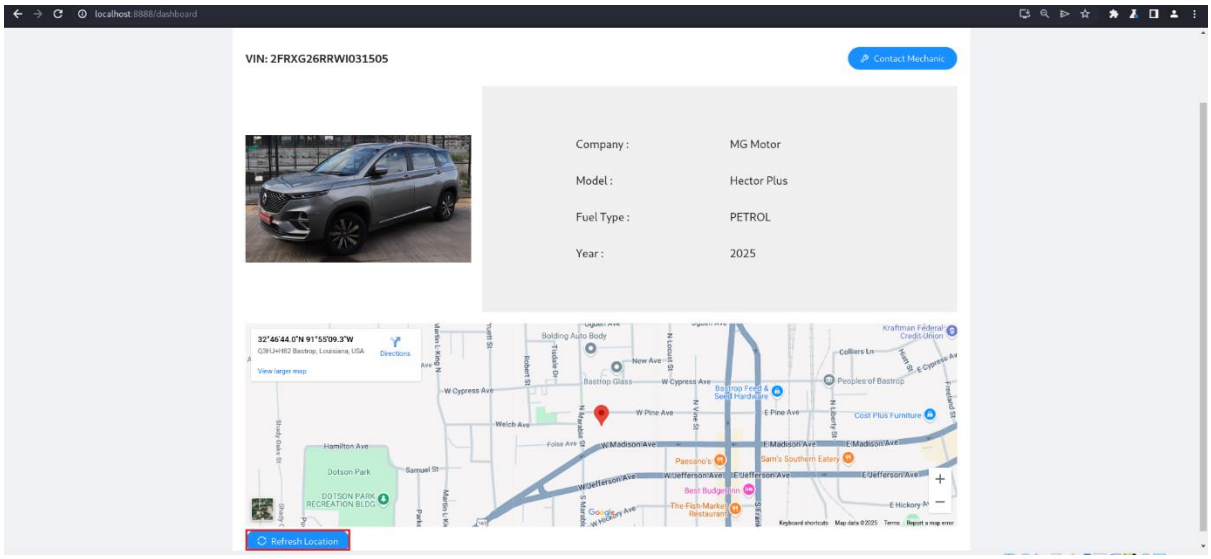
Broken Object-Level Authorization (BOLA)

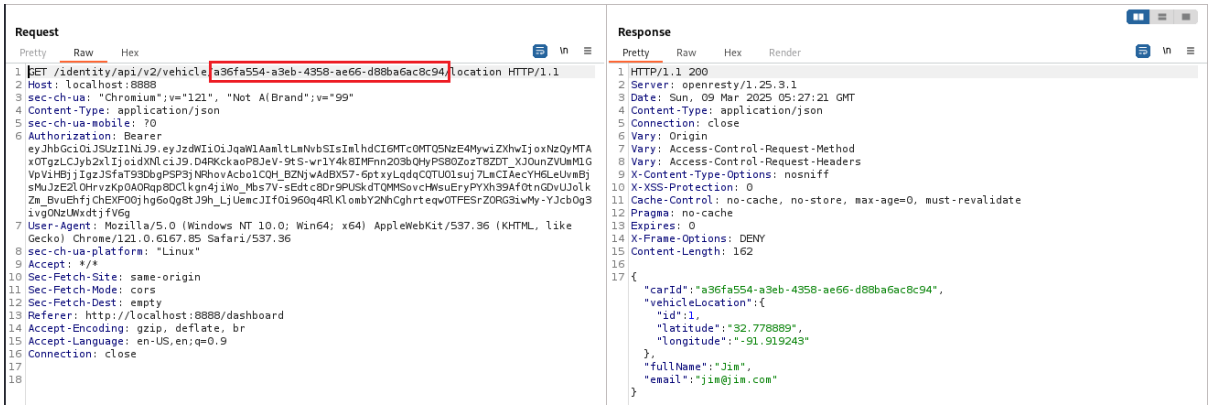
***BOLA, or Broken Object-Level Authorization**, is a type of security vulnerability that occurs when an application fails to properly enforce access controls at the object or data level. This can lead to unauthorized users gaining access to sensitive data or performing actions they should not be allowed to perform within the application.*

Challenge 1 — Access location details of another user's vehicle.

Screenshot-1: First signing up and logging into the account, it is noticed that an "Add Vehicle" option in the dashboard. After successfully adding a vehicle, the dashboard displayed detailed information about the vehicle along with its location on a Google map. Curiously, I discovered a "Refresh Location" option at the bottom of the page.



Screenshot-2: Click on 'Refresh Location' button and intercept the request. It is observed that the request is revealing an API endpoint that requires a vehicle ID, as shown below.



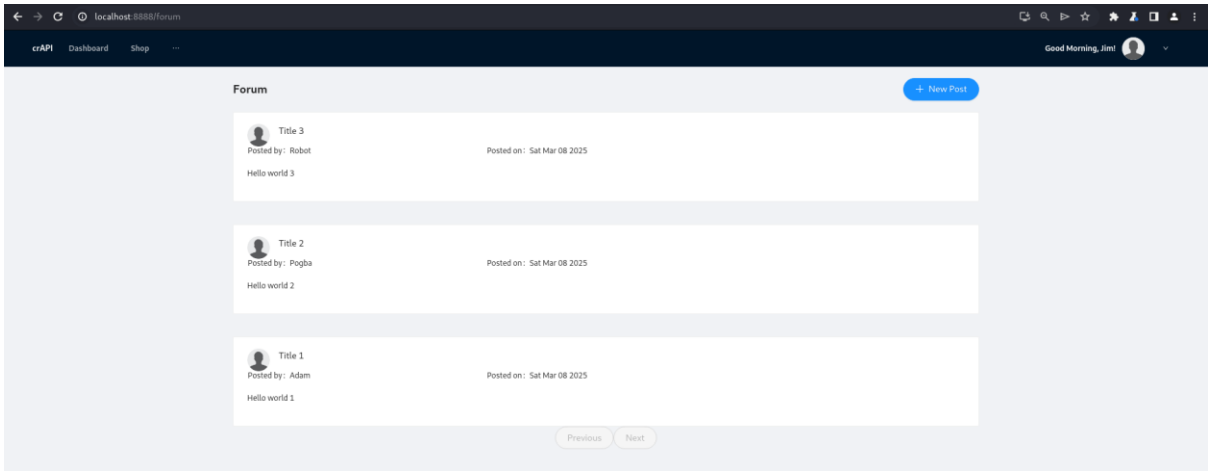
Request

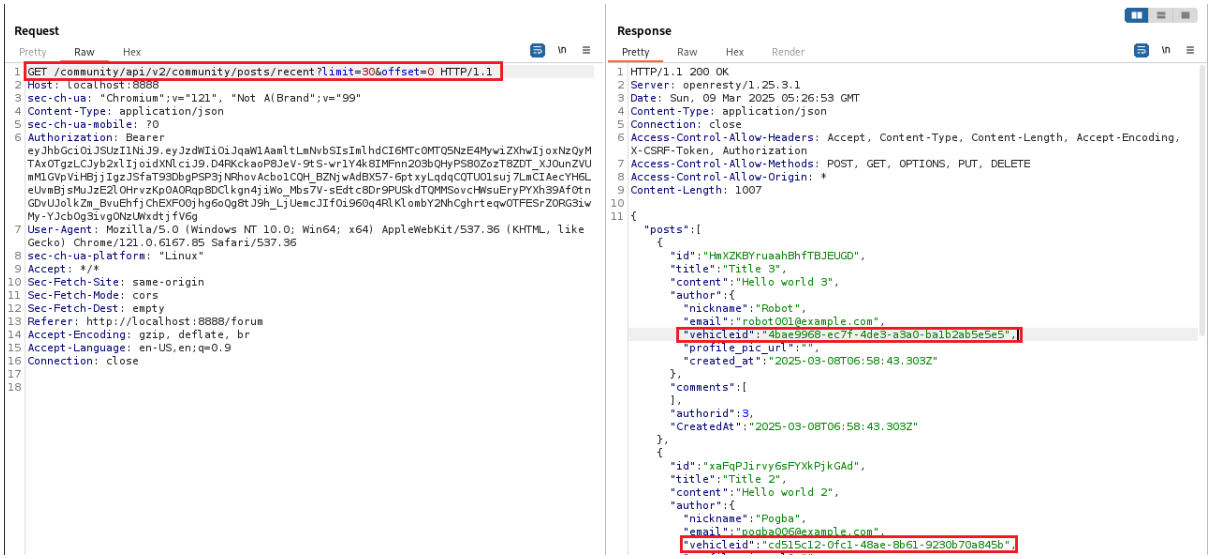
```
GET /identity/api/v2/vehicle/a36fa554-a3eb-4358-ae66-d88ba6ac8c94/location HTTP/1.1
Host: localhost:8888
sec-ch-ua: "Chromium";v="121", "Not A(Brand";v="99"
Content-Type: application/json
sec-ch-ua-mobile: ?0
Authorization: Bearer
eyJhbGciOiJSUzI1NiJ9.eyJzdWIiOiJqaW1AamltLmNvbSIsImlhdCI6MTc0MTQ5NzE4MywiZXhwIjoxNzQyMTA
xOTgzLCJyb2xlIjoidXNlciJ9.D4RKckaoP8JeV-9tS-wr1Y4k8IMFnn2O3bQHyPS8OZozT8ZDT_XJOunZVUmM1G
VpViHBjjIgzJSfaT93DbgPSP3jNRhovAcbolCQH_BZNjwAdBX57-6ptxyLqdqCQTUO1suj7LmCIAecYH6LeUvmBj
sMuJzE2lOHrvzKp0AORqp8DClkgn4jiWo_Mbs7V-sEdtc8Dr9PUSkdTQMMSovcHWsuEryPYXh39Af0tnGDvUJolk
Zm_8vuEhfjChEXFO0jhg6oQg8tJ9h_LjUemcJIfOi960q4RlKlombY2NhCghrteqwOTFESrZORG3iwMy-YJcbOg3
ivgONzUWxdtjfV6g
User-Agent: Mozilla/5.0 (Windows NT 10.0; Win64; x64) AppleWebKit/537.36 (KHTML, like
Gecko) Chrome/121.0.6167.85 Safari/537.36
sec-ch-ua-platform: "Linux"
Accept: */*
Sec-Fetch-Site: same-origin
Sec-Fetch-Mode: cors
Sec-Fetch-Dest: empty
Referer: http://localhost:8888/dashboard
Accept-Encoding: gzip, deflate, br
Accept-Language: en-US,en;q=0.9
Connection: close
```

Response

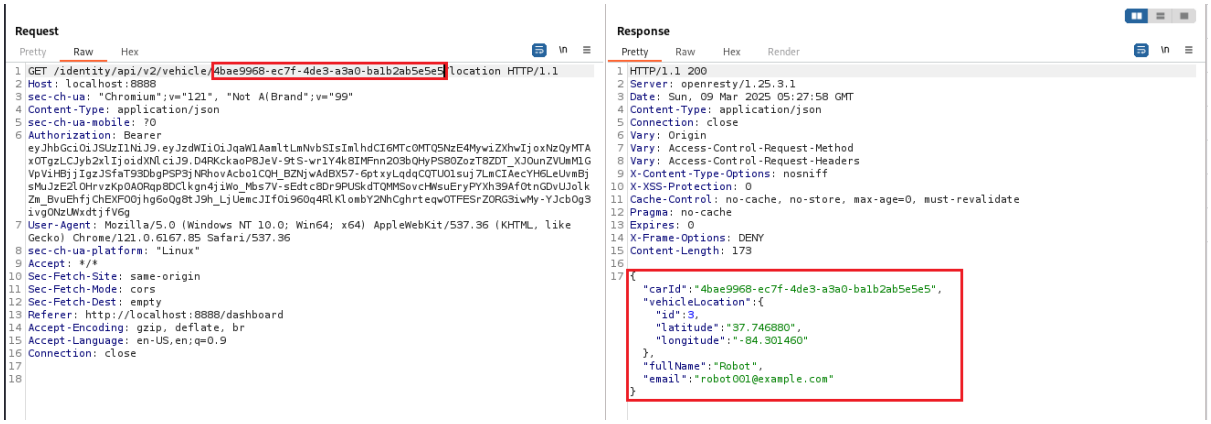
```
HTTP/1.1 200
Server: openresty/1.25.3.1
Date: Sun, 09 Mar 2025 05:27:21 GMT
Content-Type: application/json
Connection: close
Vary: Origin
Vary: Access-Control-Request-Method
Vary: Access-Control-Request-Headers
X-Content-Type-Options: nosniff
X-XSS-Protection: 0
Cache-Control: no-cache, no-store, max-age=0, must-revalidate
Pragma: no-cache
Expires: 0
X-Frame-Options: DENY
Content-Length: 162

{
  "carId":"a36fa554-a3eb-4358-ae66-d88ba6ac8c94",
  "vehicleLocation":{
    "id":1,
    "latitude":"32.778889",
    "longitude":"-91.919243"
  },
  "fullName":"Jim",
  "email":"jim@jim.com"
}
```

Screenshot-3: Navigate to the "Community" page to identify the vehicle IDs of other users. Try to access messages posted by various users and subsequently inspect the response. It is observed that "vehicle id" attribute is disclosed of other users in the response.



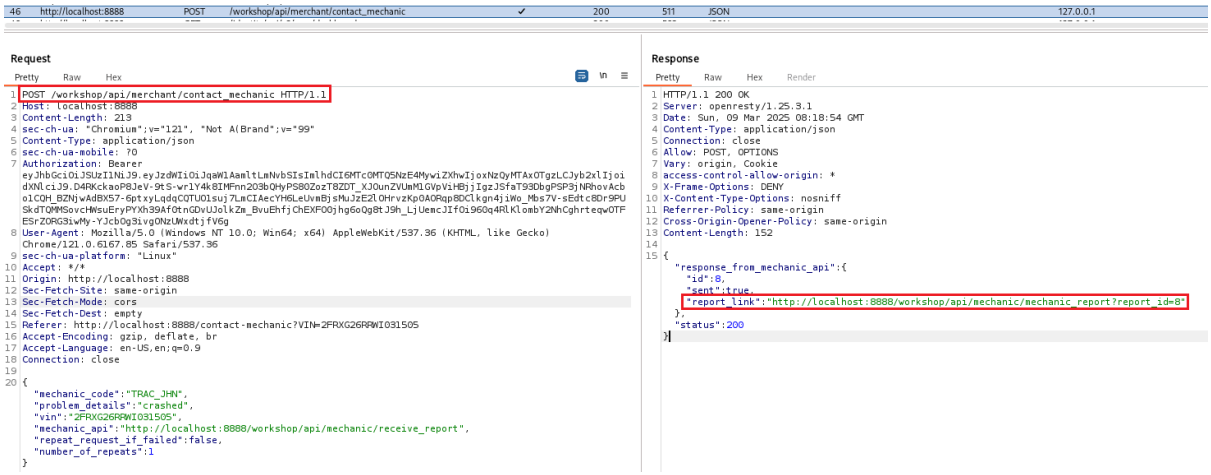


Screenshot-4: With the vehicle ID in hand, I forwarded the request to the API endpoint responsible for retrieving vehicle locations to Burp Suite's repeater. Using the vehicle ID associated with another user, I successfully gained access to the location details of their vehicle.

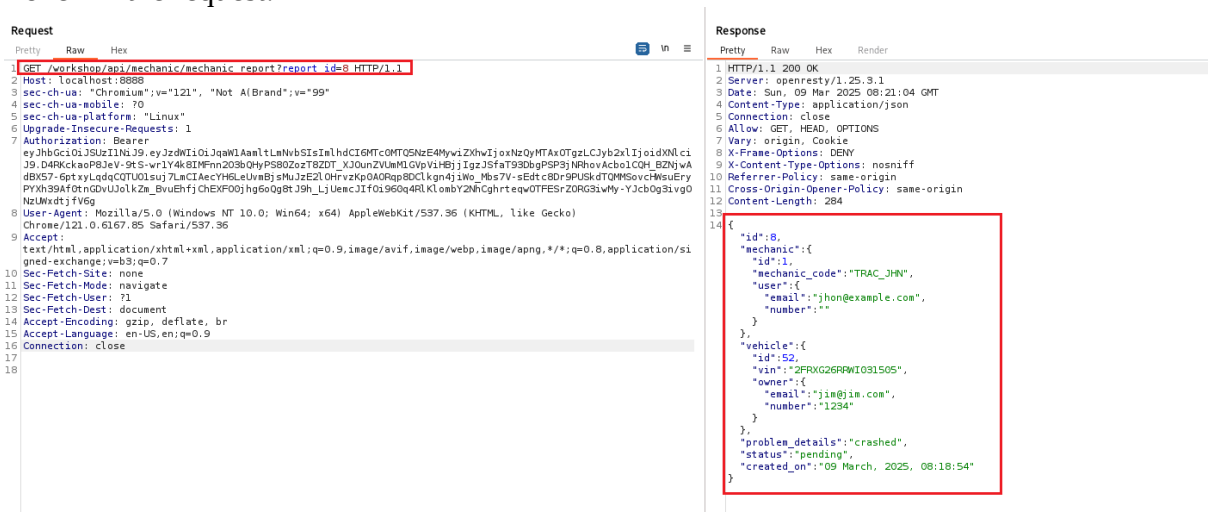


Challenge 2 — Access mechanic reports of other users

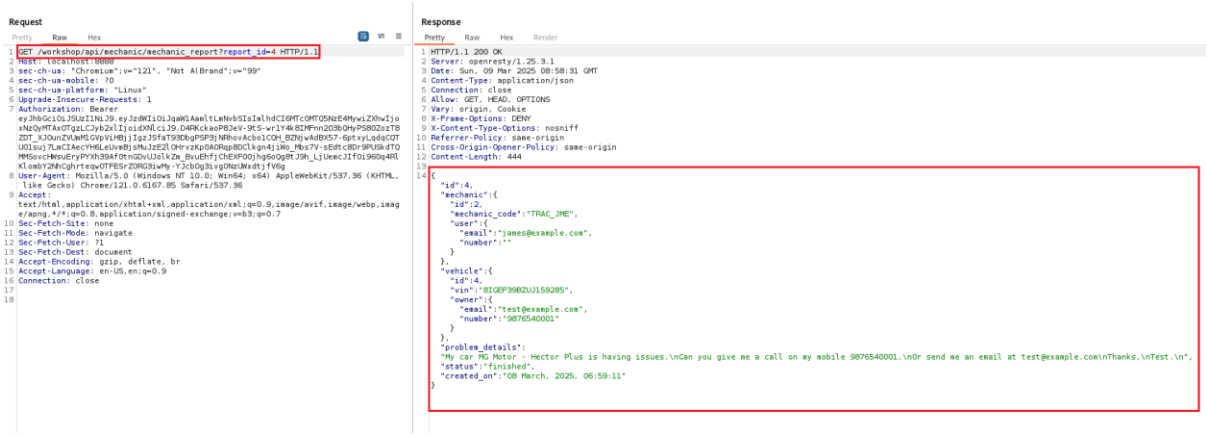
Screenshot-1: Navigate to “Contact Mechanic” page, fill out all details and submit the request. It is observed in the response section that a link revealed to access a mechanic report which is included a “report_id” parameter.



Screenshot-2: Try to open the report by using the earlier mentioned link after adding JWT Token in the request.



Screenshot-3: Now try to test it out by attempting to access the report associated with “id=4” and I successfully gained access to another user’s mechanic report.

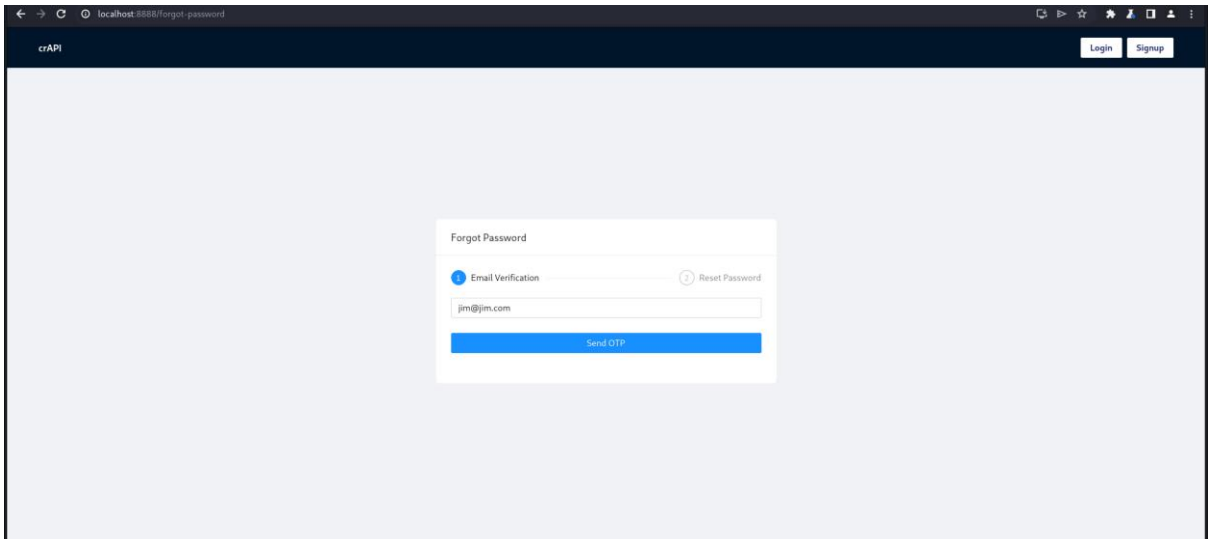


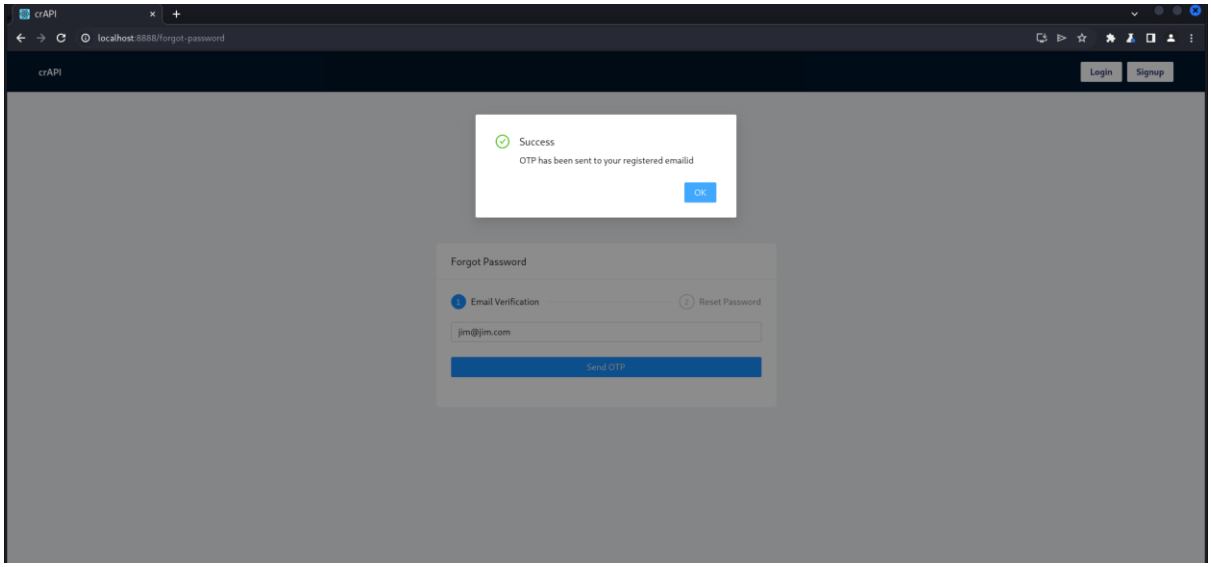
Broken User Authentication

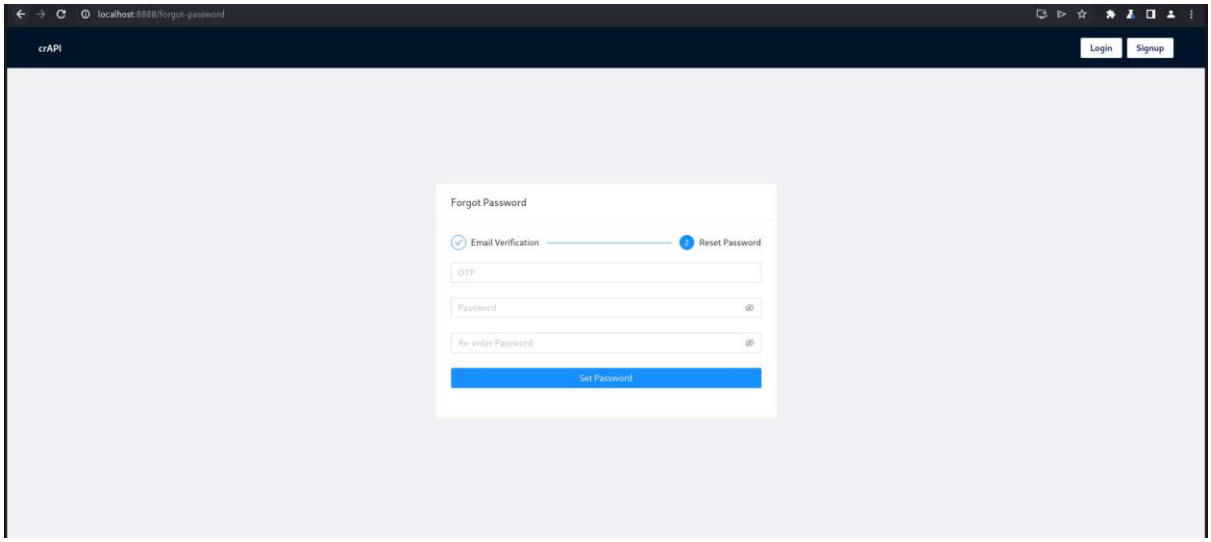
Broken user authentication is a critical security issue when an application fails to properly authenticate or authorize users, potentially allowing unauthorized individuals to access restricted resources or perform actions they shouldn't have access to.

Challenge 3 — Reset the password of a different user

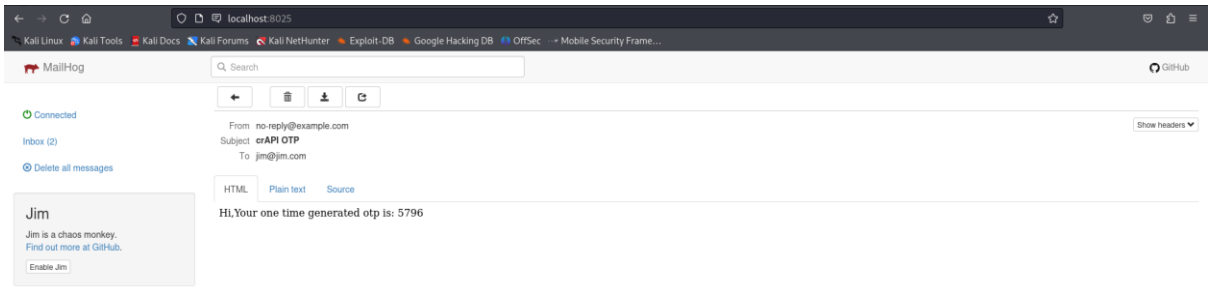
Screenshot-1: I noticed a "Forgot Password" option on the login page. After providing my email on the "Forgot Password" page, an OTP was promptly dispatched to my email.



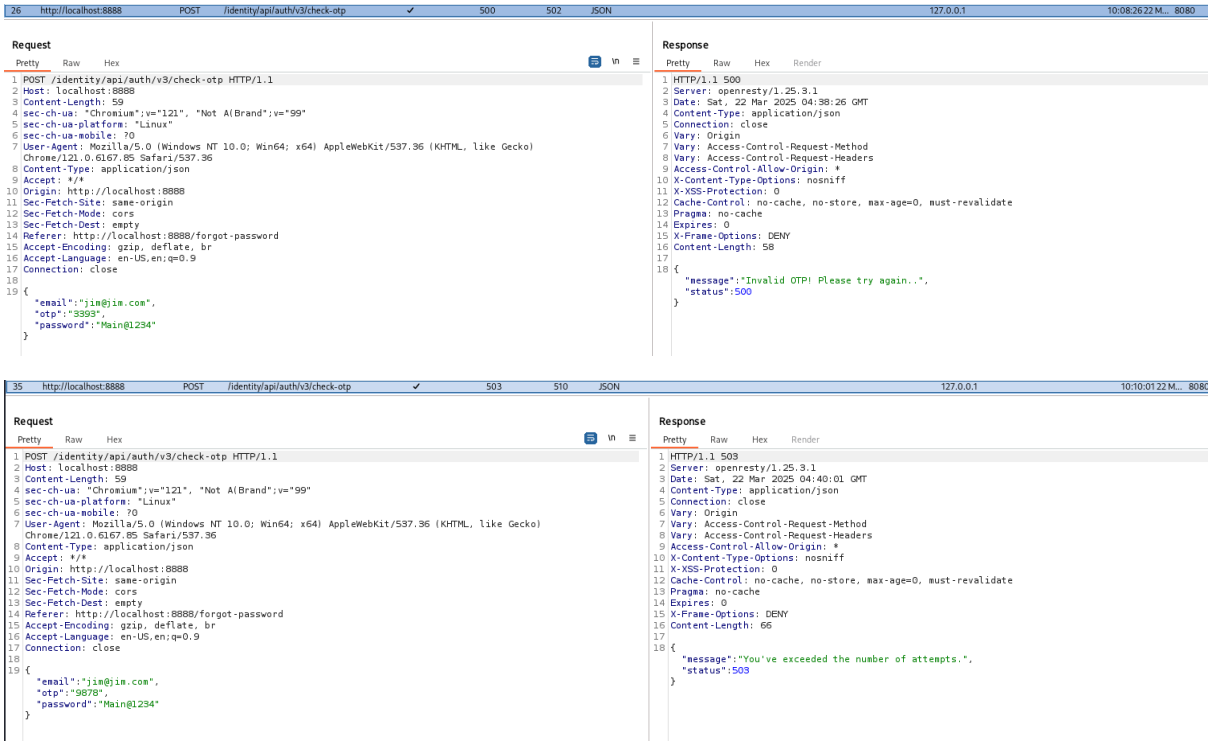




Screenshot-2: I accessed my email via http://localhost:8025 to retrieve the OTP.



Screenshot-3: I intentionally given multiple wrong OTPs and then I received a response indicating that my attempts had exceeded the limit.



Request

```
POST /identity/api/auth/v3/check-otp HTTP/1.1
Host: localhost:8888
Content-Length: 59
sec-ch-ua: "Chromium";v="121", "Not A(Brand";v="99"
sec-ch-ua-platform: "Linux"
sec-ch-ua-mobile: ?0
User-Agent: Mozilla/5.0 (Windows NT 10.0; Win64; x64) AppleWebKit/537.36 (KHTML, like Gecko)
Chrome/121.0.6167.85 Safari/537.36
Content-Type: application/json
Accept: */*
Origin: http://localhost:8888
Sec-Fetch-Site: same-origin
Sec-Fetch-Mode: cors
Sec-Fetch-Dest: empty
Referer: http://localhost:8888/forgot-password
Accept-Encoding: gzip, deflate, br
Accept-Language: en-US,en;q=0.9
Connection: close

{
  "email":"jim@jim.com",
  "otp":"3393",
  "password":"Main@1234"
}
```

Response

```
HTTP/1.1 500
Server: openresty/1.25.3.1
Date: Sat, 22 Mar 2025 04:38:26 GMT
Content-Type: application/json
Connection: close
Vary: Origin
Vary: Access-Control-Request-Method
Vary: Access-Control-Request-Headers
Access-Control-Allow-Origin: *
X-Content-Type-Options: nosniff
X-XSS-Protection: 0
Cache-Control: no-cache, no-store, max-age=0, must-revalidate
Pragma: no-cache
Expires: 0
X-Frame-Options: DENY
Content-Length: 58

{
  "message":"Invalid OTP! Please try again..",
  "status":500
}
```


Request

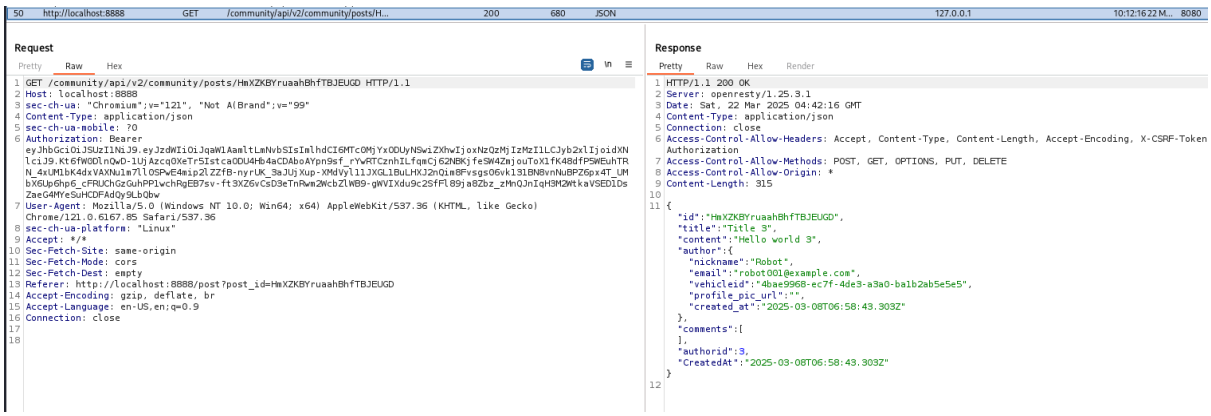
```
POST /identity/api/auth/v3/check-otp HTTP/1.1
Host: localhost:8888
Content-Length: 59
sec-ch-ua: "Chromium";v="121", "Not A(Brand";v="99"
sec-ch-ua-platform: "Linux"
sec-ch-ua-mobile: ?0
User-Agent: Mozilla/5.0 (Windows NT 10.0; Win64; x64) AppleWebKit/537.36 (KHTML, like Gecko)
Chrome/121.0.6167.85 Safari/537.36
Content-Type: application/json
Accept: */*
Origin: http://localhost:8888
Sec-Fetch-Site: same-origin
Sec-Fetch-Mode: cors
Sec-Fetch-Dest: empty
Referer: http://localhost:8888/forgot-password
Accept-Encoding: gzip, deflate, br
Accept-Language: en-US,en;q=0.9
Connection: close

{
  "email":"jim@jim.com",
  "otp":"9878",
  "password":"Main@1234"
}
```

Response

```
HTTP/1.1 503
Server: openresty/1.25.3.1
Date: Sat, 22 Mar 2025 04:40:01 GMT
Content-Type: application/json
Connection: close
Vary: Origin
Vary: Access-Control-Request-Method
Vary: Access-Control-Request-Headers
Access-Control-Allow-Origin: *
X-Content-Type-Options: nosniff
X-XSS-Protection: 0
Cache-Control: no-cache, no-store, max-age=0, must-revalidate
Pragma: no-cache
Expires: 0
X-Frame-Options: DENY
Content-Length: 66

{
  "message":"You've exceeded the number of attempts.",
  "status":503
}
```

Screenshot-4: I observed that this endpoint utilized version v3, prompting me to investigate version v2, hoping it lacked the exact protection mechanism. To my surprise, v2 did indeed lack limitations on the number of attempts.

I obtained the email addresses of other users from the community page. I proceeded to generate OTPs for another user.

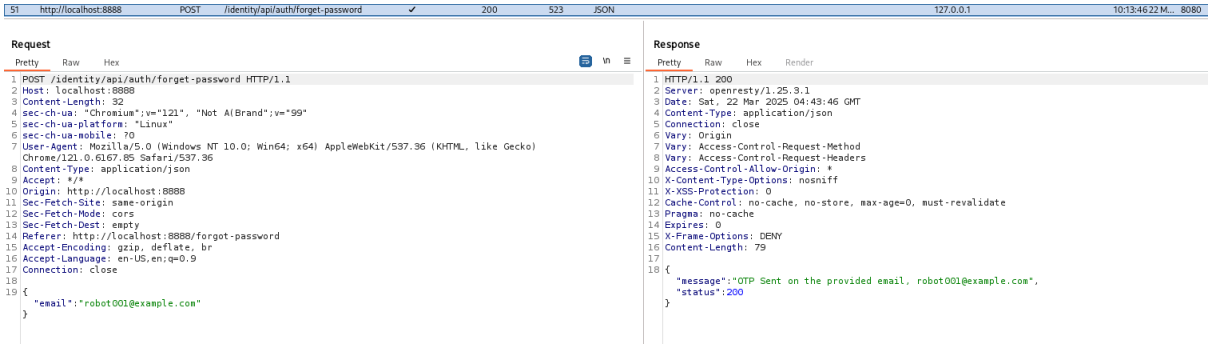


Request

```
GET /community/api/v2/community/posts/HmXZKBYruaahBhfTBJEUGD HTTP/1.1
Host: localhost:8888
sec-ch-ua: "Chromium";v="121", "Not A(Brand";v="99"
Content-Type: application/json
sec-ch-ua-mobile: ?0
Authorization: Bearer
eyJhbGciOiJSUzI1NiJ9.eyJzdWIiOiJqaW1AamltLmNvbSIsImlhdCI6MTc0MjYxODUyNSwiZXhwIjoxNzQzMjIzMzI1LCJyb2xlIjoidXN
lciJ9.Kt6fW0DlnQwD-1UjAzcqOXeTr5IstcaODU4Hb4aCDAboAYpn9sf_rYwRTCznhILfqmCj62NBKjfeSW4ZmjouToX1fK48dfP5WEuhTR
N_4xUMlbK4dxVAXNu1m7llOSPwE4mip2lZZfB-nyrUK_3aJUjXup-XMdVyl1lJXGL1BuLHXJ2nQim8FvsgsO6vk131BN8vnNuBPZ6px4T_UM
bX6UpGhp6_cFRUChQzGuhPPlwchRqEB7sv-ft3XZ6vCsD3eTnRwm2WcbZlWB9-gWVIXdu9c2SfFl89ja8Zbz_zMnQJnIqH3M2WtkaVSED1Ds
ZaeGAMfeSuHCDFAdQy9LbQbw
User-Agent: Mozilla/5.0 (Windows NT 10.0; Win64; x64) AppleWebKit/537.36 (KHTML, like Gecko)
Chrome/121.0.6167.85 Safari/537.36
sec-ch-ua-platform: "Linux"
Accept: */*
Sec-Fetch-Site: same-origin
Sec-Fetch-Mode: cors
Sec-Fetch-Dest: empty
Referer: http://localhost:8888/post?post_id=HmXZKBYruaahBhfTBJEUGD
Accept-Encoding: gzip, deflate, br
Accept-Language: en-US,en;q=0.9
Connection: close
```

Response

```
HTTP/1.1 200 OK
Server: openresty/1.25.3.1
Date: Sat, 22 Mar 2025 04:42:16 GMT
Content-Type: application/json
Connection: close
Access-Control-Allow-Headers: Accept, Content-Type, Content-Length, Accept-Encoding, X-CSRF-Token
Authorization
Access-Control-Allow-Methods: POST, GET, OPTIONS, PUT, DELETE
Access-Control-Allow-Origin: *
Content-Length: 315

{
  "id":"HmXZKBYruaahBhfTBJEUGD",
  "title":"Title 3",
  "content":"Hello world 3",
  "author":{
    "nickname":"Robot",
    "email":"robot001@example.com",
    "vehicleid":"4bae9968-ec7f-4de3-a3a0-ba1b2ab5e5e5",
    "profile_pic_url":"",
    "created_at":"2025-03-08T06:58:43.303Z"
  },
  "comments":[
  ],
  "authorid":3,
  "CreatedAt":"2025-03-08T06:58:43.303Z"
}
```



Request

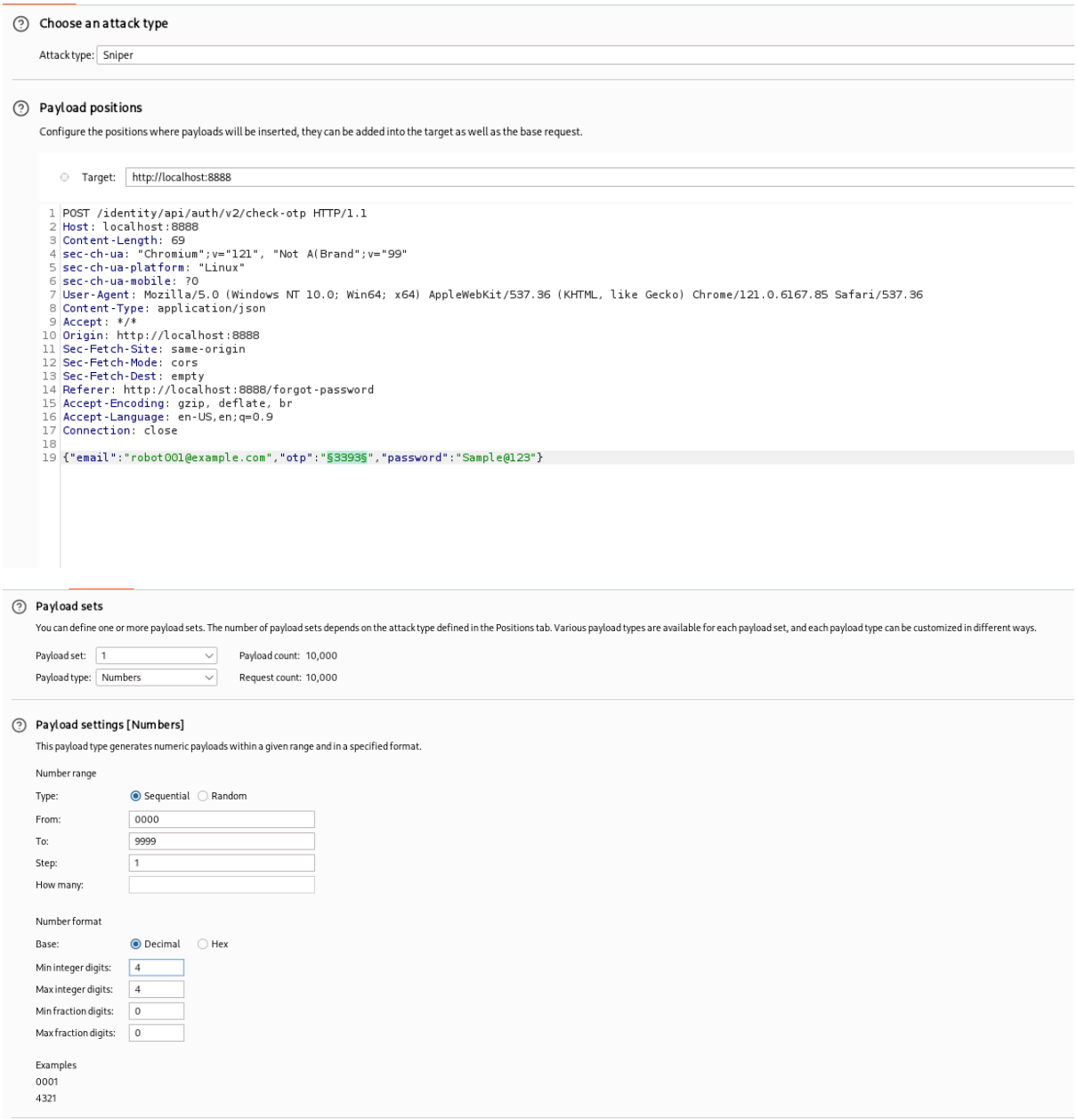
```
POST /identity/api/auth/forget-password HTTP/1.1
Host: localhost:8888
Content-Length: 32
sec-ch-ua: "Chromium";v="121", "Not A(Brand";v="99"
sec-ch-ua-platform: "Linux"
sec-ch-ua-mobile: ?0
User-Agent: Mozilla/5.0 (Windows NT 10.0; Win64; x64) AppleWebKit/537.36 (KHTML, like Gecko)
Chrome/121.0.6167.85 Safari/537.36
Content-Type: application/json
Accept: */*
Origin: http://localhost:8888
Sec-Fetch-Site: same-origin
Sec-Fetch-Mode: cors
Sec-Fetch-Dest: empty
Referer: http://localhost:8888/forgot-password
Accept-Encoding: gzip, deflate, br
Accept-Language: en-US,en;q=0.9
Connection: close

{
  "email":"robot001@example.com"
}
```

Response

```
HTTP/1.1 200
Server: openresty/1.25.3.1
Date: Sat, 22 Mar 2025 04:43:46 GMT
Content-Type: application/json
Connection: close
Vary: Origin
Vary: Access-Control-Request-Method
Vary: Access-Control-Request-Headers
Access-Control-Allow-Origin: *
X-Content-Type-Options: nosniff
X-XSS-Protection: 0
Cache-Control: no-cache, no-store, max-age=0, must-revalidate
Pragma: no-cache
Expires: 0
X-Frame-Options: DENY
Content-Length: 79

{
  "message":"OTP Sent on the provided email, robot001@example.com",
  "status":200
}
```

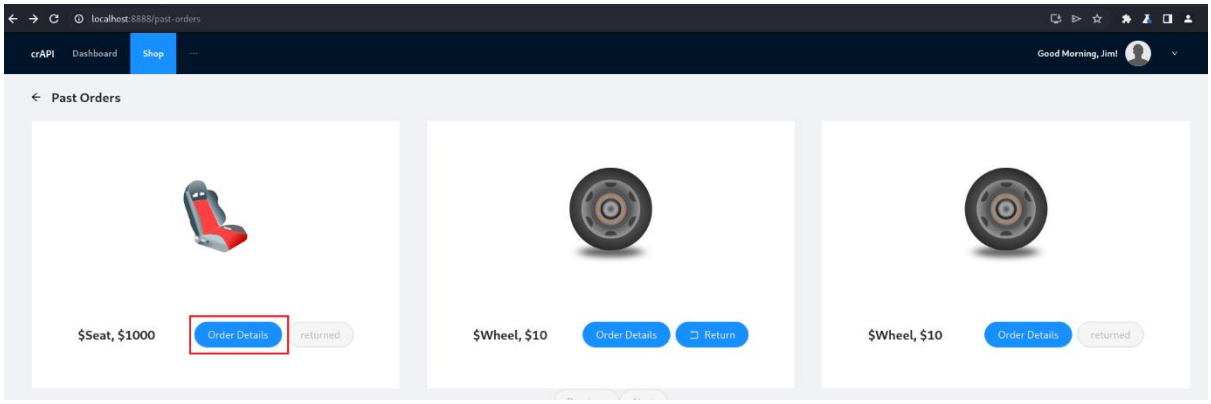
Screenshot-5: To fuzz the OTP, I employed Burp Suite's Intruder function, employing a sniper attack.

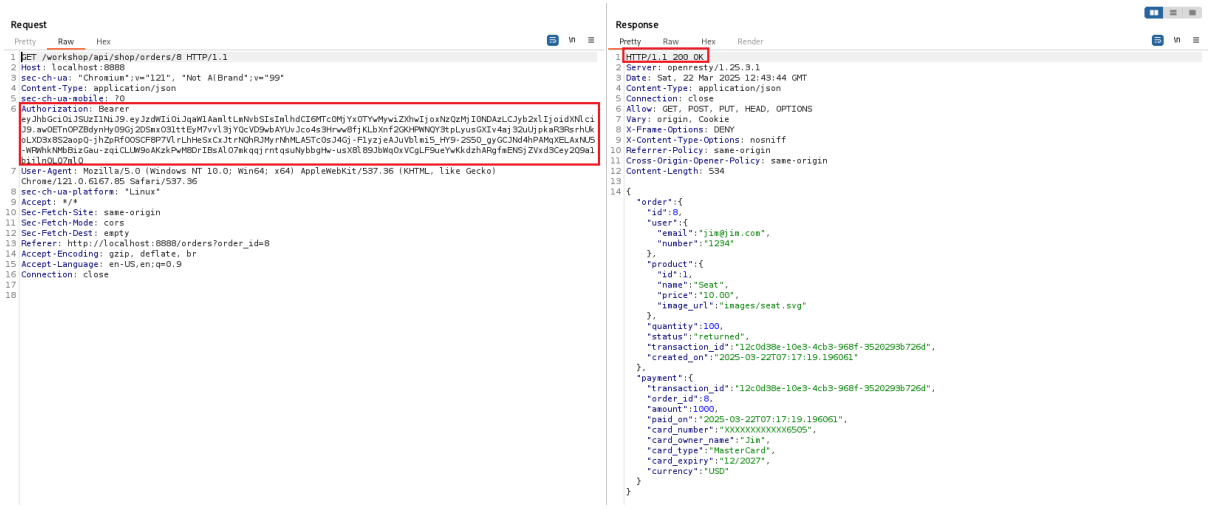
Once I had the Intruder setup in motion and initiated the attack. Finally, I received the response message "OTP verified."



Challenge 4 — Find an endpoint that does not perform authentication checks for a user.

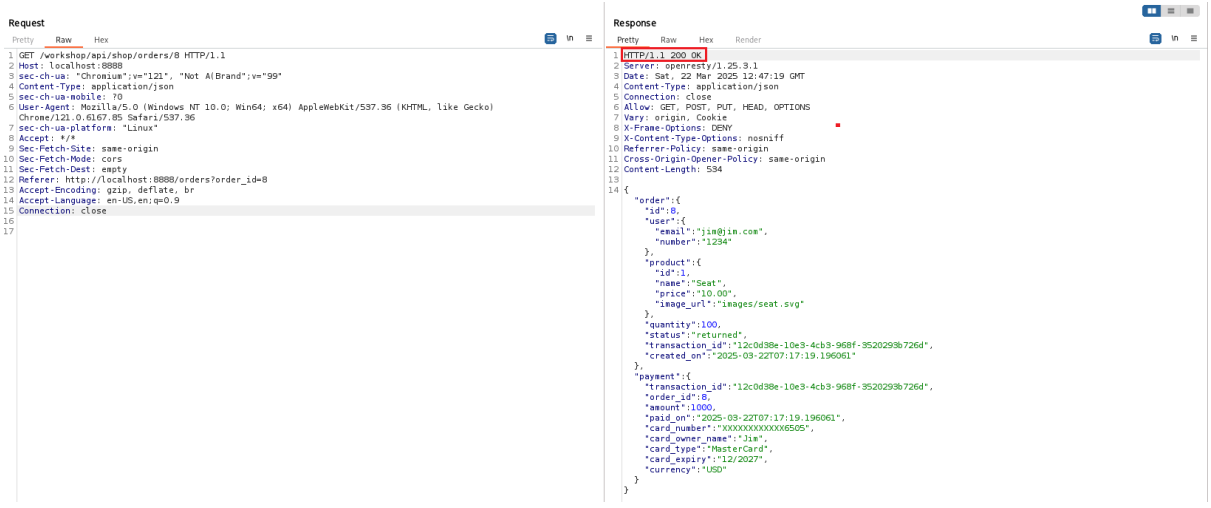
Screenshot-1: Navigate to 'Past Orders' section and click on Order Details tab. It is observed that an 'Authorization Bearer Token' is included in the request and respective successful response message is received.





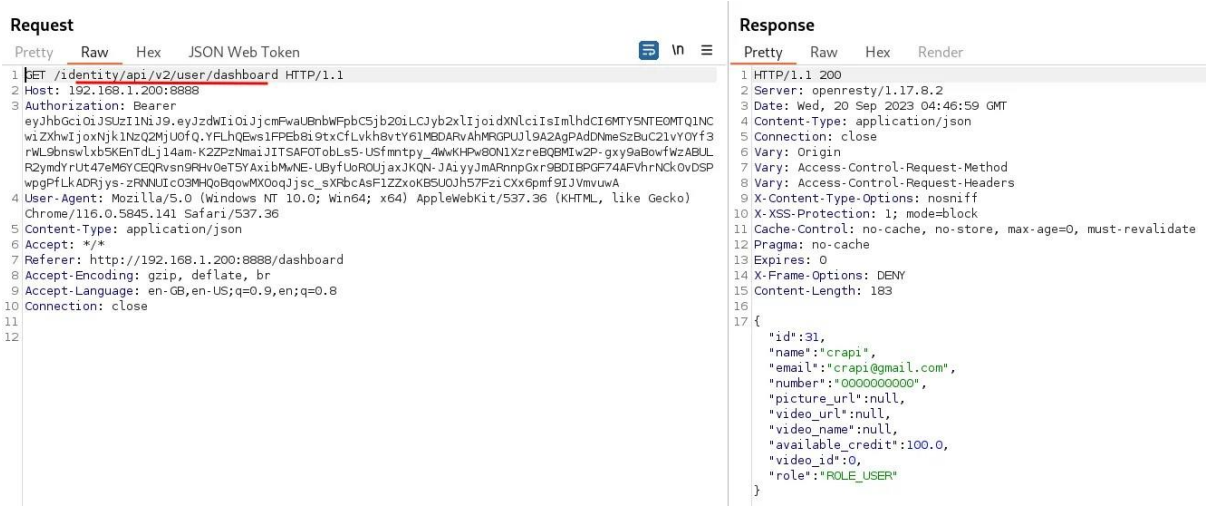
Screenshot-2: It is decided to send a request to this endpoint without including a valid 'Authorization Bearer Token'. It is observed that the successful response message came back, providing us with the order details without requiring any form of authorization.

Note: Also, we can see another user's order details by modifying the ID (8 to 9 or something else) in API endpoint.



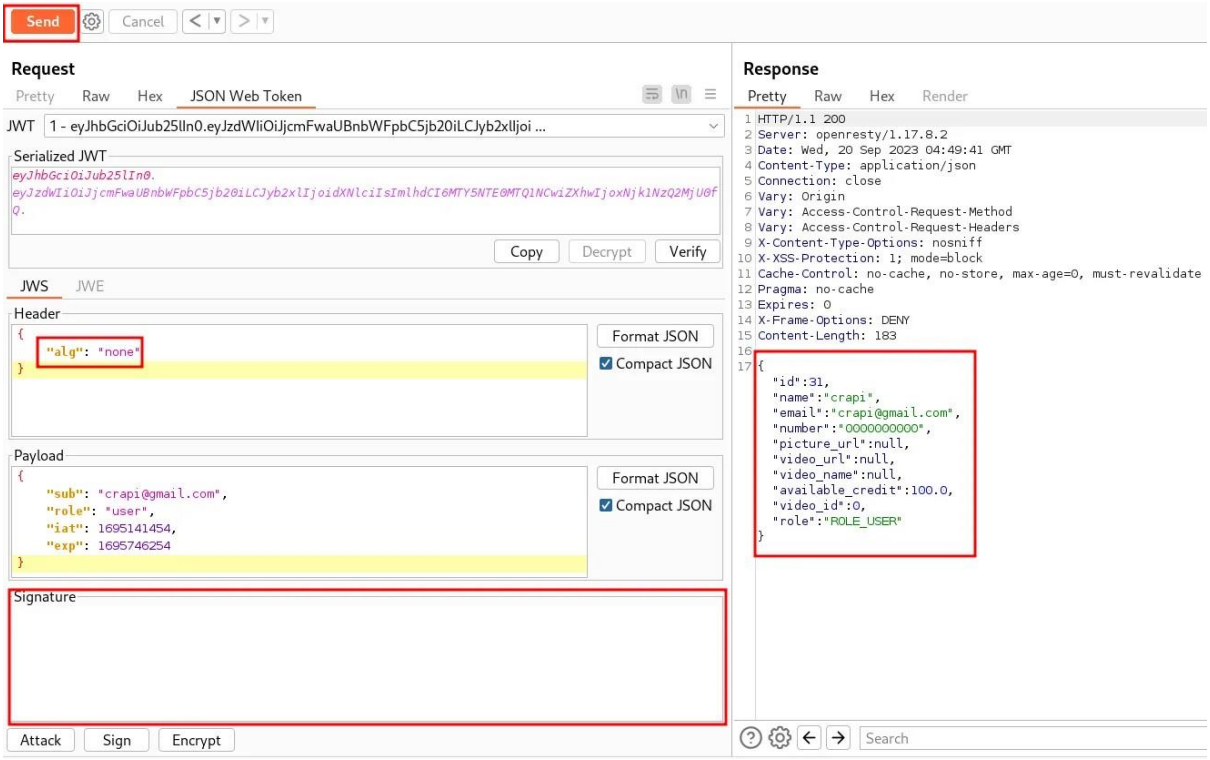
Challenge 5 — Forging Valid JWT Tokens for Full Access

Screenshot-1: Navigate to user dashboard and intercept the request.

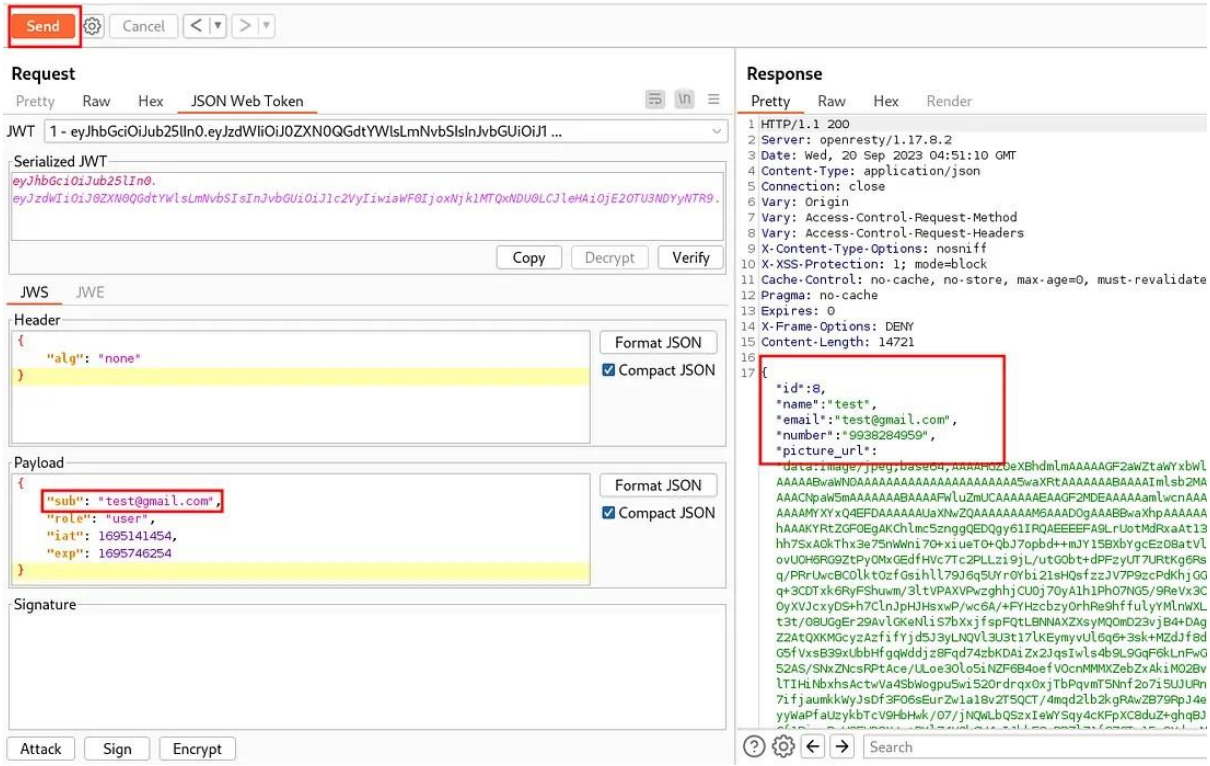


Screenshot-2: Leveraging Burp JWT Editor to forge a valid JWT token, I turned to the Burp JWT Editor. Here's what I did:

- Within the Request, I navigated to the JSON Web Token section.
- I changed the algorithm (alg) from RS256 to "none," effectively disabling the signature.
- With the signature removed, I sent the request, eagerly awaiting for the result.
- The request came back successful, granting me access to my user details without a valid signature.



Screenshot-3: I changed my Email ID to another user's Email ID and sent the request once more. The successful response message with another user's details are revealed, effectively proving that a valid JWT token is forged, circumventing the intended authentication mechanism.

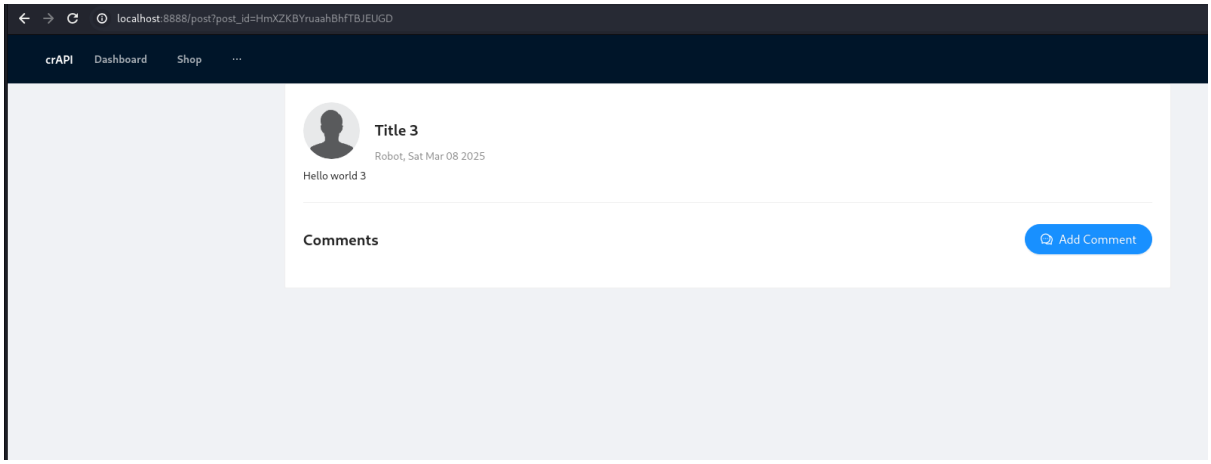


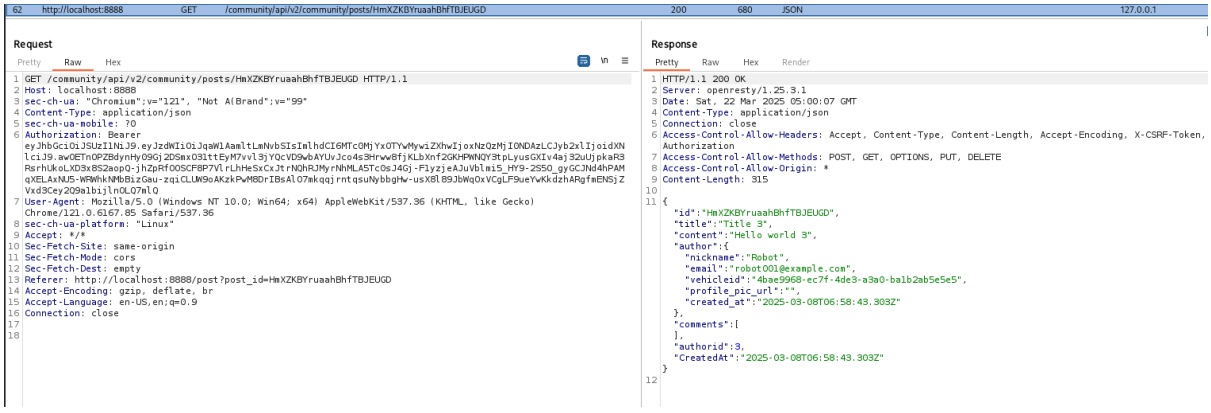
Excessive data exposure

Excessive data exposure is a critical security issue that occurs when sensitive information is unintentionally or improperly disclosed to unauthorized individuals or systems. It can have serious consequences for individuals and organizations, including data breaches, privacy violations, and legal ramifications.

Challenge 6 — Find an API endpoint that leaks sensitive information of other users

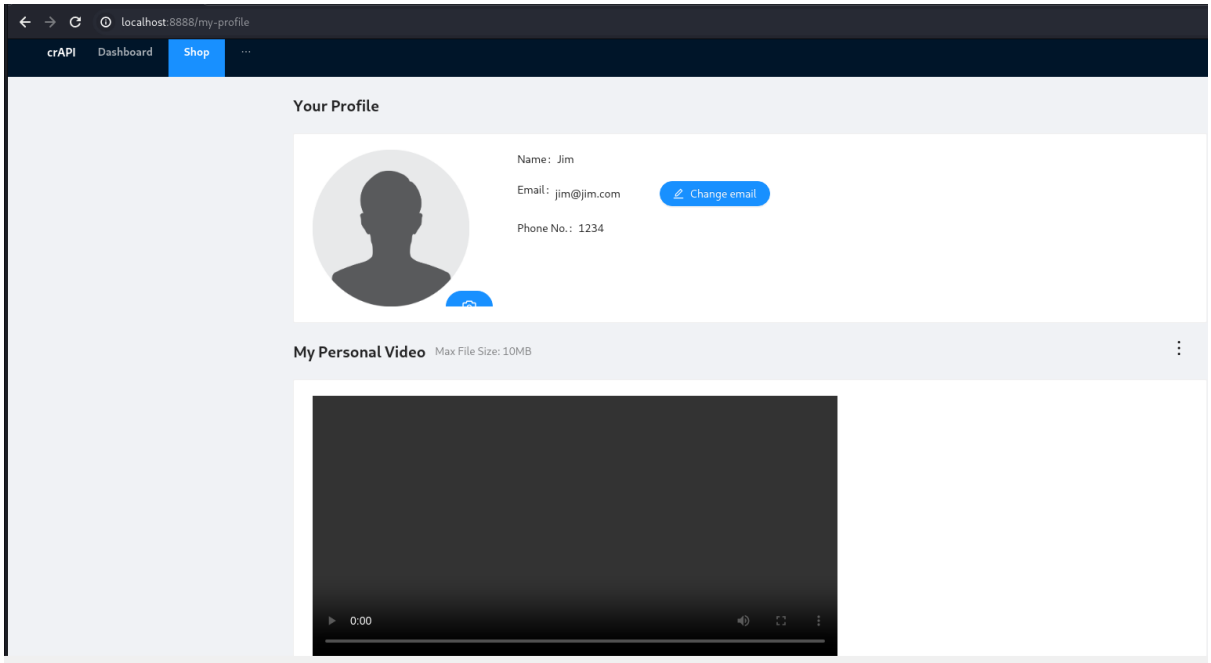
Screenshot-1: While exploring the "Community" page, I made a striking discovery. Upon inspecting the response, I found that it contained detailed information about another user.

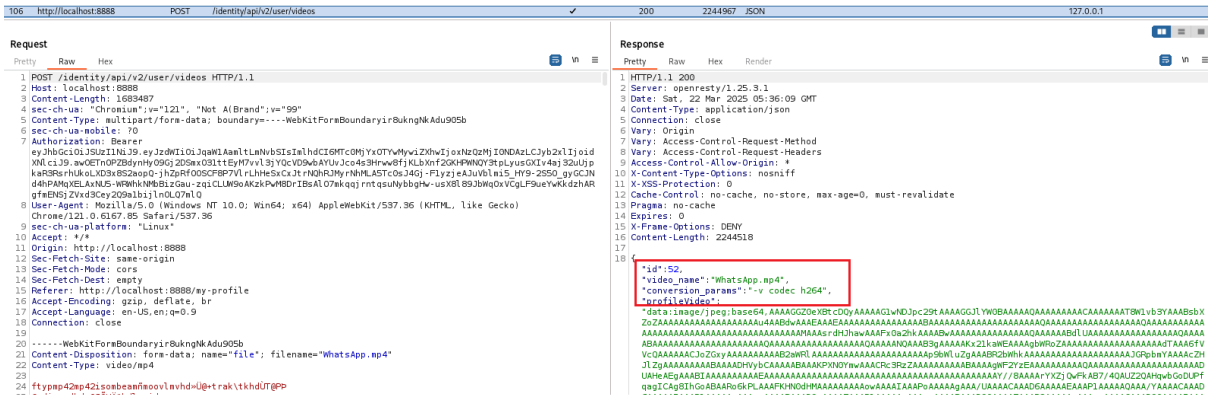




Challenge 7 — Find an API endpoint that leaks an internal property of a video

Screenshot-1: In the process of uploading a video, I made an intriguing discovery. Upon receiving the response, I noticed that it disclosed an internal property of the video.



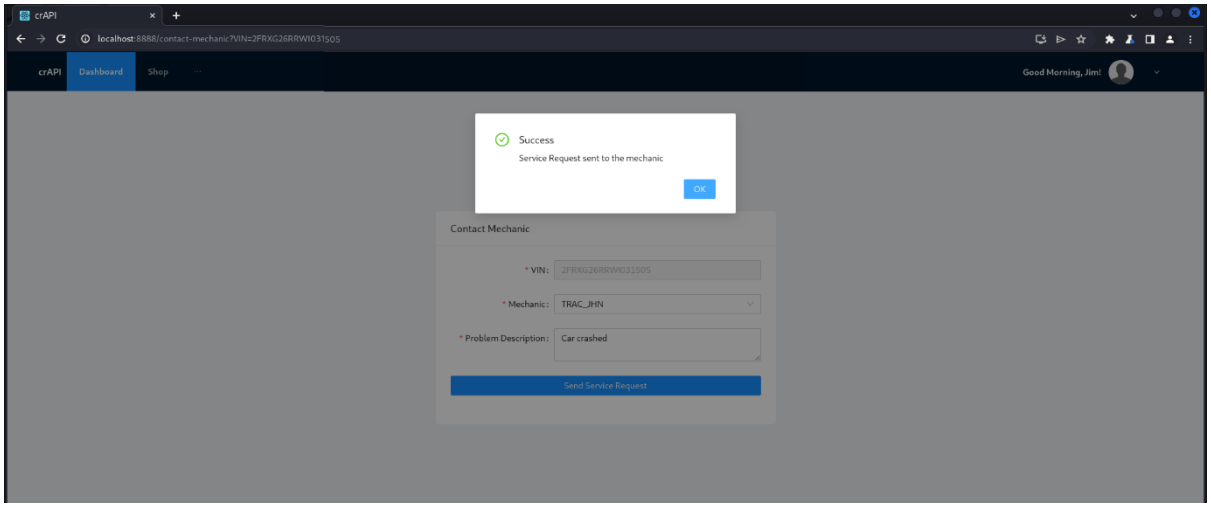


Lack of Resources and Rate Limiting

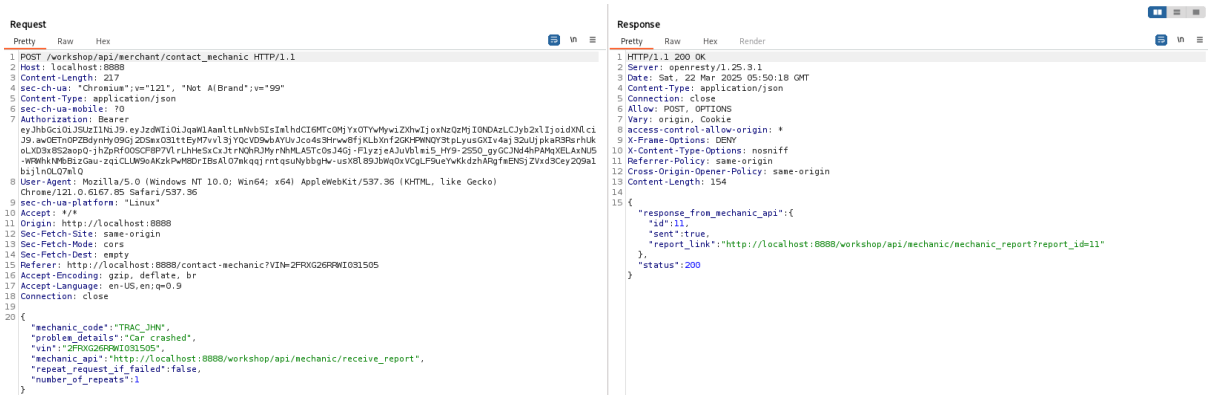
Rate limiting is a technique used in computer systems to control the rate at which requests or actions are allowed. It's often implemented to prevent abuse, protect resources, and maintain system stability. Rate limiting can be applied in various contexts, such as API rate limiting, login attempts, or even in protecting against DDoS attacks.

Challenge 8 — Perform a layer 7 Denial-of-Service (DoS) using ‘contact mechanic’ feature.

Screenshot-1: While exploring the application, I encountered the ‘Contact Mechanic’ feature, which allowed users to fill out a form for assistance.

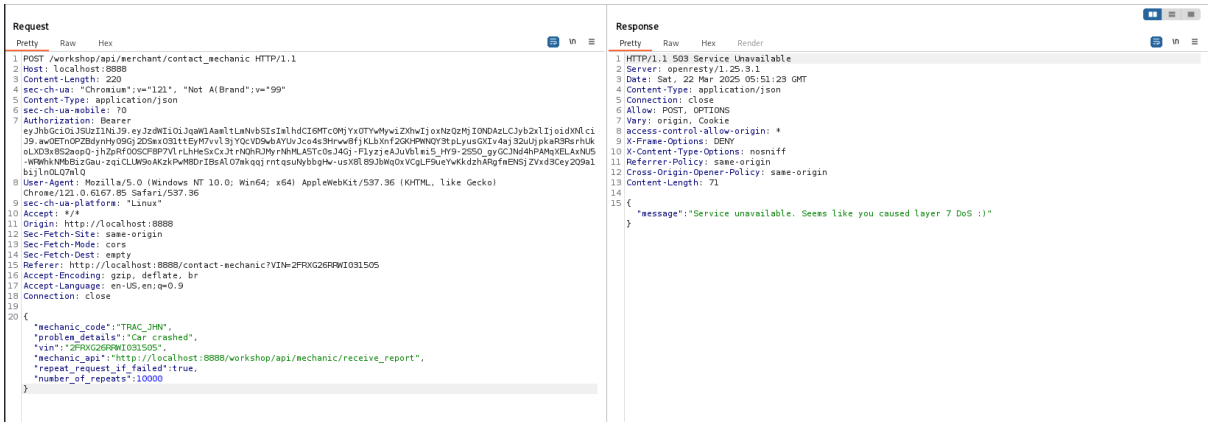


Screenshot-2: Upon examining the request, I noticed the parameter **“repeat_request_if_failed”** was set to “false,” with **“number_of_repeats”** at a value of 1.



Screenshot-3: By modifying **“repeat_request_if_failed”** to “true” and setting **“number_of_repeats”** to a whopping 10,000, I initiated a request that overwhelmed the system.

The response I received clearly indicated a Layer 7 Denial-of-Service (DoS) condition with the message: “Service unavailable. Seems like you caused a Layer 7 DoS :)”

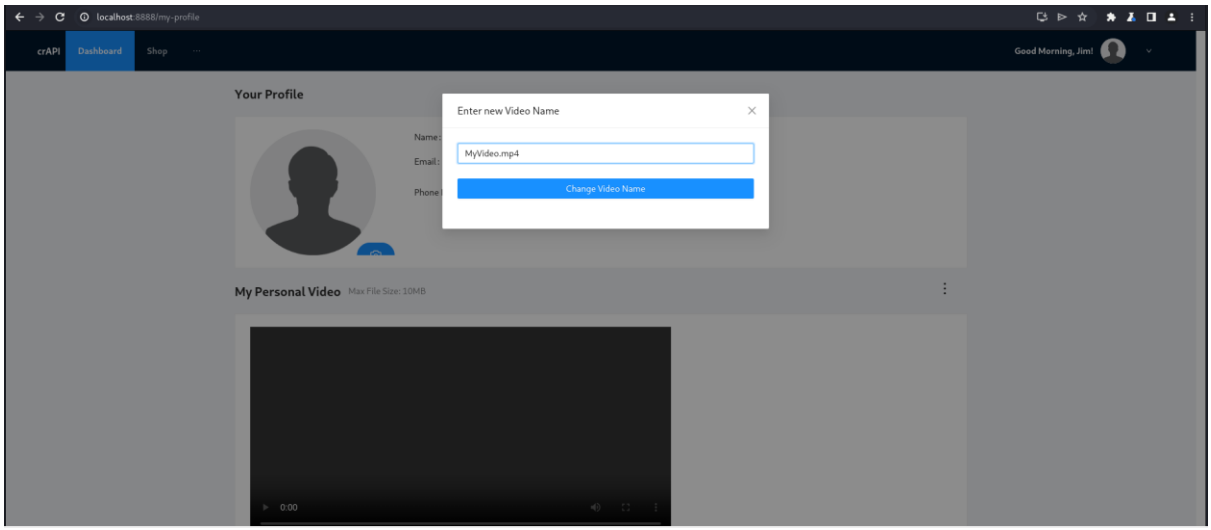


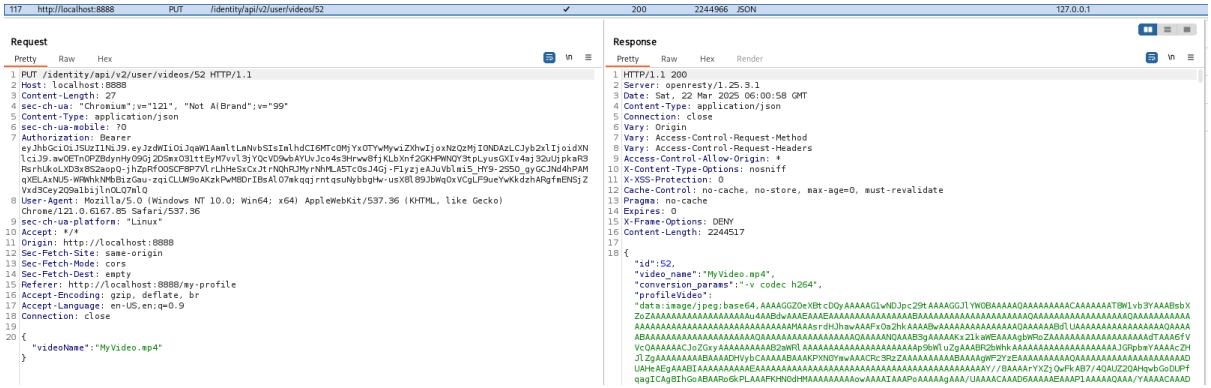
Broken Function Level Authorization (BFLA)

BFLA or Broken Function Level Authorization *is a security vulnerability that occurs when an application or system does not properly enforce access controls at the function or feature level. In other words, it allows users to perform actions or access features that they should not have permission to use.*

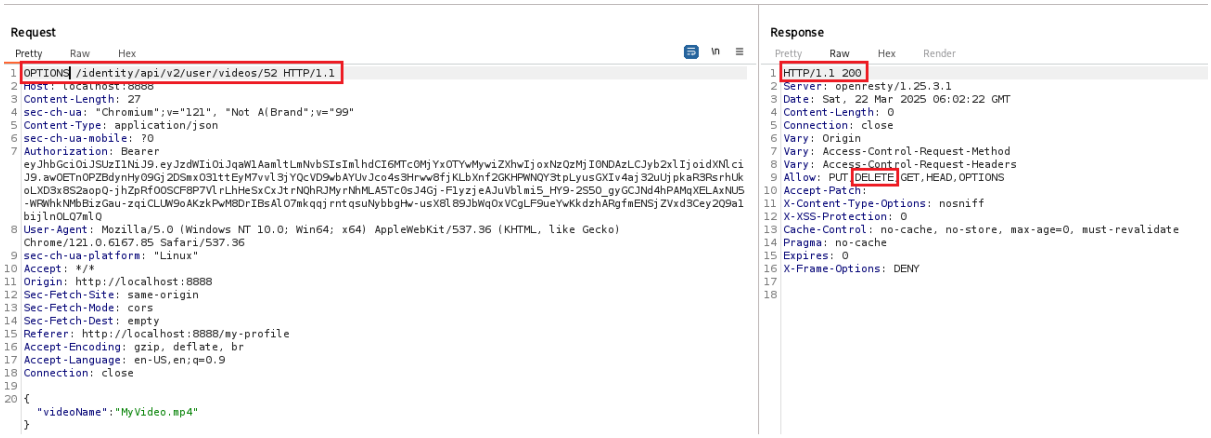
Challenge 9 — Delete a video from your profile.

Screenshot-1: I began by using the “Change Video Name” option, allowing me to modify the name of the video.

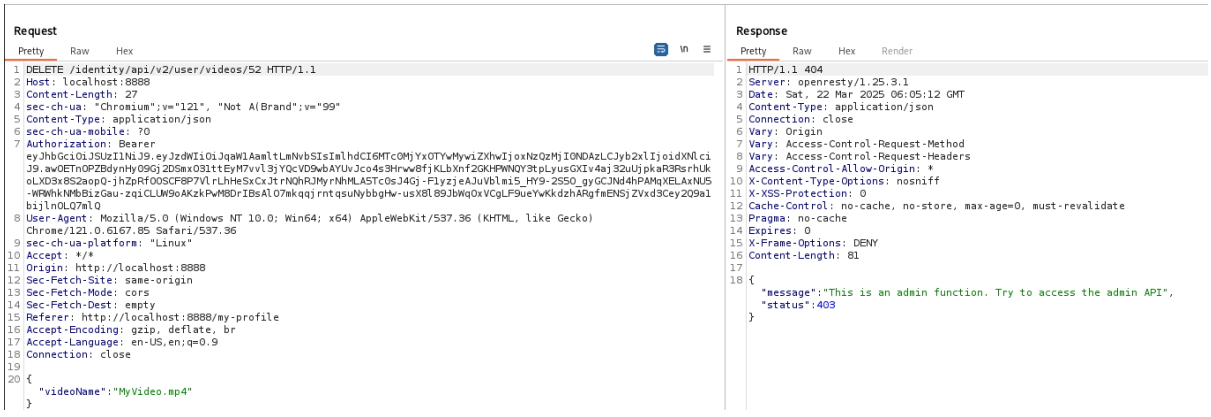




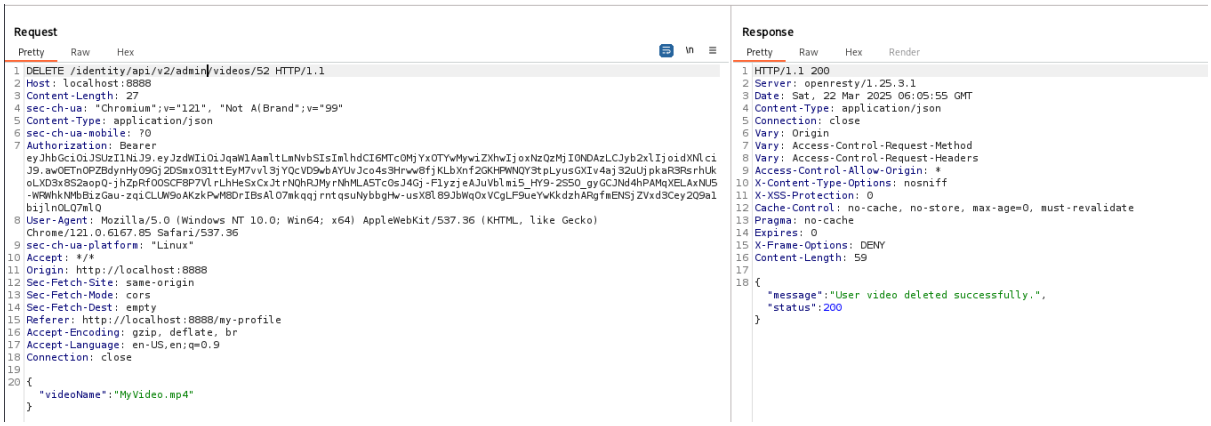
Screenshot-2: Upon renaming the video, I noticed that the PUT method was used. With my interest, prompting me to explore which methods were allowed for this endpoint. To my surprise, I discovered that the DELETE method was permitted.

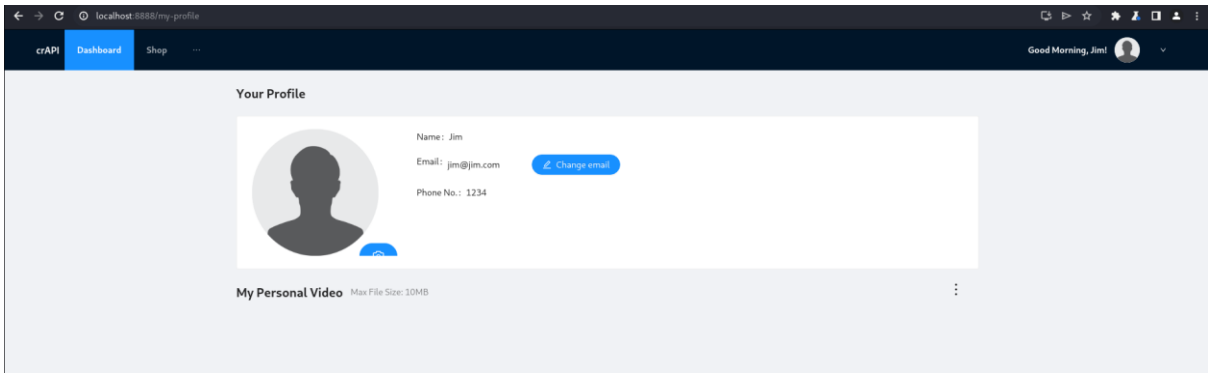


Screenshot-3: Taking advantage of the DELETE method, I initiated a request with DELETE method. The server responded with a message stating, “This is an admin function. Try to access the admin API.”



Screenshot-4: Recognizing the need for admin privileges, I decided to manipulate the request further. By changing the user to “admin” and sending the request, I successfully deleted the video.



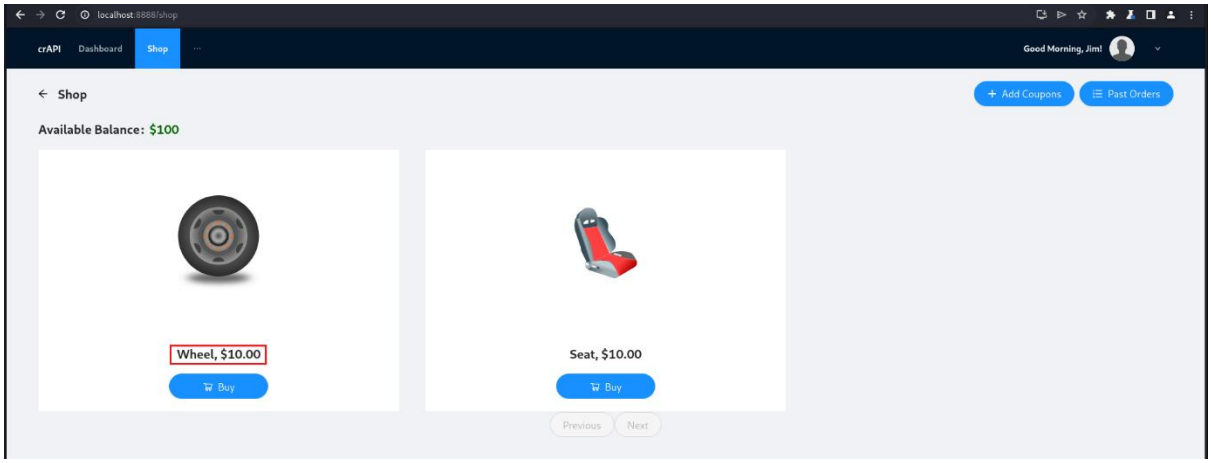


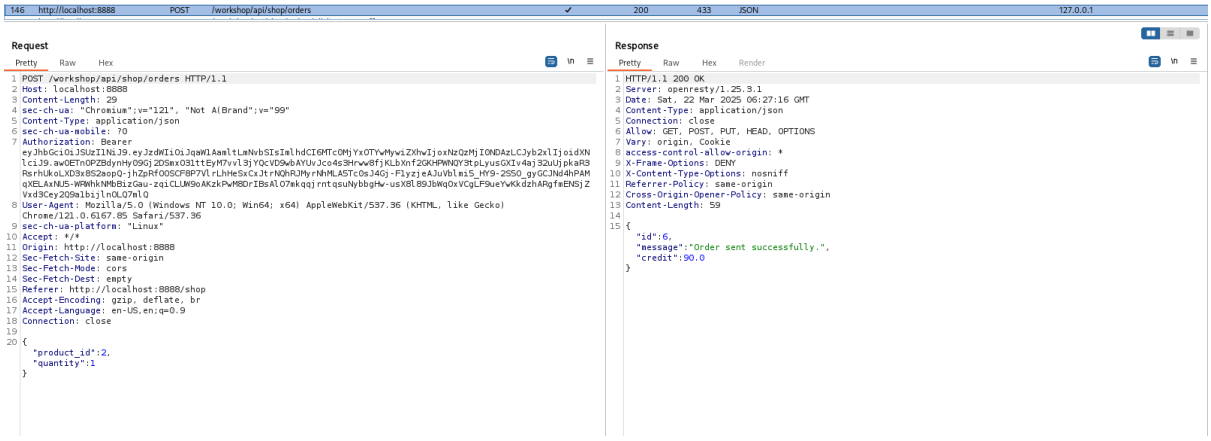
Mass Assignment

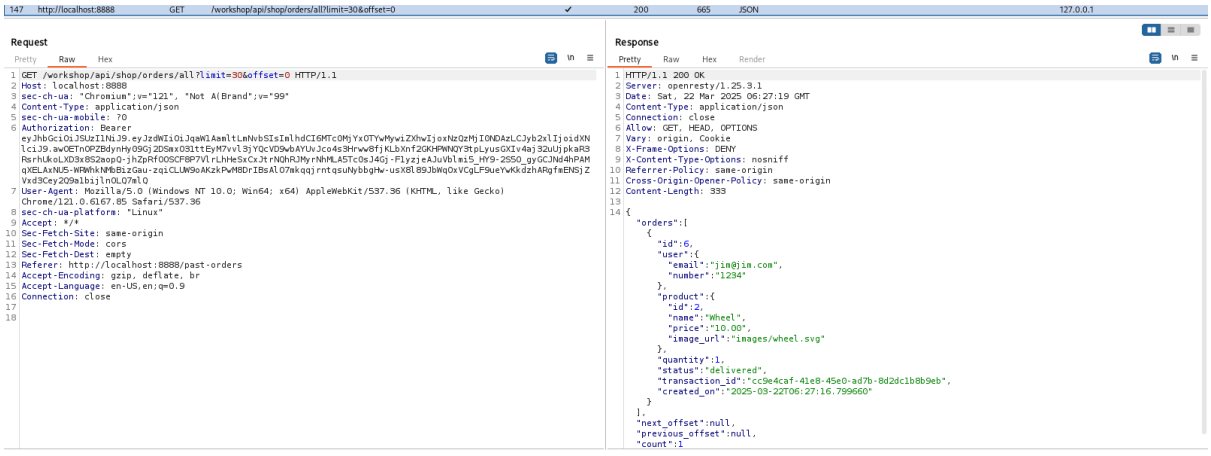
Mass assignment is a security vulnerability that occurs when an attacker can manipulate input data to modify an object's properties, often leading to unauthorized changes in a system. This can happen when developers don't properly validate and sanitize user inputs or fail to restrict which properties can be modified in an object.

Challenge 10 — Get an item for free.

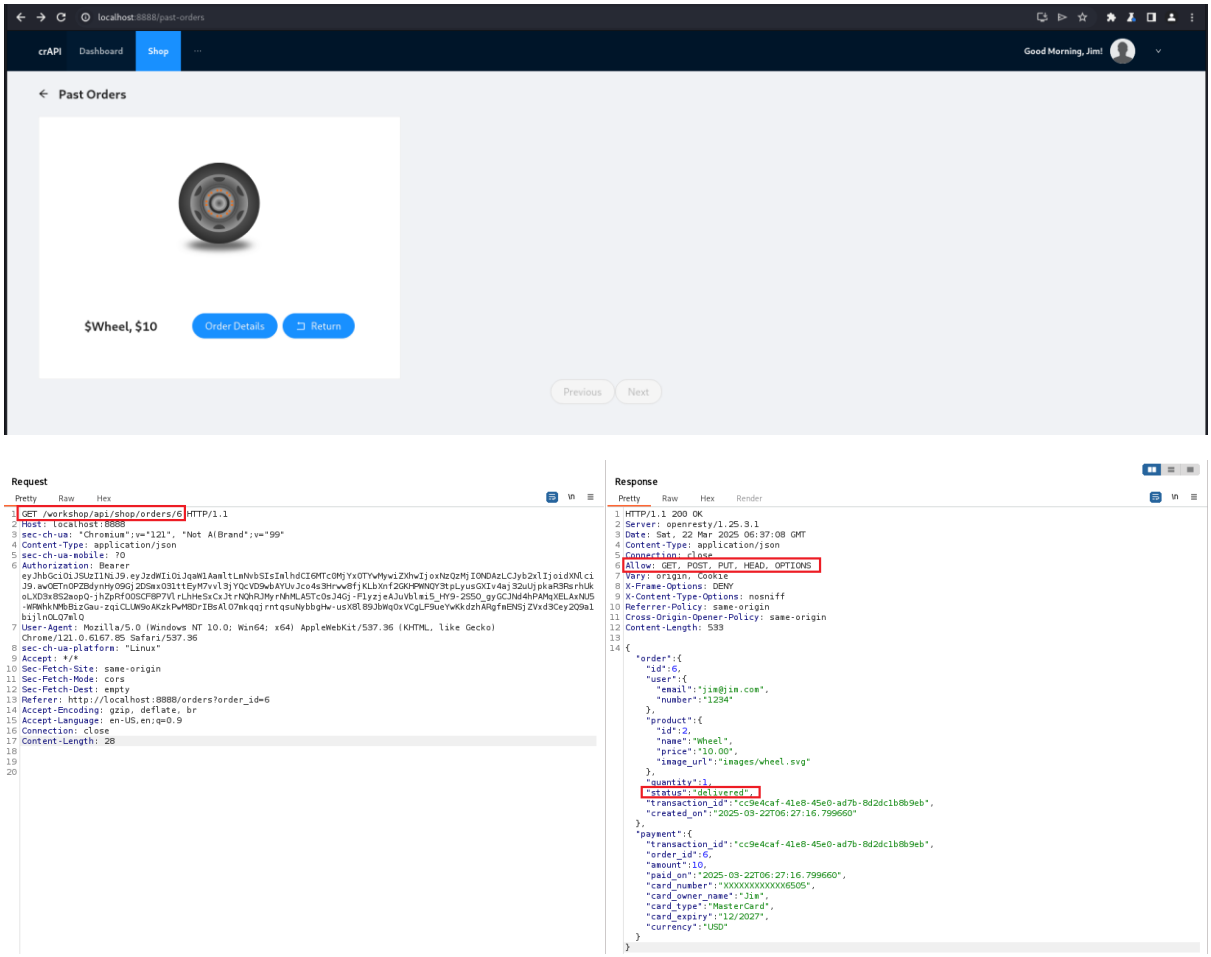
Screenshot-1: We initiated the challenge by exploring the 'shop' page, where we noticed an available balance of $100 and two items: 'Seat' and 'Wheel'. We placed an order and closely examined the request and response from the workshop API. It is also observed that product is delivered.



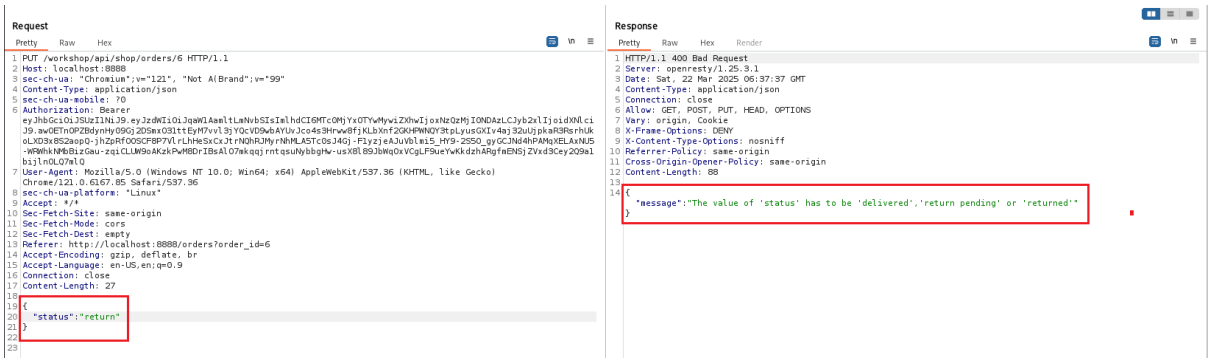




Screenshot-2: Navigate to ‘Past-Orders’ section where a list of all orders for the user are displayed. Now click on ‘Order Details’ tab and closely observe the response where all HTTP allow methods are disclosed.



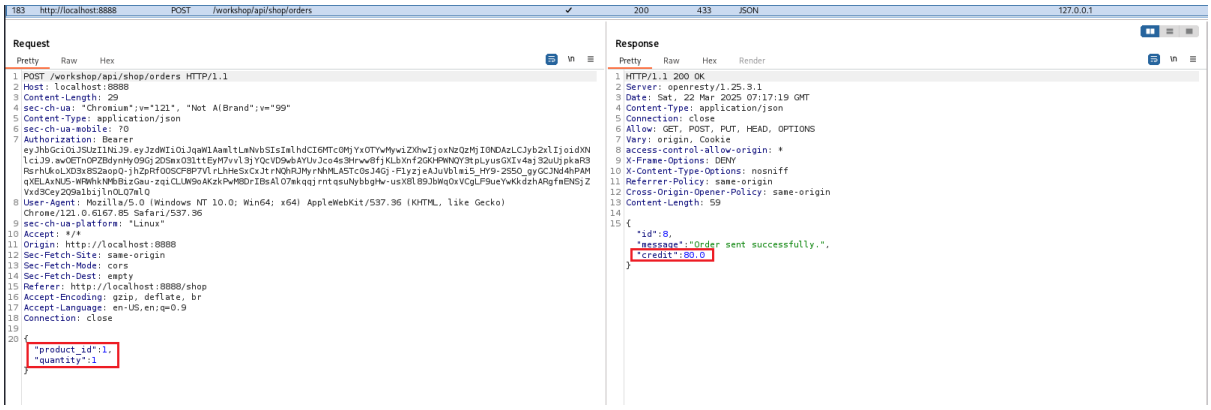
Screenshot-3: Now change the request method from GET to PUT, add the parameter ‘status’ with value ‘return’ in the request body and then send the modified request to the server. A valuable message is revealed in the response. It appeared that the ‘status’ could have only three distinct values ‘delivered’, ‘return pending’ or ‘returned’.



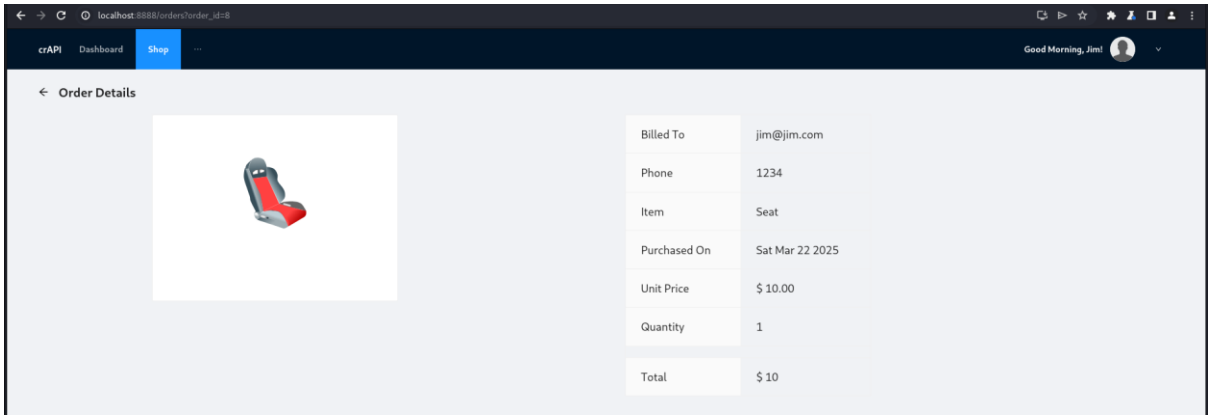
Screenshot-4: We modified the ‘status’ to ‘returned’. The result? The item was successfully returned, and a refund was added to the account.

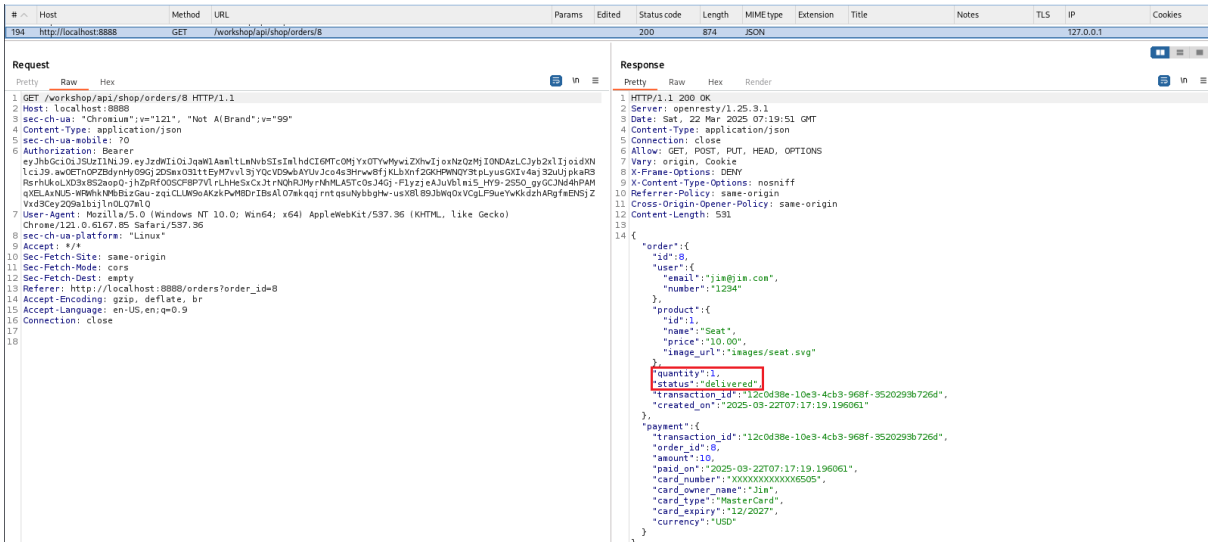
Challenge 11 — Increase your balance by $1,000 or more

Screenshot-1: A new product is ordered from the shop section.

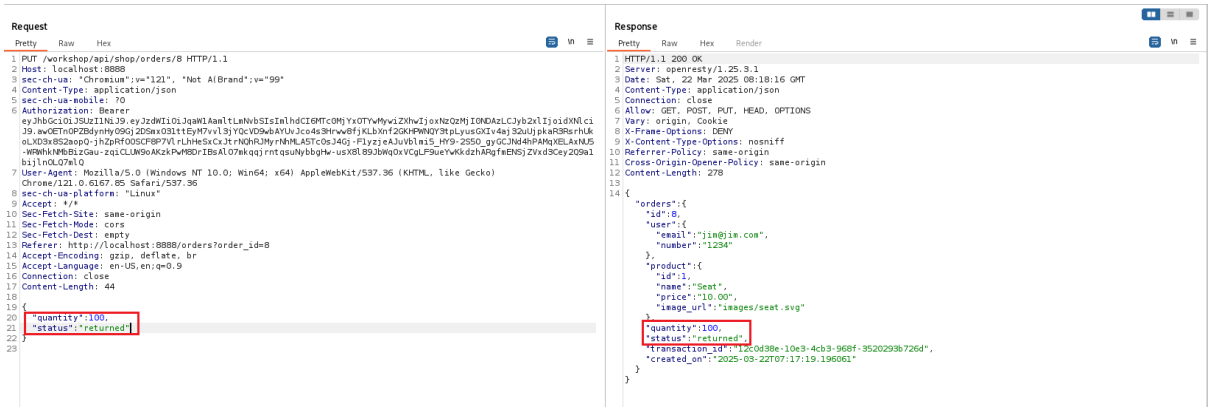


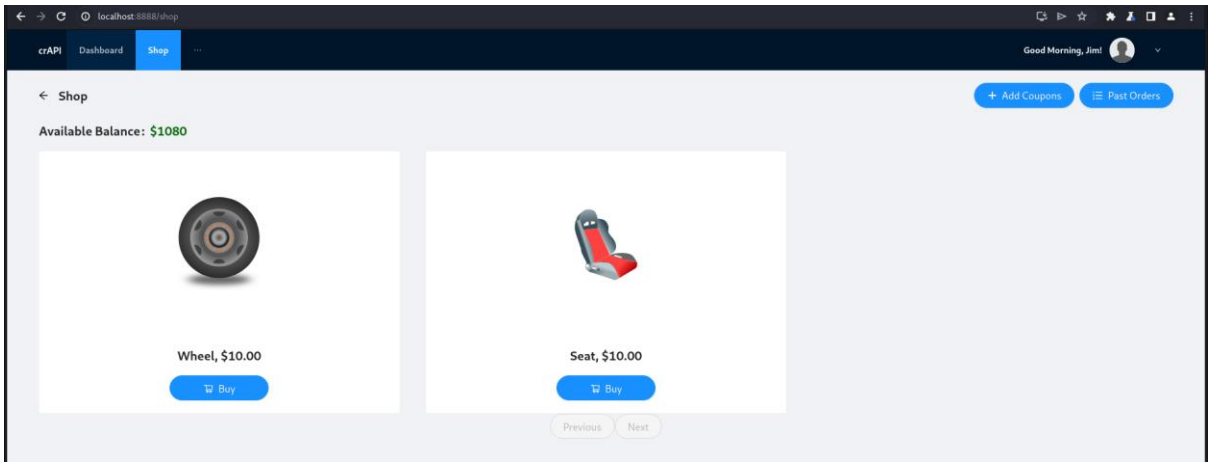
Screenshot-2: Navigate to ‘Past-Orders’ section where a list of all orders for the user are displayed. Now click on ‘Order Details’ tab for the last ordered product and closely observe the response.





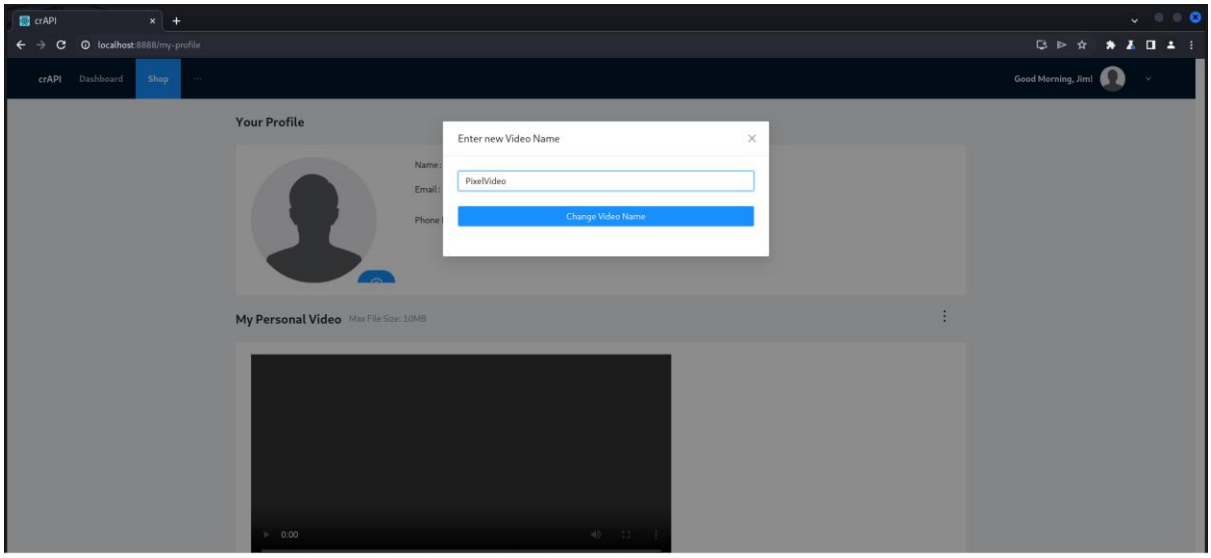
Screenshot-3: Now change the request method from GET to PUT, add the parameters ‘quantity’ to ‘100’ and the ‘status’ to ‘returned’ in the request body and then send the modified request to the server. By performing this, we triggered a refund of $1,000, and same amount is effectively added to our account.



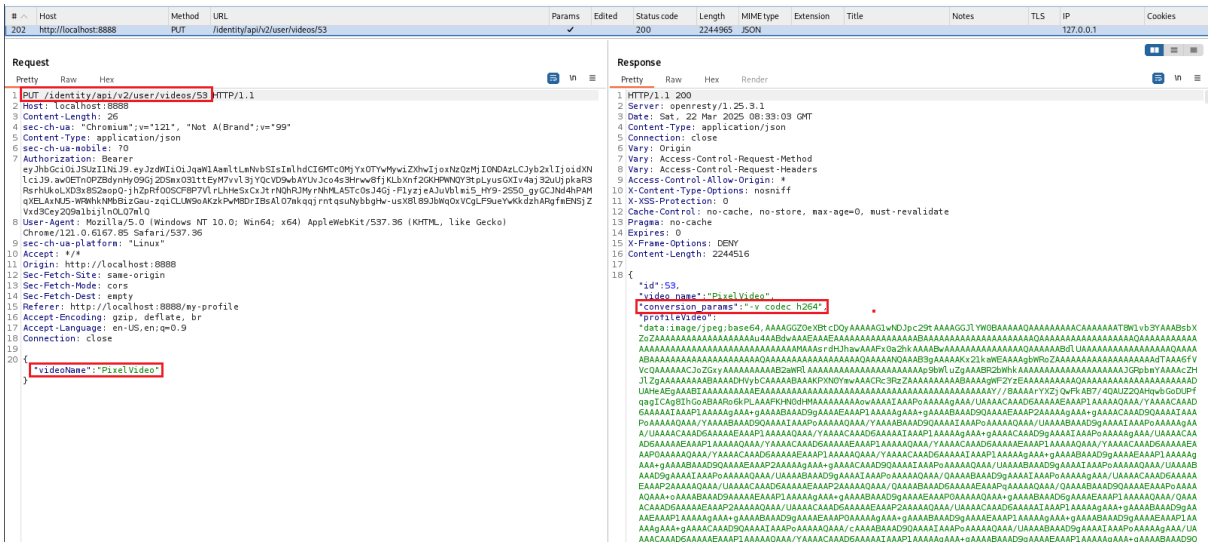


Challenge 12 — Update internal video properties

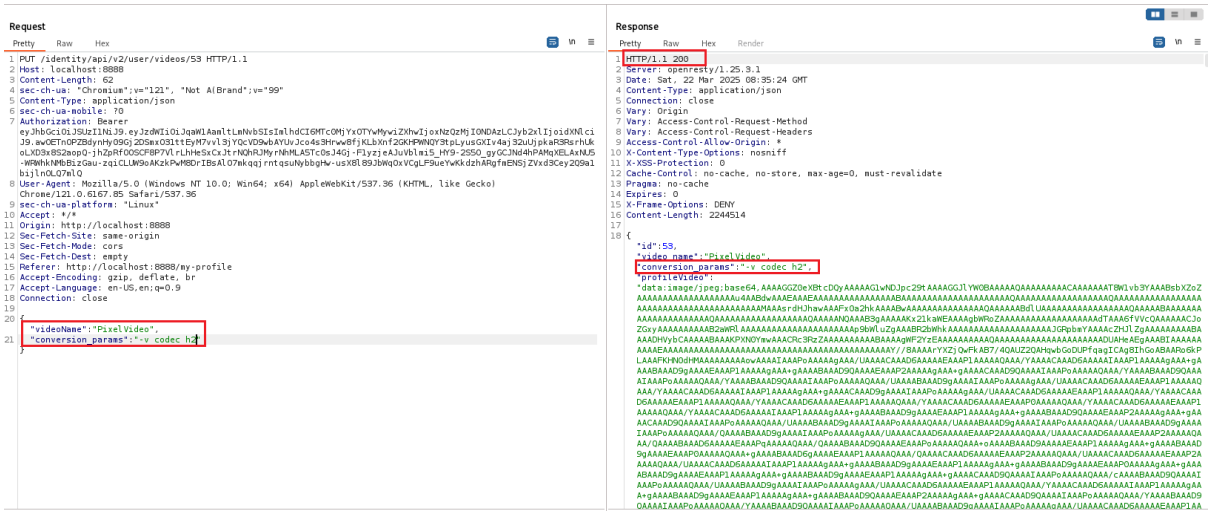
Screenshot-1: Navigate to “Change Video Name” option, which allowed us to modify the video’s name. Upon renaming the video, carefully observe the request and response section.



Screenshot-2: It is noticed that HTTP PUT request method is used and the response provided valuable information, specifically the “conversion_params,” which exposed internal video properties.



Screenshot-3: The parameter “conversion_params” with different value added in the request body and send the crafted request to the server to alter the video properties. It is observed that video properties are changed successfully.



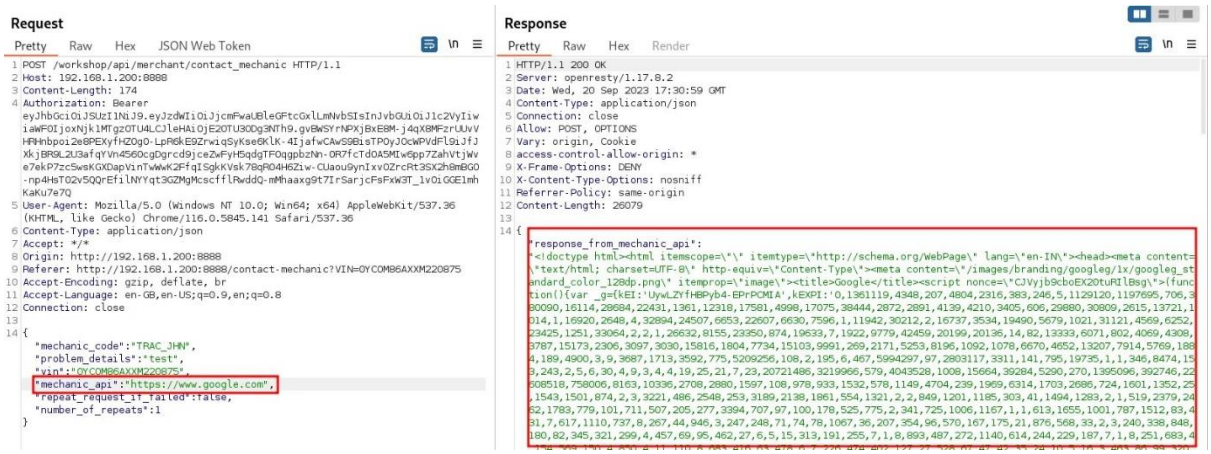
Server-Side Request Forgery (SSRF)

SSRF is a type of security vulnerability that occurs when an attacker can manipulate the requests made by a web application to access resources on the server or other internal systems that they should not have access to.

Challenge 13 — Make crAPI send an HTTP call to “www.google.com" and return the HTTP response.

Screenshot-1: Navigate to ‘Contact Mechanic’ section and submit the service request. It is observed that the request is sent to the server to call the mechanic_api mentioned in request body; and successfully received the report link URL displayed in the response tab.

Screenshot-2: We decided to modify the request URL. By substituting the URL, we successfully prompted the server to make an HTTP call to “www.google.com."



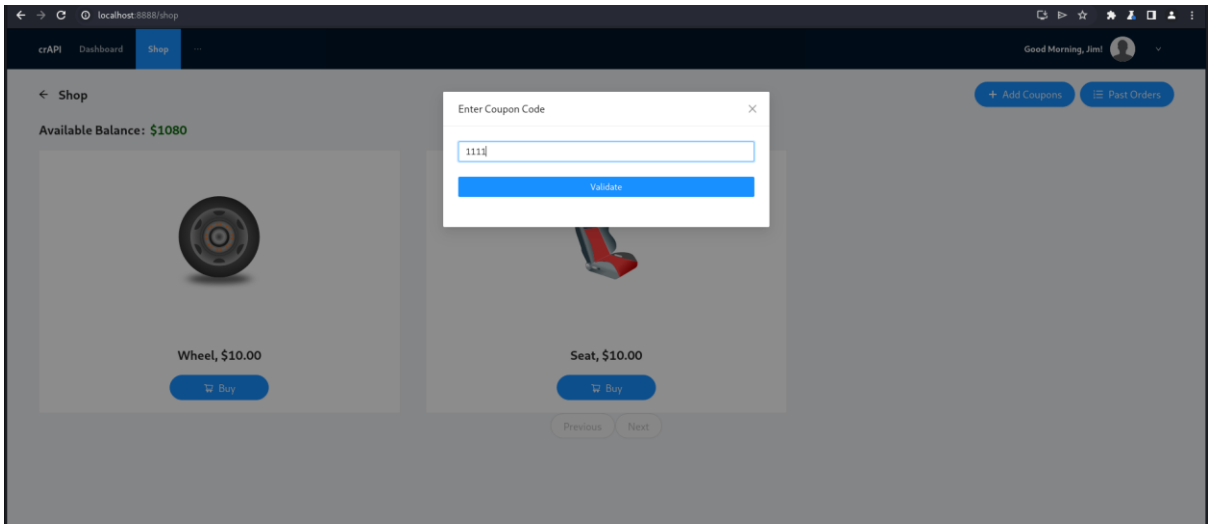
Injection

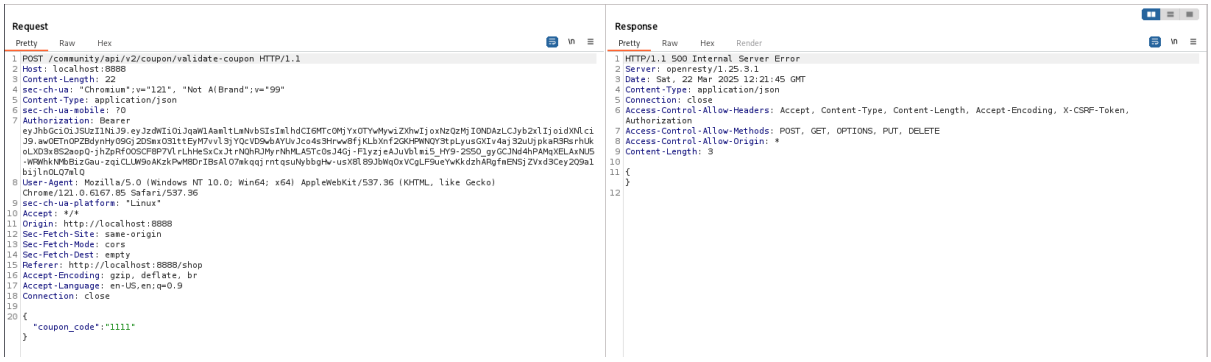
An injection vulnerability occurs when an application fails to properly sanitize or validate user input, allowing attackers to insert malicious code or data that can be interpreted and executed by the application, potentially leading to data breaches, system compromise, or other security issues.

NoSQL injection: *It is a vulnerability that lets a malicious hacker introduce (inject) undesired code into database queries executed by NoSQL databases such as MongoDB, Cassandra, or Redis.*

Challenge 14 — Find a way to get free coupons without knowing the coupon code.

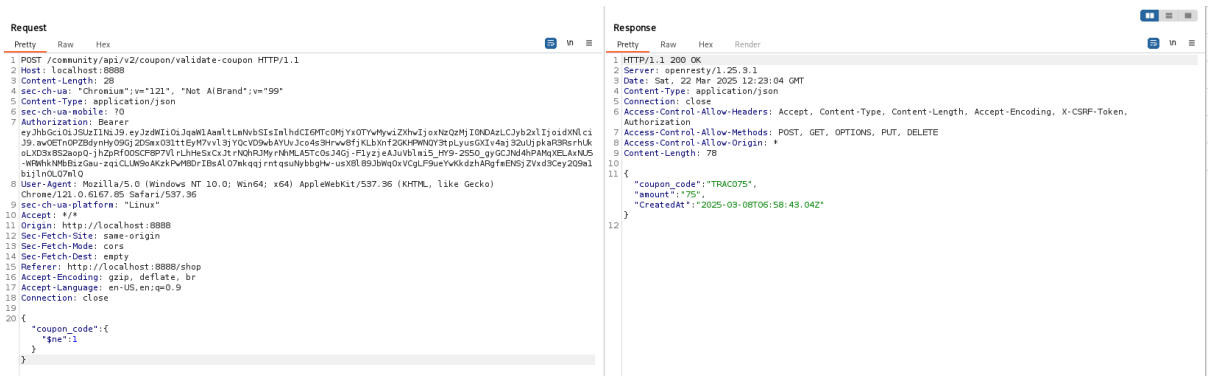
Screenshot-1: We initiated the challenge by intercepting the validate-coupon request in Burp Suite.



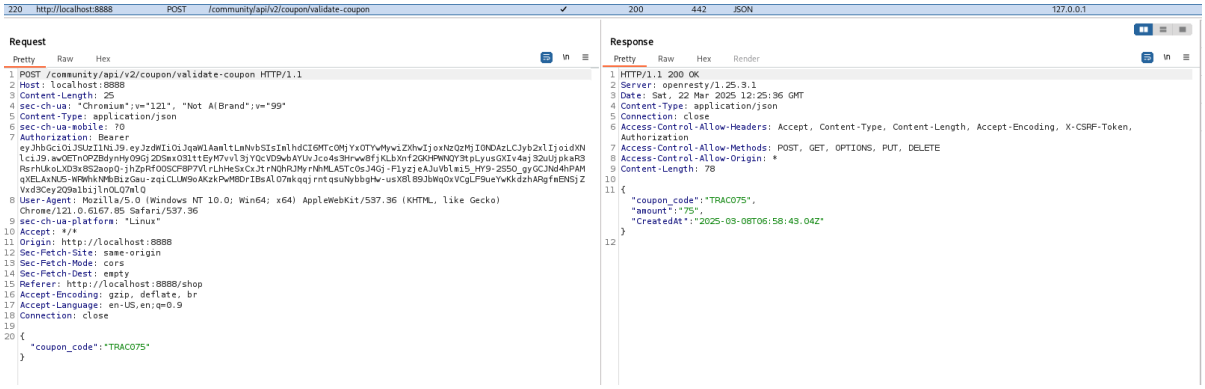


Screenshot-2: Employing a NoSQL Payload

We took inspiration from a collection of NoSQL injection payloads available on GitHub at https://github.com/swisskyrepo/PayloadsAllTheThings/tree/master/NoSQL%20Injection
Specifically, we used the payload `{ "$ne": 1 }`, which proved effective.



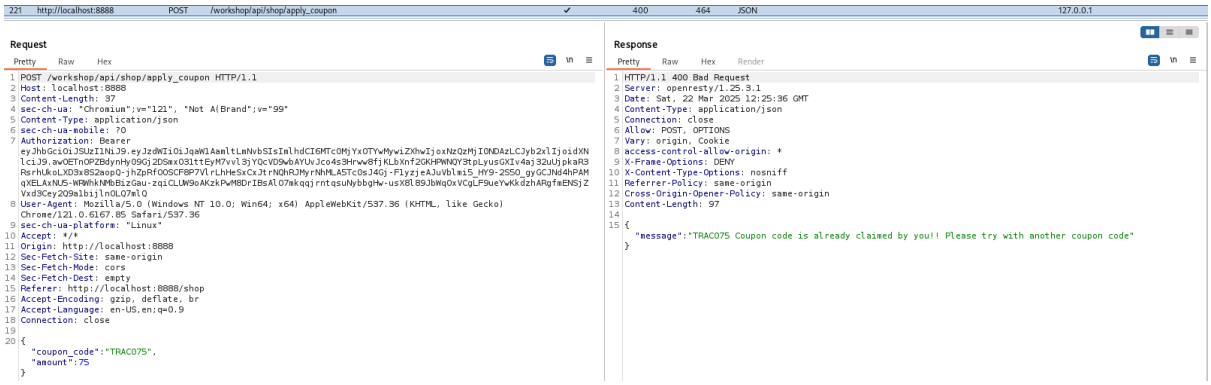
Screenshot-3: Now apply the coupon code and it is observed that amount is credited.



SQL injection: *It is a type of security vulnerability that occurs in web applications when user-supplied data is not properly validated or sanitized before being included in SQL queries. This allows malicious users to manipulate these queries to gain unauthorized access to a database or perform unintended actions on the database.*

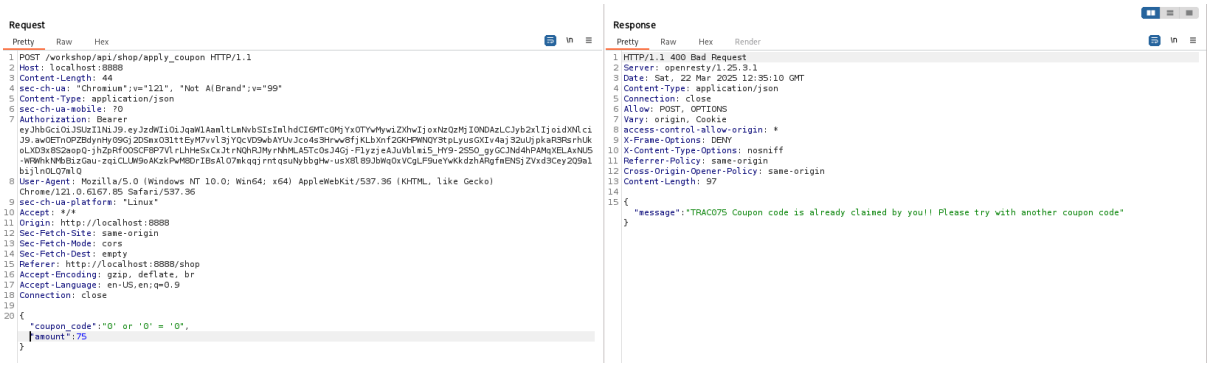
Challenge 15 — Find a way to redeem a coupon that you have already claimed by modifying the database

Screenshot-1: Try to apply the previously used coupon code and observe the response.



Screenshot-2: To manipulate the database and redeem the coupon, we turned to a repository of offensive payloads available at [https://github.com/InfoSecWarrior/Offensive-Payloads/]. We used a basic SQL injection payload: `0' or '0' = '0`.

The response we received was a success message indicating that the coupon code had already been claimed by us.



Screenshot-3: We attempted to identify the SQL version by using the payload `0'; select version() --+`, which revealed the SQL version in the response.