API Pentesting

- By Alham Rizvi

1. API Recon

1.1 What is API Recon?

API recon (reconnaissance) means **collecting information about an API before testing or attacking it**. Think of it like **exploring a building** before checking how secure it is.

Step 1: Find API Endpoints

An **API endpoint** is just a **URL path where the API listens for requests**.

Example:

GET /api/books

- `/api/books` is the **endpoint**
- It tells the API: *"Give me the list of books"*

Another example:

GET /api/books/mystery

- This endpoint asks for **mystery books only**

Each endpoint usually does **one specific job** (get data, add data, delete data, etc.).

Step 2: Understand How to Use Each Endpoint

Once you know the endpoints, you must learn **how to talk to them correctly**.

1. Input Data (Parameters)

APIs often expect **data from the user**, such as:

- Required data (must be provided)
- Optional data (extra filters or options)

Example:

/api/books?author=John

- `author=John` is input data

1. 2. Request Types (HTTP Methods)

APIs accept different **HTTP methods**, each with a purpose:

- **GET** → Read data
- **POST** → Create new data
- **PUT / PATCH** → Update data
- **DELETE** → Remove data

Example:

GET /api/books → Get books

```
POST /api/books      → Add a new book
```

1.3. Data Format (Media Types)

APIs usually send and receive data in formats like:

- **JSON** (most common)
- XML

Example JSON response:

```
{ "title": "Harry Potter", "author": "J.K. Rowling" }
```

1 . 4 Authentication (Login / Access Control)

Some APIs require **authentication**, such as:

- API keys
- Tokens (JWT, Bearer tokens)
- Username & password

Example:

Authorization: Bearer <token>

This checks **who you are and what you're allowed to access**.

1.5. Rate Limits

Rate limits control **how many requests you can send**.

Example:

- 100 requests per minute
- Too many requests → API blocks you

This protects the API from abuse.

1.6 Why API Recon Is Important?

API recon helps you:

- Understand **what endpoints exist**
- Know **what data is expected**
- Learn **what actions are allowed**
- Find **security weaknesses**

2. What is API Documentation?

2.1 Introduction

API documentation is a **guide that explains how to use an API**.
It tells developers **what the API can do and how to talk to it correctly**.

Think of it like a **user manual** for an API.

2.2 Types of API DocUmentation

1. Human-Readable Documentation

This is written **for people**.

It usually includes:

- What each API endpoint does
- Which HTTP methods to use (GET, POST, etc.)
- Required and optional parameters
- Example requests and responses
- Error messages and explanations

Example:

> "To get a list of books, send a GET request to /api/books."

This helps developers **understand and use the API easily**.

2. Machine-Readable Documentation

This is written **for computers, not humans**.

It uses structured formats like:

- **JSON**
- **XML**

Common examples:

- OpenAPI / Swagger files

These files help tools:

- Automatically generate API requests
- Validate inputs and responses

- Create client code or test cases

2.3 Where to Find API Documentation

- Often **publicly available**
- Common URLs:
    - /docs
    - /api/docs
    - /swagger
    - /v3/api-docs

If an API is meant for **external developers**, its documentation is usually **open to everyone**.

2.4 Why Documentation Is Important for API Recon

When doing **API reconnaissance**, documentation is the **best place to start** because it can reveal:

- All available endpoints
- Supported HTTP methods
- Required input fields
- Authentication methods
- Rate limits
- Expected responses

This saves time and gives a **clear picture of the API's attack surface**.

3. Discovering API Documentation

Sometimes **API documentation is not public**, but you can still **find it by exploring the app that uses the API**.

Think of it like **finding a hidden instruction manual** by looking around the building.

3.1 How You Can Find Hidden API Documentation

1. Browse the Application

Applications (websites or apps) that use an API often **link to the documentation internally**.

You can:

- Click through the app normally
- Watch network requests (using tools like Burp)
- Look for URLs that hint at documentation

2. Use Burp to Crawl the API

Burp Scanner can:

- Automatically **crawl the app**
- Discover API endpoints
- Reveal **hidden paths** that may point to documentation

This helps you find things you might miss manually.

3. Look for Common Documentation Paths

Developers often use **standard paths** for API docs.

Common examples:

- /api
- /swagger/index.html
- /openapi.json

If any of these load, you may find:

- A full list of endpoints
- Request methods
- Parameters
- Authentication details

4. Check the Base Path

If you find a **specific endpoint**, don't stop there.

Example:

/api/swagger/v1/users/123

You should also check:

- /api/swagger/v1
- /api/swagger
- /api

Why?
Documentation is often stored **higher up in the path**, not at the exact endpoint.

5. Use Common Path Lists (Intruder)

You can also:

- Use a **list of common API doc paths**
- Send many requests automatically (with tools like Intruder)
- See which paths exist

This helps uncover **forgotten or exposed documentation**.

3.2 Why This Matters

Hidden API documentation can reveal:

- All available endpoints
- Internal APIs not meant for users
- Sensitive details about how the API works

That's why **discovering documentation is a powerful recon step**.

3.3 Key Takeaway

Even if API documentation isn't public, **it's often still accessible** if you know where to look.

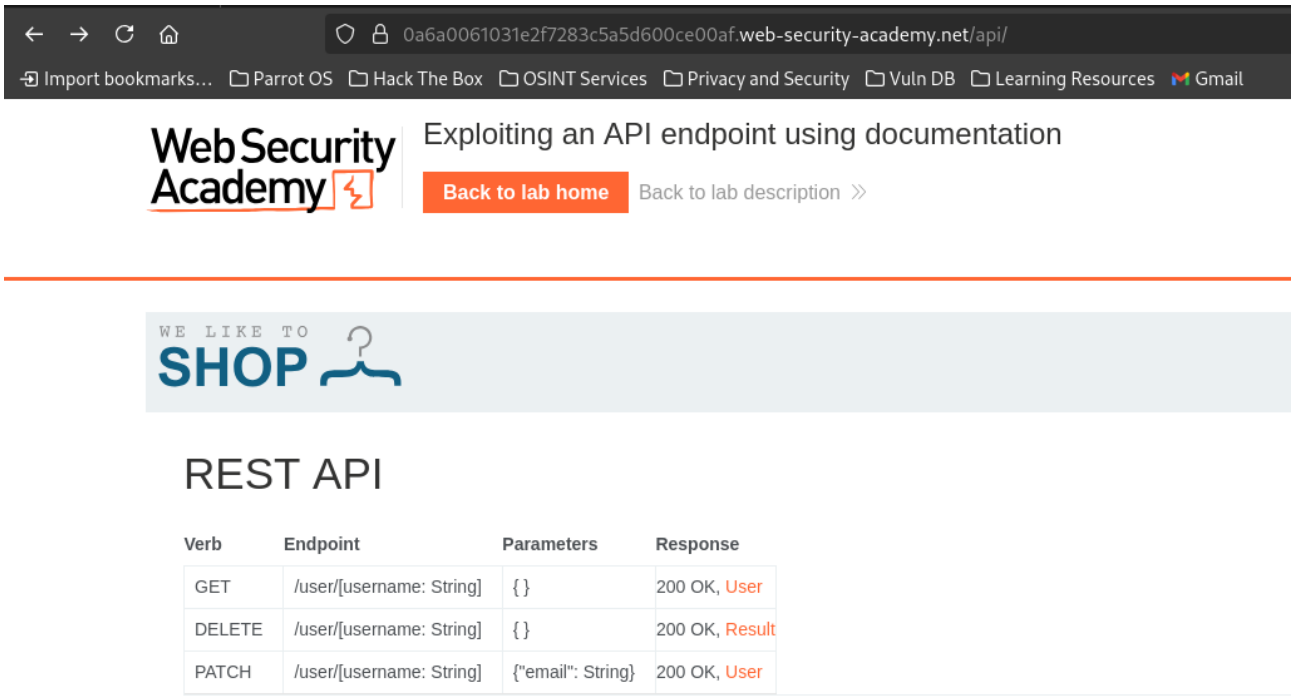
If you want, I can explain this with:

- A **real-world example**
- A **step-by-step recon flow**
- Or explain **why exposed Swagger files are risky**

Practical Demonstration

lab 1: Exploiting an API endpoint using documentation

This is PortSwigger's first API testing lab. I logged in using the default credentials and then sent requests to identify available API endpoints. After clicking Add and adding the path /api, I discovered a REST API.



Basically, this means the API can be abused. Using Burp Suite, an attacker can **GET**, **PATCH**, or **DELETE** *any* user account by simply specifying the username in the request. There appears to be no proper authorization or access control in place to restrict these actions.

This is extremely dangerous, as it allows an attacker to:

- View other users' details
- Modify user information (such as email addresses)
- Delete arbitrary user accounts

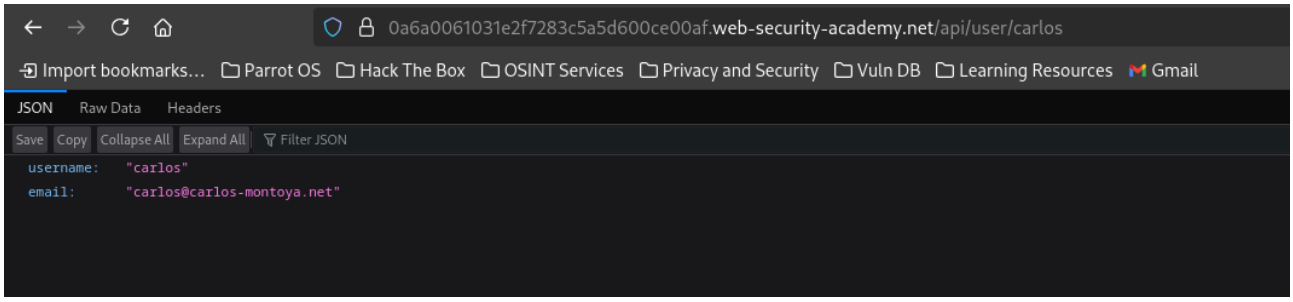
All of this can be done directly through the API using Burp Suite, without owning or authenticating as the target user.

For example, this lab instructs us to delete the user **carlos**. However, before doing that, we can inspect Carlos's API endpoints directly. For instance, we can access his user data via:

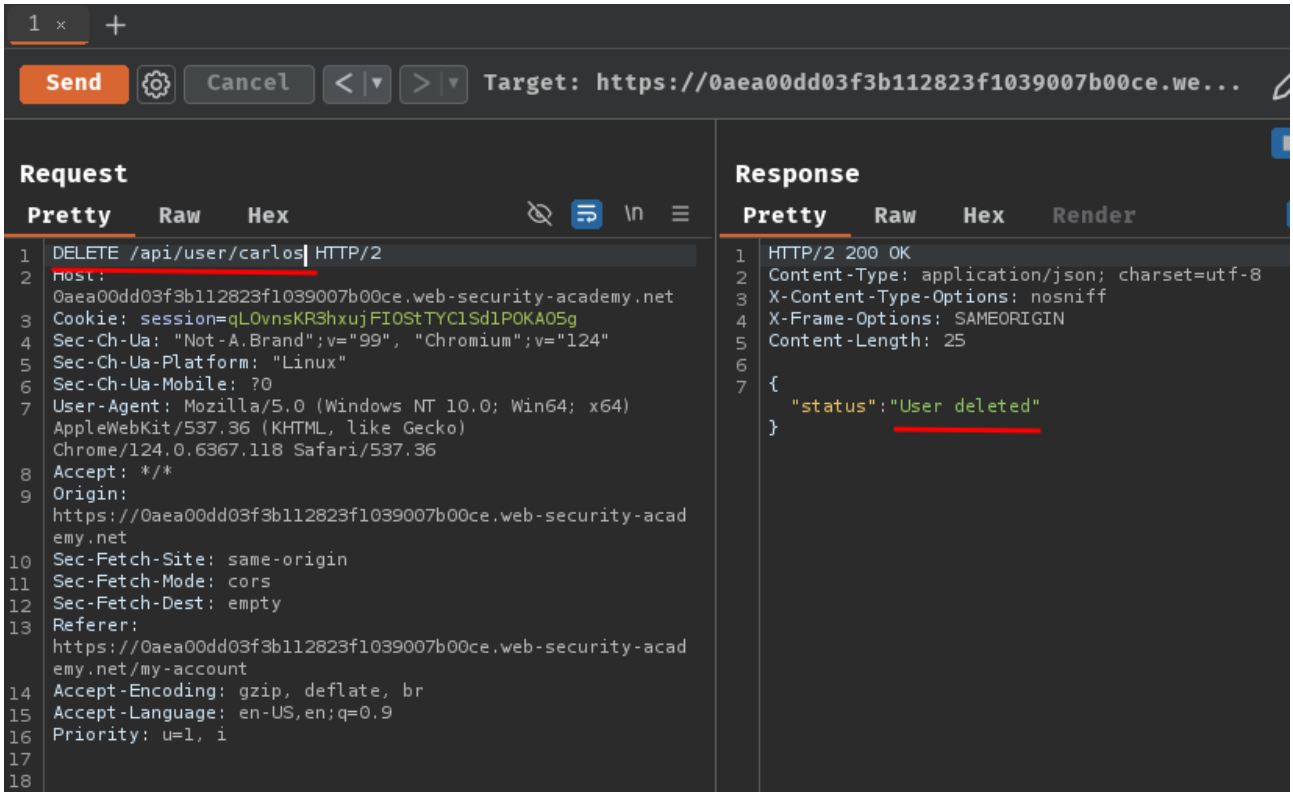
/api/user/carlos

(The exact domain depends on your specific lab instance.)

Using Burp Suite, we can exploit these exposed API endpoints by sending unauthorized **GET**, **PATCH**, or **DELETE** requests. This allows us to interact with Carlos's account directly through the API, ultimately enabling us to delete his user account as required by the lab.



Next, open **Burp Suite**, configure the browser to use Burp as a proxy, and refresh the page. You will see a request related to the user **carlos** appear in the **Proxy** tab. Send this request to the **Repeater** so it can be modified and analyzed further.



The request initially appears as a **GET** request. Change the HTTP method from **GET** to **DELETE**, then send the request. After clicking **Send** and checking the **Response** tab, you will see a confirmation indicating that the user **carlos** has been successfully deleted.

Congratulations! You have successfully completed the lab.

4. Identifying API endpoints

4.1 What does “identifying API endpoints” mean?

An **API endpoint** is just a **URL that an app uses to send or receive data** from a server.
For example, when a website loads your profile or posts a comment, it often talks to an API behind the scenes.

4.2 Why should you look for API endpoints yourself?

Even if you have **API documentation**, it might:

- Be **old**
- Be **missing some endpoints**
- Contain **mistakes**

So, looking at the **actual application** can show you what APIs are really being used.

4.3 How can you find API endpoints by browsing an app?

1. **Browse the website or app**
    - Click buttons, open pages, log in, etc.
    - Watch what network requests are happening in the background.
2. **Use Burp Scanner to crawl the site**
    - Crawling means Burp automatically visits pages to discover links and requests.
    - It helps reveal parts of the app you might miss manually.
3. **Look at the URL patterns**
    - API URLs often look like:
        - /api/
        - /api/users
        - /api/login
    - Seeing /api/ in a URL is a strong hint it’s an API endpoint.

4.4 Why are JavaScript files important?

- Websites use **JavaScript** to talk to APIs.
- JavaScript files often **contain hidden API URLs**.
- Some of these APIs may:
    - Not be visible just by clicking around

    - Only be used in special situations

4.5 How Burp helps with JavaScript

- **Burp Scanner** automatically finds some API endpoints.
- For deeper searching:
    - Tools like **JS Link Finder** can extract more API URLs from JavaScript.
- You can also **manually open and read JavaScript files** inside Burp to spot API links yourself.

In short,

- Don’t rely only on documentation.
- Browse the app to see what APIs are really used.
- Watch for /api/ in URLs.
- Check JavaScript files because they often reveal hidden API endpoints.
- Burp tools help automate and simplify this process.

5. HTTP methods

5.1 What are HTTP headers?

HTTP headers are extra pieces of information sent with a request or response.
They tell the server (or client):

- Who you are
- What kind of data you're sending
- How the request should be handled
- Security rules

Think of headers like the **instructions and labels on a package**, not the package itself.

5.2 IMPORTANT REQUEST HEADERS (Client → Server)

These are the headers **you send to the API**.

1. **Host**

Host: example.com

- Tells the server **which website or API** you want.
- Required in HTTP/1.1.
- Sometimes useful for **virtual host testing**.

2. **Authorization**

Authorization: Bearer eyJhbGciOi...

- Used for **authentication**.
- Common formats:
    - Bearer token (JWT, OAuth)
    - Basic base64(username:password)
- Very important for:
    - Access control testing
    - Privilege escalation
    - Broken authentication

3. **Content-Type**

Content-Type: application/json

- Tells the server **what format the data is in**.
- Common values:
    - application/json
    - application/xml
    - application/x-www-form-urlencoded
- Changing this can sometimes:
    - Bypass validation
    - Trigger errors or unexpected behavior

4. **Accept**

Accept: application/json

- Tells the server **what response format you want**.
- Sometimes APIs support:
    - JSON
    - XML
- Testing different values can reveal **hidden features**.

5. **User-Agent**

User-Agent: Mozilla/5.0

- Identifies the client (browser, mobile app, tool).
- Some APIs:
    - Block unknown User-Agents
    - Behave differently for mobile vs browser
- Changing it may bypass restrictions.

6. **Cookie**

Cookie: sessionid=abc123

- Stores **session data**.
- Very important for:
    - Session hijacking

        - Authentication testing
        - Authorization flaws
- APIs sometimes use cookies instead of tokens.

7. **Origin**

Origin: https://example.com

- Used for **CORS (Cross-Origin Resource Sharing)**.
- Important for:
    - CORS misconfiguration testing
    - API access from other websites

8. **Referer**

Referer: https://example.com/dashboard

- Shows **where the request came from**.
- Some apps trust this header (bad idea).
- Can be manipulated to bypass weak checks.

9. **X-Requested-With**

X-Requested-With: XMLHttpRequest

- Indicates the request came from JavaScript (AJAX).
- Sometimes APIs:
    - Allow requests only if this header exists
- Removing or changing it can reveal issues.

10. **HTTP Method Override Headers**

X-HTTP-Method-Override: DELETE

or

X-Method-Override: PUT

- Used to **override the HTTP method**.
- Example:
    - POST request that acts like DELETE
- Very important for:
    - Method bypass testing

- Hidden functionality

5.3 IMPORTANT RESPONSE HEADERS (Server → Client)

These are headers **returned by the API**.

11. **Allow**

Allow: GET, POST, DELETE

- Shows **which HTTP methods are supported**.
- Often returned with OPTIONS requests.
- Great for discovering hidden methods.

12. **Access-Control-Allow-Origin**

Access-Control-Allow-Origin: *

- Part of **CORS**.
- Dangerous if combined with:
    - Credentials allowed
- Important for API security testing.

13. **Access-Control-Allow-Credentials**

Access-Control-Allow-Credentials: true

- Allows cookies or auth headers in cross-origin requests.
- Combined with weak origin rules = serious risk.

14. **Set-Cookie**

Set-Cookie: sessionid=abc123; HttpOnly; Secure

- Creates or updates cookies.
- Look for missing flags:
    - HttpOnly
    - Secure
    - SameSite

15. **WWW-Authenticate**

WWW-Authenticate: Bearer

- Tells how authentication works.
- Useful for identifying:
    - Auth type
    - Token requirements

5.4 WHY HEADERS MATTER IN API TESTING

Changing or removing headers can:

- Bypass authentication
- Bypass authorization
- Reveal hidden endpoints
- Trigger different behavior
- Expose security misconfigurations

In Short,

- HTTP headers control **how APIs behave**
- APIs may trust headers too much
- Testing different headers is essential
- Burp makes it easy to modify and replay requests

Important: A*PI endpoints often expect data in a specific format. They may therefore behave differently depending on the content type of the data provided in a request. Changing the content type may enable you to:*

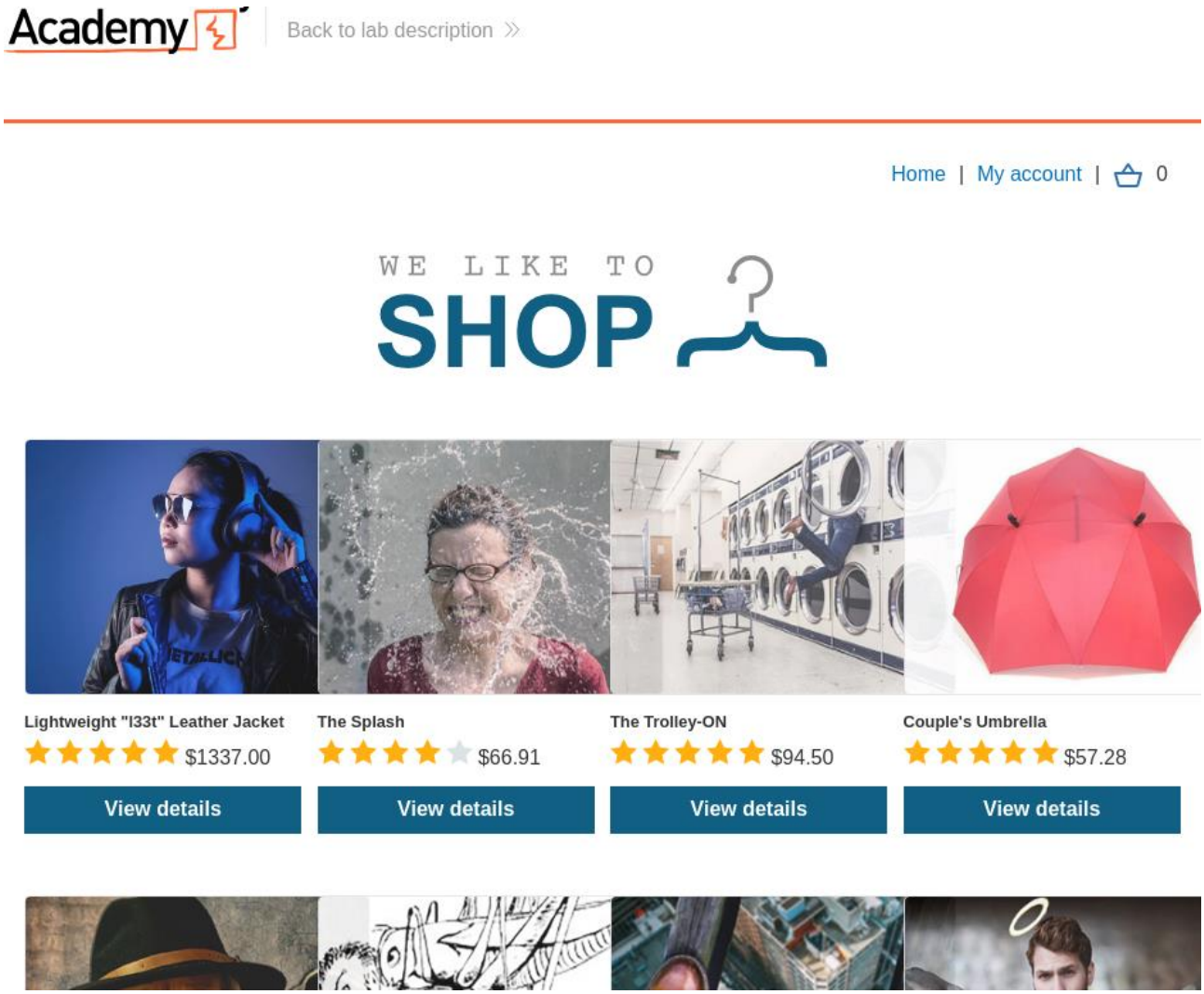
- *Trigger errors that disclose useful information.*
- *Bypass flawed defenses.*
- *Take advantage of differences in processing logic. For example, an API may be secure when handling JSON data but susceptible to injection attacks when dealing with XML.*

To change the content type, modify the Content-Type header, then reformat the request body accordingly. You can use the Content type converter BApp to automatically convert data submitted within requests between XML and JSON.

6. Finding and exploiting an unused API endpoint

This is practical demonstration Finding and Exploiting an Unused API endpoint By portswigger

First of all, I log in using the credentials I was given, and this is how the homepage looks:

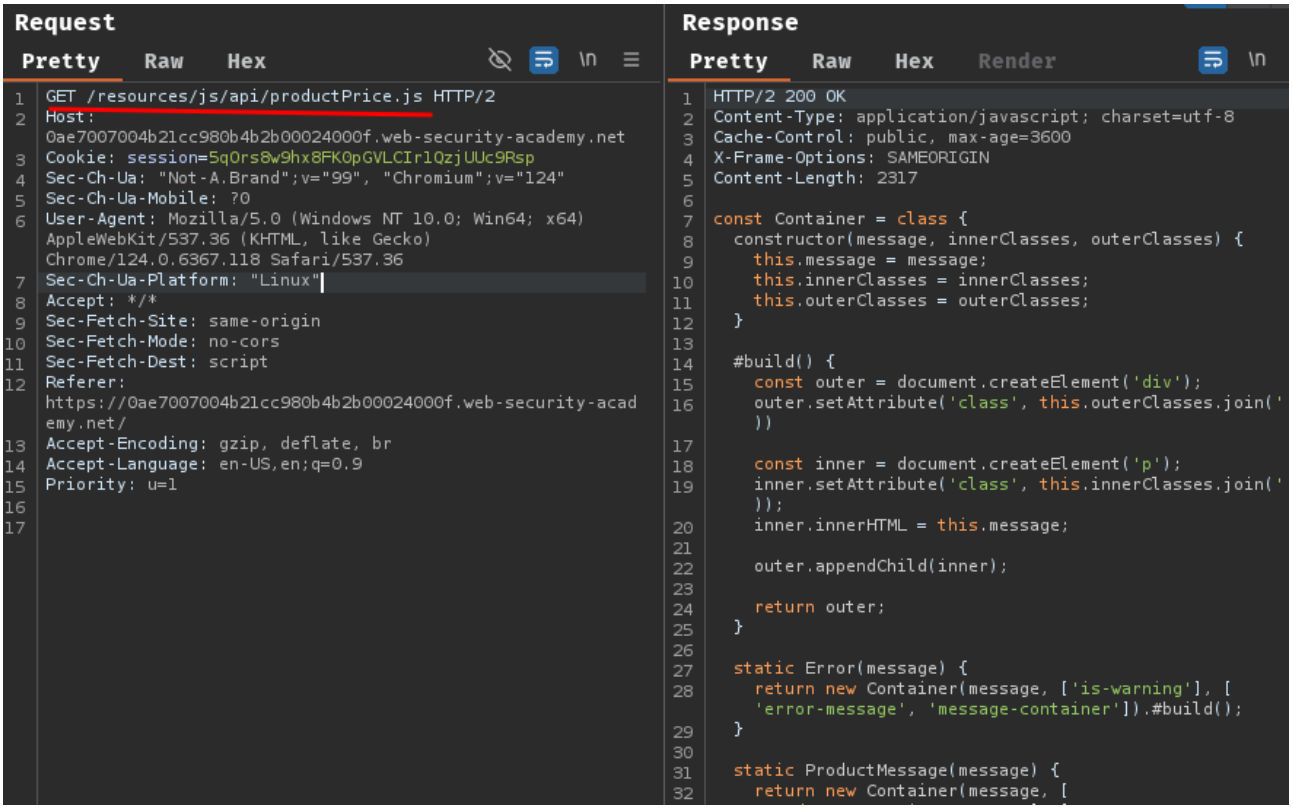


In this lab, we need to change the price of a product. **Do not try this in real life**—this is only a **practice demonstration** done in a lab environment using API endpoints.

For API testing, we can learn a lot by browsing the application that uses the API. This is useful even if API documentation is available, because the documentation may be outdated or incorrect.

We can use **Burp Scanner** to crawl the application and then manually explore interesting areas using **Burp's browser**.

While browsing, look for URL patterns that suggest API endpoints, such as /api/. Also check **JavaScript files**, as they may contain API endpoints that are not directly visible in the browser. Burp Scanner finds some endpoints automatically, and for deeper analysis, we can use the **JS Link Finder** extension or review the JavaScript files manually.



When we first visit the application, we see that a JavaScript file is loaded from:

/resources/js/api/productPrice.js

To change the content type, update the **Content-Type** header and then reformat the request body to match it. We can use the **Content-Type Converter BApp** to automatically convert request data between XML and JSON.

First, try adding an empty JSON object, and then add data to it. At this point, the application tells us that a **price** parameter is required in the JSON object.

If we want to change the price of **product ID 2**, we can use the following JSON data:

```
{ "price": 0.00 }
```

Request

Pretty Raw Hex

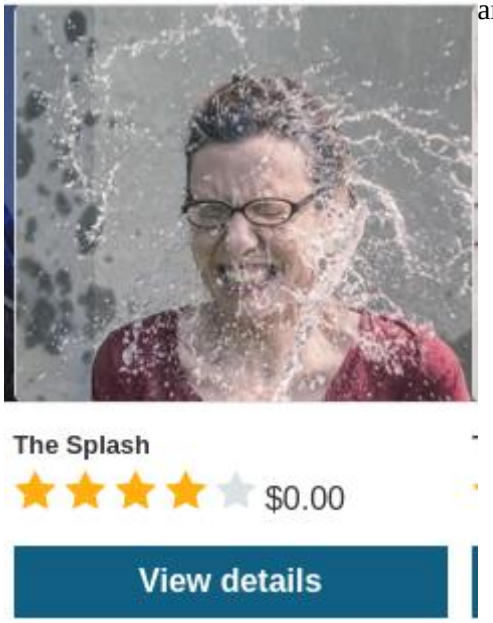
```
1  PATCH /api/products/2/price HTTP/2
2  Host:
   0ae7007004b21cc980b4b2b00024000f.web-security-academy.net
3  Cookie: session=muuF6nfbhL4DqYXvY9tVCJ3fw22UbJqY
4  Sec-Ch-Ua: "Not-A.Brand";v="99", "Chromium";v="124"
5  Sec-Ch-Ua-Mobile: ?0
6  User-Agent: Mozilla/5.0 (Windows NT 10.0; Win64; x64)
   AppleWebKit/537.36 (KHTML, like Gecko)
   Chrome/124.0.6367.118 Safari/537.36
7  Sec-Ch-Ua-Platform: "Linux"
8  Accept: */*
9  Sec-Fetch-Site: same-origin
10 Sec-Fetch-Mode: no-cors
11 Sec-Fetch-Dest: script
12 Referer:
   https://0ae7007004b21cc980b4b2b00024000f.web-security-acad
   emy.net/
13 Accept-Encoding: gzip, deflate, br
14 Accept-Language: en-US,en;q=0.9
15 Priority: u=1
16 Content-Type: application/json
17 Content-Length: 11
18
19 {
     "price":0
   }
```

Response

Pretty Raw Hex Render

```
1  HTTP/2 200 OK
2  Content-Type: application/json; charset=utf-8
3  X-Frame-Options: SAMEORIGIN
4  Content-Length: 17
5
6  {
     "price":"$0.00"
   }
```

and yah the page literally shows$0.00



Lab Summary: Changing Product Price via API

1. **Login and Explore:**
   - Logged in to the web application using provided credentials.
   - Observed the homepage and explored its structure.
2. **API Exploration:**
   - Browsed the application to identify API endpoints.
   - Looked for URL patterns like /api/ and checked JavaScript files for hidden endpoints.
   - Used **Burp Scanner** and optionally **JS Link Finder** to discover API endpoints.

3. **Content-Type Handling:**
    - Learned how to change the **Content-Type** header in requests.
    - Used the **Content-Type Converter BApp** to switch request data between XML and JSON.
4. **Modifying Data:**
    - Created a JSON object for a request.
    - Added the required price parameter to update a product.
    - Constructed a JSON body to change the price of **product ID 2**:

- {

"price": 0.00

}

- **Key Takeaways:**
- Browsing and analyzing applications helps find API endpoints.
- JavaScript files often reveal hidden API endpoints.
- Modifying headers and request data is important for testing APIs.

API Pentesting Pt 2

-By Alham Rizvi

6. Mass assignment vulnerabilities

6.1 Introduction

Mass assignment (also known as **auto-binding**) can unintentionally introduce hidden parameters into an application.

This happens when a software framework automatically maps all incoming request parameters to fields in an internal object. If this process isn't carefully controlled, the application may accept and process parameters that the developer never intended to expose.

As a result, attackers may be able to modify sensitive fields such as roles, permissions, or internal flags simply by including unexpected parameters in their requests.

6.2 Identifying hidden parameters

Because **mass assignment** binds request parameters directly to object fields, you can often spot hidden parameters by comparing **what the API accepts** with **what it returns**.

A useful technique is to examine objects returned by the API and look for fields that aren't documented or editable through normal requests.

Example

Suppose the API allows users to update their profile using:

PATCH /api/users/

```
{
  "username": "wiener",
  "email": "wiener@example.com"
}
```

At the same time, a request to retrieve a user object returns:

GET /api/users/123

```
{
  "id": 123,
  "name": "John Doe",
  "email": "john@example.com",
  "isAdmin": false
}
```

What this suggests

The presence of fields like `id` and `isAdmin` in the response may indicate that these fields are part of the same internal user object as `username` and `email`. If the application uses mass assignment without proper controls, these fields could potentially be accepted as request parameters—even if they were never meant to be user-editable.

Key takeaway

By comparing **editable fields** with **returned object fields**, you can identify parameters that may be unintentionally exposed due to mass assignment. This helps highlight areas where developers should apply explicit allowlists or validation to prevent misuse.

6.3 Testing mass assignment vulnerabilities

To test whether you can modify the enumerated isAdmin parameter value, add it to the PATCH request:

```
{ "username": "wiener", "email": "wiener@example.com", "isAdmin": false, }
```

In addition, send a PATCH request with an invalid isAdmin parameter value:

```
{ "username": "wiener", "email": "wiener@example.com", "isAdmin": "foo", }
```

If the application behaves differently, this may suggest that the invalid value impacts the query logic, but the valid value doesn't. This may indicate that the parameter can be successfully updated by the user.

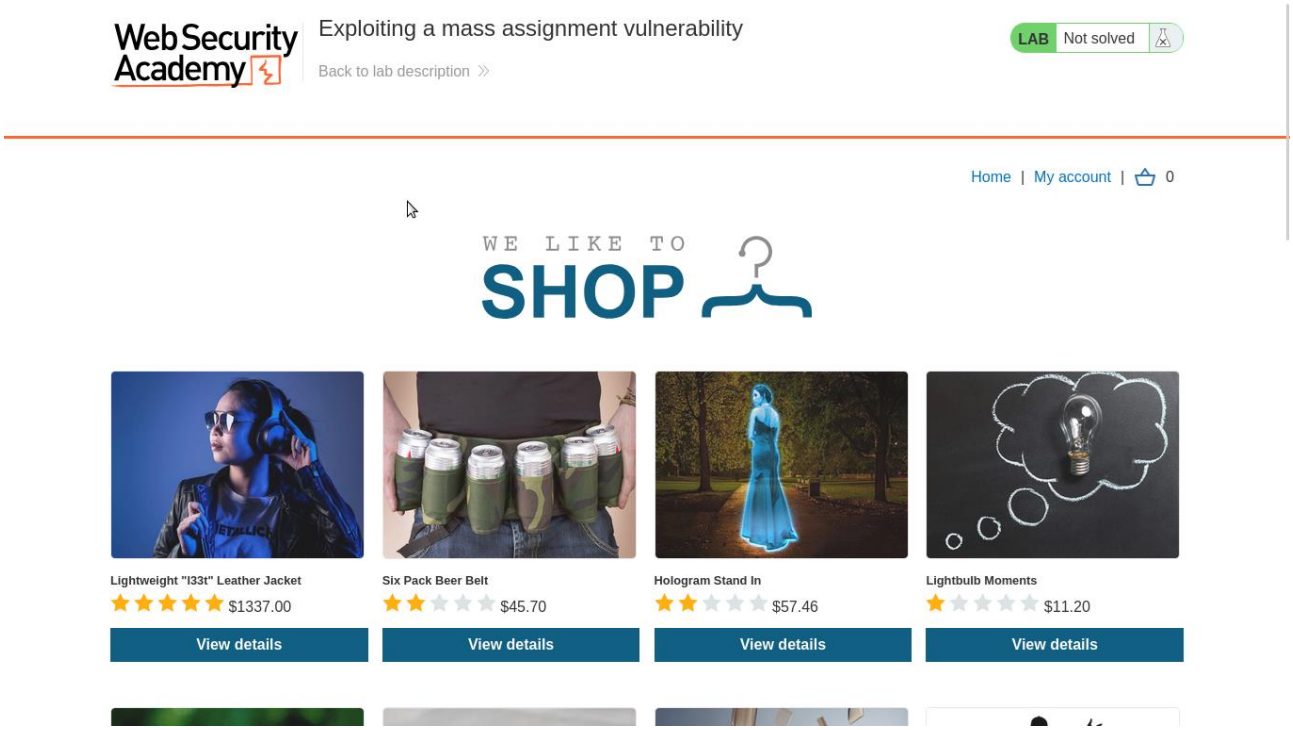
You can then send a PATCH request with the isAdmin parameter value set to true, to try and exploit the vulnerability:

```
{ "username": "wiener", "email": "wiener@example.com", "isAdmin": true, }
```

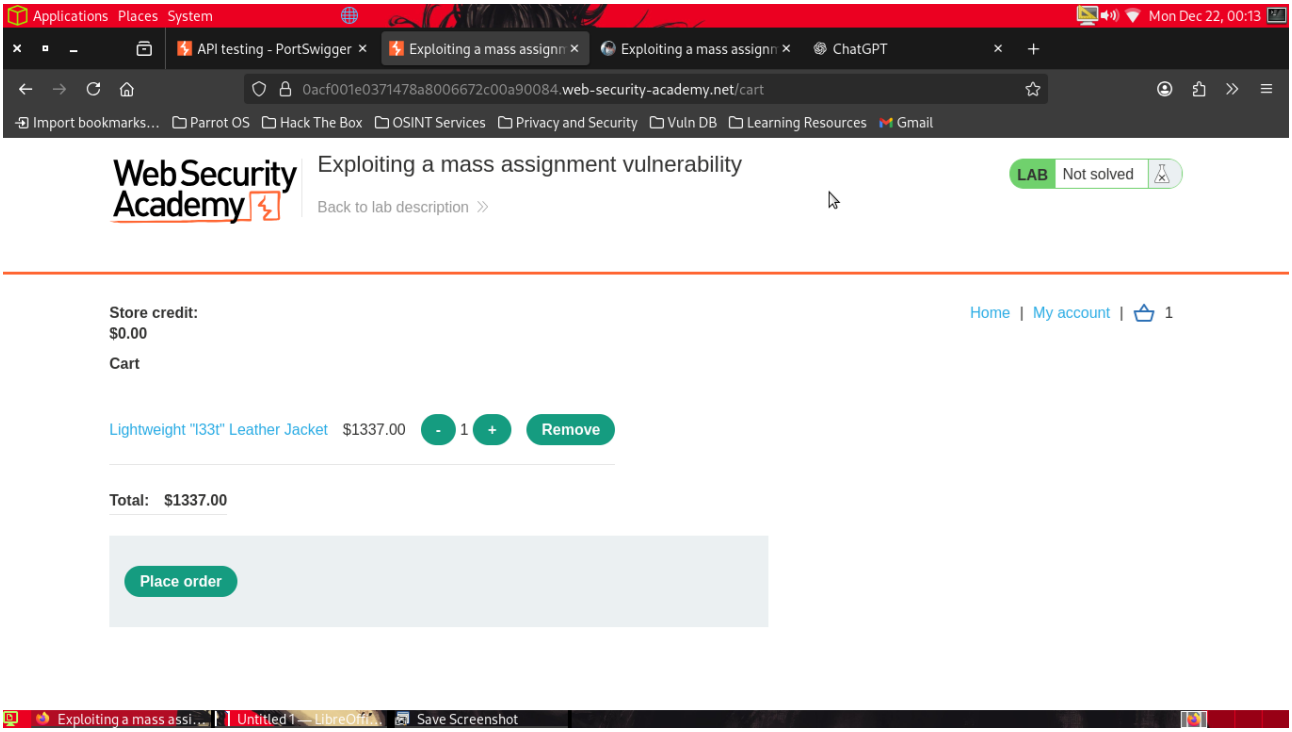
If the isAdmin value in the request is bound to the user object without adequate validation and sanitization, the user wiener may be incorrectly granted admin privileges. To determine whether this is the case, browse the application as wiener to see whether you can access admin functionality.

Practical Demonstration

In this demonstration, I used a PortSwigger lab to explain how the vulnerability occurs and how it can be exploited. The following is the home page:



One of the most important things is finding the API endpoint. After playing around for about 30 minutes, I found the API endpoint when I clicked on any product and then clicked **View Details > Add to cart and check in the cart**



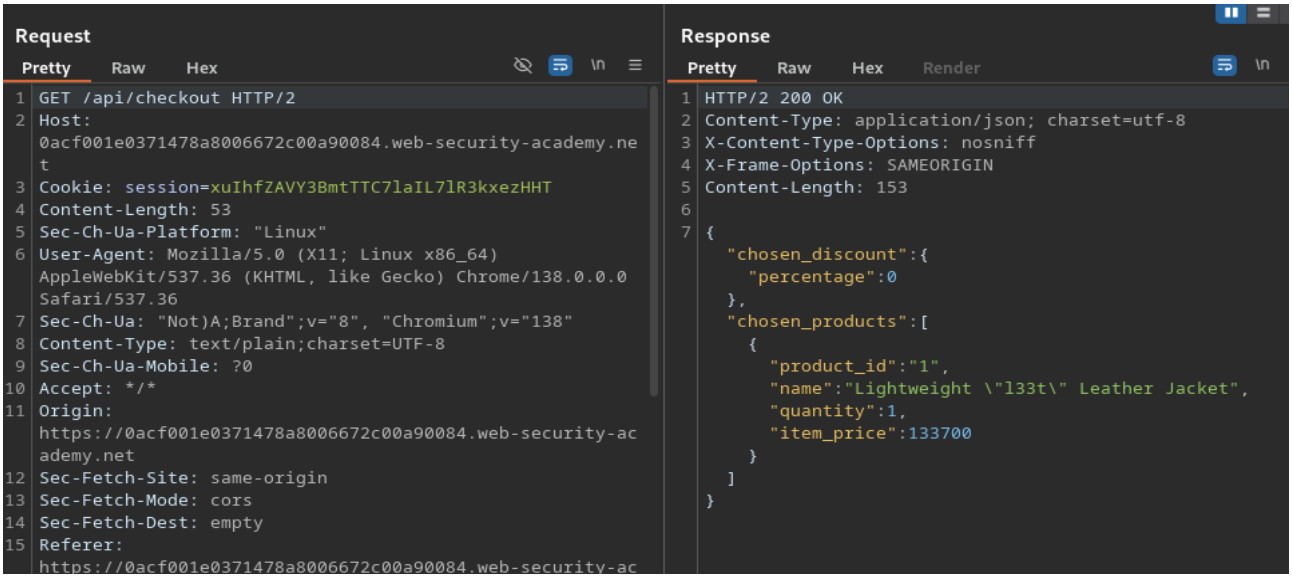
After viewing the page source using **Ctrl + U**, I discovered that the endpoint is /api/checkout.

```
                </table>
                <table>
                    <tbody>
                        <tr>
                            <th>Total:</th>
                            <th id='cart-total'></th>
                        </tr>
                    </tbody>
                </table>
                <script type='text/javascript'>
                    const getApiEndpoint = () => '/api/checkout';
                    const buildProductRow = (id, name, price, quantity) => `        <tr
        <td>
            <a href=/product?productId=${encodeURIComponent(id)}>${name}</a>
        </td>
        <td>${price}</td>
        <td>
            <form action=/cart method=POST style='display: inline'>
```

Another endpoint

```
        </tr>
                    `;
                    </script>
                    <script type='text/javascript' src='/resources/js/api/checkout.js'></script>
                    <form class='login-form' onsubmit='doCheckout(event)'>
                        <button class='button' type='submit'>Place order</button>
                    </form>
                </div>
            </section>
            <div class="footer-wrapper">
```

when I clicked **Place Order**, I started Burp Suite and sent that request to the Repeater. Then, I changed the request from **POST** to **GET** and received this data,



Now, I added the following JSON payload and explained what it does in Burp Suite.

Payload you added

```
{
   "chosen_discount": {
      "percentage": 100
   },
   "chosen_products": [
      {
         "product_id": "1",
         "quantity": 1
      }
   ]
}
```

Explanation (what it does in Burp Suite)

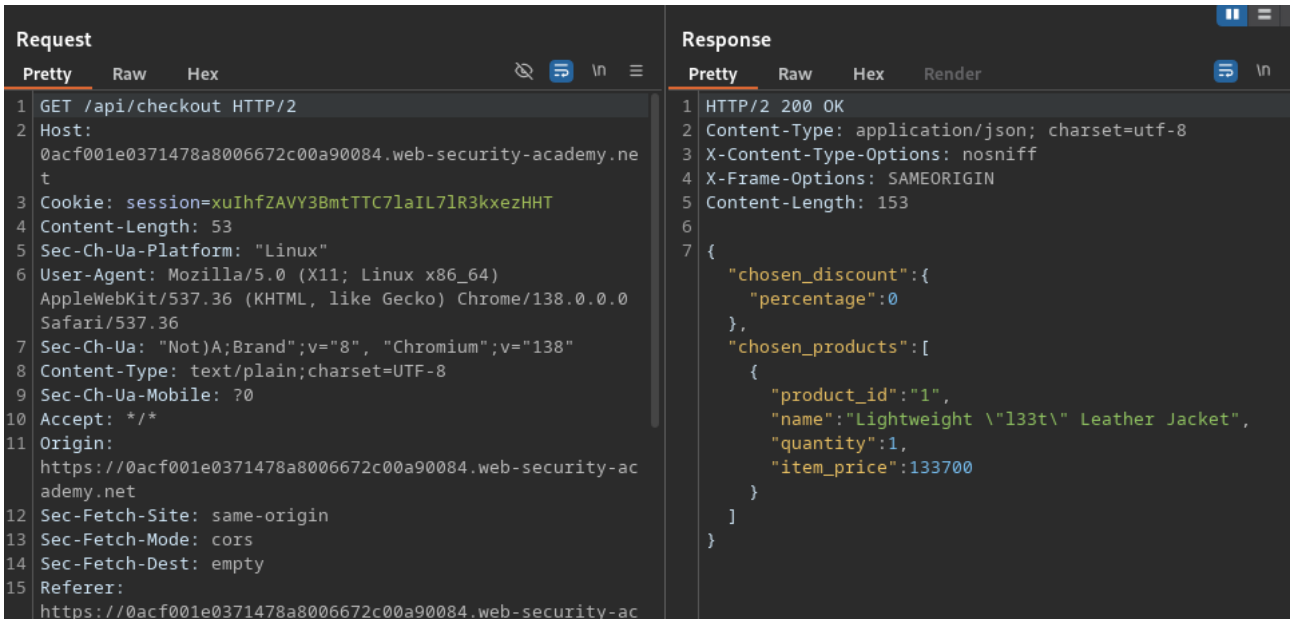
- chosen_products
  This tells the backend which product you want to buy.
  - product_id: "1" → selects product ID 1
  - quantity: 1 → buys one unit of that product
- chosen_discount
  This manually sets a discount value.
  - percentage: 100 → applies a **100% discount**, meaning the total price becomes **free (0 cost)**

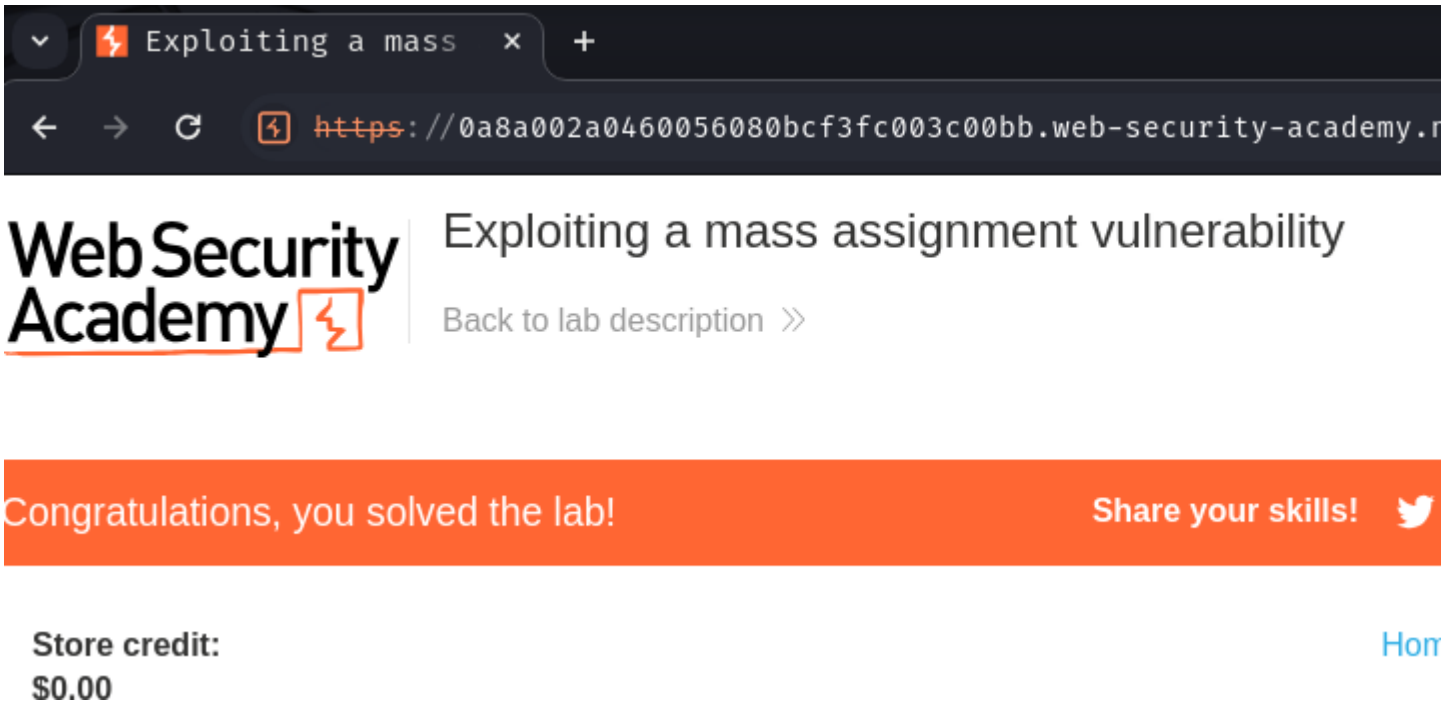
Why this matters

By modifying the request in **Burp Suite Repeater**, you are:

- Directly controlling the API request
- Injecting a discount value that the frontend normally wouldn't allow
- Exploiting a **lack of server-side validation**, where the backend blindly trusts user input

This demonstrates a **business logic/API vulnerability**, where sensitive values like discounts should never be controlled by the client.





hahhaha, We bought the product for free.

7. Server-Side Parameter Pollution

7.1 Server-Side Parameter Pollution (SSPP)

Some systems contain internal APIs that are not directly accessible from the internet. **Server-side parameter pollution** occurs when a website embeds user-controlled input into a server-side request to an internal API **without proper encoding or validation**.

As a result, an attacker may be able to manipulate or inject additional parameters, which can allow them to:

- Override existing parameters
- Modify application behavior
- Access unauthorized or sensitive data

Testing for Server-Side Parameter Pollution

Any user-controlled input can potentially be tested for parameter pollution, including:

- Query parameters
- Form fields
- HTTP headers
- URL path parameters

If these inputs are not correctly handled before being passed to internal APIs, the application may become vulnerable to SSPP.

7.2 Example of Server-Side Parameter Pollution

Suppose a website sends this request to an internal API:

`/internal-api/checkout?product_id=1&price=100`

If user input is added to this request **without validation**, an attacker could modify it like this:

`/internal-api/checkout?product_id=1&price=100&price=0`

The internal API may process the second `price` parameter, allowing the attacker to change the product price to **0**.

This shows how server-side parameter pollution can be used to **override parameters and manipulate application behavior**.

7.3 Testing for server-side parameter pollution in the query string

To test for server-side parameter pollution in the query string, you can insert characters like `#`, `&`, or `=` into user input and observe how the application responds.

Example:

A vulnerable application lets you search for users by username. When you search for "peter," your browser sends the following request:

`GET /userSearch?name=peter&back=/home`

The server then queries an internal API like this:

`GET /users/search?name=peter&publicProfile=true`

If the input isn't properly validated, an attacker could manipulate the query string to override or inject additional parameters, potentially exposing unauthorized data.

7.4 Truncating Query Strings

You can test for server-side parameter pollution by using a URL-encoded `#` character (`%23`) to try to truncate the server-side request. To better understand the response, you can also add extra text after the `#`.

Example request:

`GET /userSearch?name=peter%23foo&back=/home`

The front-end may construct the following internal API request:

```
GET /users/search?name=peter#foo&publicProfile=true
```

Important:

The # character must be URL-encoded. If it is not encoded, the browser treats it as a fragment and does not send it to the server or internal API.

How to identify truncation

- If the response returns the user **peter**, the query may have been truncated at the #.
- If an **Invalid name** error appears, the application likely treated `foo` as part of the username, meaning truncation did not occur.

Why this is useful

If truncation is successful, the `publicProfile=true` parameter may be removed from the request. This could allow access to **non-public user profiles**, demonstrating a server-side parameter pollution vulnerability.

7.5 Injecting Invalid Parameters

You can test for server-side parameter pollution by using a URL-encoded `&` character (`%26`) to try to inject an extra parameter into the server-side request.

Example request:

```
GET /userSearch?name=peter%26foo=xyz&back=/home
```

This may cause the server to send the following request to an internal API:

```
GET /users/search?name=peter&foo=xyz&publicProfile=true
```

How to analyze the response

- If the response does not change, the extra parameter may have been injected but ignored.
- If the behavior changes or an error appears, this suggests the application is parsing the injected parameter.

Next steps

To fully confirm the vulnerability, you should continue testing with different parameter names and values to understand how the server handles injected parameters.

7.6 Injecting Valid Parameters

If you are able to modify the query string, you can try injecting a **second valid parameter** into the server-side request.

If you have already identified a valid parameter (such as `email`), you can add it to the query string like this:

Example request:

```
GET /userSearch?name=peter%26email=foo&back=/home
```

This may result in the following internal API request:

```
GET /users/search?name=peter&email=foo&publicProfile=true
```

How to analyze the response

Review the response carefully to see how the injected parameter is handled. Any change in behavior, data returned, or error messages may indicate that the parameter was successfully injected and processed by the server.

\

7.7 Overriding Existing Parameters

To confirm whether an application is vulnerable to **server-side parameter pollution**, you can try overriding an existing parameter by injecting a second parameter with the same name.

Example request:

```
GET /userSearch?name=peter%26name=carlos&back=/home
```

This may result in the following internal API request:

```
GET /users/search?name=peter&name=carlos&publicProfile=true
```

The internal API now receives **two** `name` **parameters**. How this is handled depends on the backend technology:

- **PHP**: Uses the **last parameter** → searches for `carlos`
- **ASP.NET**: Combines parameters → searches for `peter,carlos`, which may cause an error
- **Node.js / Express**: Uses the **first parameter** → searches for `peter` (no change)

Why this matters

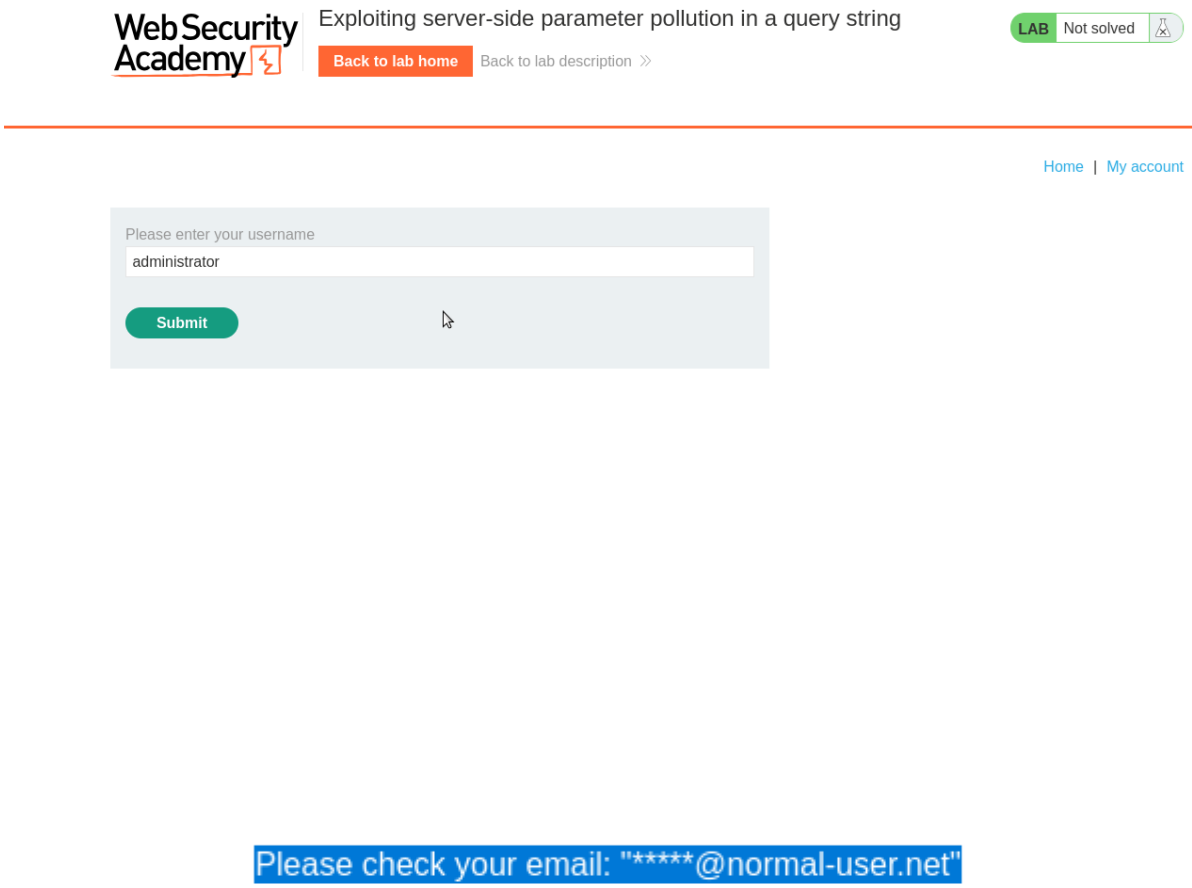
If the application allows overriding parameters, it may be possible to exploit this behavior. For example, injecting `name=administrator` could allow access to **privileged or sensitive accounts**.

Practical Demonstration

Portswigger Lab: Exploiting server-side parameter pollution in a query string

> *To complete the lab, log in as the administrator account and delete the user* ***carlos****.*

First, I went to the login page and tried to gain access to the administrator account, since only an admin can perform these actions. I then clicked **Forgot Password** because I didn’t know the actual password.



It seems like the /forgot-password page sends an email to the administrator. The page also loads a JavaScript file that runs when the DOM (Document Object Model) is fully loaded.

The script does the following:

- It retrieves the value of the reset-token GET parameter.
- If the parameter exists, it redirects the browser to:
- /forgot-password?reset_token=<resetToken>

If we send a GET request with an invalid reset token, like:

/forgot-password?reset_token=<resetToken>

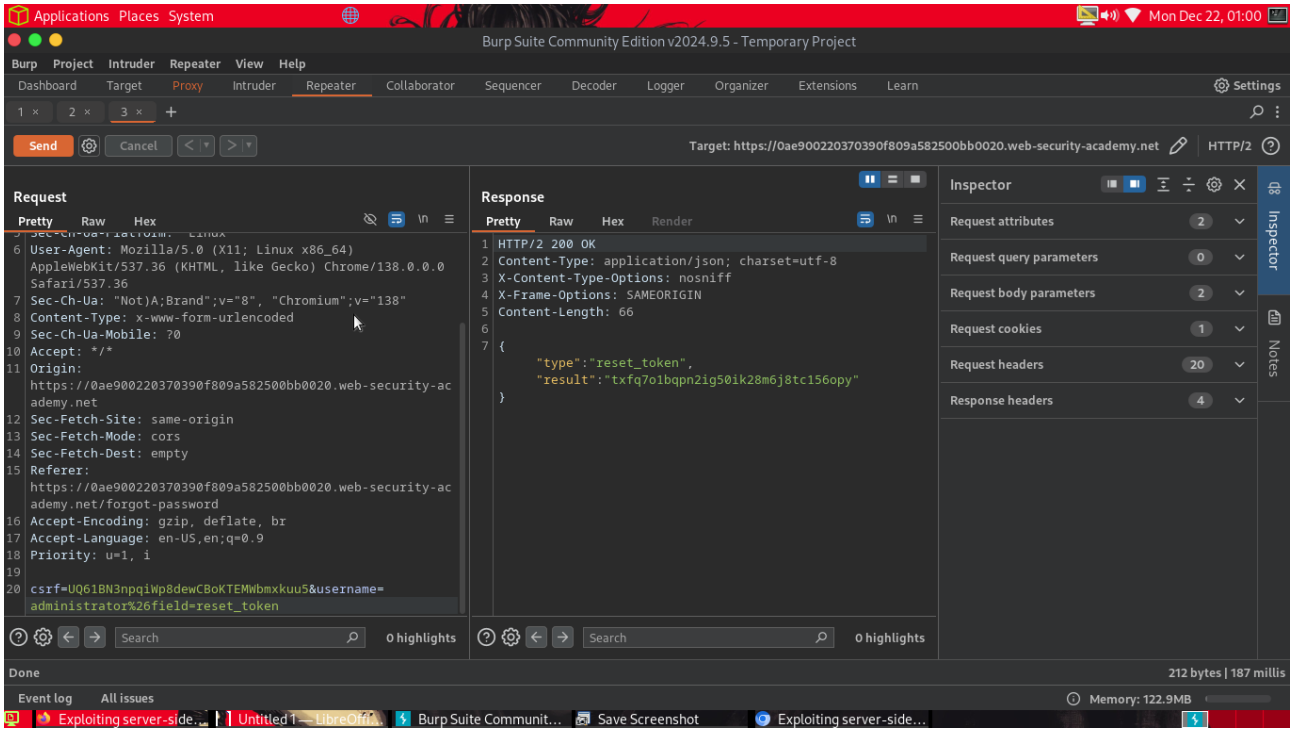
The server responds with:

Invalid token

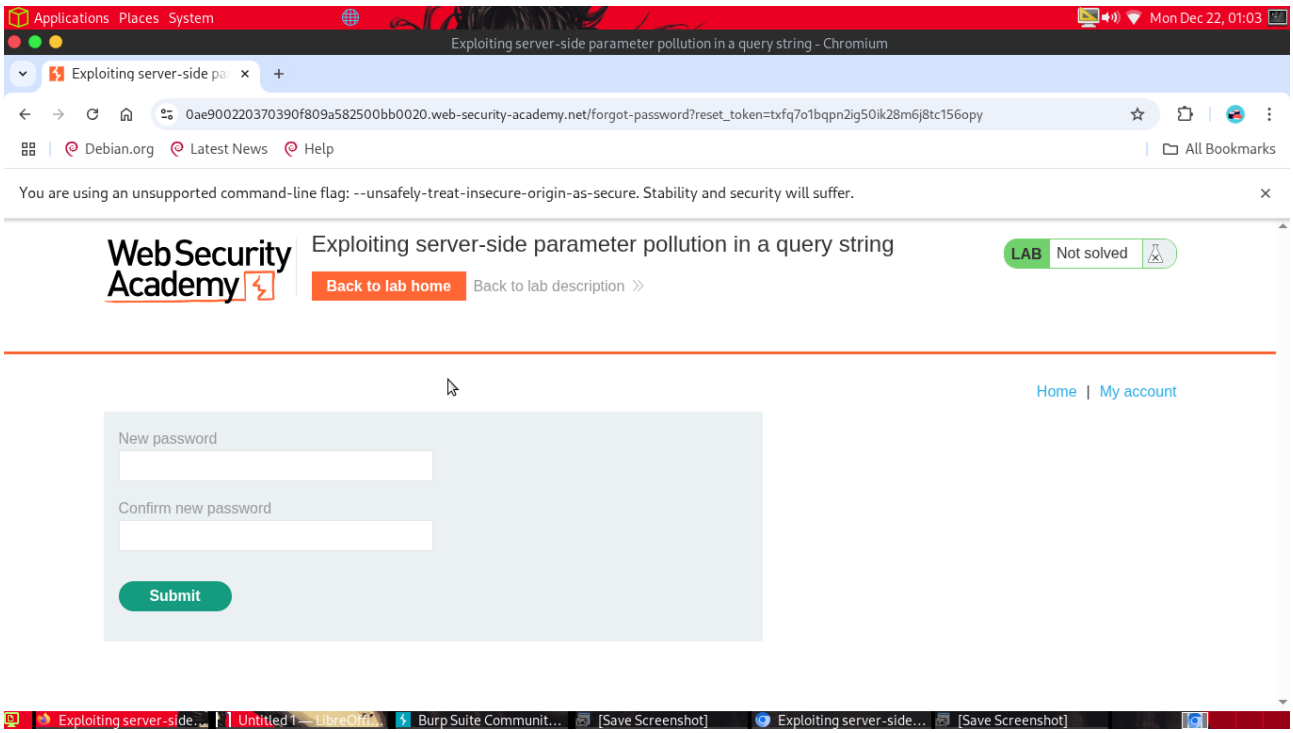
So, I injected the following payload:
csrf=UQ61BN3npqiWp8dewCBoKTEMWbmxkuu5&username=administrator%26field=reset_token

Explanation:

- The username=administrator targets the administrator account.
- The URL-encoded & (%26) is used to inject an extra parameter.
- This causes the server-side request to include field=reset_token, which may override or manipulate how the reset token is handled.

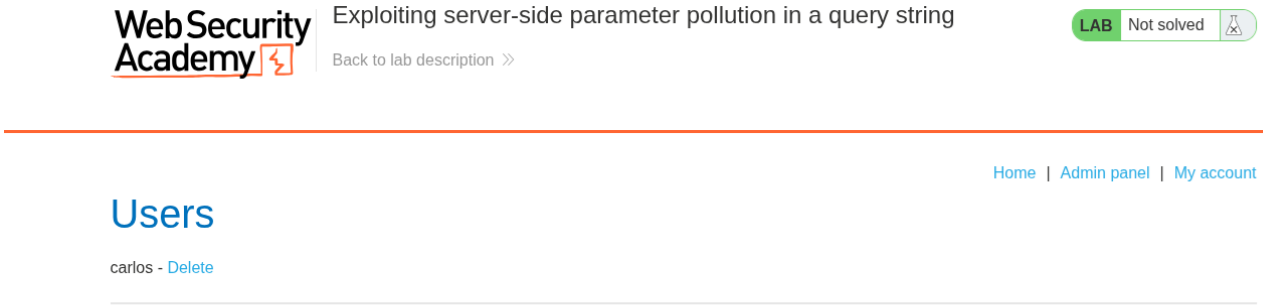


Using the resulting reset token, it is possible to change the administrator's password, demonstrating the severe impact of this vulnerability and how it can lead to complete account compromise.



We can see the admin panel,

And after accessing the admin panel, click **Delete Carlos**



8. Testing for Server-Side Parameter Pollution in REST Paths

8.1 Introduction

In some RESTful APIs, parameters are placed in the **URL path** instead of the query string.
For example:

`/api/users/123`

This path can be broken down as:

- `/api` → root API endpoint
- `/users` → resource name
- `/123` → parameter (user identifier)

Example scenario

Suppose an application allows users to edit profiles using a username. The front-end sends:

`GET /edit_profile.php?name=peter`

The server then makes an internal API request:

`GET /api/private/users/peter`

8.2 Exploiting path parameters

An attacker may be able to manipulate path parameters by injecting **path traversal sequences**.
For example:

`GET /edit_profile.php?name=peter%2f..%2fadmin`

This could result in the following server-side request:

`GET /api/private/users/peter/../admin`

If the server or backend API **normalizes the path**, it may resolve to:

`/api/private/users/admin`

Impact

If successful, this technique could allow an attacker to access or modify **unauthorized user data**, demonstrating a server-side parameter pollution vulnerability in REST paths.

9. Testing for Server-Side Parameter Pollution in Structured Data Formats

9.1 Introduction

Server-side parameter pollution can also occur when applications process **structured data formats** such as **JSON** or **XML**. To test for this, you can inject unexpected structured data into user input and observe how the server responds.

Example scenario

Suppose an application allows users to edit their profile. When a user updates their name, the browser sends:

```
POST /myaccount

name=peter
```

The server then sends the following request to an internal API:

```
PATCH /users/7312/update

{"name":"peter"}
```

Injecting additional parameters

An attacker can try to inject a new parameter into the structured data:

```
POST /myaccount

name=peter","access_level":"administrator
```

If the input is added to the JSON without proper validation or sanitization, the server-side request may become:

```
PATCH /users/7312/update

{"name":"peter","access_level":"administrator"}
```

Impact

This could result in the user **peter** being granted **administrator privileges**, demonstrating a server-side parameter pollution vulnerability in structured data formats.

9.2 Structured Data Injection Using JSON Input

Consider a similar scenario where user input is sent in **JSON format**. When a user edits their name, the browser sends the following request:

```
POST /myaccount

{"name": "peter"}
```

This results in the following server-side request to an internal API:

```
PATCH /users/7312/update

{"name":"peter"}
```

Injecting additional parameters

An attacker can attempt to inject a new parameter into the JSON input:

```
POST /myaccount

{"name":"peter\",\"access_level\":\"administrator"}
```

If the input is decoded and added to the server-side JSON without proper encoding, the internal API request may become:

```
PATCH /users/7312/update

{"name":"peter","access_level":"administrator"}
```

Impact

This could grant the user **peter administrator privileges**, demonstrating a server-side parameter pollution vulnerability in JSON-based inputs.

Injection in responses

Structured format injection can also occur in **responses**. For example, user input may be safely stored in a database but later embedded into a JSON response from a backend API without proper encoding. These vulnerabilities can often be detected and exploited using the same techniques as request-based injections.

9.3 Preventing Server-Side Parameter Pollution

To prevent server-side parameter pollution, applications should **strictly validate and encode user input** before including it in server-side requests.

Key prevention measures include:

- Use an **allowlist** to define which characters are permitted
- **Encode all other characters** before passing input to internal APIs
- Ensure all input matches the **expected format and structure**
- Avoid directly embedding user-controlled data into server-side requests

Final Words

Congratulations! You have completed the entire chapter on **API penetration testing**. I created this paper to explain API pentesting in the **simplest way possible**, and I hope you found it helpful.

If you liked it, please don't forget to **share this PDF with your friends**. Thank you for supporting me!

-Alham Rizvi

References

- PortSwigger Web Security Academy – **API Testing**
  https://portswigger.net/web-security/api-testing
- PortSwigger Web Security Academy – **Server-Side Parameter Pollution**
  https://portswigger.net/web-security/server-side-parameter-pollution
- PortSwigger Documentation – **Burp Suite**
  https://portswigger.net/burp