SQL Injection Vulnerability Exploitation

Labs by PortSwigger: Web Security Academy

Write-Up

Pavel Pecheniuk

Introduction

SQL injection (SQLi) is a web security vulnerability that allows an attacker to interfere with the queries that an application makes to its database. This can allow an attacker to view data that they are not normally able to retrieve. This might include data that belongs to other users, or any other data that the application can access. In many cases, an attacker can modify or delete this data, causing persistent changes to the application's content or behaviour. In some situations, an attacker can escalate a SQL injection attack to compromise the underlying server or other back-end infrastructure. It can also enable them to perform denial-of-service attacks.

Some common SQL injection examples include:

- Retrieving hidden data, where you can modify a SQL query to return additional results.
- Subverting application logic, where you can change a query to interfere with the application's logic.
- UNION attacks, where you can retrieve data from different database tables.
- Blind SQL injection, where the results of a query you control are not returned in the application's responses.

In this write-up, I document the completion of the lab challenges where I am practicing my exploitation skills of common SQLi vulnerabilities, concentrating on ones listed above in the Introduction section. Throughout the labs I am using Burp Suite as a web vulnerability penetrating tool.

Lab 1. SQL injection vulnerability in WHERE clause allowing retrieval of hidden data

Task:

This lab contains a SQL injection vulnerability in the product category filter. When the user selects a category, the application carries out a SQL query like the following:

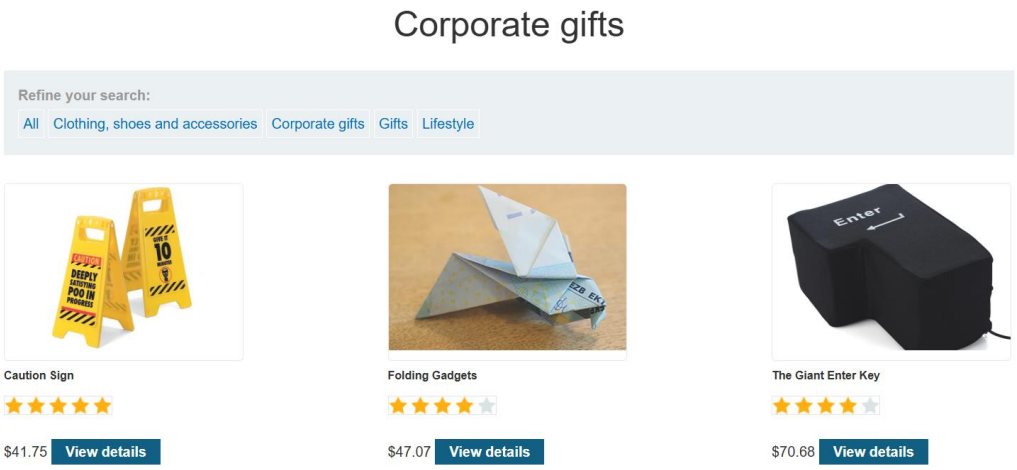
*SELECT * FROM products WHERE category = 'Gifts' AND released = 1*

To solve the lab, perform a SQL injection attack that causes the application to display one or more unreleased products.

Solution:

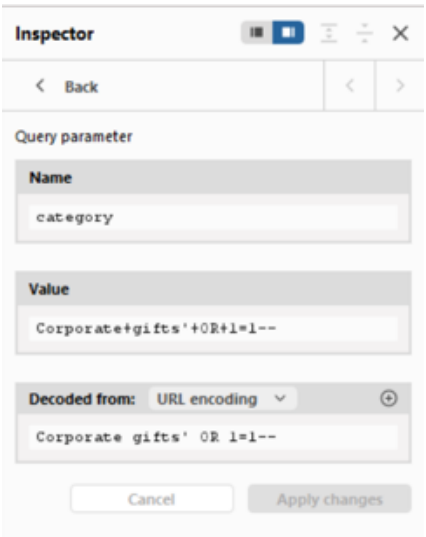
Let's explore the field of work – an online shop web page.

We know that a vulnerability is exposed when dealing with the product categories, so let's navigate to one of the categories, in our case, to the Corporate gifts.



Before requesting the Corporate Gifts category, we enable interception of the web traffic which is done via *Main bar* > *Proxy* > *Intercept on*. Then we right click on the necessary request that sets the product category filter and select *Send to Repeater*.

After navigating to Repeater, we need modify the request in a way that allows disclosure of the unreleased products. To do so, we change the value *Corporate gifts* of the *category* parameter by adding a construction *'+OR+1=1--*.



Let's break this step down:

When we clicked the Corporate gifts category, our browser requested the URL that structurally looks like this:

https://insecure-website.com/products?category=Corporategifts

An application (online shop) requests the database to retrieve information about the released products by making a following query:

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

Where database should return all details (*) from the table named *products* where the *category* is *Corporate gifts* and hide products that haven't been released (*released = 1*).

With applied the construction *'+OR+1=1--* a request to the application looks like this:

https://insecure-website.com/products?category=Corporategifts'+OR+1=1--

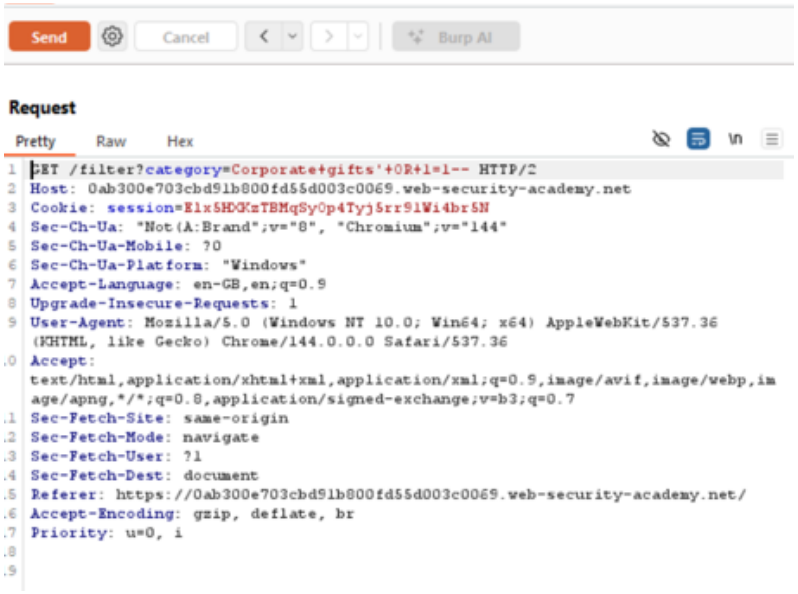
And a SQL query to the database is:

*SELECT * FROM products WHERE category = 'Corporategifts' OR 1=1--' AND released = 1*

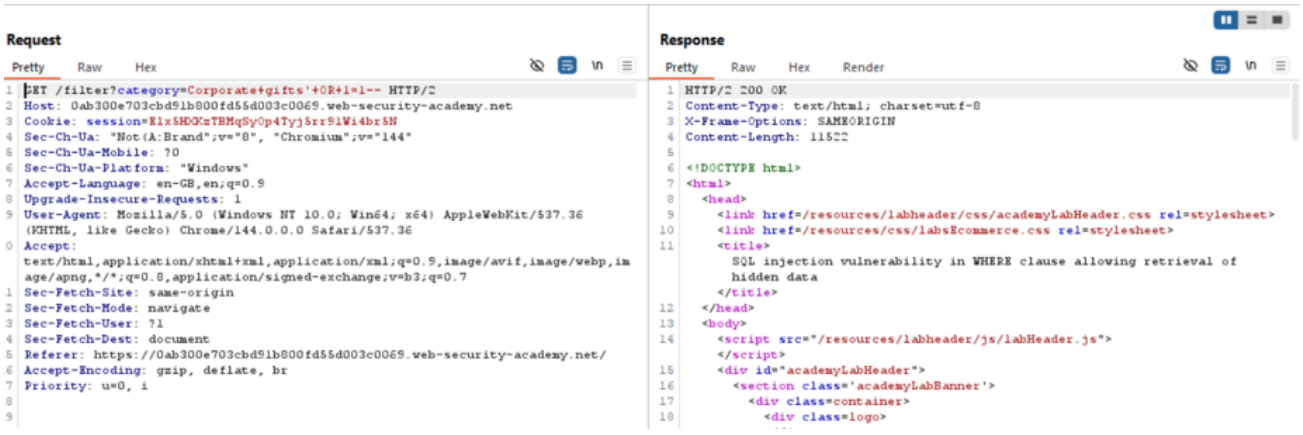
SQL's sign -- is the comment indicator, so the input after this sign is interpreted as a comment. As a result, the condition *released = 1* is ignored and all the products, released and unreleased, will be displayed.

The query thus returns all products where either the *category* is *Corporate gifts*, or *1* is equal to *1*. Regardless of the first condition, the second condition 1=1 is obviously always true, which allows to successfully execute this query and return all products from the Corporate gifts category.

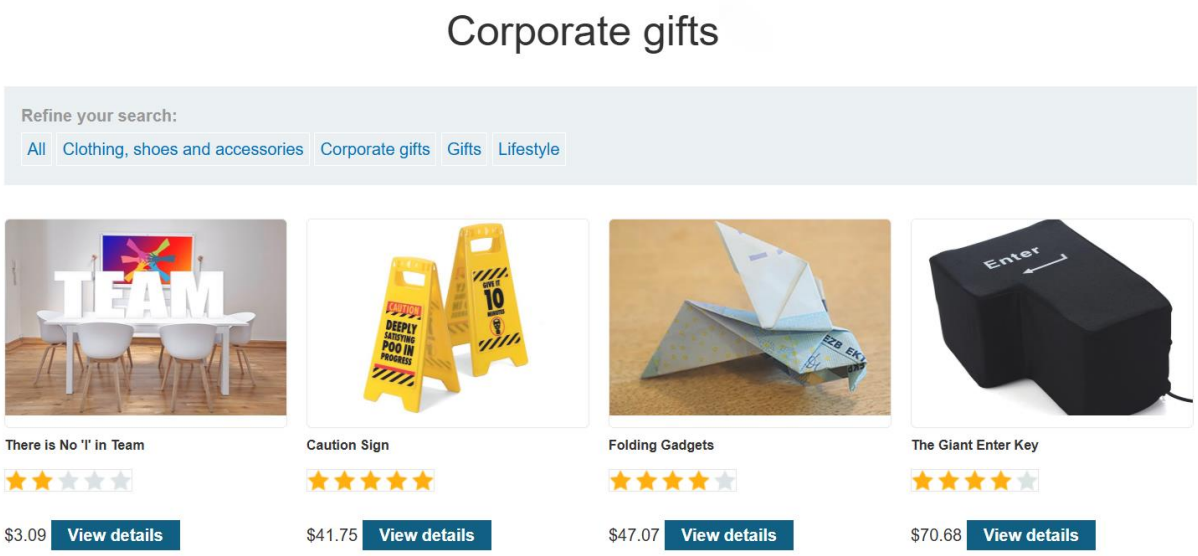
Let's get back to our lab. After modification and clicking *Apply changes* our request looks like this:



Let's disable interception in Proxy and send this updated request by clicking the orange button *Send* and observe the successful response (200 OK) displayed in Burp Suite.



And verify the successful response in the web page of online shop, since now we can see one unreleased product in the Corporate gifts. Lab is solved.



Lab 2. SQL injection vulnerability allowing login bypass

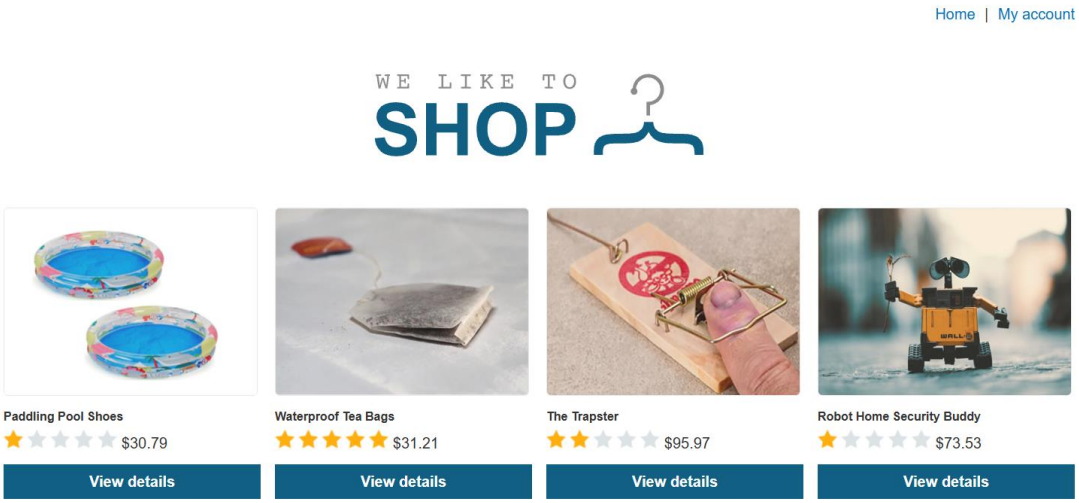
Task:

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

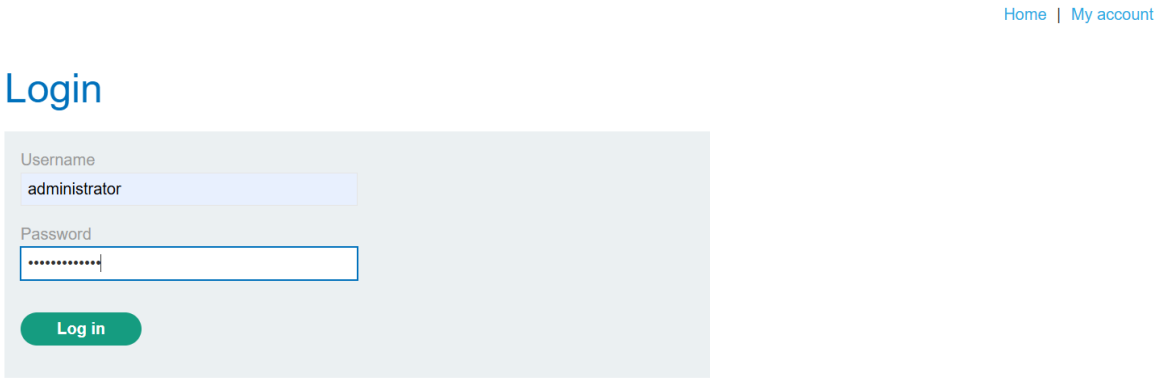
To solve the lab, perform a SQL injection attack that logs in to the application as the administrator user.

Solution:

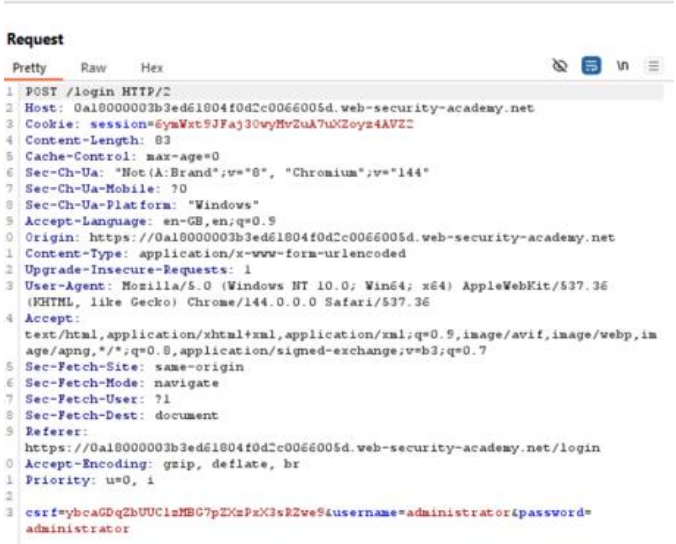
We enter another online shopping webpage and navigate to My Account tab.



Interception is enabled, and we are logging in with the username *administrator* and the password *administrator*.



Here is the login request at Repeater. To make use of the vulnerability in the login logic, we need to modify the value of the *username* parameter, changing it from *administrator* to *administrator'*--.



Here we submitted our username administrator and password administrator to the application to log in. The application checks provided credentials by performing the following SQL query:

*SELECT * FROM users WHERE username = 'administrator' AND password = 'administrator'*

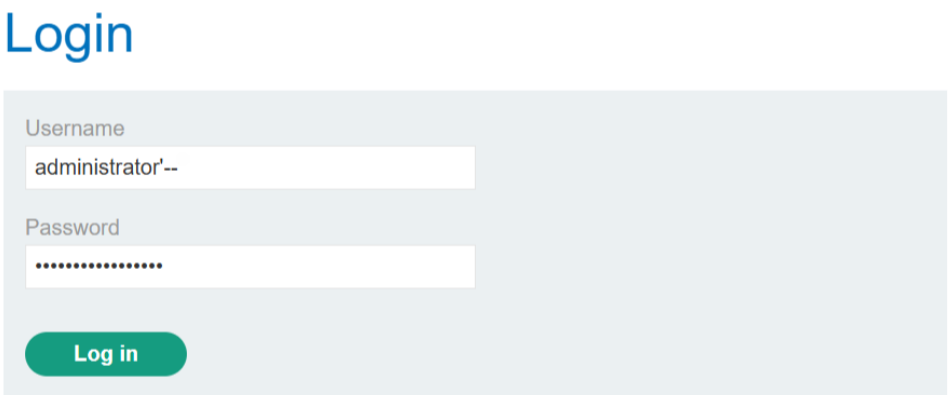
Database then checks the table *users* to find details of the user with these credentials. If the query returns details of that user, then the login is successful and we are logged in. Otherwise, access is denied.

In our case, we can log in as any user without needing a password. We again utilize the SQL comment indicator -- to remove the password part within the WHERE clause. Here how the query would look like if we submit the username *administrator'*-- and the blank password:

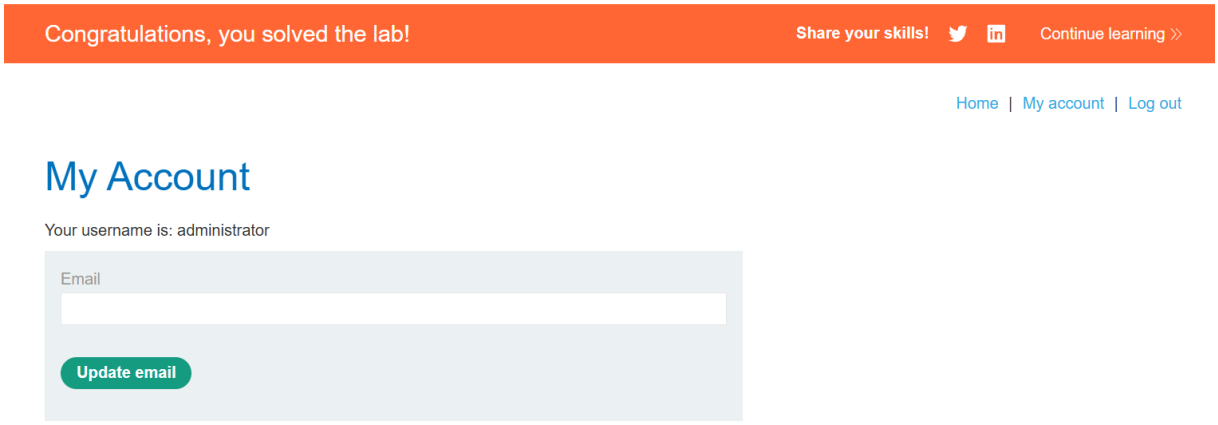
*SELECT * FROM users WHERE username = 'administrator'--' AND password = ''*

The user with the name administrator will be confirmed and the attacker will be successfully logged in as that user.

We could actually use Intruder to change the values of the username and the password, forward updated request and obtain the needed result. However, another approach here is to simply try to log in with the username *administrator'*-- and the password *administrator*, for example, right in the browser as shown below.



And we successfully logged in abusing the vulnerability in the login logic.



Lab 3. SQL injection UNION attack, determining the number of columns returned by the query

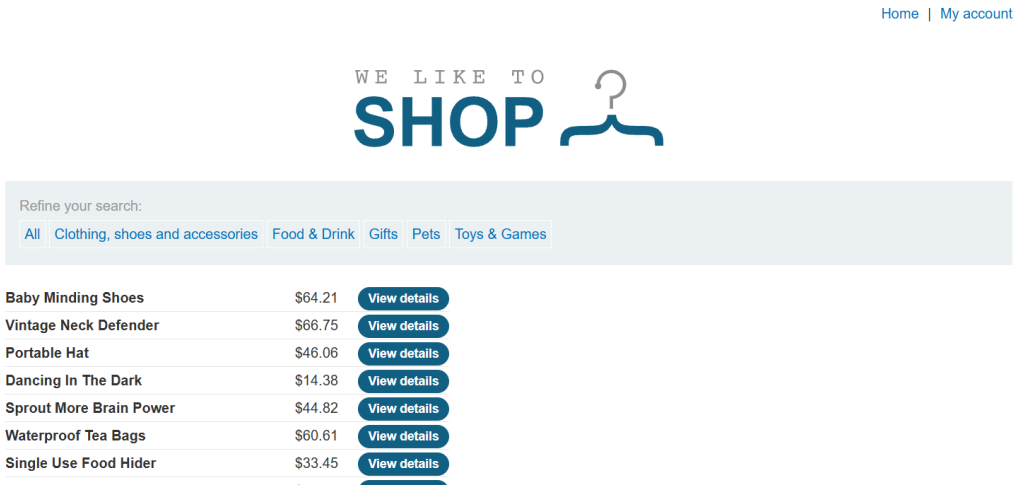
Task:

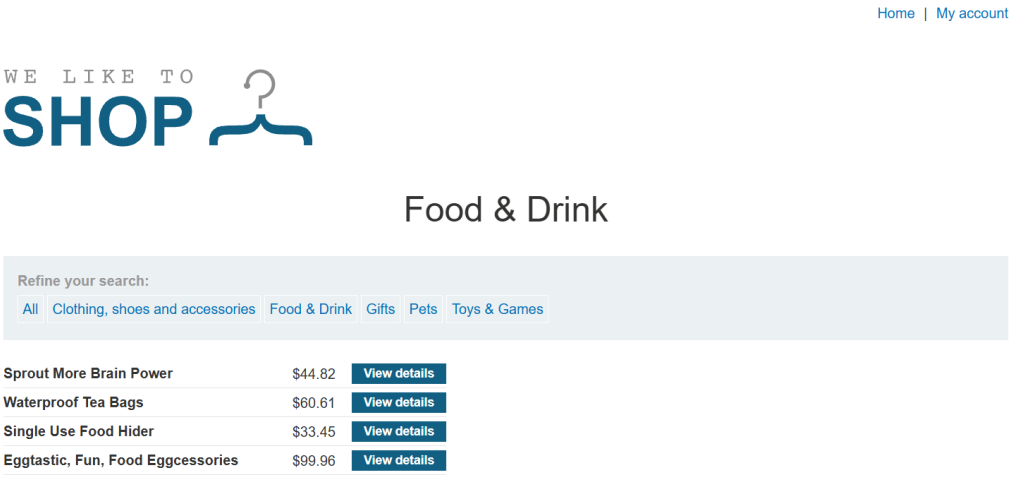
This lab contains a SQL injection vulnerability in the product category filter. The results from the query are returned in the application's response, so you can use a UNION attack to retrieve data from other tables. The first step of such an attack is to determine the number of columns that are being returned by the query.

To solve the lab, determine the number of columns returned by the query by performing a SQL injection UNION attack that returns an additional row containing null values.

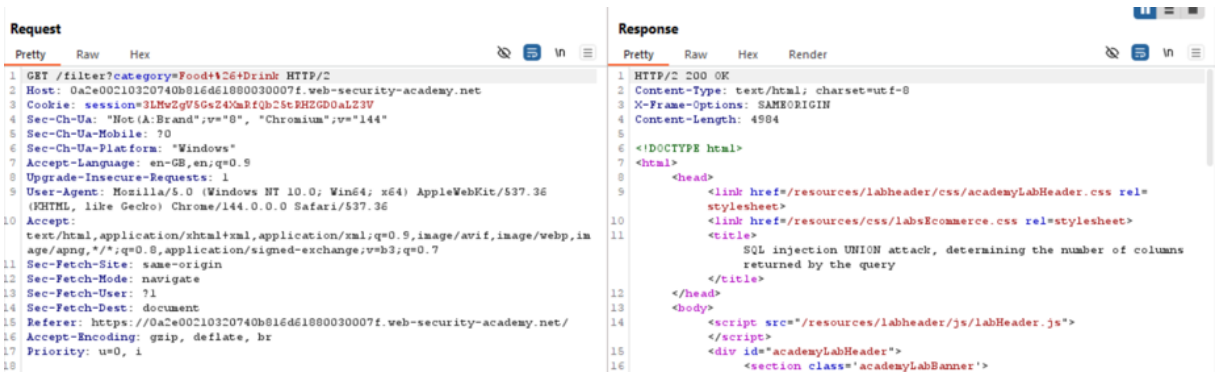
Solution:

Let’s navigate to an online shopping page and select one of the categories since a vulnerability to be exploited is located in the product category filter. Let it be the Food & Drink category.

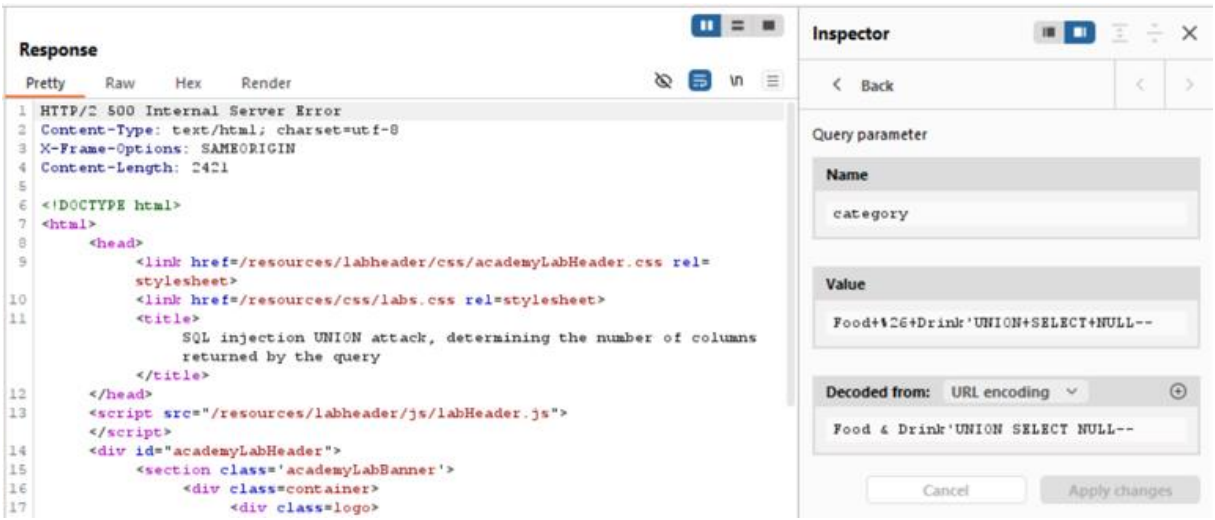




Intercepting the category request and sending it to Intruder.



Let's modify the *category* parameter by adding *'+UNION+SELECT+NULL--*. Submitting this updated request shows us an error response.



In general, the *UNION* SQL keyword enables to execute one or more *SELECT* queries and append the results to the original query. For example:

SELECT a, b FROM table1 UNION SELECT c, d FROM table2

Interpretation: This query returns a single entity with results containing two columns with values from columns *a* and *b* in *table1* and columns *c* and *d* in *table2*.

One important requirement for *UNION* query to work is that individual queries which *UNION* query consists of must return the same number of columns. For the lab that's particularly relevant.

How to determine the number of columns that are returned by the original query? To do this we can submit the *UNION SELECT* payloads, where each next payload would contain incrementing number of *NULL* values, such as:

UNION SELECT NULL--

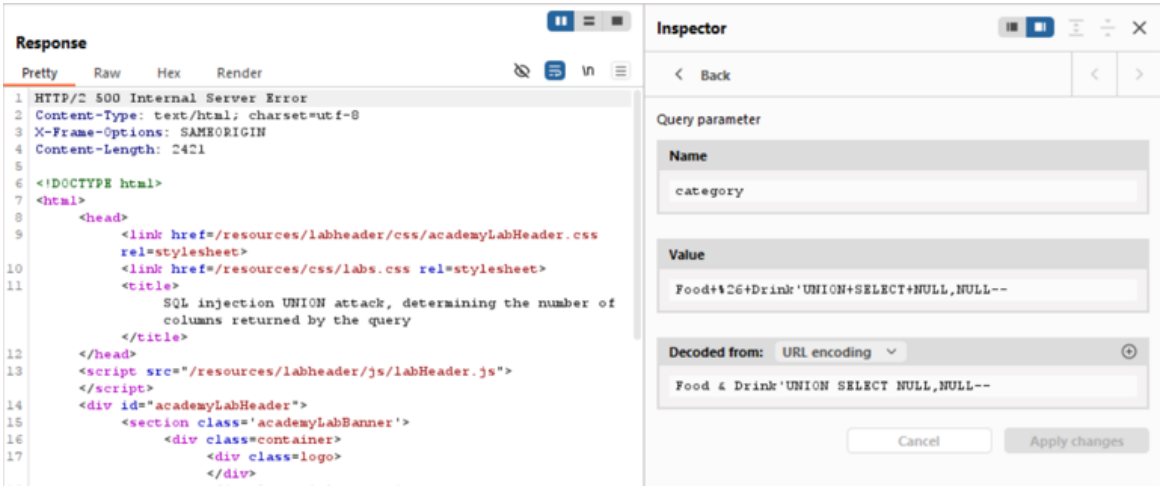
UNION SELECT NULL, NULL--

If the number of nulls does not match the number of columns, the database returns an error and there will be no successful responses.

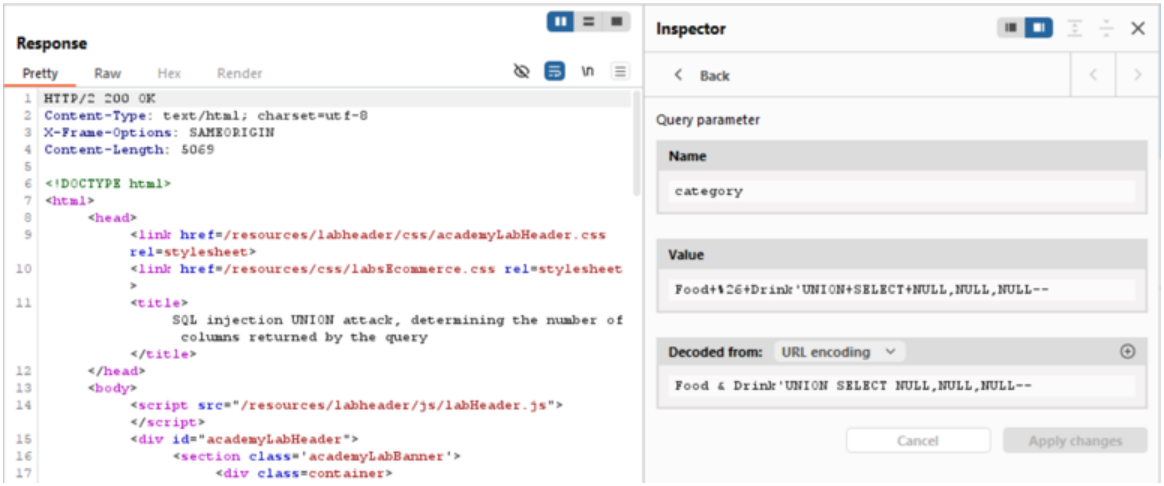
All queries combined using a UNION, INTERSECT or EXCEPT operator must have an equal number of expressions in their target lists.

Since the data types in each column must be compatible between the individual queries, which is also a must-meet requirement for a *UNION* query to work, we use *NULL*, as this data type can be converted to every common data type, maximizing the odds for the payload being successfully executed.

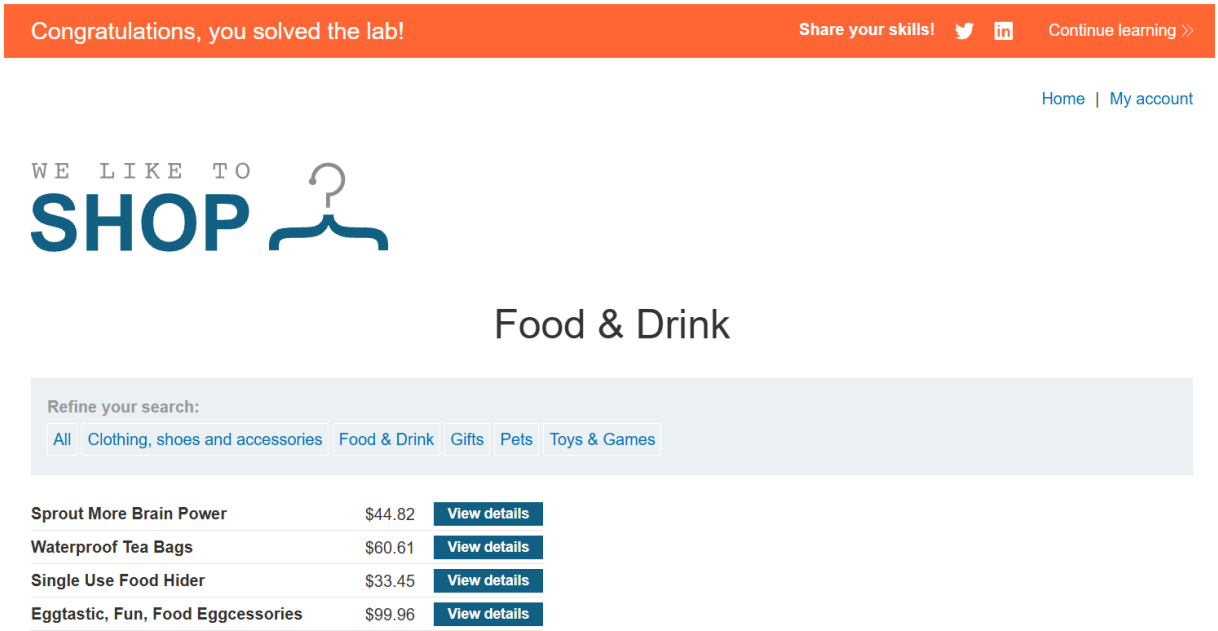
Back to our lab. We encountered an error, so there are more columns which number is to be determined. Let's add another *NULL* and submit the request.



Error again. Let's add another one and see the result.



Success! We established that there are 3 columns returned by the query and managed to exploit the vulnerability. The lab is solved.



Lab 4. Blind SQL injection with time delays

Task:

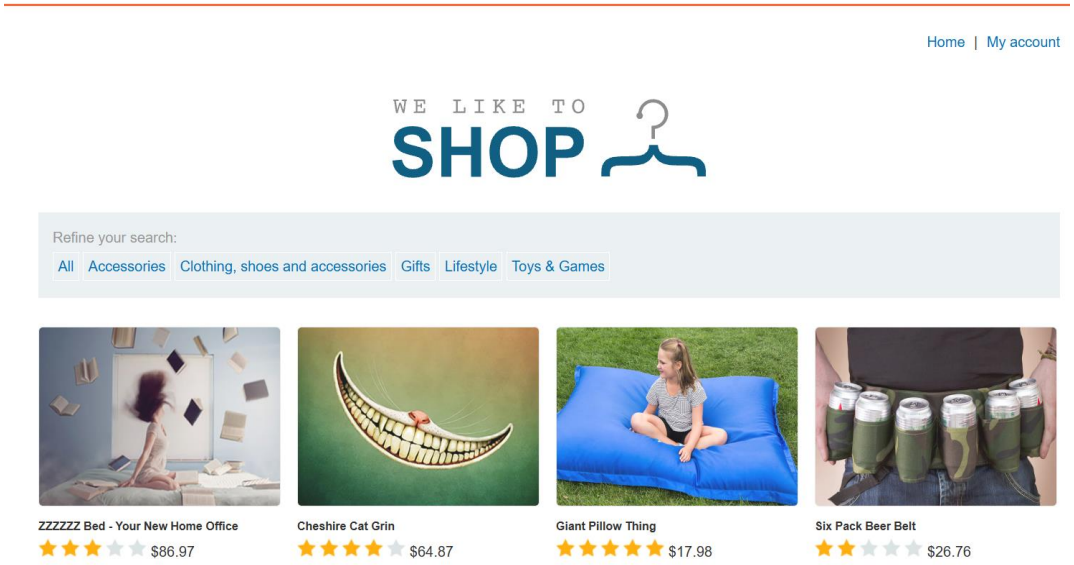
This lab contains a blind SQL injection vulnerability. The application uses a tracking cookie for analytics, and performs a SQL query containing the value of the submitted cookie.

The results of the SQL query are not returned, and the application does not respond any differently based on whether the query returns any rows or causes an error. However, since the query is executed synchronously, it is possible to trigger conditional time delays to infer information.

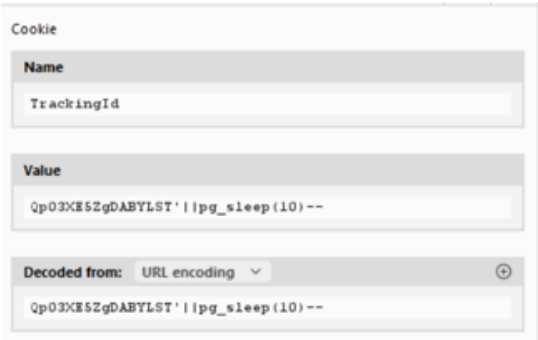
To solve the lab, exploit the SQL injection vulnerability to cause a 10 second delay.

Solution:

Let's visit the front page of the shop and simultaneously intercept the request.



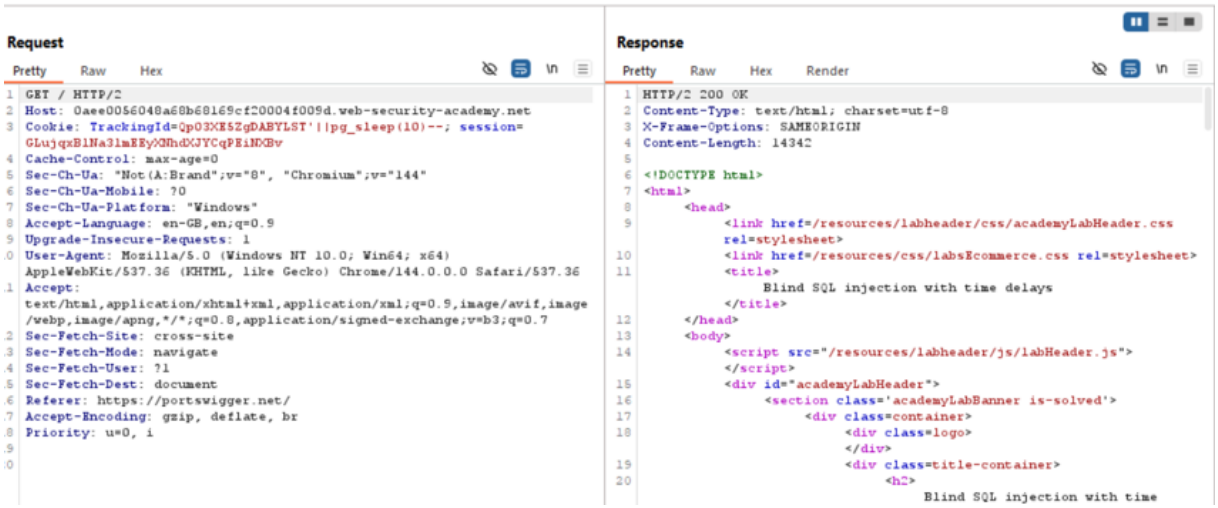
Sending the request to Intruder. *Cookie* and its parameter *TrackingId* are of our interest. Let's change the value of *TrackingId* accordingly as shown below.



'||*pg_sleep(10)*—is the construction that causes an unconditional time delay of 10 seconds during the process of a query. It's also important to note that this construction's syntax is exclusively for the MySQL databases. It would look different if using it against other databases like PostgreSQL, Microsoft and so on.

It is hard to demonstrate successful time delay execution in the write-up 😊

However, after submission of the updated request we obtain the successful response.



Verification in the browser shows that everything worked correct and lab is solved.