# PortSwigger API Testing Lab Notes

1. Exploiting an API endpoint using documentation

To solve the lab, find the exposed API documentation and delete carlos. You can log in to your own account using the following credentials: wiener:peter.

**Required knowledge**

To solve this lab, you'll need to know:

* What API documentation is.
* How API documentation may be useful to an attacker.
* How to discover API documentation.

Solution

1. In Burp's browser, log in to the application using the credentials wiener:peter and update your email address.
2. In Proxy > HTTP history, right-click the ***PATCH /api/user/wiener*** request and select Send to Repeater.
3. Go to the Repeater tab. Send the ***PATCH /api/user/wiener*** request. Notice that this retrieves credentials for the user wiener.
4. Remove /wiener from the path of the request, so the endpoint is now /api/user, then send the request. Notice that this returns an error because there is no user identifier.
5. Remove /user from the path of the request, so the endpoint is now /api, then send the request. Notice that this retrieves API documentation.
6. Right-click the response and select Show response in browser. Copy the URL.
7. Paste the URL into Burp's browser to access the documentation. Notice that the documentation is interactive.
8. To delete Carlos and solve the lab, click on the DELETE row, enter carlos, then click Send request.
9. Exploiting server-side parameter pollution in a query string

To solve the lab, log in as the administrator and delete carlos.

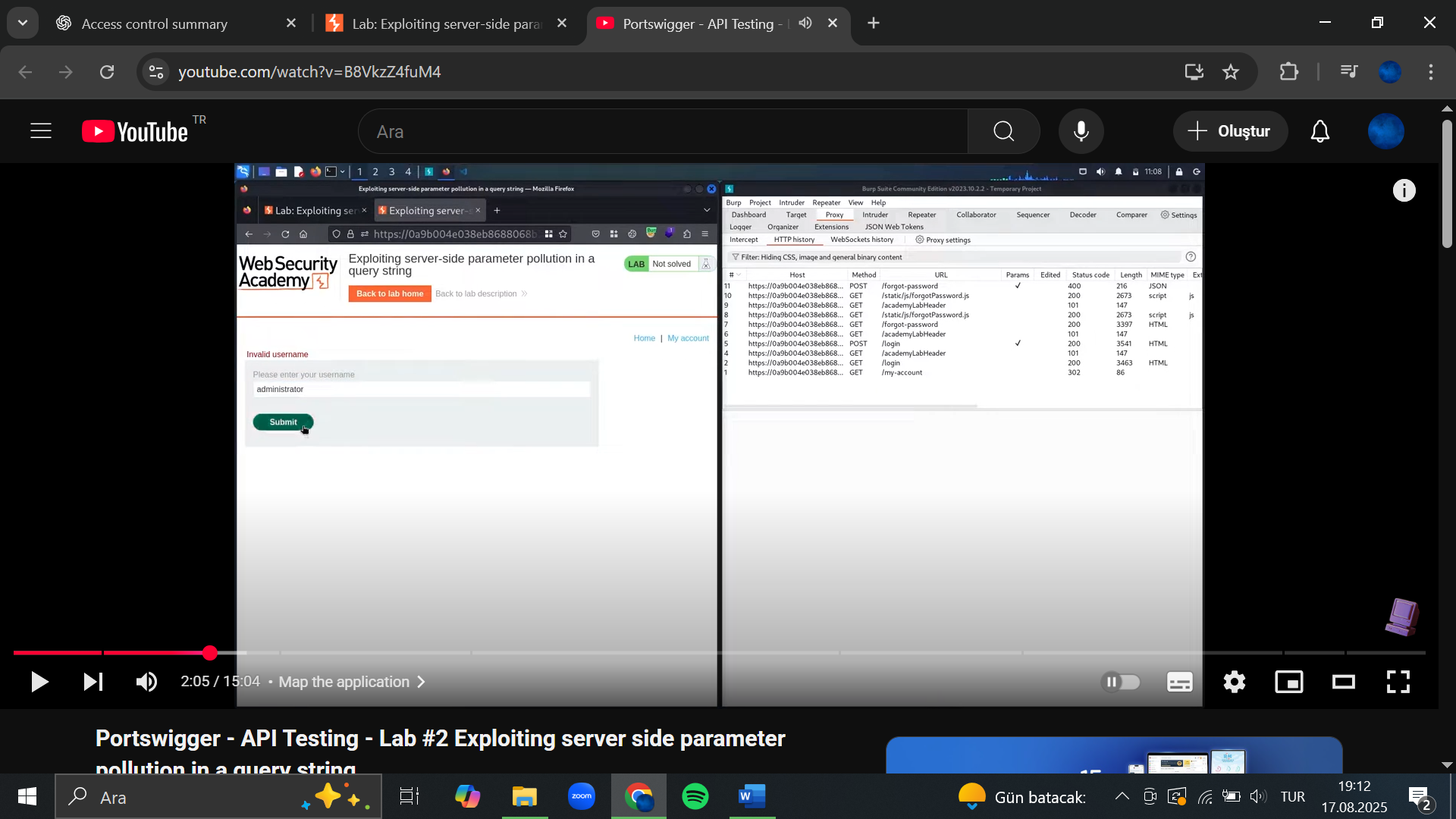
**Required knowledge**

To solve this lab, you'll need to know:

* How to use URL query syntax to attempt to change a server-side request.
* How to use error messages to build an understanding of how a server-side API processes user input.

Solution

1. In Burp's browser, trigger a password reset for the administrator user.



1. In **Proxy > HTTP history**, notice the POST /forgot-password request and the related /static/js/forgotPassword.js JavaScript file.

A screenshot of a computer

AI-generated content may be incorrect.

A screenshot of a computer

AI-generated content may be incorrect.

1. Right-click the ***POST /forgot-password*** request and select **Send to Repeater**.
2. In the **Repeater** tab, resend the request to confirm that the response is consistent.
3. Change the value of the username parameter from administrator to an invalid username, such as administratorx. Send the request. Notice that this results in an Invalid username error message.
4. Attempt to add a second parameter-value pair to the server-side request using a URL-encoded & character. For example, add URL-encoded &x=y:

***username=administrator%26x=y***

Send the request. Notice that this returns a Parameter is not supported error message. This suggests that the internal API may have interpreted &x=y as a separate parameter, instead of part of the username.

A screenshot of a computer

AI-generated content may be incorrect.

1. Attempt to truncate the server-side query string using a URL-encoded # character:

***username=administrator%23***

Send the request. Notice that this returns a Field not specified error message. This suggests that the server-side query may include an additional parameter called field, which has been removed by the # character.

1. Add a field parameter with an invalid value to the request. Truncate the query string after the added parameter-value pair. For example, add URL-encoded &field=x#:

***username=administrator%26field=x%23***

Send the request. Notice that this results in an Invalid field error message. This suggests that the server-side application may recognize the injected field parameter.

1. Brute-force the value of the field parameter:
   1. Right-click the POST /forgot-password request and select **Send to Intruder**.
   2. In the **Intruder** tab, add a payload position to the value of the field parameter as follows:

***username=administrator%26field=§x§%23***

* 1. In the **Payloads** side panel, under **Payload configuration**, click **Add from list**. Select the built-in **Server-side variable names** payload list, then start the attack.
  2. Review the results. Notice that the requests with the username and email payloads both return a 200 response.

1. Change the value of the field parameter from x# to email:

***username=administrator%26field=email%23***

Send the request. Notice that this returns the original response. This suggests that email is a valid field type.

email acts as a **stepping stone** → it confirms your injection approach is valid before you go for the juicy target (reset\_token)

1. In **Proxy > HTTP history**, review the /static/js/forgotPassword.js JavaScript file. Notice the password reset endpoint, which refers to the reset\_token parameter:

***/forgot-password?reset\_token=${resetToken}***

1. In the **Repeater** tab, change the value of the field parameter from email to reset\_token:

***username=administrator%26field=reset\_token%23***

Send the request. Notice that this returns a password reset token. Make a note of this.

1. In Burp's browser, enter the password reset endpoint in the address bar. Add your password reset token as the value of the reset\_token parameter . For example:

***/forgot-password?reset\_token=123456789***

1. Set a new password.
2. Log in as the administrator user using your password.
3. Go to the **Admin panel** and delete carlos to solve the lab.
4. Finding and exploiting an unused API endpoint

To solve the lab, exploit a hidden API endpoint to buy a **Lightweight l33t Leather Jacket**. You can log in to your own account using the following credentials: wiener:peter.

**Required knowledge**

To solve this lab, you'll need to know:

* How to use error messages to construct a valid request.
* How HTTP methods are used by RESTful APIs.
* How changing the HTTP method can reveal additional functionality.

Solution

1. In Burp's browser, access the lab and click on a product.
2. In **Proxy > HTTP history**, notice the API request for the product. For example, /api/products/3/price.
3. Right-click the API request and select **Send to Repeater**.
4. In the **Repeater** tab, change the HTTP method for the API request from GET to OPTIONS, then send the request. Notice that the response specifies that the GET and PATCH methods are allowed.
5. Change the method for the API request from GET to PATCH, then send the request. Notice that you receive an Unauthorized message. This may indicate that you need to be authenticated to update the order.
6. In Burp's browser, log in to the application using the credentials wiener:peter.
7. Click on the **Lightweight "l33t" Leather Jacket** product.
8. In **Proxy > HTTP history**, right-click the API/products/1/price request for the leather jacket and select **Send to Repeater**.
9. In the **Repeater** tab, change the method for the API request from GET to PATCH, then send the request. Notice that this causes an error due to an incorrect Content-Type. The error message specifies that the Content-Type should be application/json.
10. Add a Content-Type header and set the value to application/json.
11. Add an empty JSON object {} as the request body, then send the request. Notice that this causes an error due to the request body missing a price parameter.
12. Add a price parameter with a value of 0 to the JSON object ***{"price":0}.*** Send the request.
13. In Burp's browser, reload the leather jacket product page. Notice that the price of the leather jacket is now $0.00.
14. Add the leather jacket to your basket.
15. Go to your basket and click **Place order** to solve the lab.
16. Exploiting a mass assignment vulnerability

To solve the lab, find and exploit a mass assignment vulnerability to buy a **Lightweight l33t Leather Jacket**. You can log in to your own account using the following credentials: wiener:peter.

**Required knowledge**

To solve this lab, you'll need to know:

* What mass assignment is.
* Why mass assignment may result in hidden parameters.
* How to identify hidden parameters.
* How to exploit mass assignment vulnerabilities.

Mass assignment happens when a web app automatically takes **all parameters from user input** (like a JSON body or form submission) and directly maps them onto backend objects or database fields **without filtering or validation**.

Solution

1. In Burp's browser, log in to the application using the credentials wiener:peter.
2. Click on the **Lightweight "l33t" Leather Jacket** product and add it to your basket.
3. Go to your basket and click **Place order**. Notice that you don't have enough credit for the purchase.
4. In **Proxy > HTTP history**, notice both the GET and POST API requests for /api/checkout.
5. Notice that the response to the GET request contains the same JSON structure as the POST request. Observe that the JSON structure in the GET response includes a ***chosen\_discount*** parameter, which is not present in the POST request.
6. Right-click the POST /api/checkout request and select **Send to Repeater**.
7. In Repeater, add the chosen\_discount parameter to the request. The JSON should look like the following:

***{***

***"chosen\_discount":{***

***"percentage":0***

***},***

***"chosen\_products":[***

***{***

***"product\_id":"1",***

***"quantity":1***

***}***

***]***

***}***

1. Send the request. Notice that adding the chosen\_discount parameter doesn't cause an error.
2. Change the chosen\_discount value to the string "x", then send the request. Observe that this results in an error message as the parameter value isn't a number. This may indicate that the user input is being processed.
3. Change the chosen\_discount percentage to 100, then send the request to solve the lab.