Chapter 2

**Types of attack:**

1. **Sql injection**

* Error Based sqli
* Boolean-Based sqli
* Union-Based sqli
* Time-Based blind sqli

1. **XSS**

* Reflected XSS
* Stored XSS

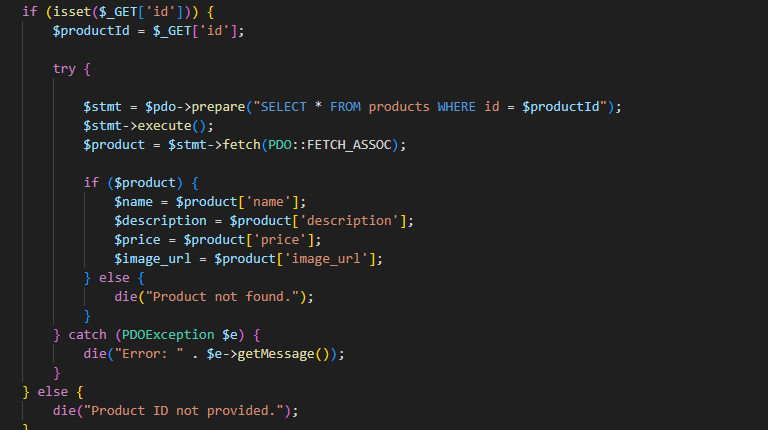
1. **LFI**
2. **Brute Force**
3. **IDOR**
4. **SSRF**

**Ways to prevent this attack:**

* Firewall
* Code review

**Research** **papers**

* **Sql injection**



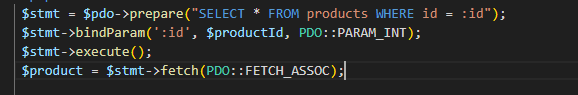
This PHP code get **id** from the URL **(http://localhost/web-test-app/lab/sqli/shop/product.php?id=1)** in the web address. This **id** i used to look up a product in a database. The code does this by create an SQL query, which is like a question to the database.

The problem is that the id from the URL is directly put into the SQL query without any checks, so if attacker changes the URL to [**http://localhost/web-app/lab/sqli/shop/product.php?id=1**](http://localhost/web-app/lab/sqli/shop/product.php?id=1)**’** , by add single quote to test for sql injection vulnerabilities the SQL query will become :

SELECT \* FROM products WHERE id = '1’'

And the output of this query will return a syntax error message in the browser, so the attacker can be sure that the code is vulnerable to sql injection and add more SQL queries or use tools such as SQLMAP to retrieve all the data from the tables .

To prevent SQL injection, the programmer should use prepared statements with parameterized queries, as shown in the following code:



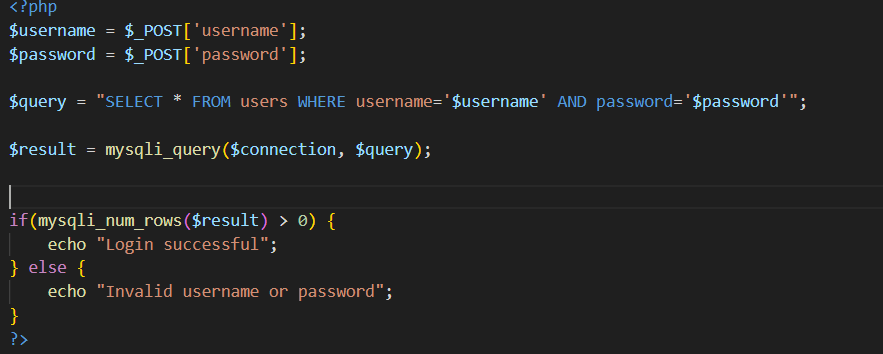
Instead of directly put $productId into the SQL query, we use **:id** as a placeholder. This tells the database engine to expect a value for :id but not to treat it as a part of the SQL command.

By bind the actual value of $productId to the :id placeholder using bindParam(':id', $productId, PDO::PARAM\_INT). So this function bindParam to Take the value of $productId and replace :id with it, but treat it as an integer (PDO::PARAM\_INT) to avoid any unintended interpretation as SQL code, This ensures that user input is always treated as data, not as executable SQL code.

ــــــــــــــــــــــــــــــــــــــــــــــــــــــــــــــــــــــــــــــــــــــــــــــــــــــــــــــــــــــــــــــــــــــــــــــــــــــــــــــــــــــــــــــــــــــــــــ

* Sql injection

Let's say you have a website with a login page that takes a username and password and performs a SQL query to check if the provided credentials match any user in the database. Here's a vulnerable PHP code snippet for such a scenario:



An attacker can exploit this vulnerability by entering a specially crafted input into the username or password field that alters the SQL query's behavior. For example, they could input ' OR '1'='1 as the username or password. The resulting SQL query would become:

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

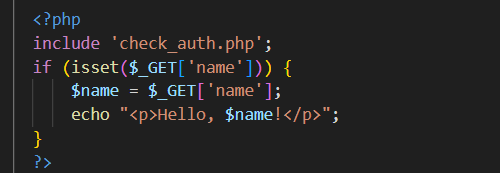
This query always returns true because '1'='1' is always true, allowing the attacker to log in without a valid username or password.

To prevent SQL injection attacks, you should use prepared statements (also known as parameterized queries) instead of directly inserting user input into SQL queries. Here's the same code snippet with prepared statements:



In this code, mysqli\_prepare() creates a prepared statement with placeholders (?) for the user inputs. Then, mysqli\_stmt\_bind\_param() binds the actual values to these placeholders, ensuring that they are treated as data rather than part of the SQL query structure. This approach prevents SQL injection attacks by separating SQL code from data.

* Reflected XSS



To execute a reflected XSS attack using the vulnerability in the provided code, the attacker can input malicious data into the name field and submit the form.

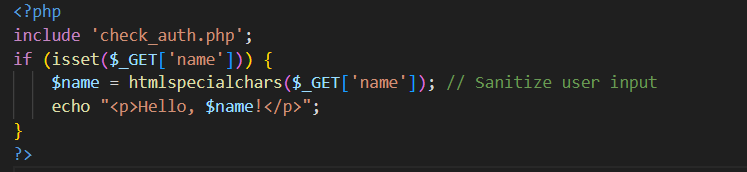
The code accepts the value of 'name' from the GET method ($\_GET['name']) without any sanitization or escaping, then directly prints this value onto the page using echo. This means that if an attacker inputs data containing HTML or JavaScript commands, these commands will be directly executed in the user's browser when they visit the infected page.

To execute the attack, the attacker can send a link like this:

http://example.com/page.php?name=<script>alert('XSS Attack!')</script>

When the user opens this link, the script <script>alert('XSS Attack!')</script> will be executed directly in their browser, displaying an alert window with the message "XSS Attack!".

To prevent XSS attacks in the provided code, you need to properly sanitize and escape user input before displaying it on the web page. This can be achieved by using functions such as htmlspecialchars() in PHP. Here's how you can modify the code to mitigate XSS vulnerabilities:



By using htmlspecialchars(), any HTML special characters in the user input will be converted to their respective HTML entities, preventing them from being interpreted as HTML or script code by the browser. This effectively neutralizes the XSS threat by ensuring that user input is treated as plain text rather than executable code.