**EXPERIMENT 07**

**AIM:** Implementation of Data Validation Experiment (SQL Injection)

**TO DO:**

1. What is SQL Injection?
2. How and why SQL Injection attack is performed?
3. How to prevent SQL Injection?
4. SQL Injection Process
   1. SQL Injection vulnerability allowing login bypass
   2. SQL Injection vulnerability allowing retrieval of hidden data

**THEORY:**

**SQL Injection:**

SQL injection (SQLi) is a web security vulnerability that allows an attacker to interfere with the queries that an application makes to its database. It generally allows an attacker to view data that they are not normally able to retrieve. This might include data belonging to other users, or any other data that the application itself is able to 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, or perform a denial-of-service attack.

**Impact of SQL Injection Attack:**

A successful SQL injection attack can result in unauthorised access to sensitive data, such as passwords, credit card details, or personal user information. Many high-profile data breaches in recent years have been the result of SQL injection attacks, leading to reputational damage and regulatory fines. In some cases, an attacker can obtain a persistent backdoor into an organisation's systems, leading to a long-term compromise that can go unnoticed for an extended period.

**How is the SQL Injection attack performed?**

A SQL injection attack targets vulnerabilities in dynamic SQL statements. Think of a dynamic SQL statement like a multivariate function in mathematics, of which the parameters are fixed, while the values substituted in the independent variables determine the result.

Similarly, a dynamic SQL statement also consists of a predetermined set of parameters (such as a web form), of which the complete statement is only generated when a user fills in their inputs. See the following example of a SQL statement of a login form:

| SELECT \* FROM users WHERE username = ‘$username’ AND password = bcrypt (‘$password’) |
| --- |

After the user enters their username and password, the statement would be completed, after which a query would be sent to the server to retrieve the user’s information from the database.

When a vulnerability exists in a dynamic SQL statement, the attacker would be able to enter complex scripts into the forms to interfere with the preexisting parameters to alter the meaning of the complete statement.

**Why is the SQL Injection attack performed?**

To perform an SQL injection attack, an attacker must locate a vulnerable input in a web application or webpage. When an application or webpage contains a SQL injection vulnerability, it uses user input in the form of an SQL query directly. The hacker can execute a specifically crafted SQL command as a malicious cyber intrusion. Then, leveraging malicious code, a hacker can acquire a response that provides a clear idea about the database construction and thereby access to all the information in the database.

SQL serves as the way of communication to the database. SQL statements are used to retrieve and update data in the database. Attackers use malicious SQL statements in the input box, and in response, the database presents sensitive information. This exploit of security aims at gaining access to the unauthorised data of a website or application. Several websites and web applications store data in SQL databases.

**How to prevent SQL?**

Most instances of SQL injection can be prevented by using parameterized queries (also known as prepared statements) instead of string concatenation within the query.

The following code is vulnerable to SQL injection because the user input is concatenated directly into the query:

| String query = "SELECT \* FROM products WHERE category = '"+ input + "'";  Statement statement = connection.createStatement();  ResultSet resultSet = statement.executeQuery(query); |
| --- |

This code can be easily rewritten in a way that prevents the user input from interfering with the query structure:

| PreparedStatement statement = connection.prepareStatement("SELECT \* FROM products WHERE category = ?");  statement.setString(1, input);  ResultSet resultSet = statement.executeQuery(); |
| --- |

Parameterized queries can be used for any situation where untrusted input appears as data within the query, including the WHERE clause and values in an INSERT or UPDATE statement. They can't be used to handle untrusted input in other parts of the query, such as table or column names, or the ORDER BY clause. Application functionality that places untrusted data into those parts of the query will need to take a different approach, such as white-listing permitted input values, or using different logic to deliver the required behaviour.

For a parameterized query to be effective in preventing SQL injection, the string that is used in the query must always be a hard-coded constant, and must never contain any variable data from any origin. Do not be tempted to decide case-by-case whether an item of data is trusted, and continue using string concatenation within the query for cases that are considered safe. It is all too easy to make mistakes about the possible origin of data, or for changes in other code to violate assumptions about what data is tainted.

**SQL Injection Process:**

1. **SQL injection vulnerability allowing login bypass**

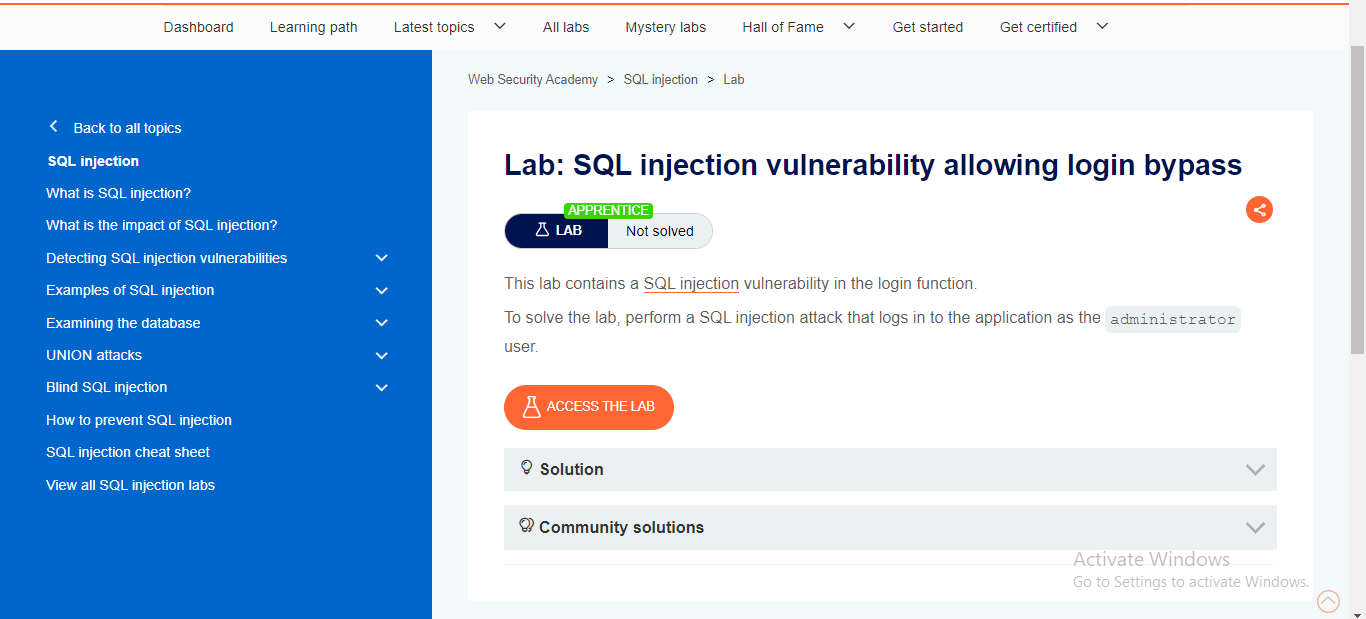
In cases where the results of an SQL query are returned within the application's responses, an attacker can leverage an SQL injection vulnerability to retrieve data from other tables within the database. Here, an attacker can log in as any user without a password simply by using the SQL comment sequence -- to remove the password check from the WHERE clause of the query. For example, submitting the username administrator'-- and a blank password results in the following query:

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

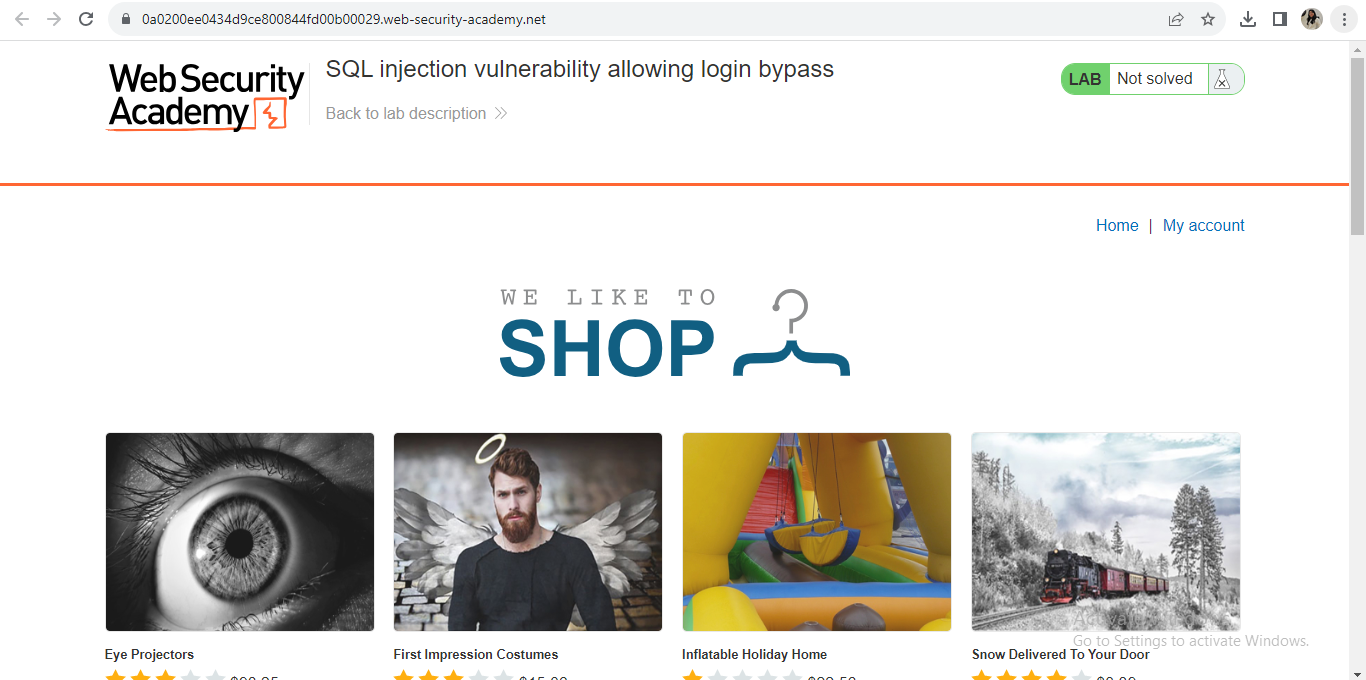
This query returns the user whose username is administrator and successfully logs the attacker in as that user. Consider a shopping application that displays products in different categories.

**IMPLEMENTATION:**

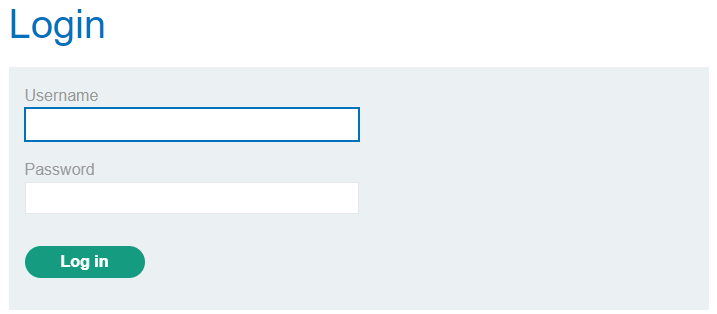
1. Go to Burp Labs Bypass: [Link](https://portswigger.net/web-security/sql-injection/lab-login-bypass)
2. Click on Access Lab.



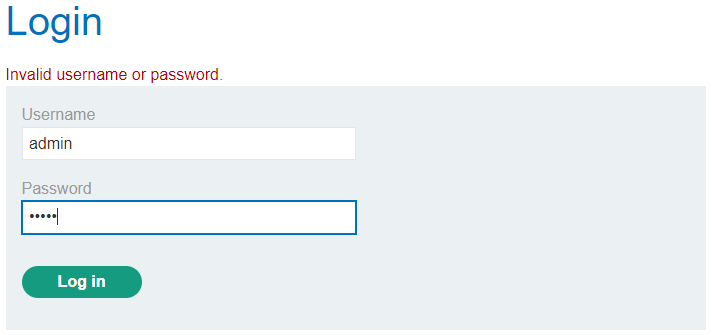
1. Here we can see that there is a My Account button in which we will enter the username and password for login as admin.



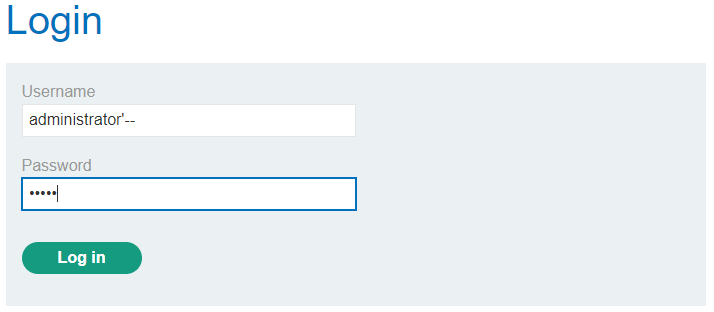
1. After clicking to My Account the screen appears for username and password.



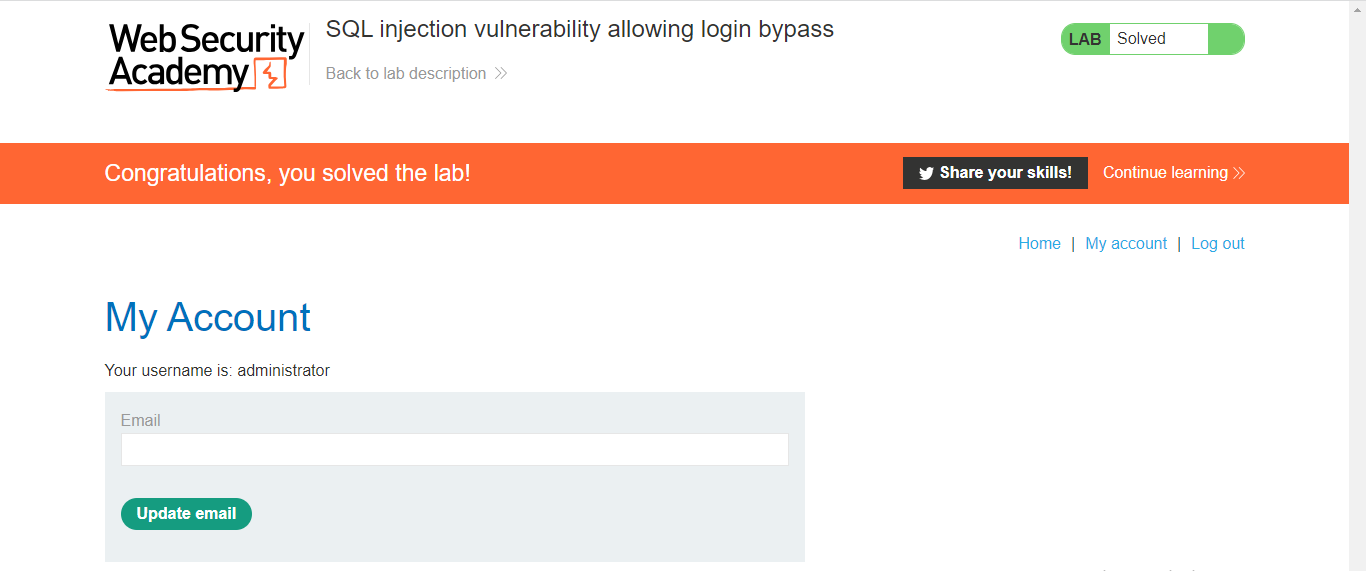
1. After that we will enter username as admin and password as admin and try to get into the website. After clicking the LOGIN button an error will come and show as Invalid username and password.



1. After this we will perform an SQL injection attack that logs in to the application as the administrator user.



1. After then we can see that we got success in LOGIN as Administrator.



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

Consider a shopping application that displays products in different categories. This causes the application to make an SQL query to retrieve details of the relevant products from the database:

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

This SQL query asks the database to return all details (\*) from the products table where the category is Gifts and released is 1. The restriction released = 1 is being used to hide products that are not released. For unreleased products, presumably released = 0.

The key thing here is that the double-dash sequence -- is a comment indicator in SQL, and means that the rest of the query is interpreted as a comment. This effectively removes the remainder of the query, so it no longer includes AND released = 1. This means that all products are displayed, including unreleased products.

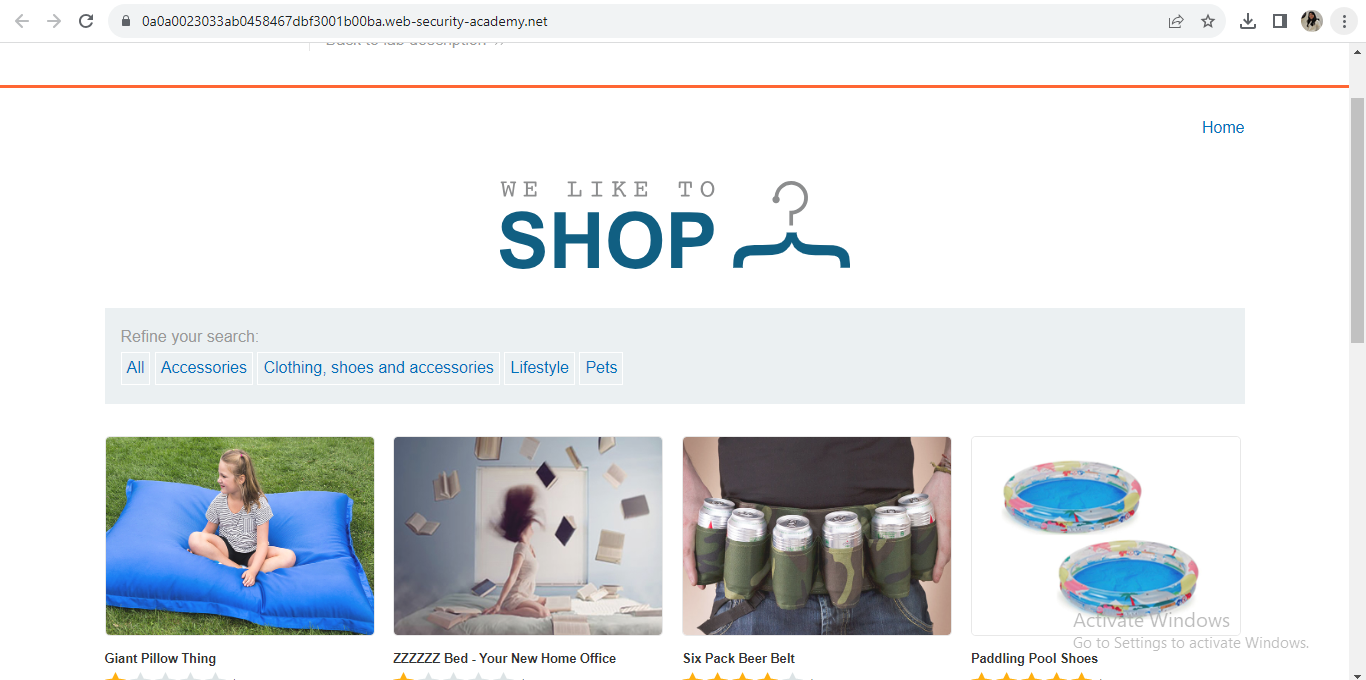
Going further, an attacker can cause the application to display all the products in any category, including categories that they don't know about. This results in the SQL query:

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

The modified query will return all items where either the category is Gifts, or 1 is equal to 1. Since 1=1 is always true, the query will return all items.

**IMPLEMENTATION:**

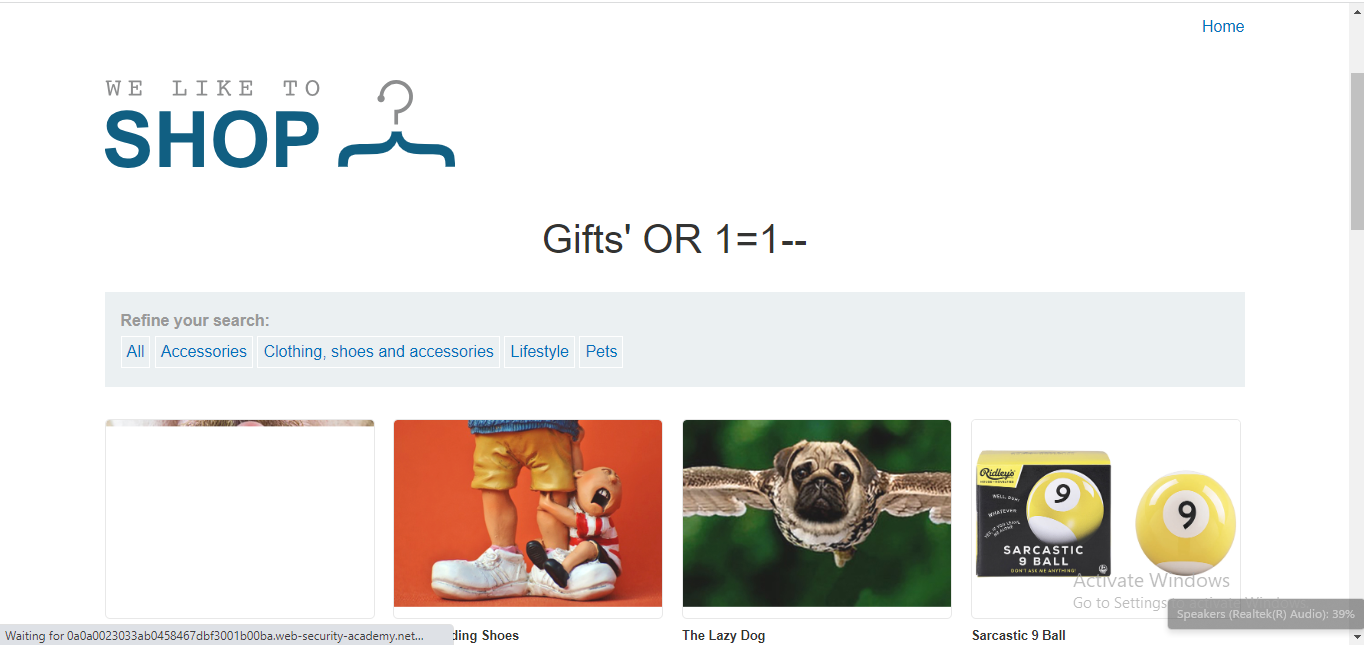
1. Use Burp Suite to intercept and modify the request that sets the product category filter.



1. Modify the category parameter, giving it the value filter?category=Gifts%27+OR+1=1--



1. Submit the request, and verify that the response now contains additional items.



**CONCLUSION:**

In this experiment we learned how to use BURP Proxy to test Web Applications.