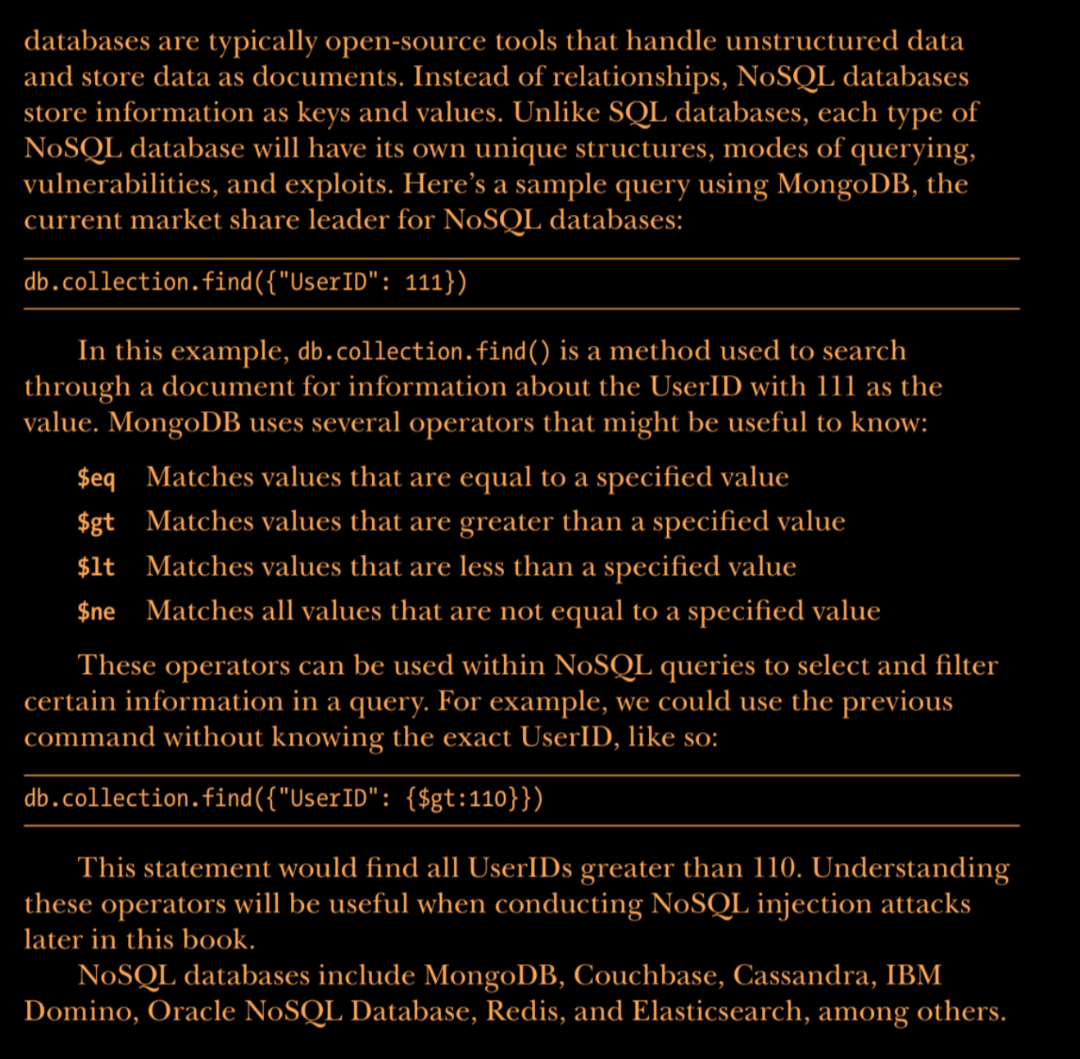
Local Storage (e.g., localStorage.setItem('cart', 'item1,item2,item3'))

Session Storage (e.g.,sessionStorage.setItem('cart','item1,item2,item3'))

1. SQL



1. NO SQL



## **Ride-Sharing App Integration Summary**

A ride-sharing app can leverage existing APIs to enhance its functionality without specializing in each field.

Here's how:

Google Maps API

Routing and Navigation: Utilize Google Maps for accurate route calculation, traffic updates, and turn-by-turn directions.

Geocoding and Places: Integrate Google Maps to convert addresses to coordinates and provide location suggestions.

Twilio API

Communication: Use Twilio for seamless communication between drivers and passengers, including:

SMS and voice notifications

In-app messaging

Phone number masking for privacy

Stride API (for identity verification)

Identity Verification: Integrate Stride API to verify user identities, ensuring a secure and trustworthy platform.

1. CRUD

CREATE = Resource is created using POST

READ= Using GET request

UPDATE= Using PUT request

DELETE=Using wither PUT OR DELETE

## **AUTHORISATION HEADERS**

Authorization headers are used to pass a token or credentials to the API provider. The format of these headers is

**Authorization: <type> <token/credentials>.**

For example, take a look at the following authorization header: Authorization: Bearer Ab4dtok3n There are different authorization types. Basic uses base64-encoded credentials. Bearer uses an API token. Finally, AWS-HMAC-SHA256 is an AWS authorization type that uses an access key and a secret key. Content Type Content-Type headers are used to indicate

## **X HEADERS**

Middleware (X) Headers X-<anything> headers are known as middleware headers and can serve all sorts of purposes. They are fairly common outside of API requests as well.

X-Response-Time can be used as an API response to indicate how long a response took to process. X-API-Key can be used as an authorization header for API keys. X-Powered-By can be used to provide additional information about backend services. X-Rate-Limit can be used to tell the consumer how many requests they can make within a given time frame.

X-RateLimit-Remaining can tell a consumer how many requests remain before they violate rate-limit enforcement. (There are many more, but you get the idea.) X-<anything> middleware headers can provide a lot of useful

## **HTML FORMS {Form Data)**

It's the forms on the HTML websites where the users are able to submit the forms to provide data to the web servers.

1. **When the string “hAPI hacker” is encoded in UTF-8,** it becomes the following: \x68\x41\x50\x49\x20\x68\x61\x63\x6B\x65\x72 Here is the UTF-16 version of the string: \u{68}\u{41}\u{50}\u{49}\u{20}\u{68}\u{61}\u{63}\u{6b}\u{65}\u{72} Finally, here is the base64-encoded version: aEFQSSBoYWNrZXI= Recognizing these encoding

# **GraphQL API**

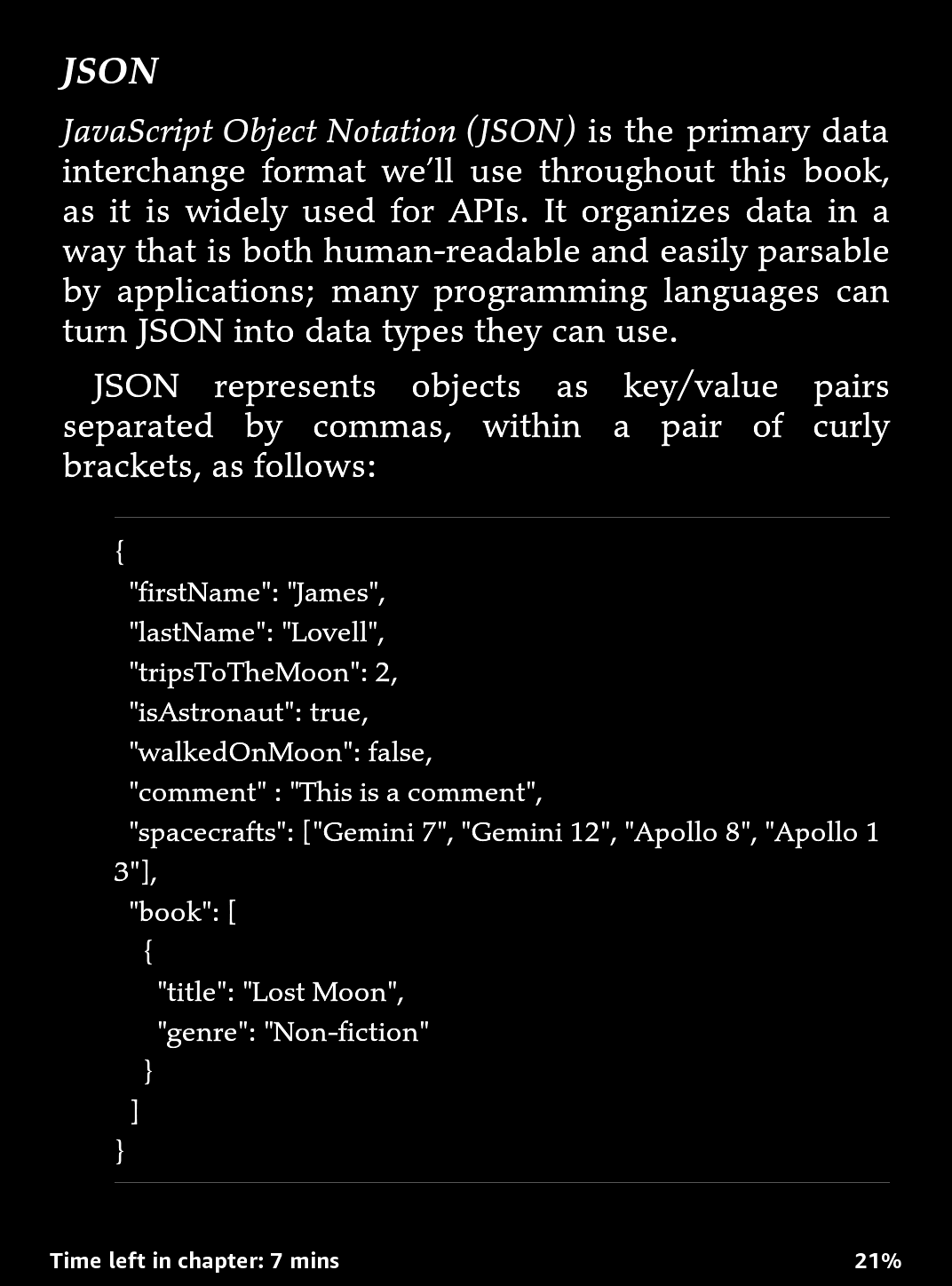
To access a **GraphQL API**, you’ll typically access the URL where it is hosted and submit an authorized request that contains query parameters as the body of a POST request, similar to the following: query { users { username id email } } In the right context, this query would provide you with the usernames, IDs, and emails of the requested resources. A GraphQL response to this query would look like the following: { "data": { "users": { "username": "hapi\_hacker", "id": 1111, "email": "hapihacker@email.com" } } }

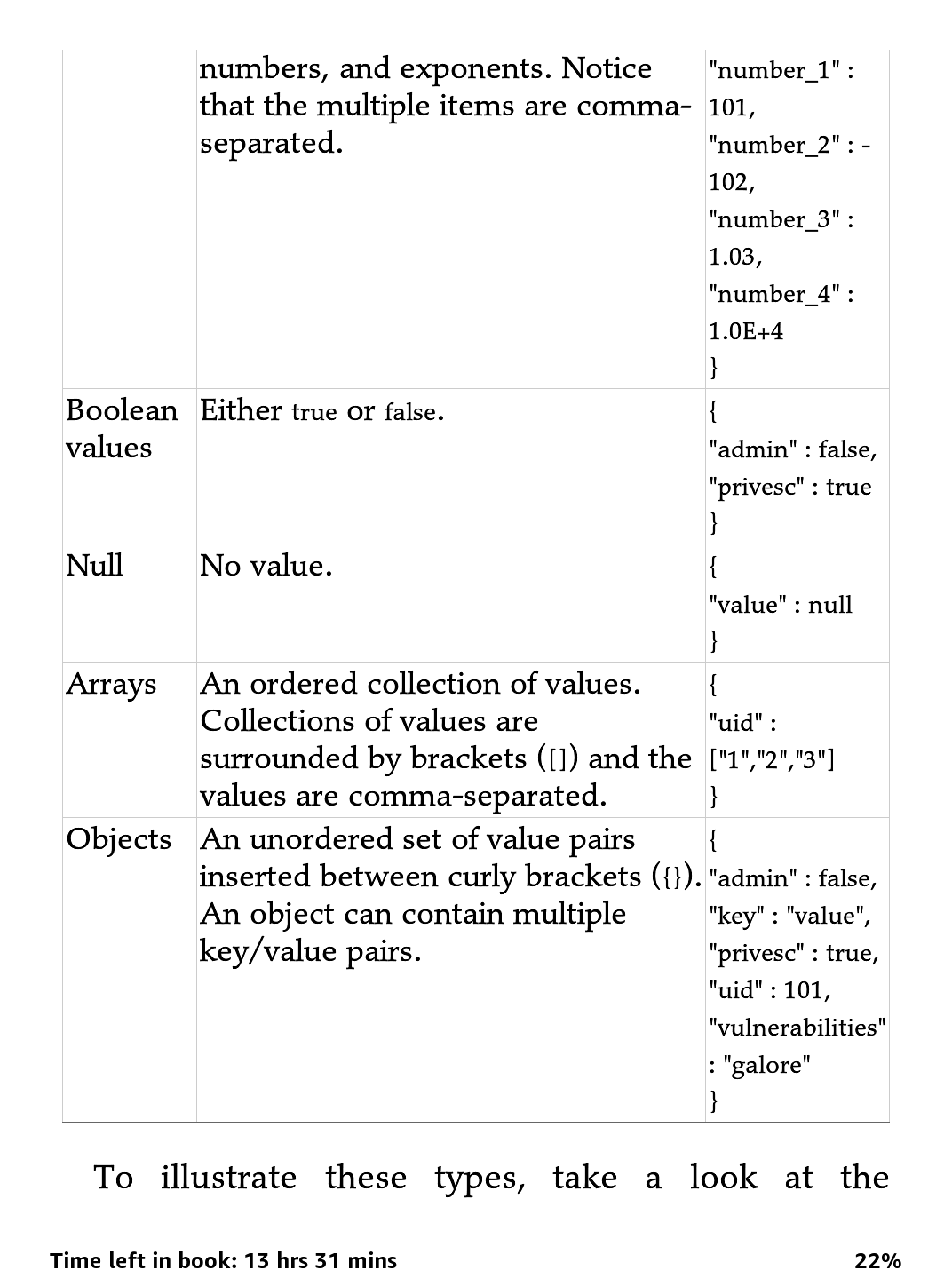
* Graph QL also uses the HTTP method , it typically depends on single entry point (URL) using the POST method.
* In Graph QL the body of the post request is what the provider processes.
* If this had been a REST API, it might have been necessary to send requests to different endpoints to get the quantity and then the brand of the graphics card, but with GraphQL you can build out a query for the specific information you are looking for from a single endpoint.

This above statement does not mean that REST is stateful API, it simply means that each endpoint in REST handles a separate resource .This is why the first bullet point states that GRAPH has a single point of entry.

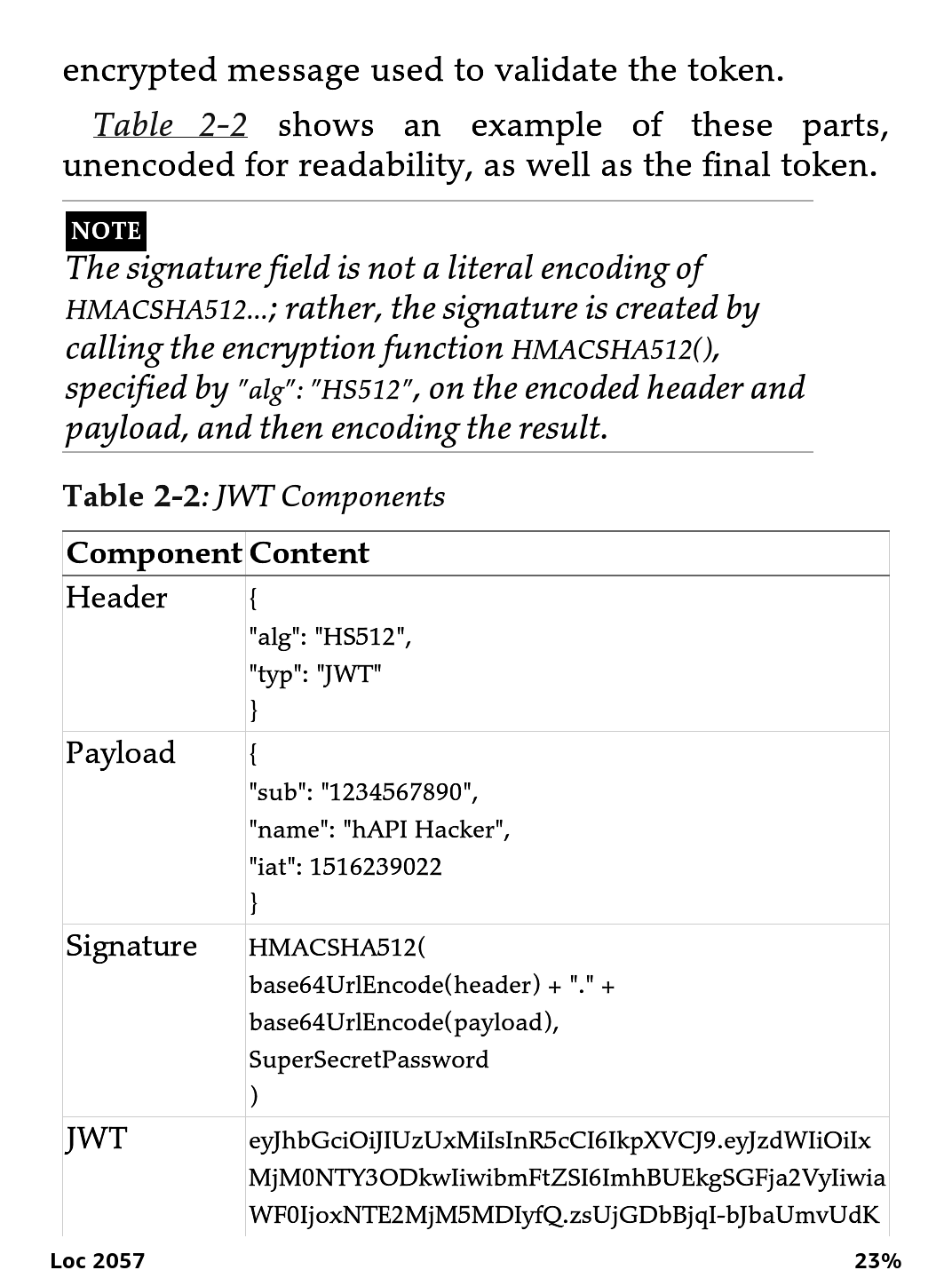
* GraphQL still functions using CRUD, which may sound confusing at first since it relies on POST requests. However, GraphQL uses three operations within the POST request to interact with GraphQL APIs: query, mutation, and subscription. Query is an operation to retrieve data (read). Mutation is an operation used to submit and write data (create, update, and delete). Subscription is an operation used to send data (read) when an event occurs. Subscription is a way for GraphQL clients to listen to live updates from the server.

# **JSON**





JSON WEB TOKEN COMPONENETS



* JSON does not allow inline comments, so any sort of comment-like communications must take place as a key/value pair like "comment" : "This is a comment". Alternatively, you can find comments in the API documentation or HTTP response.

# **SOAP API**

SOAP API messages are made up of four parts: the envelope ❶ and header ❷, which are necessary, and the body ❸ and fault ❹, which are optional.

1. Envelope is the XML tag in the beginning of a message that signifies that the following is a SOAP message.
2. Header is used to process the message. For example Content type request header allows the SOAP provider know what type of content is being sent in the POST request.

Header is essentially a Machine to machine agreement between the provider and the consumer concerning the expectations within the request.

1. The body is the primary payload of the XML message, meaning it contains the data sent to the application.
2. The fault is an optional part of a SOAP response that can be used to provide error messaging.

## XML

XML tags are surrounded by angle brackets (< and >). They typically have a structure like this:

XML

<tagName attribute="value">content</tagName>

Here's a breakdown:

* <tagName>: The opening tag, which starts with a less-than sign (<) followed by the tag name.
* attribute="value": Optional attributes, which provide additional information about the tag. Attributes consist of a name and a value separated by an equals sign (=).
* content: The text or data contained within the tag.
* </tagName>: The closing tag, which starts with a less-than sign (<) followed by a forward slash (/) and the tag name.

Example:

XML

<person id="123">John Doe</person>

In this example:

* <person> is the opening tag.
* id="123" is an attribute.
* John Doe is the content.
* </person> is the closing tag.

XML tags can also be self-closing, meaning they don't have a separate closing tag:

XML

<img src="image.jpg" />

1. KEY differences between REST, JSON , SOAP

REST API (Representational State of Resource)

* Architecture: Client-Server
* Data Format: JSON (usually)
* Request/Response: HTTP methods (GET, POST, PUT, DELETE)
* Endpoints: Multiple URLs for different resources

Graphical Representation:

+---------------+

| Client |

+---------------+

|

| HTTP Request

|

v

+---------------+

| Server |

| (Multiple |

| Endpoints) |

+---------------+

|

| HTTP Response

|

v

+---------------+

| Client |

+---------------+

XML Tags (Extensible Markup Language)

* Purpose: Data representation and formatting
* Structure: Tags (elements) with attributes and content
* Example: <person id="123">John Doe</person>

Graphical Representation:

+---------------+

| <person> |

| id="123" |

| John Doe |

| </person> |

+---------------+

How would the XML request will look like

The Twitter JSON you just saw would look like the following if converted to XML: <?xml version="1.0" encoding="UTF-8" ?> ❶ <root> ❷ <id>1278533978970976300</id> <id\_str>1278533978970976256</id\_str> <full\_text>1984: William Gibson published his debut novel, Neuromancer. It&#x27;s a cyberpunk tale about Henry Case, a washed up computer hacker who&#x27;s offered a chance at redemption by a mysterious dude named Armitage. Cyberspace. Hacking. Virtual reality. The matrix. Hacktivism. A must read. https://t.co/R9hm2LOKQi </full\_text> <truncated>false</truncated> </root>

SOAP (Simple Object Access Protocol)

* Purpose: Web service communication
* Format: XML-based
* Request/Response: Envelope, Header, Body
* Example: <soap:Envelope>...</soap:Envelope>

Graphical Representation:

+---------------+

| <soap:Envelope> |

| <soap:Header> |

| <soap:Body> |

| ... |

| </soap:Envelope> |

+---------------+

GraphQL

* Purpose: Query language for APIs
* Format: JSON-like
* Request/Response: Query, Mutation, Subscription
* Example: query { person(id: 123) { name } }

Graphical Representation:

+---------------+

| query { |

| person(id: |

| 123) { |

| name |

| } |

+---------------+

JSON Web Request

+---------------+

| Client |

+---------------+

|

| JSON Request

| (e.g., {"name": "John", "age": 30})

|

v

+---------------+

| Server |

| (JSON |

| Endpoint) |

+---------------+

|

| JSON Response

| (e.g., {"id": 123, "name": "John"})

|

v

+---------------+

| Client |

+---------------+

JSON REQUEST FORMAT

{ "id":1278533978970976256, ❶ "id\_str":"1278533978970976256", ❷ "full\_text":"1984: William Gibson published his debut novel, Neuromancer. It's a cyberpunk tale about Henry Case, a washed up computer hacker who's offered a chance at redemption by a mysterious dude named Armitage. Cyberspace. Hacking. Virtual reality. The matrix. Hacktivism. A must read. https:\/\/t.co\/R9hm2LOKQi", "truncated":false ❸

* JSON Web Request:
  + Uses JSON format for requests and responses.
  + Typically uses HTTP methods (GET, POST, PUT, DELETE).
  + No specific endpoint structure.
* REST:
  + Focuses on resource-based architecture.
  + Uses HTTP methods to interact with resources.
  + Multiple endpoints for different resources.
* SOAP:
  + Uses XML format for requests and responses.
  + Typically uses a single endpoint.
  + Envelope, Header, and Body structure.

## **YAML**

Initially, YAML was an acronym for "Yet Another Markup Language," but it was later changed to "YAML Ain't Markup Language" to reflect its purpose as a data serialization language, rather than a markup language.

YAML documents begin with --- and end with ... instead of with curly brackets.

XML is often used for documentation and validation, YAML is often used for configuration and serialization

# AUTHENTICATION TO WEB SERVER COOKIES

After a client sends credentials, the web server compares what was sent to the credentials it has stored. If the credentials provided match the credentials stored, the web server will create a user session and issue a cookie to the client. When the session ends between the web app and user, the web server will destroy the session and remove the associated client cookies.

REST and GraphQL APIs are stateless, so when a consumer authenticates to these APIs, no session is created between the client and server. Instead, the API consumer must prove their identity within every request sent to the API provider’s web server.

Usually with rest and Graphql after the user has initially authenticated the server might simply provide an **API key** or **token** (usually a **JWT** – JSON Web Token) that the client can use for the session instead of sending the credentials in every single request.

The client includes this token in the **Authorization** header of subsequent requests, like:

Http

Authorization: Bearer <JWT\_TOKEN>

API keys can be transmitted either via URL(query parameter)

**/api/v1/users?apikey=ju574n3x4mpl34p1k3y**

Via headers

**"API-Secret": "17813fg8-46a7-5006-e235-45be7e9f2345"**

Or via cookies

**Cookie: API-Key=4n07h3r4p1k3y**

| **Component** | **Description** | **Example** |
| --- | --- | --- |
| **Header** | **Contains metadata about the token. Usually includes:** | **{"alg": "HS256", "typ": "JWT"}** |
|  | **- Algorithm: Signing algorithm (e.g., HS256, RS256).** |  |
|  | **- Token Type: Usually "JWT".** |  |
| **Payload** | **Contains the claims, which are the actual data being transmitted. Claims can be:** | **{"sub": "1234567890", "name": "John Doe", "admin": true, "iat": 1516239022}** |
|  | **- Registered Claims: Predefined claims (e.g., iss, sub, exp).** |  |
|  | **- Public Claims: Custom claims agreed upon by the parties.** |  |
|  | **- Private Claims: Custom claims used between parties.** |  |
| **Signature** | **Ensures the token hasn’t been tampered with. Created by signing the encoded header and payload using a secret.** | **HMACSHA256(base64UrlEncode(header) + "." + base64UrlEncode(payload), secret)** |

### 

**Example of a JWT :**

**eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJzdWIiOiIxMjM0NTY3ODkwIiwibmFtZSI6IkpvaG4gRG9lIiwiYWRtaW4iOnRydWUsImlhdCI6MTUxNjIzOTAyMn0.SflKxwRJSMeKKF2QT4fwpMeJf36POk6yJV\_adQssw5c**

**It is composed of:**

* **Header: eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9**
* **Payload: eyJzdWIiOiIxMjM0NTY3ODkwIiwibmFtZSI6IkpvaG4gRG9lIiwiYWRtaW4iOnRydWUsImlhdCI6MTUxNjIzOTAyMn0**
* **Signature: SflKxwRJSMeKKF2QT4fwpMeJf36POk6yJV\_adQssw5c**

1. Linux command - to encode or decode base 64 text strings

$ echo "username:password"|base64 dXNlcm5hbWU6cGFzc3dvcmQK



$ echo "dXNlcm5hbWU6cGFzc3dvcmQK"|base64 -d username:password

# HMAC

* Hash-based message authentication code (HMAC) is the primary API authentication method used by Amazon Web Services (AWS). When using HMAC, the provider creates a secret key and shares it with consumer.
* When a consumer interacts with the API, an HMAC hash function is applied to the consumer’s API request data and secret key. The resulting hash (also called a message digest) is added to the request and sent to the provider.
* The provider calculates the HMAC, just as the consumer did, by running the message and key through the hash function, and then compares the output hash value to the value provided by the client. If the provider’s hash value matches the consumer’s hash value, the consumer is authenticated.

1. OAUTH

* OAuth 2.0 OAuth 2.0, or just OAuth, is an authorization standard that allows different services to access each other’s data, often using APIs to facilitate the service-to-service communications.
* Let’s say you want to automatically share your Twitter tweets on LinkedIn.
* In OAuth’s model, we would consider Twitter to be the service provider and LinkedIn to be the application or client.
* In order to post your tweets, LinkedIn will need authorization to access your Twitter information.
* Since both Twitter and LinkedIn have implemented OAuth, instead of providing your credentials to the service provider and consumer every time you want to share this information across platforms, you can simply go into your LinkedIn settings and authorize Twitter. Doing so will send you to api.twitter.com to authorize LinkedIn to access your Twitter account.

When you authorize LinkedIn to access your Twitter posts, Twitter generates a limited, time-based access token for LinkedIn. LinkedIn then provides that token to Twitter to post on your behalf, and you don’t have to give LinkedIn your Twitter credentials.

# BOLA Broken object level authorization (BOLA).

* BOLA vulnerabilities occur when an API provider allows an API consumer access to resources they are not authorized to access.
* If an API endpoint does not have object-level access controls, it won’t perform checks to make sure users can only access their own resources. When these controls are missing, User A will be able to successfully request User B’s resources.

In this situation, we might check for these problems by using another identification number that is close to Cloud’s ID of 5501. Say we are able to obtain information about another user by sending a request for https://bestgame.com/api/v3/users?id=5502 and receiving the following response: { "id": "5502", "first\_name": "Zack", "last\_name": "Fair", "link": " https://www.bestgame.com/user/shinra-number-1", "name": "Zack Fair", "dob": "2007-09-13", "username": "shinra-number-1" } In this case, Cloud has discovered a BOLA. Note that predictable object IDs don’t necessarily indicate that you’ve found a BOLA. For the application to be vulnerable, it must fail to verify that a given user is only able to access their own resources.

# What to look for in an url

The bolded elements 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 bolded values: GET /api/resource/3 GET /user/account/find?**user\_id=23** POST /company/account/Google/balance POST /admin/pwreset/account/**111**

1. Mass Assignment

* While reading the API documentation regarding the account creation process, suppose you discover that there is an additional key, "isAdmin", that consumers can use to become administrators.
* You could use a tool like Postman or Burp Suite to add the attribute to a request and set the value to true: { "User": "scuttleph1sh", "Password": "GreatPassword123", "isAdmin": true }
* If the API does not sanitize the request input, it is vulnerable to mass assignment, and you could use the updated request to create an admin account.
* On the backend, the vulnerable web app will add the key/value attribute, {"isAdmin":"true"}, to the user object and make the user the equivalent of an administrator.

==========

### Reading x response time in api response

* Organizations often include versioning information in their endpoint names to distinguish between older and newer versions, such as /v1/, /v2/, /v3/, and so on.
* APIs still in development often use paths such as /alpha/, /beta/, /test/, /uat/, and /demo/.
* If you know that an API is now using apiv3.org/admin but part of the API documentation refers to apiv1.org/admin, you could try testing different endpoints to see if apiv1 or apiv2 is still active.
* Additionally, the organization’s changelog may disclose the reasons why v1 was updated or retired. If you have access to v1, you can test for those weaknesses.

### Business Logic Flaw

Business logic vulnerabilities (also known as business logic flaws, or BLFs) are intended features of an application that attackers can use maliciously. For example, if an API has an upload feature that doesn’t validate encoded payloads, a user could upload any file as long as it was encoded. This would allow end users to upload and execute arbitrary code, including malicious payloads.

# CHROME DEV TOOLS

| **DevTool** | **Description** |
| --- | --- |
| **Elements** | - Allows you to inspect and modify the DOM and CSS of the current page. |
|  | - View the structure of the HTML document, edit elements, and see changes in real-time. |
|  | - CSS styles can be modified directly, and you can see the computed styles for selected elements. |
| **Console** | - Displays messages, warnings, and errors generated by the JavaScript on the page. |
|  | - Enables execution of JavaScript commands and provides an interactive command line interface. |
|  | - Useful for logging and debugging code with functions like console.log(). |
| **Sources** | - Displays the files that make up the web application, including JavaScript, CSS, and HTML files. |
|  | - Allows you to set breakpoints and debug JavaScript code step-by-step. |
|  | - View and edit files directly and check the call stack. |
| **Network** | - Monitors network activity, showing requests and responses made by the page. |
|  | - Provides detailed information about each request, including status codes, headers, and response times. |
|  | - Useful for analyzing performance and troubleshooting API calls. |
| **Performance** | - Records and analyzes the runtime performance of your web application. |
|  | - Helps identify performance bottlenecks, including rendering and scripting times. |
|  | - Provides flame graphs and other visualizations to understand how resources are being utilized. |
| **Memory** | - Helps profile memory usage in your application. |
|  | - Allows you to take heap snapshots and analyze memory leaks. |
|  | - Useful for optimizing performance and ensuring efficient memory usage. |
| **Application** | - Provides access to information about the web application’s storage, including cookies, local storage, and session storage. |
|  | - Allows you to manage service workers and view the manifest for Progressive Web Apps (PWAs). |
|  | - Useful for debugging storage issues and understanding offline capabilities. |
| **Security** | - Displays information about the security of the current page, including HTTPS status and certificate details. |
|  | - Useful for checking security-related issues and understanding mixed content warnings. |
| **Lighthouse** | - Provides automated audits for performance, accessibility, best practices, SEO, and Progressive Web Apps. |
|  | - Generates reports to help you improve your web application’s quality and user experience. |
| **Accessibility** | - Tools for auditing and improving the accessibility of your web applications. |
|  | - Provides insights into accessibility issues and recommendations for improvements. |
| **Device Mode** | - Allows you to simulate different device screen sizes and resolutions. |
|  | - Useful for testing responsive designs and mobile layouts. |
|  | - You can also emulate different user agents to test how the application behaves across devices. |

### 

1. WEB CRAWLING or SPIDERING

* Spidering, or web crawling, is a method that bots use to automatically detect the URL paths and resources of a host. Typically, spidering is done by scanning the HTML of web pages for hyperlinks.
* Spidering is a good way to get a basic idea of the contents of a web page, but it won’t be able to find hidden paths, or the ones that do not have links found within web pages.
* To find hidden paths, we’ll need to use a tool like Kiterunner that effectively performs directory brute-force attacks. In such an attack, an application will request various possible URL paths and validate whether they actually exist based on the host’s responses.
* Robots.txt is a key element in web crawling , other elements can include mega tags , site maps and link structure.
* mega tags

<meta name="robots" content="noindex"> prevents a page from being indexed.

<meta name="robots" content="nofollow"> instructs the bot not to follow links on that page.

* Site maps
* A sitemap is an XML file that lists all the important pages on a website, helping search engines understand the site's structure and find pages that might not be easily discoverable through links.
* Submitting a sitemap through search engine webmaster tools (like Google Search Console) can improve crawling efficiency.
* Link structure

The internal linking structure of a website significantly affects how spiders discover and index pages. Pages with strong internal links are more likely to be crawled.

Sitemaps for website crawling are typically provided by the **server**.

# KALI FOXYPROXY BURP INSTALLATION

1. FOXYPROXY

install FoxyProxy Standard and add it to your browser.

* Click the fox icon at the top-right corner of your browser (next to your URL) and select Options.
* Select Proxies▶Add New Proxy▶Manual Proxy Configuration.
* Add 127.0.0.1 as the host IP address.
* Update the port to 8080 (Burp Suite’s default proxy settings).
* Under the General tab, rename the proxy to Hackz (I will refer to this proxy setting throughout the labs)

1. Installing the certificate in burp suite and your browser

Start Burp Suite. Open your browser of choice. Using FoxyProxy, select the Hackz proxy. Navigate to http://burpsuite, as seen in Figure 4-3, and click CA Certificate.

This will initiate the download of the Burp Suite CA certificate. Figure 4-3: The landing page you should see when downloading Burp Suite’s CA certificate Save the certificate somewhere you can find it. Open your browser and import the certificate.

In Firefox, open Preferences and use the search bar to look up certificates. Import the certificate.

In Chrome, open Settings, use the search bar to look up certificates, select More▶Manage Certificates▶Authorities, and import the certificate (see Figure 4-4). If you do not see the certificate, you may need to expand the file type

1. BURP SUITE TABS AND OPTIOSN

| **Tab** | **Functionality** | **Key Options/Features** |
| --- | --- | --- |
| **Dashboard** | Provides an overview of your Burp Suite's scanning activities, including automated scans and task statuses. | - Scan queue  - Issue activity  - Logger  - Alerts |
| **Target** | Defines the target scope for testing and provides detailed information on the target web application. | - Site map  - Scope definition  - Filter by URL or folder structure |
| **Proxy** | Allows interception, modification, and forwarding of HTTP(S) requests between the browser and target application. | - Intercept tab  - HTTP history  - WebSockets history  - Options (interception rules, SSL certificates, etc.) |
| **Intruder** | Automates attacks, allowing you to test how inputs can be manipulated to exploit vulnerabilities.  Any part of a captured HTTP request can be transformed into a variable, or attack position, by surrounding it with **§** symbols | - Positions (define where to inject payloads)  - Perform fuzzing and brute-force attacks.  - Payloads (set attack types)  - Attack types (Sniper, Battering Ram, Pitchfork, Cluster Bomb)  - Options (request timing, payload encoding) |
| **Repeater** | Allows you to manually modify and re-send individual requests to observe how the target web application responds. | - Request editor  - Response viewer (Raw, Headers, Hex, Rendered)  - History |
| **Sequencer** | Analyzes the quality of randomness in tokens and other session-related data generated by the web application. | - Token collection (manual or automated)  - Analysis metrics (randomness distribution, entropy, etc.) |
| **Decoder** | Encodes or decodes data in various formats (URL, Base64, Hex, etc.) and can perform hashing functions. | - Encode/decode options  - Hash options (MD5, SHA-256, etc.) |
| **Comparer** | Compares two sets of data (e.g., responses or requests) to identify differences. | - Side-by-side comparison view  - Automated or manual comparison |
| **Extender** | Allows you to add extensions to Burp Suite, either from the BApp Store or custom-built using the Burp Extender API. | - BApp Store  - Installed extensions  - Extension options |
| **Scanner** | Scans the target for vulnerabilities using Burp's automated or semi-automated scanning tools. | - Active scan  - Passive scan  - Issues list (detailed view of vulnerabilities)  - Request/response pair for each vulnerability |
| **Logger** | Logs all HTTP(S) traffic, WebSocket messages, and errors in the proxy. | - HTTP history  - Error logs |
| **Alerts** | Displays notifications for critical events like security issues, failed requests, and errors during testing. | - Alert messages with severity levels (info, warning, critical) |
| **Options** | Provides global settings and configurations for Burp Suite, including proxy settings, SSL certificates, and scanning options. | - Proxy options  - HTTP request handling  - SSL/TLS settings  - Miscellaneous settings |

### 

### **Intruder Payload Type Description Example**

### **Sniper** Sends a single payload at a time, useful for testing specific vulnerabilities. Send a specific username and password combination to a login form: - Payload: admin:password

### **Battering Ram** Sends the same payload repeatedly, useful for testing rate limiting or flood protection. Send the same request multiple times to a server: - Payload: GET /login HTTP/1.1 (Repeat this request many times)

### **Pitchfork** Sends a list of payloads one by one, useful for testing various input values. Send a list of usernames to a login form: - Payloads: admin, user1, user2, ...

### **Cluster Bomb** Sends a combination of payloads, useful for testing complex interactions. Send different combinations of username and password pairs to a login form: - Payloads: - admin:password1 - user1:password2 - user2:password3

### 

### **NOTE :**

### If you’re fuzzing a single attack position, use sniper.

### If you’re fuzzing several attack positions at once, use battering ram.

### When you need to test set combinations of payloads, use pitchfork.

### For password-spraying efforts, use cluster bomb.

1. EXTENSTION FOR BURP SUITE :

**AUTORISE**

Autorize checks if restricted resources or actions can be accessed by unauthorized users.

The extension sends each request twice:

1. Once with the session cookies of an authenticated user.
2. Once without those session cookies or with a low-privilege user’s session.

It compares the two responses to see if there is any difference, which helps in identifying unauthorized access

**IP Rotate**

IP Rotate allows you to alter the IP address you are attacking from to indicate different cloud hosts in different regions. This is extremely useful against API providers that simply block attacks based on IP address.

**Bypass WAF**

* The WAF Bypass extension adds some basic headers to your requests in order to bypass some web application firewalls (WAFs).
* Some WAFs can be tricked by the inclusion of certain IP headers in the request. WAF Bypass saves you from manually adding headers such as X-Originating-IP, X-Forwarded-For, X-Remote-IP, and X-Remote-Addr.
* These headers normally include an IP address, and you can specify an address that you believe to be permitted, such as the target’s external IP address (127.0.0.1) or an address you suspect to be trusted.

# KALI FOXYPROXY POSTMAN INSTALLATION

1. set up Postman on Kali, open your terminal and enter the following commands:

$ sudo wget https://dl.pstmn.io/download/latest/linux64 -O postman-linux-x64.tar.gz

$ sudo tar -xvzf postman-linux-x64.tar.gz -C /opt

$ sudo ln -s /opt/Postman/Postman /usr/bin/postman

Configure another proxy within in Foxyproxy so that the intercepted traffic can be amended in postman

Add the same host IP address, 127.0.0.1, and set the port to 5555, the default port for Postman’s proxy. Update the name of the proxy under the General tab to Postman and save.

1. POSTMAN TABS

| **Tab** | **Description** | **Example** |
| --- | --- | --- |
| **New** | Provides options to create new requests, collections, environments, mock servers, monitors, or APIs. | You can create a **new HTTP request** or start a new **collection** of API requests from here. |
| **Request Tab** | The primary interface for sending HTTP requests (GET, POST, PUT, DELETE, etc.) to a server and viewing the response. | Sending a **GET request** to https://api.example.com/users to fetch user data. |
| **Params** | Allows you to add URL parameters (query strings) to the request URL. | Adding ?id=123 to the URL to filter user data: https://api.example.com/users?id=123. |
| **Authorization** | Provides options for adding authentication credentials (Basic Auth, Bearer Token, OAuth 2.0, etc.) for the request. | Using **Bearer Token** for API authorization: adding a token to access a restricted API. |
| **Headers** | Allows you to set custom headers (e.g., Content-Type, Authorization) for the request. | Adding a **Content-Type: application/json** header for a POST request. |
| **Body** | Used to specify the body of the request (for POST, PUT, PATCH methods). You can send raw data, JSON, form data, etc. | Sending JSON data in a POST request: { "name": "John", "email": "john@example.com" }. |
| **Pre-request Script** | Write JavaScript code that runs before the request is sent (useful for setting dynamic values like tokens). | Generating a timestamp dynamically for each request. |
| **Tests** | Write JavaScript test scripts to validate responses (e.g., check for status codes or response body values). | Checking if a response status is 200 OK after sending a request. |
| **Response Tab** | Displays the response from the server, including the body, headers, and status code. | Viewing a 200 OK status with JSON data returned from the API after a GET request. |
| **Cookies** | Displays cookies set by the server in the response. | Viewing cookies returned after sending a login request. |
| **Code** | Generates ready-to-use code snippets for different programming languages (e.g., Python, JavaScript, cURL) based on the current request. | Generating a **cURL** command for the request you just configured. |
| **Save** | Saves your request to a collection or a specific folder for future use. | Saving a **GET request** to the "User APIs" collection. |
| **Collections** | Displays a list of saved collections and folders that group related API requests.  If an API provider offers a collection, you won’t have to physically type in every single request. Instead, you can just import its collection. | Creating a collection of **user management API requests**. |
| **History** | Shows a list of previous API requests made in the current or past sessions. | Viewing a history of all recent requests made to test different APIs. |
| **Mock Servers** | Allows you to simulate APIs with mock responses to test client-side applications without needing the actual API. | Creating a mock server that returns a preset JSON response for testing purposes. |
| **Monitor** | Provides an option to set up automated API testing at regular intervals. | Setting up a monitor to run tests every 10 minutes to check if your API is up and running. |
| **Environment** | Allows you to create and manage environments with variables (e.g., dev, staging, production) to switch between different API settings easily. | Using environment variables like {{base\_url}} for switching between staging and production URLs. |
| **Variables** | Displays and manages variables that you can use in requests (global, collection-level, or environment-level variables). | Using {{api\_key}} variable to store API keys for different environments. |
| **Runner** | Executes a series of API requests (collections) with multiple iterations and data-driven testing using files like CSV/JSON. | Running a **collection of requests** to automate the testing of APIs in bulk. |
| **Settings** | Allows you to configure Postman settings, such as SSL verification, proxy, or behavior of request redirects. | Disabling **SSL certificate verification** when testing an API with self-signed certificates. |
| **LINK** | Select the Link tab Paste the URL to the API specification and click Continue. On the Confirm Your Import screen, click Import. Figure 4-20: Importing an API specification in Postman using the Link tab in the Import panel |  |

### 

1. Configure postman with burp suite

Adding proxy to POSTMAN and turning the SSL certificate verification off

Click the checkbox for adding a custom proxy configuration. Make sure to set the proxy server to 127.0.0.1. Set the proxy server port to 8080. Select the General tab and turn SSL certificate verification Off. In Burp Suite, select the Proxy tab. Click the button to turn Intercept On.

# EXTRA TOOLS

* The following tools are excellent at what they do, open source, and free.
* In particular, the API scanning tools covered here serve several purposes when you’re actively testing your target.
* Tools such as Nikto and OWASP ZAP can help you actively discover API endpoints, security misconfigurations, and interesting paths, and they provide some surface-level testing of an API.
* In other words, they are useful when you start actively engaging with a target, whereas tools such as Wfuzz and Arjun will be more useful once you’ve discovered an API and want to narrow the focus of your testing.
* Use these tools to actively test APIs to discover unique paths, parameters, files, and functionality.

### OWASP AMASS

We can use Amass to discover the attack surface of our target organisation.

With as little as the target domain name, we can use amass to scan through many internet sources for the associated domains to get a list of potential URLs to attack.

If amass is not installed we can use:

$ sudo apt-get install amass

* However, you can make it into an information collection powerhouse by setting it up with API keys from various sources.
* I recommend at least setting up accounts with GitHub, Twitter, and Censys.
* Once you’ve set up these accounts, you can generate API keys for these services and plug them into Amass by adding them to Amass’s configuration file, config.ini.

#https://urlscan.io (Free) #URLScan can be used without an API key #apikey = # https://virustotal.com (Free) #[data\_sources.URLScan] #apikey

### KITERUNNER

Discovering API Endpoints with Kiterunner Kiterunner (https://github.com/assetnote/kiterunner) is a content discovery tool designed specifically for finding API resources. Kiterunner is built with Go, and while it can scan at a speed of 30,000 requests per second, it takes into account the fact that load balancers and web application firewalls will likely enforce rate limiting.

To install Kiterunner, run the following commands:

$ git clone https://github.com/assetnote/kiterunner.git $ cd kiterunner

$ make build

$ sudo ln -s $(pwd)/dist/kr /usr/local/bin/kr

You should then be able to use Kiterunner from the command line by entering the following:

$ kr

kite is a context based webscanner that uses common api paths

CREATE A WORD LISTS FOR KITERUNNER , ASSEST.IO

Create an API wordlists directory as follows:

$ mkdir -p ~/api/wordlists

You can then select your desired wordlists and download them to the /api/wordlists directory:

$ curl https://wordlists-cdn.assetnote.io/data/automated/httparchive\_apiroutes\_2021\_06\_28.txt > latest\_api\_wordlist.txt

% Total % Received % Xferd Average Speed Time Time Time Current Dload Upload Total Spent Left Speed 100 6651k 100 6651k 0 0 16.1M 0 --:--:-- --:--:-- --:--:-- 16.1M

You can replace httparchive\_apiroutes\_2021\_06\_028.txt with whichever wordlists suit you best.

### NIKTO

Nikto is a command line web application vulnerability scanner that is quite effective at information gathering. I use Nikto immediately after discovering the existence of a web application, as it can point me toward the application’s interesting aspects.

Nikto will provide you with information about the target web server, security misconfigurations, and other web application vulnerabilities. Since Nikto is included in Kali, it should not require any special setup.

To scan a domain, use the following command: $ nikto -h https://example.com

To see the additional Nikto options, enter nikto -Help on the command line.

A few options you may find useful include -output filename for saving the Nikto results to a specified file and -maxtime #ofseconds to limit how long a Nikto scan will take.

### OWASP ZAP

ZAP has two components: automated scan and manual explore.

ZAP’s automated scan performs web crawling, detects vulnerabilities, and tests web application responses by altering request parameters.

Automated scan is great for detecting the surface directories of a web application, which includes discovering API endpoints.

To run it, enter the target URL into the ZAP interface and click the button to start the attack.

Once the scan has run its course, you’ll receive a list of alerts that are categorized by the severity of the finding.

The issue with ZAP’s automated scan is that it can be riddled with false positives, so it is important to examine and validate the alerts. The testing is also limited to the surface of a web application.

### Wfuzz

You can use Wfuzz to inject a payload within an HTTP request by replacing occurrences of the word FUZZ with words from a wordlist; Wfuzz will then rapidly perform many requests (around 900 requests per minute) with the specified payload. Since so much of the success of fuzzing depends on the use of a good wordlist, we’ll spend a decent amount of time discussing wordlists in Chapter 6.

Here’s the basic request format of Wfuzz: $ wfuzz options -z payload,params url

To run Wfuzz, use the following command:

$ wfuzz -z file,/usr/share/wordlists/list.txt http://targetname.com/FUZZ

This command replaces FUZZ in the URL http://targetname.com/FUZZ with words from /usr/share/wordlists/list.txt. The -z option specifies a type of payload followed by the actual payload. In this example, we specified that the payload is a file and then provided the wordlist’s file path.

You can use the range option to easily scan a series of numbers:

$ wfuzz -z range,500-1000 http://targetname.com/account?user\_id=FUZZ

This will automatically fuzz all numbers from 500 to 1000. This will come in handy when we test for BOLA vulnerabilities. To specify multiple attack positions, you can list off several -z flags and then number the corresponding FUZZ placeholders, such as FUZZ, FUZ1, FUZ2, FUZ3, and so on, like so: $ wfuzz -z list,A-B-C -z range,1-3 http://targetname.com/FUZZ/user\_id=FUZZ2

### Arjun

$ cd /opt/

$ sudo git clone https://github.com/s0med3v/Arjun.git

* Arjun works by first performing a standard request to the target API endpoint. If the target responds with HTML forms, Arjun will add the form names to the parameter list during its scan.
* Arjun then sends a request with parameters it expects to return responses for nonexistent resources. This is done to note the behavior of a failed parameter request.
* Arjun then kicks off 25 requests containing the payload of nearly 26,000 parameters, compares the API endpoint’s responses, and begins additional scans of the anomalies.
* To run Arjun, use the following command:

$ python3 /opt/Arjun/arjun.py -u http://target\_address.com

If you would like to have the output results in a certain format, use the -o option with your desired file type:

$ python3 /opt/Arjun/arjun.py -u http://target\_address.com -o arjun\_results.json

# DOCKER

Installing Docker and Docker Compose Once you’ve configured your host operating system, you can use Docker to host the vulnerable applications in the form of containers.

Docker and Docker Compose will make it incredibly easy to download the vulnerable apps and launch them within a few minutes.

Follow the official instructions at https://docs.docker.com/engine/install/ubuntu to install Docker on your Linux host.

Installed using :

$ sudo apt update -y

$ sudo apt install docker.io

$ sudo apt install docker-compose

$groups

Will show us all the current groups that we have for the logged in user.

You’ll know that Docker Engine is installed correctly when you can run the hello-world image:

$ sudo docker run hello-world



If you can run the hello-world container, you have successfully set up Docker. Congrats! Otherwise, you can troubleshoot using the official Docker instructions.

Docker Compose is a tool that will enable you to run multiple containers from a YAML file. Depending on your hacking lab setup, Docker Compose could allow you to launch your vulnerable systems with the simple command docker-compose up. The official documentation for installing Docker Compose can be found at https://docs.docker.com/compose

# Which vulnerable apps have to be installed

Installing Vulnerable Applications I have selected these vulnerable applications to run in the lab:

* OWASP crAPI,
* OWASP Juice Shop,
* OWASP DevSlop’s
* Pixi, and
* Damn Vulnerable GraphQL.

These apps will help you develop essential API hacking skills such as discovering APIs, fuzzing, configuring parameters, testing authentication, discovering OWASP API Security Top 10 vulnerabilities, and attacking discovered vulnerabilities. This section describes how to set up these applications.

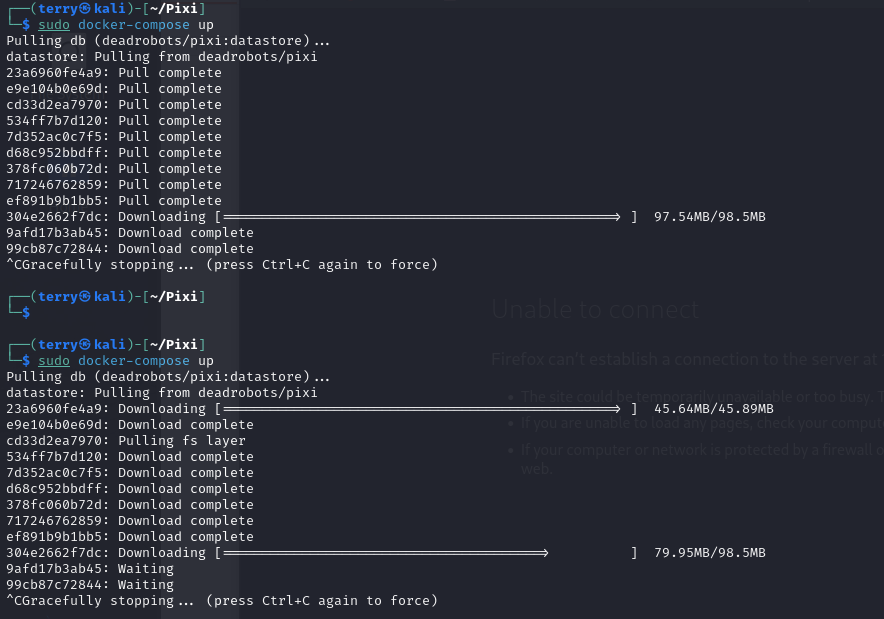
1. Setting up pixi

$ git clone <https://github.com/DevSlop/Pixi.git>

$ cd Pixi

$ sudo docker-compose up

Then use a browser and visit http://localhost:8000 to see the landing page. If Docker and Docker Compose have been set up, as described previously in this chapter, launching Pixi should really be as easy as that.

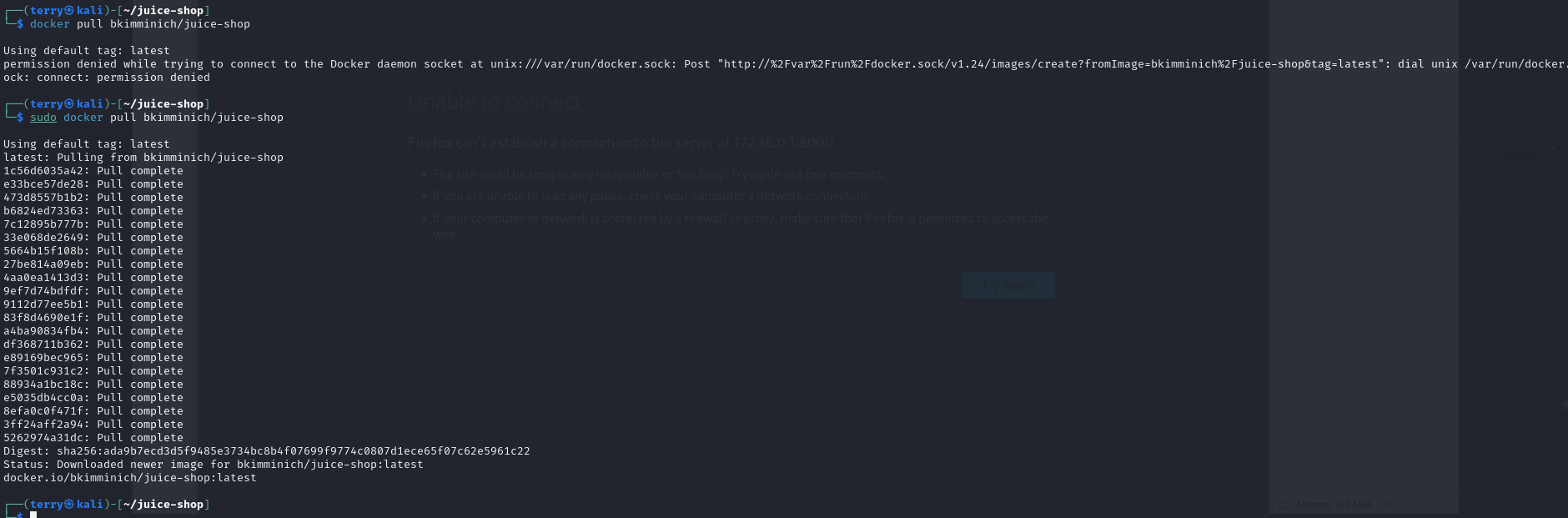


1. Juiceshop

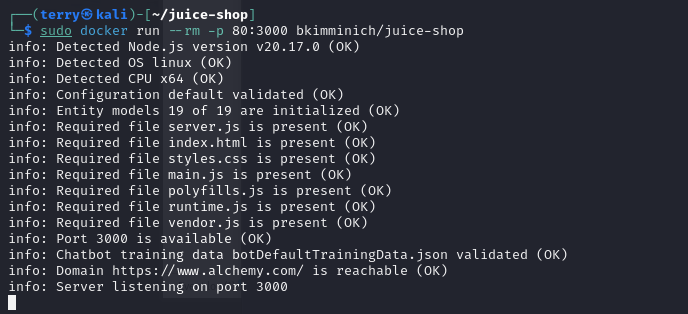
To download and launch Juice Shop, run the following commands:

$ docker pull bkimminich/juice-shop

$ docker run --rm -p 80:3000 bkimminich/juice-shop



Juice Shop and Damn Vulnerable GraphQL Application (DVGA) both run over port 3000 by default.



To avoid conflict, the -p 80:3000 argument in the docker-run command sets Juice Shop up to run over port 80 instead.

To access Juice Shop, browse to http://localhost. (On macOS and Windows, browse to http://192.168.99.100 if you are using Docker Machine instead of the native Docker installation.)

Damn Vulnerable GraphQL Application DVGA is a deliberately vulnerable GraphQL application developed by Dolev Farhi and Connor McKinnon. I’m including DVGA in this lab because of GraphQL’s increasing popularity and adoption by organizations such as Facebook, Netflix, AWS, and IBM. Additionally, you may be surprised by how often a GraphQL integrated development environment (IDE) is exposed for all to use. GraphiQL is one of the more popular GraphQL IDEs you will come across. Understanding how to take advantage of the GraphiQL IDE will prepare you to interact with other GraphQL APIs with or without a friendly user interface (see Figure 5-4).

DVGA

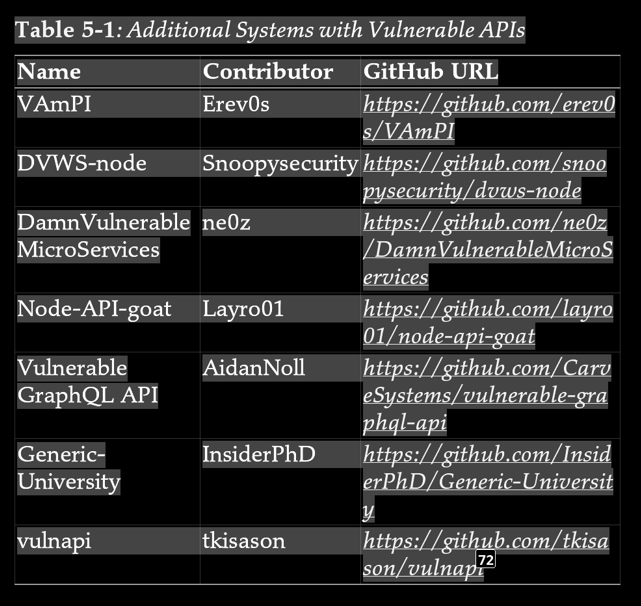
To download and launch DVGA, run the following commands from your Ubuntu host terminal:

$ sudo docker pull dolevf/dvga

$ sudo docker run -t -p 5000:5000 -e WEB\_HOST=0.0.0.0 dolevf/dvga

To access it, use a browser and visit http://localhost:5000.

1. OTHER VULNERABLE APPS



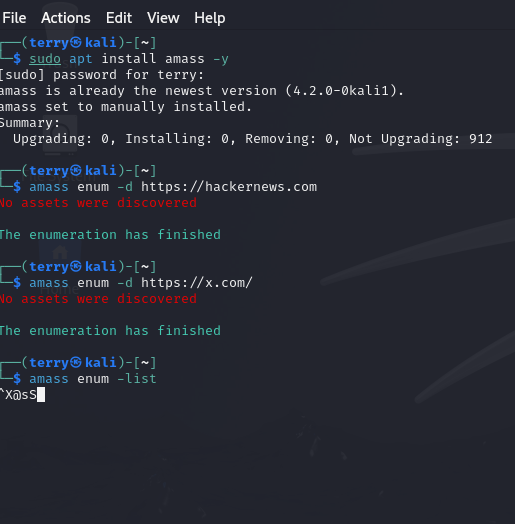
## OWASP JUICE SHOP

FIrstly, we will use the command sudo netdiscover to find the IP address of the device that is hosting the juice shop

Nmap hostip

Should reveal port 3000 is open on the host but unfortunately we only see port 80 open

1. Also, in a cloud environment, it would be beneficial to block access to the IP address 169.254.169.254, which contains metadata for the deployed cloud server, including possibly sensitive information. An attacker can bypass this by registering a subdomain on their own domain with a DNS record that points to the IP Address 169.254.169.254.
2. An ASN ID refers to an Autonomous System Number (ASN), which is a unique identifier assigned to an Autonomous System (AS). An Autonomous System is a collection of IP networks and routers under the control of a single organization that presents a common routing policy to the internet. These systems typically belong to Internet Service Providers (ISPs), large enterprises, universities, and government bodies.
3. **OWASP AMASS**

****

****

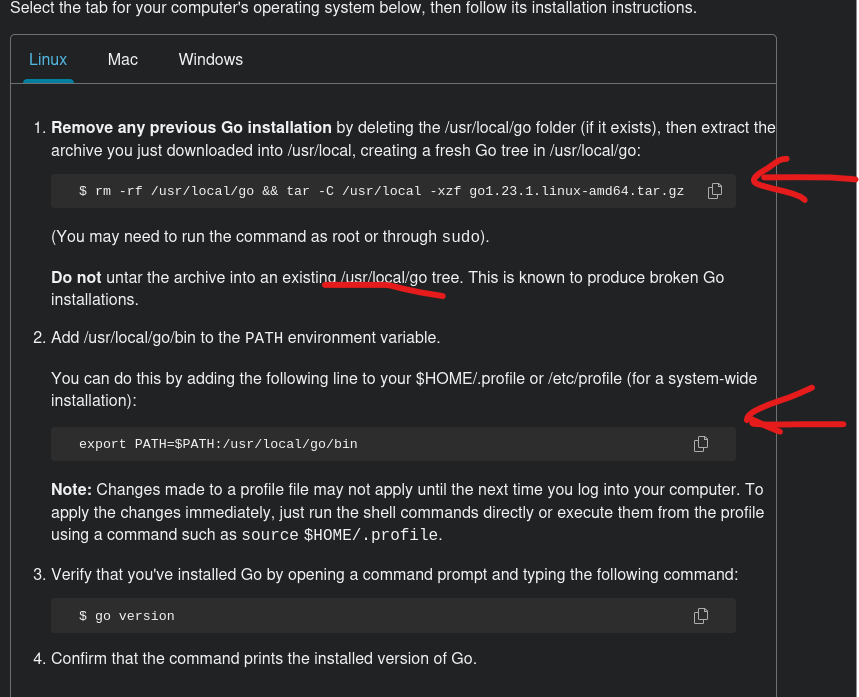
* **As we can see above that all the api’s do not have the key configured**
* **Which would mean that our amass is currently not looking for IP data in all the sub domains for our searches**

1. **Run tools from anywhere**

**/usr/bin**

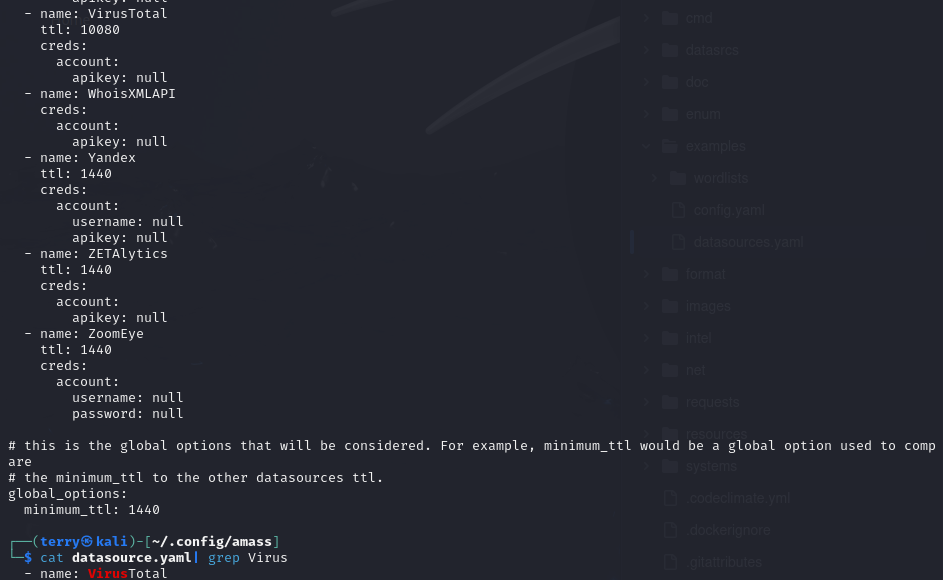
**For example if we have installed go and we want to run it from any directory in the kali . We can move it to the /usr/local**

**Add step 2 from the image below so that go is installed as a local command line tool that can be access from any directory within kali.**

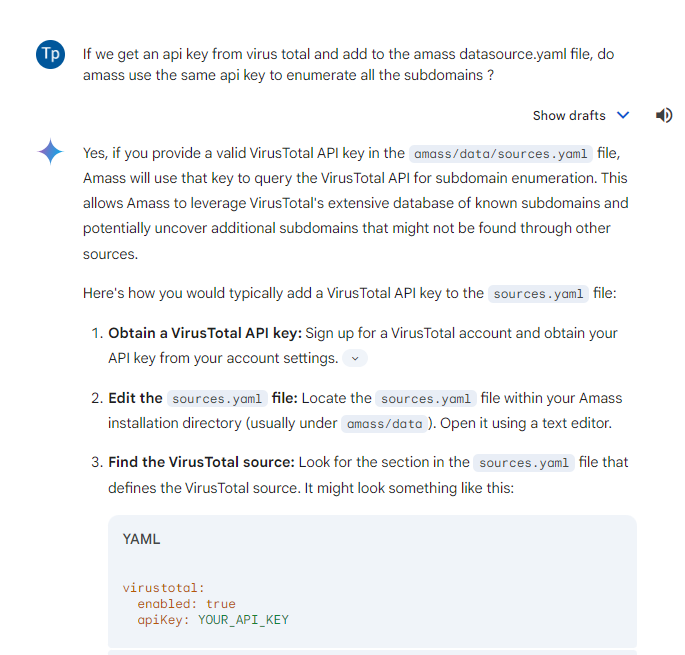
****

**One that is set up , we are required to setup the config.yaml and datasource.yaml file from the github repo**

**Into .config file that is located in the root repository**

****

**In order to add the api key we have to create an account in virus total and get an api key.**

****

**Same has to be actioned from github etc…**