**Experiment 7**

**Aim**: Data Validation Experiment.

**Theory**:

**SQL Injection** is one of the most common and severe vulnerabilities in web applications. It occurs when an attacker manipulates a web application's input fields to inject malicious SQL code, which is then executed by the database. This exploitation allows the attacker to interfere with the queries made to the database, often leading to unauthorized access, data theft, or even full system compromise.

SQL Injection attacks exploit the way web applications interact with their databases. In typical web applications, user input (such as form data, URL parameters, or cookies) is used to construct SQL queries that retrieve, modify, or delete data. If the application does not properly validate or sanitize this input, an attacker can craft input in such a way that alters the intended SQL query, often bypassing authentication mechanisms or gaining access to sensitive data.

### **How SQL Injection Works**

When a web application uses user input directly in SQL queries without sanitization, an attacker can input special SQL commands to change the query's logic. Here’s a simple example:

Consider a login form where the user enters a username and password, and the application checks these values against the database using a query like this:

sql

Copy code

SELECT \* FROM users WHERE username = 'user\_input' AND password = 'user\_password';

An attacker could enter the following input in the username field:

' OR '1' = '1

This would result in the following query:

SELECT \* FROM users WHERE username = '' OR '1' = '1' AND password = 'user\_password';

In this modified query, the condition OR '1' = '1' is always true, allowing the attacker to bypass authentication and potentially log in as any user without knowing their credentials.

### **Types of SQL Injection**

1. **Classic SQL Injection**: This type involves modifying an existing SQL query by injecting malicious input. The goal is usually to extract data from the database or bypass security checks.
2. **Blind SQL Injection**: In this variation, the attacker cannot directly see the results of their queries. However, they can infer the structure of the database or the success of their query based on application behavior, like whether a page loads correctly or returns an error. Blind SQL Injection often involves sending multiple requests to extract small bits of information iteratively.
3. **Error-Based SQL Injection**: In error-based SQL Injection, the attacker exploits database error messages to gather information about the structure of the database. By causing the database to return an error (e.g., through malformed SQL queries), attackers can learn about table names, column names, or other details that can help them refine their attacks.
4. **Union-Based SQL Injection**: This type of injection involves using the SQL UNION operator to combine the results of two queries. By crafting input that appends a malicious query via UNION, the attacker can retrieve additional information from the database.
5. **Time-Based Blind SQL Injection**: In cases where no error messages or responses are returned by the application, attackers can use time-based attacks. They use SQL functions that cause a delay in the query's execution (such as SLEEP()), allowing them to infer whether a certain condition is true based on the time it takes for the server to respond.

### **Impact of SQL Injection**

SQL Injection can have catastrophic consequences for web applications, including:

* **Data Theft**: Attackers can retrieve sensitive data such as usernames, passwords, credit card information, and other personal or financial details stored in the database.
* **Data Manipulation**: Attackers can alter, delete, or add data in the database, potentially compromising the integrity of the application.
* **Authentication Bypass**: Attackers can exploit SQL Injection to log in as other users, including administrators, without knowing their credentials.
* **Remote Code Execution**: In certain situations, SQL Injection vulnerabilities can allow attackers to execute arbitrary commands on the server, gaining full control over the application and underlying systems.
* **Denial of Service (DoS)**: Attackers can issue queries that overload the database, causing it to crash or become unresponsive.

### **Mitigation of SQL Injection**

To protect web applications from SQL Injection attacks, developers must implement best practices and security measures. These include:

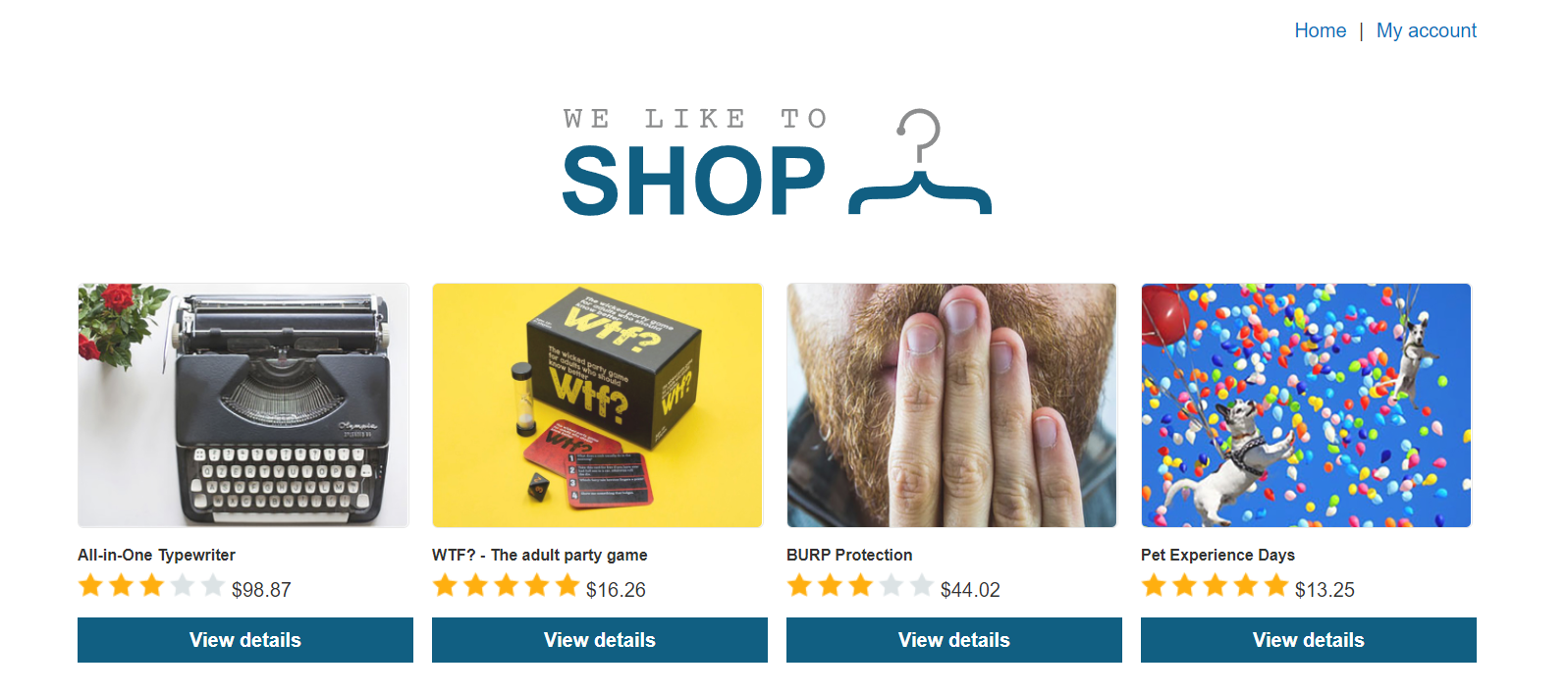
**Parameterized Queries/Prepared Statements**: One of the most effective defenses against SQL Injection is the use of parameterized queries, also known as prepared statements. Instead of dynamically constructing SQL queries with user input, prepared statements separate the SQL code from the data. This ensures that user input is treated as data, not as part of the query itself. Here's an example of how to use a prepared statement in PHP:  
php  
Copy code  
$stmt = $pdo->prepare('SELECT \* FROM users WHERE username = ? AND password = ?');

$stmt->execute([$username, $password]);

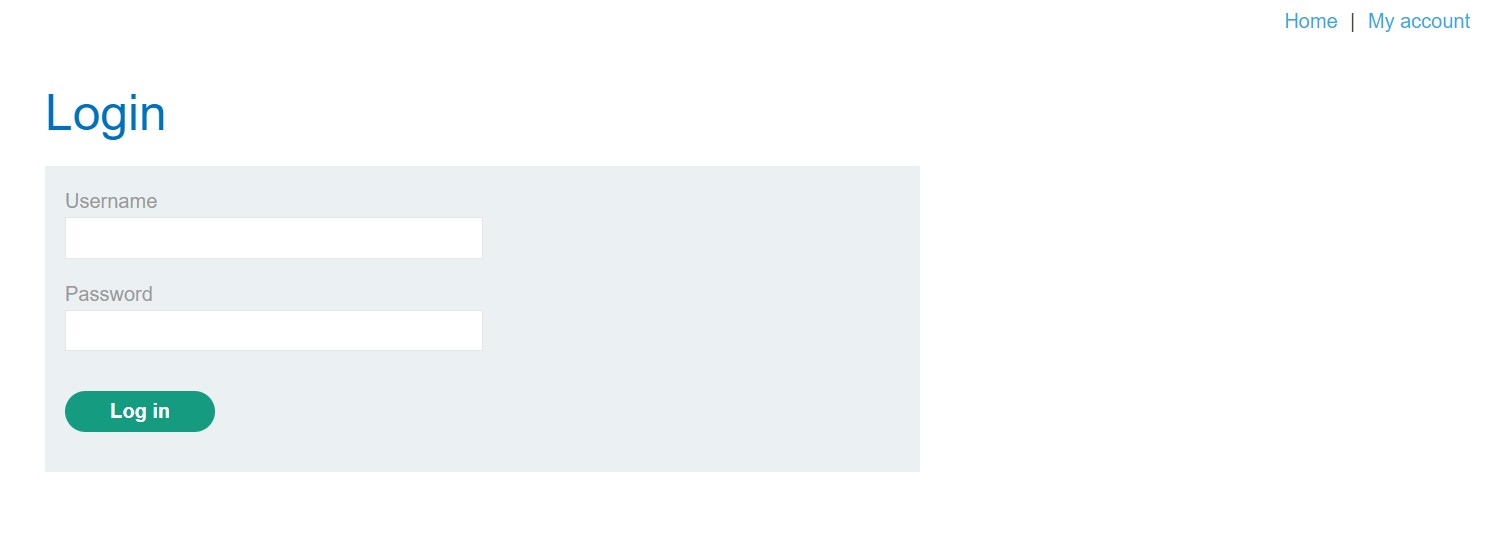
* **Stored Procedures**: Stored procedures are precompiled SQL statements stored in the database that can be executed with specific parameters. They help reduce the risk of SQL Injection by avoiding dynamic SQL query construction in the application code.
* **Input Validation**: All user input should be validated and sanitized to ensure it conforms to expected formats. This includes checking the length, data type, and content of input fields. Rejecting unexpected characters (like SQL meta-characters) can help prevent SQL Injection.
* **Least Privilege Principle**: Database accounts used by applications should have the minimum privileges necessary to perform their functions. For example, if an application only needs to read data from the database, the account it uses should not have permission to modify or delete data.
* **Escaping User Input**: In cases where parameterized queries cannot be used, developers must ensure that special characters (like quotes) in user input are properly escaped before including them in SQL queries.
* **Error Handling**: Proper error handling should be implemented to prevent the application from displaying detailed error messages to users. Database error messages can give attackers valuable information about the structure of the database and the types of queries being executed.
* **Web Application Firewalls (WAFs)**: WAFs can detect and block SQL Injection attempts by monitoring incoming requests for malicious patterns. Although WAFs should not be relied upon as the primary defense, they can provide an additional layer of security.

Consider a shopping application that displays products in different categories.

Step 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.



Step 2: After clicking to My Account the screen appears for username and password.



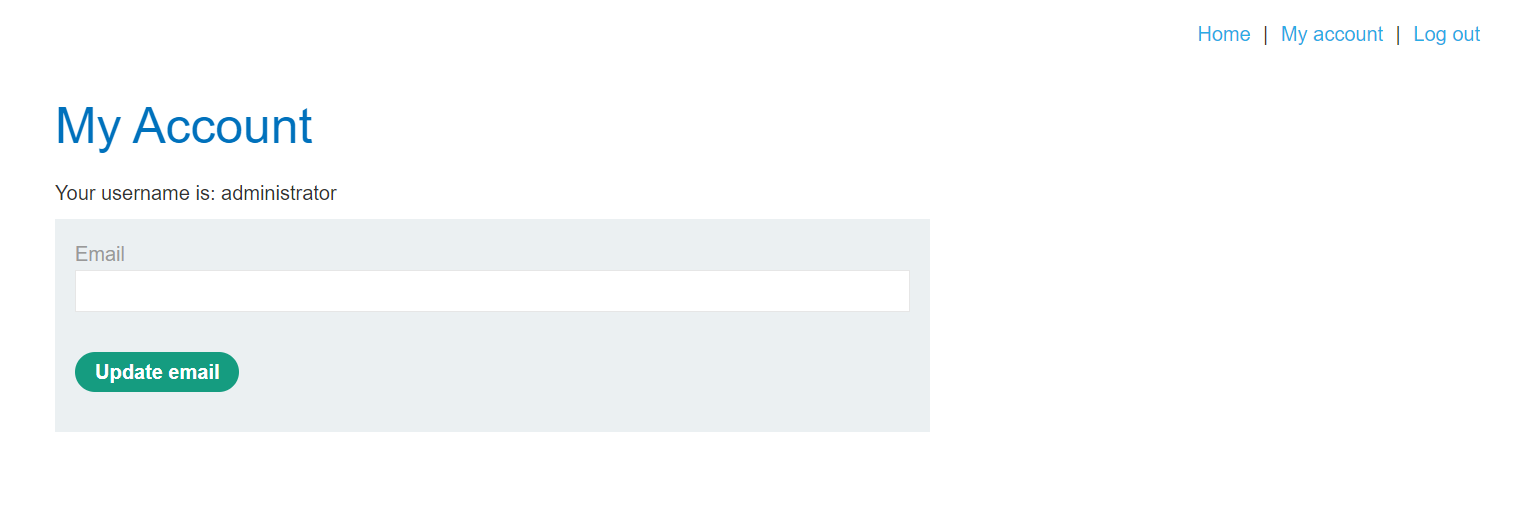
Step 3: After that we will enter username as **admin** and password as **admin** and try to get into the website. After clicking the LOGIN button an error will come and show as Invalid username and password.



Step 4: After this we will perform an SQL injection attack that logs in to the application as the administrator user.



Step 5: After then we can see that we got success in LOGIN as Administrator.



**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.

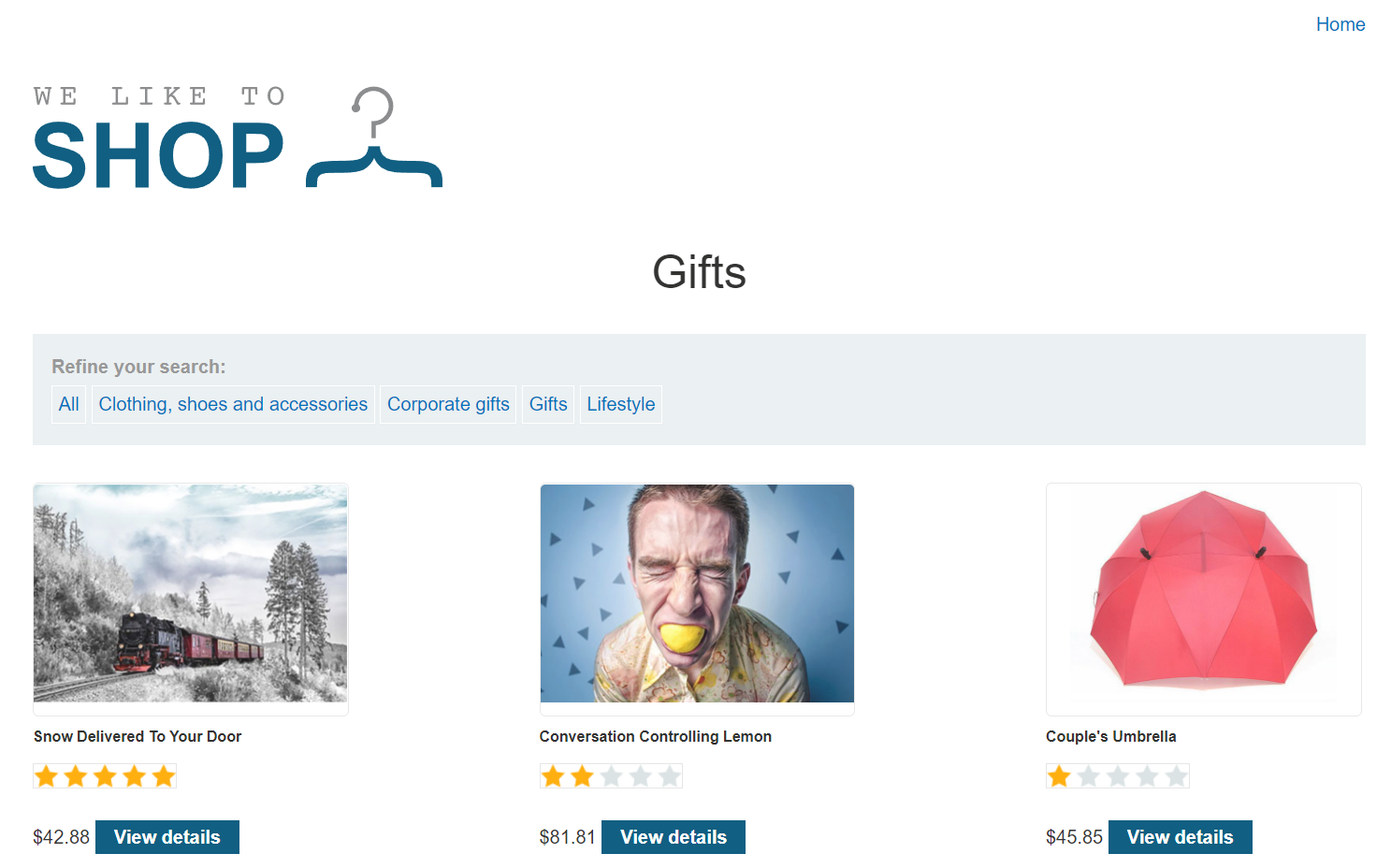
The key thing here is that the double-dash sequence -- is a comment indicator in SQL, and means that the rest of the query is interpreted as a comment. This effectively removes the remainder of the query, so it no longer includes AND released = 1. This means that all products are displayed, including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

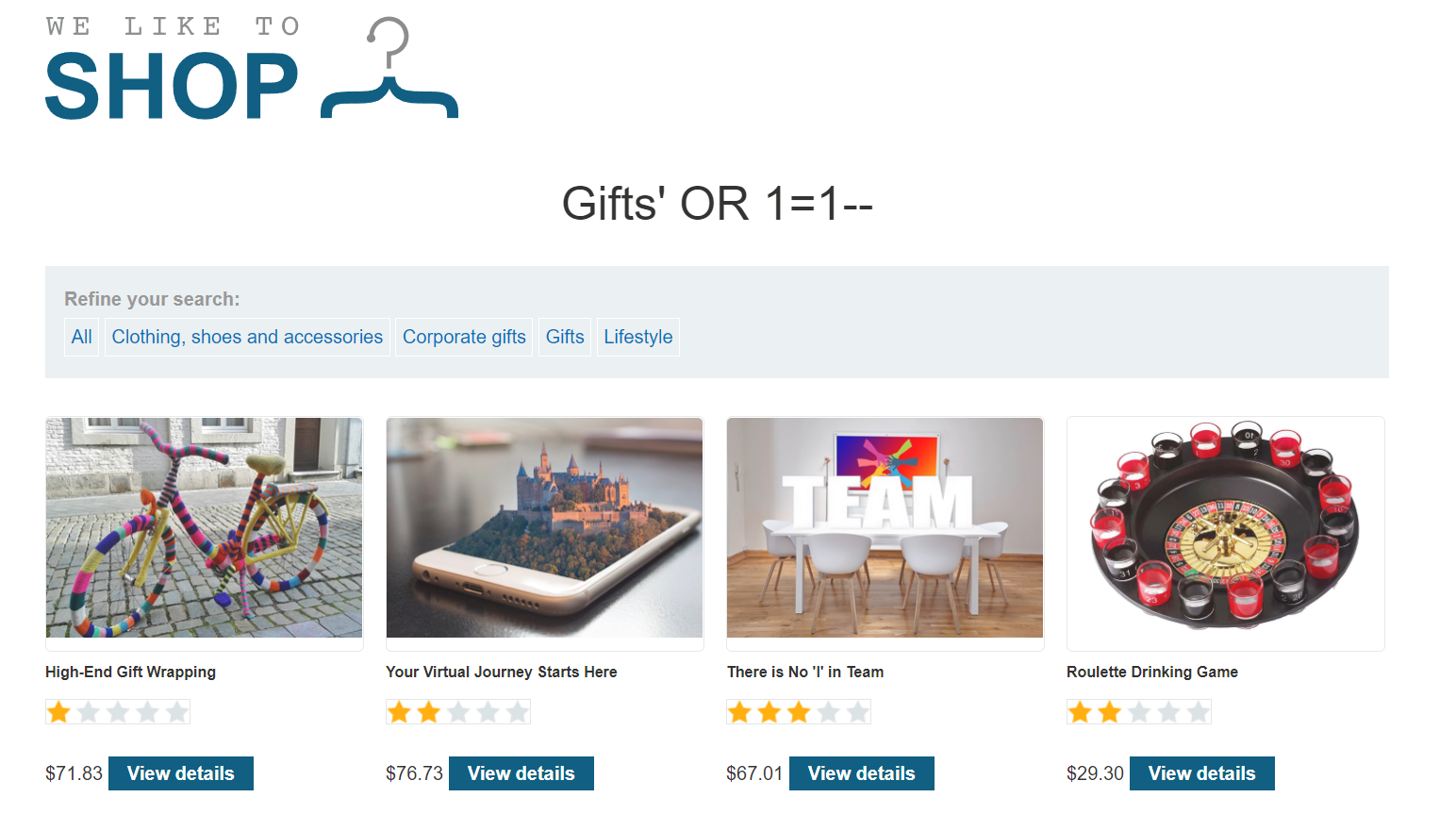
Step 1: Use Burp Suite to intercept and modify the request that sets the product category filter.



Step 2: Modify the category parameter, giving it the value '+OR+1=1–



Step 3: Submit the request, and verify that the response now contains additional items.



**Conclusion**:

Thus we have studied how to validate the data and how to perform SQL injection attacks.