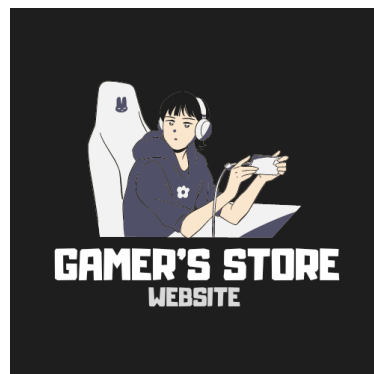




Gamer's Store

Group 3 - Web Technology and Security



[GitHUB Link](#)

Ahmed Sami Al Arfaj	2200004813
Abdulaziz Saud Al Huwaishil	2200003842
Khalifah Waleed Alzwaimel	2200003609
Naeem Mohammed Owaida	2200006454
Khaled Juaitheh Alhajeri	2200001019
Hassan Ali Al Saleem	2200002597



Table of Contents

1. INTRODUCTION	3
2. WEBSITE FUNCTIONALITY	3
A. Website Sections.....	3
B. Sections Functionality	4
3. ATTACKS & COUNTERMESURES	11
A. SQL INJECTION ATTACK.....	11
B. DIRECTORY BRUTE-FORCE WITH DirBuster TOOL.....	21
4. References	23

Table of Figures

Figure 1 Product Page.....	4
Figure 2 Main Page.	5
Figure 3 Register Page.....	6
Figure 4 Login Page	7
Figure 5 About us Page	8
Figure 6 Contact Us page.	9
Figure 7 Validate Chosen item.	10
Figure 8 User Cart	10
Figure 9 a failed attack on the search function.....	12
Figure 10 The search function code	12
Figure 11 Time Based SQL Injection in Register Page.	13
Figure 12 sanitizes the input data before storing it in variables.....	13
Figure 13 Input Sanitization Code.....	14
Figure 14 Input validation code.	15
Figure 15 SQLi Command that used to bypass the login authentication.....	16
Figure 16 login.php code.....	17
Figure 17 filter_var() function.....	18
Figure 18 Running SQL MAP.....	19
Figure 19 Result of SQL MAP	19
Figure 20 PHP code of closing SQLi vulnerability.....	19
Figure 21 DirBuster tool.....	21
Figure 22 Brute-force results	22



1. INTRODUCTION

Welcome to Gamer's Store, the ultimate online destination for all things gaming. Our website is designed to cater to the needs of passionate gamers who are looking for the latest video games, gaming platforms, and accessories. Our team of skilled developers has utilized a range of programming languages, including HTML, CSS, PHP, and JavaScript, to create a seamless and user-friendly experience for our customers.

To ensure the highest level of security and safety, we have implemented robust security protocols and utilized the powerful XAMPP tool during the website's development. XAMPP provides the Apache server and MySQL DB, guaranteeing a reliable and secure platform for our customers to shop on.

At Gamer's Store, our website is designed to be easy to navigate, and our goal is to ensure that our customers can find the products they need quickly and efficiently. Whether you're a casual gamer or a hardcore enthusiast, you can find something that suits your needs on our website.

2. WEBSITE FUNCTIONALITY

A. Website Sections

The game store aims to provide a productive and informative website to ensure users have an optimal shopping experience. To achieve this, the website is divided into six main sections, which include:

1-Products

2-Main page (Best Sellers)

3-Register

4-Contact Us

5-About Us

6-Login

7-Shopping Cart



B. Sections Functionality

1- Products:

On the product page, all the items will appear here, so the customer can view all the products that our website sells. Also, there's a search field that will make finding the desired item easier for the customer.

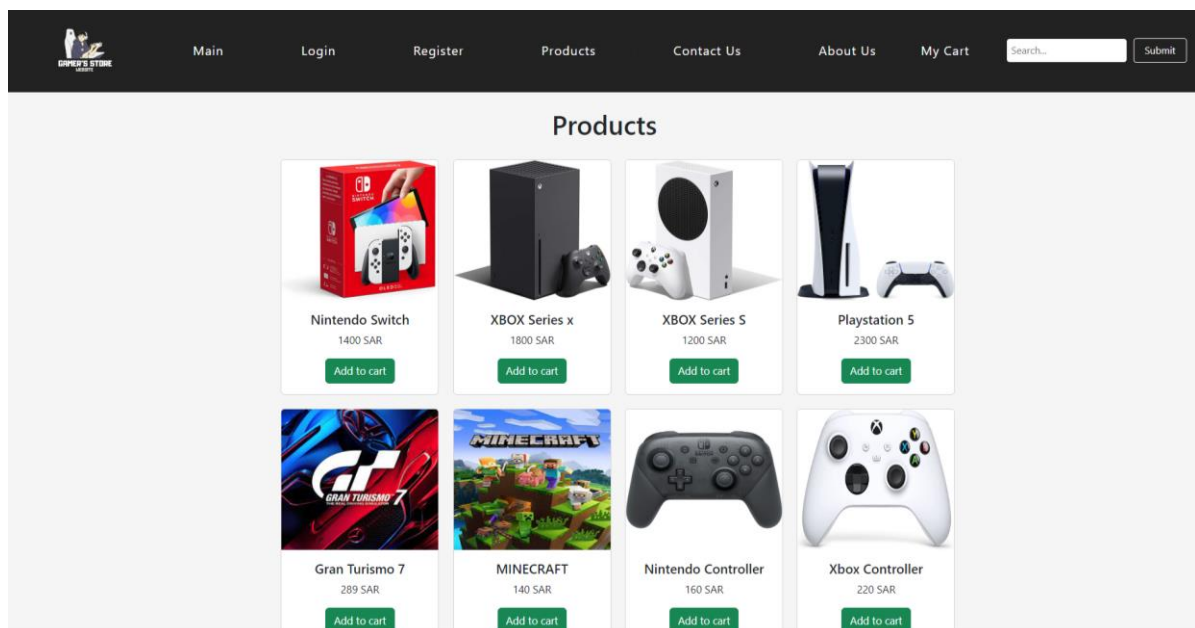


Figure 1 Product Page.



2- Main page (Best Sellers):

Including a "Best Sellers" section on a website can be a valuable tool for both customers and the website itself. By providing customers with easy access to the most popular items and helping the website increase sales and establish trust, a "Best Sellers" section can effectively improve the website's functionality and user experience.

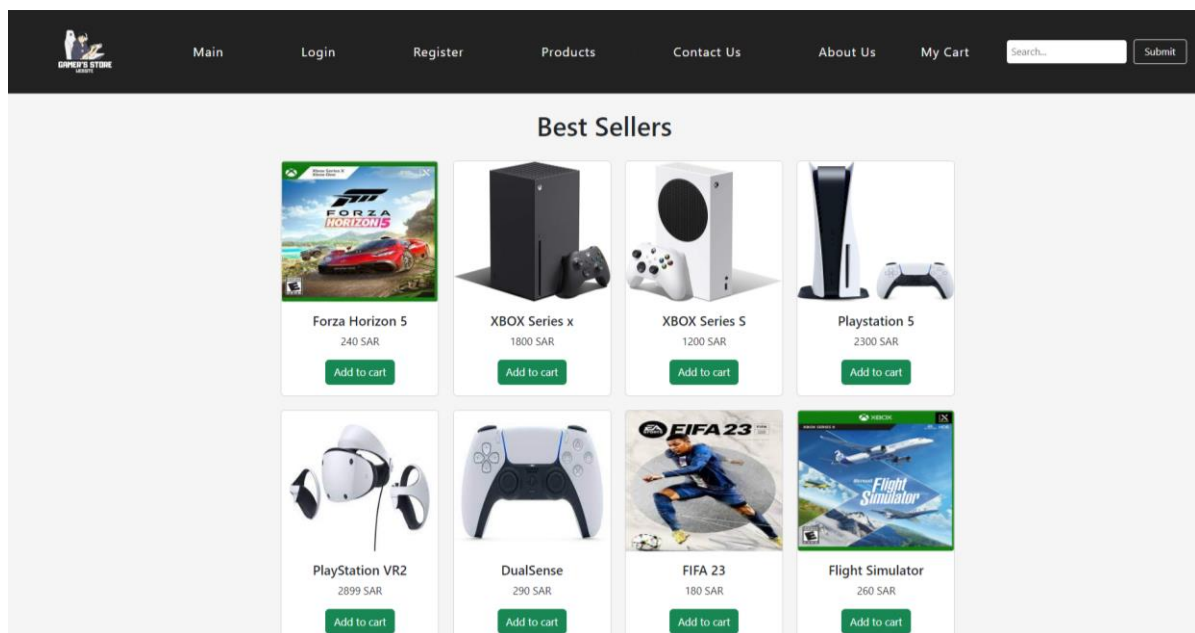


Figure 2 Main Page.



3-Register:

This section allows new users to create an account on the game store website. It typically requires users to provide their name, email address and create a password and phone number. When users register for an account on a game store website, their information is typically sent to a database. This database securely stores the user's information and allows the website to retrieve it as needed.

The screenshot shows a web page for a 'Game Store'. At the top is a dark navigation bar with a logo on the left and links for 'Main', 'Login', 'Register', 'Products', 'Contact Us', and 'About Us'. The main body of the page is light gray and features a centered white box titled 'Register Form'. Inside this box are four text input fields labeled 'Name', 'Email', 'Password', and 'Phonenumber', stacked vertically. Below these fields is a green button with the text 'Register'.

Figure 3 Register Page



4-Login:

This section allows users to sign into their account on the game store website. It may require an email and password or let users sign into the website.

Figure 4 Login Page



5- About Us:

This section of a website game store typically provides an overview of the website's purpose and mission, along with information about the store's team members.

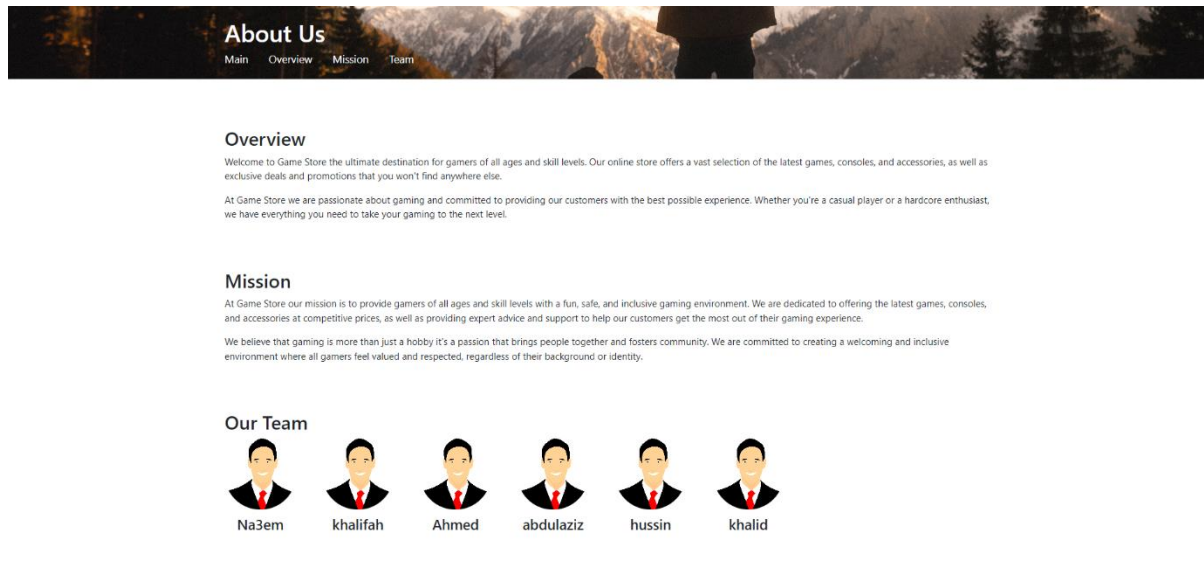


Figure 5 About us Page



6- Contact Us:

section of a game store website provides users with a way to get in touch with the store for any questions, concerns, or feedback. This section typically includes contact information for the store, such as phone numbers, email addresses, and your name. It may also include a contact form that users can fill out to send a message to the game store directly.

Figure 6 Contact Us page.



7- Shopping Cart:

In this section, we display the user and allow the user to add any item we have to his cart by simply pressing the add to cart button. We have created two pages, one validating the user-chosen item as we see in figure 7 and the other displaying the user's cart, including all the items he chooses. We also provide the ability to delete an item from the cart as we see in figure 8.

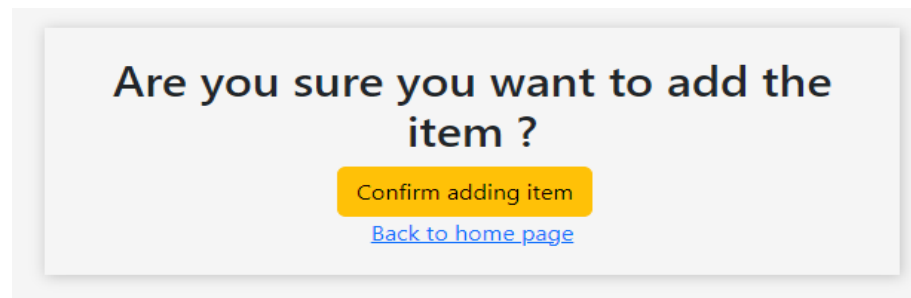


Figure 7 Validate Chosen item.

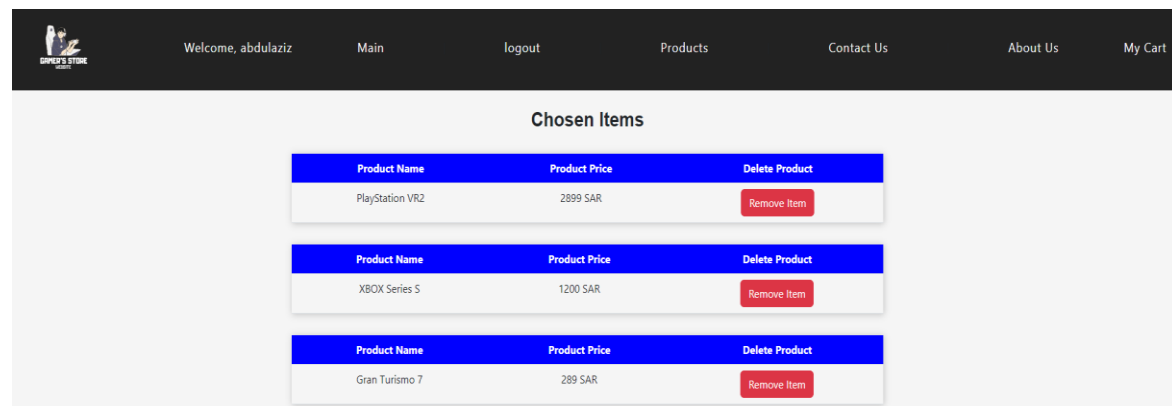


Figure 8 User Cart



3. ATTACKS & COUNTERMEASURES

A. SQL INJECTION ATTACK

SQL injection is a web application security vulnerability that poses a significant threat to the confidentiality, integrity, and availability of sensitive data. It occurs when an attacker can inject malicious SQL code into an application's input fields, such as a login form or search box, which is then executed by the application's database. This can lead to unauthorized access to sensitive data, modification of data, or even complete control of the affected system. One of the main reasons SQL injections is such a pervasive threat is that it can exploit vulnerabilities in almost any web application that interacts with a database. SQL injection attacks can have serious consequences, including data breaches, financial losses, and reputational damage. As a result, organizations must take steps to protect against SQL injection. This includes appropriately validating and sanitizing all user input, using parameterized queries, and limiting the privileges of database users. By implementing these best practices, organizations can significantly reduce the risk of SQL injection attacks and ensure the security of their applications and data. [1] Organizations can follow several best practices to prevent SQL injection attacks. First and foremost, it is important to properly validate and sanitize all user input to ensure it does not contain malicious SQL code. This can be done using input validation techniques such as white-listing and black-listing and input sanitization techniques such as escaping special characters. Additionally, organizations should use parameterized queries, which separate the SQL code from the user input, to prevent attackers from injecting malicious code into the application's queries. It is also important to limit the privileges of database users and to regularly test applications for vulnerabilities using tools such as vulnerability scanners and penetration testing. By following these best practices, organizations can significantly reduce the risk of SQL injection attacks and ensure the security of their applications and data. [2]



- Simple SQL Injection

When trying to use an injunction on the search function, it does not allow it to work, and this is what it displays in the URL:

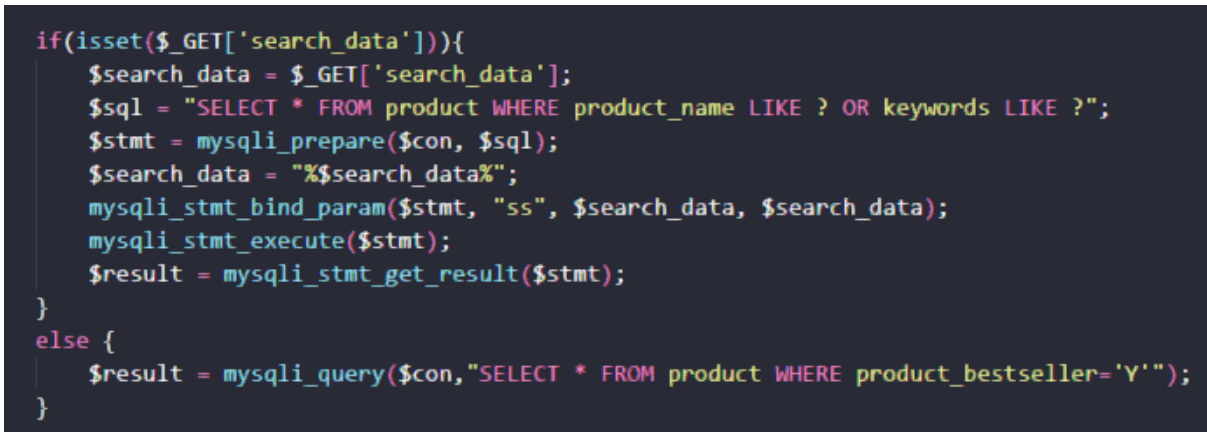


localhost/PROJECT/index.php?search_data=%27%3B--

Figure 9 a failed attack on the search function

The reason being the search function is one of the most overlooked functions when it comes to security, and it is one of the easiest to use injunctions against. Therefore, we designed our search functions based on the prepared statement method.

- COUNTERMESURES



```
if(isset($_GET['search_data'])){
    $search_data = $_GET['search_data'];
    $sql = "SELECT * FROM product WHERE product_name LIKE ? OR keywords LIKE ?";
    $stmt = mysqli_prepare($con, $sql);
    $search_data = "%$search_data%";
    mysqli_stmt_bind_param($stmt, "ss", $search_data, $search_data);
    mysqli_stmt_execute($stmt);
    $result = mysqli_stmt_get_result($stmt);
}
else {
    $result = mysqli_query($con, "SELECT * FROM product WHERE product_bestseller='Y'");
}
```

Figure 10 The search function code

What is a prepared statement? It's when the code pre-compiles an SQL statement before executing it. By using a specified template, the SQL will use placeholders for input values and after being executed the placeholders will be supplied with specific values. Not only is the prepared statement more efficient than normal SQL functions but it also helps prevent injections. Because it treats the input values as data and not as part of the SQL statement. In this code the 'search_data' parameter is set before using it in the query. And the user input will be used in the LIKE clause of the SQL query, which means that the input is being used as a part of a string comparison.



- SQLi on the Register Page.

Figure 11 Time Based SQL Injection in Register Page.

As we can see in figure 11, when we try to inject using Time-Based injection by command (" or sleep(5)#), the injection does not succeed because we used Input Validation, Input Sanitization, and input filtering, which increases security on the page [3].

- COUNTERMESURES

```
}
if(isset($_POST['submit'])){
include 'conn-db.php';
$name=filter_var($_POST['name'],FILTER_SANITIZE_STRING);
$password=filter_var($_POST['password'],FILTER_SANITIZE_STRING);
$email=filter_var($_POST['email'],FILTER_SANITIZE_EMAIL);
$phonenumber=filter_var($_POST['phonenumber'],FILTER_SANITIZE_STRING);
```

Figure 12 sanitizes the input data before storing it in variables.



```
$errors=[];
// validate name
if(empty($name)){
    $errors[]="Must Write username";
}elseif(strlen($name)>100){
    $errors[]="Must user name less than 100";
}
// validate name
if(empty($phonenummer)){
    $errors[]="Must Write phonenummer";
}elseif(strlen($name)==10){
    $errors[]="Must phonenummer equal 100";
}

// validate email
if(empty($email)){
    $errors[]="Must Write email";
}elseif(filter_var($email,FILTER_VALIDATE_EMAIL)==false){
    $errors[]="Email is not valid";
}

$stmt="SELECT email FROM users WHERE email ='$email'";
$q=$con->prepare($stm);
$q->execute();
$data=$q->fetch();

if($data){
    $errors[]="Email is already exist";
}
```

Figure 13 Input Sanitization Code.

The code implements several countermeasures to prevent SQL injection attacks and other security vulnerabilities, including input validation, prepared statements, and password hashing. These measures help to ensure that user input is properly sanitized and validated and that sensitive data is stored securely in the database. By limiting errors and validating user input, the code also helps prevent data breaches and other security incidents resulting from malicious attacks. Filters such as `FILTER_SANITIZE_STRING` and `FILTER_SANITIZE_EMAIL` help ensure user input is properly formatted and adheres to expected standards. Lastly, the code uses regular expressions to enforce password strength requirements and prevent weak passwords from being used.

Use of prepared statements: The `INSERT INTO users (name, email, password, phonenummer) VALUES (?, ?, ?, ?)` query uses prepared statements. This means that the data is not directly inserted into the query string but is instead passed to the database as parameters. This prevents attackers from injecting malicious SQL code into the query.

Escaping of special characters: The `filter_var()` function is used to escape special characters such as single quotes, double quotes, and backslashes. This prevents these characters from being interpreted as special characters by the database.

Validating user input: The `empty()` function validates user input. This ensures that the input is not empty and that it does not contain any invalid characters. User input should be validated before being inserted into a query to ensure it is safe. This can be done using functions such as `filter_input()` and `preg_match()` to check for invalid characters and patterns.



```
1 const passwordInput = document.querySelector('input[name="password"]');
2
3 passwordInput.addEventListener('input', () => {
4   const password = passwordInput.value.trim();
5   const letterRegex = /[a-zA-Z]/; // at least one letter
6   const numberRegex = /\d/; // at least one number
7   const capitalRegex = /[A-Z]/; // at least one capital letter
8
9   if (password.length >= 8 && letterRegex.test(password) && numberRegex.test(password) && capitalRegex.test(password)) {
10     passwordInput.setCustomValidity('');
11   } else {
12     passwordInput.setCustomValidity('Password must be at least 8 characters long and'
13     + ' contain at least one letter, one number, and one capital letter');
14   }
15
16   passwordInput.reportValidity();
17 });
```

Figure 14 Input validation code.

This code is implemented in how to use JavaScript to validate a password input field in a registration form. The code listens for input events on the password input field and checks whether the entered password meets certain criteria. The criteria include a minimum length of 8 characters, at least one letter, one number, and one capital letter. If the password meets these criteria, the `setCustomValidity()` method is called with an empty string, indicating that the password is valid. If the password does not meet the criteria, the `setCustomValidity()` method is called with an error message, which is displayed to the user using the `reportValidity()` method. This code provides a basic level of password validation to ensure that users enter a password that meets certain security requirements.



- **SQLi on Login Page.**

The first SQLi command that I have entered to test the security of the login page is “ ' or 1=1# ”, the goal of this command is to end the default condition in the SQL Query by ‘ and then 1=1 will be always true, # is to comment the rest of the conditions (if there’s). So if the code is vulnerable this command should it let the attacker to bypass the login authentication.

Figure 15 SQLi Command that used to bypass the login authentication.

As it's shown in Fig.15, the attack has been successfully prevented and the attacker has failed in his attempt.



• COUNTERMEASURES

```
login.php
7  if(isset($_POST['submit'])) {
8      include 'conn-dbb.php';
9      $password=filter_var($_POST['password'],FILTER_SANITIZE_STRING);
10     $email=filter_var($_POST['email'],FILTER_SANITIZE_EMAIL);
11     $errors=[];
12
13     // validate email
14     if(empty($email)){
15         $errors[]="Must Write Email";
16     } elseif (!filter_var($email, FILTER_VALIDATE_EMAIL)) {
17         $errors[]="Invalid Email Format";
18     }
19
20     // validate password
21     if(empty($password)){
22         $errors[]="Must Write Password";
23     }
24
25     // check for errors
26     if(empty($errors)){
27         // prepare and execute query
28         $stm="SELECT * FROM users WHERE email = ?";
29         $q=$conn->prepare($stm);
30         $q->execute([$email]);
31         $data=$q->fetch();
32
33         if(!$data){
34             $errors[] = "Login Error";
35         } else {
36             // ...
37         }
38     }
39 }
```

Figure 16 login.php code.

Let's have a deep explanation for this code, and why it's secure from SQLi Attacks. in this login.php code we use prepared statements to prevent SQL injection attacks.

Prepared statements are a way of preventing SQL injection attacks by separating the SQL query from the user input. This means that the user input is not directly inserted into the SQL query, but instead is passed as a parameter. This prevents the attacker from being able to inject malicious SQL code into the query.



`$q->execute([$email]);`

This line of code passes the user-supplied email address as a parameter to the `execute()` method of the `PDOStatement` object. This prevents the attacker from being able to inject malicious SQL code into the query.

In addition to using prepared statements, the code also uses the `filter_var()` function to sanitize the user input. This function helps to prevent the attacker from injecting malicious code into the input.

The following lines of code use the `filter_var()` function to sanitize the user input:

```
$password=filter_var($_POST['password'],FILTER_SANITIZE_STRING);  
$email=filter_var($_POST['email'],FILTER_SANITIZE_EMAIL);
```

Figure 17 `filter_var()` function

The `FILTER_SANITIZE_STRING` filter removes all HTML and JavaScript code from the input. This helps to prevent the attacker from injecting malicious code into the input.

The `FILTER_SANITIZE_EMAIL` filter validates the email address and removes any invalid characters. This helps to prevent the attacker from submitting an invalid email address.

By using prepared statements and the `filter_var()` function, the code you provided helps to prevent SQL injection attacks.



- **SQLi using SQL MAP**

SQL injection is a type of web application vulnerability that allows an attacker to manipulate or inject malicious SQL code into a database query. It occurs when user input is not properly sanitized or validated, allowing the attacker to inject malicious SQL code into the query [4]. Our store is for selling products so it must have a secure authentication mechanism that provides a safe way to make authorized users access their accounts. We take care strongly about the input validation and sanitization, we will provide an example of using SQL MAP on our website.

We will use SQL MAP to scan the website and see if there is any SQL vulnerability [5]. SQL MAP recommends that the URL contain parameters so I will scan the cart because it contains a parameter.

```
object-sqlmap-290a8e7>python3 sqlmap.py -u http://localhost/websecurity4/validate_item.php?%20id=5 --dbs
```

Figure 18 Running SQL MAP

The result was there is no SQL injection vulnerability on the website:

```
[21:30:05] [INFO] testing 'Generic UNION query (NULL) - 1 to 10 columns'
[21:30:05] [WARNING] GET parameter 'id' does not seem to be injectable
[21:30:05] [CRITICAL] all tested parameters do not appear to be injectable. Try to increase values for '--level/'
--risk' options if you wish to perform more tests. If you suspect that there is some kind of protection mechanism
involved (e.g. WAF) maybe you could try to use option '--tamper' (e.g. '--tamper=space2comment') and/or switch '--
random-agent'
```

Figure 19 Result of SQL MAP

- **COUNTERMESURES**

We strongly close all the vulnerabilities that are related to SQLi by : input validation and sanitization in login , register and all pages that take an input from user, use of prepared statements.

```
<?php
include('conn-db.php');
$ID = $_GET['id'];
$stmt = mysqli_prepare($con, "SELECT * FROM product WHERE id = ?");
mysqli_stmt_bind_param($stmt, "i", $ID);
mysqli_stmt_execute($stmt);
$result = mysqli_stmt_get_result($stmt);
$data = mysqli_fetch_array($result);
?>
```

Figure 20 PHP code of closing SQLi vulnerability



We close the SQL injection in the URL that is in Figure 20 by adding prepared statement as you can see in Figure 3 (\$stmt), and id represented by question marks ?. The prepared statement function is to prepare the SQL statement, which creates a statement object that can be executed repeatedly with different parameters.

After that we use “mysqli_stmt_bind_param” which is responsible to bind the value of the “ID”, then using “mysqli_stmt_execute(\$stmt)” this one executes the prepared statement using the mysqli_stmt_execute function. The execution replaces the placeholder (?) with the actual value of the ID variable.

By utilizing prepared statements (mysqli_prepare, mysqli_stmt_bind_param, mysqli_stmt_execute) in this code, you're effectively guarding against SQL injection vulnerabilities. The use of placeholders and binding variables ensures that user input is treated as data rather than executable SQL code.



B. DIRECTORY BRUTE-FORCE WITH DirBuster TOOL

DirBuster tool can be used to enumerate directories and files within a server by simply trying all possible names and extensions. This tool comes with Kali Linux and can easily be used by giving it the desired website URL and specifying the option we want like what type of extension files to look for [6].

- **Attack:**

Open the dirbuster tool in Kali and give it the URL of the website which should be like `hostip/websitepath/` in our case it is `192.168.1.27/gamerstore/`

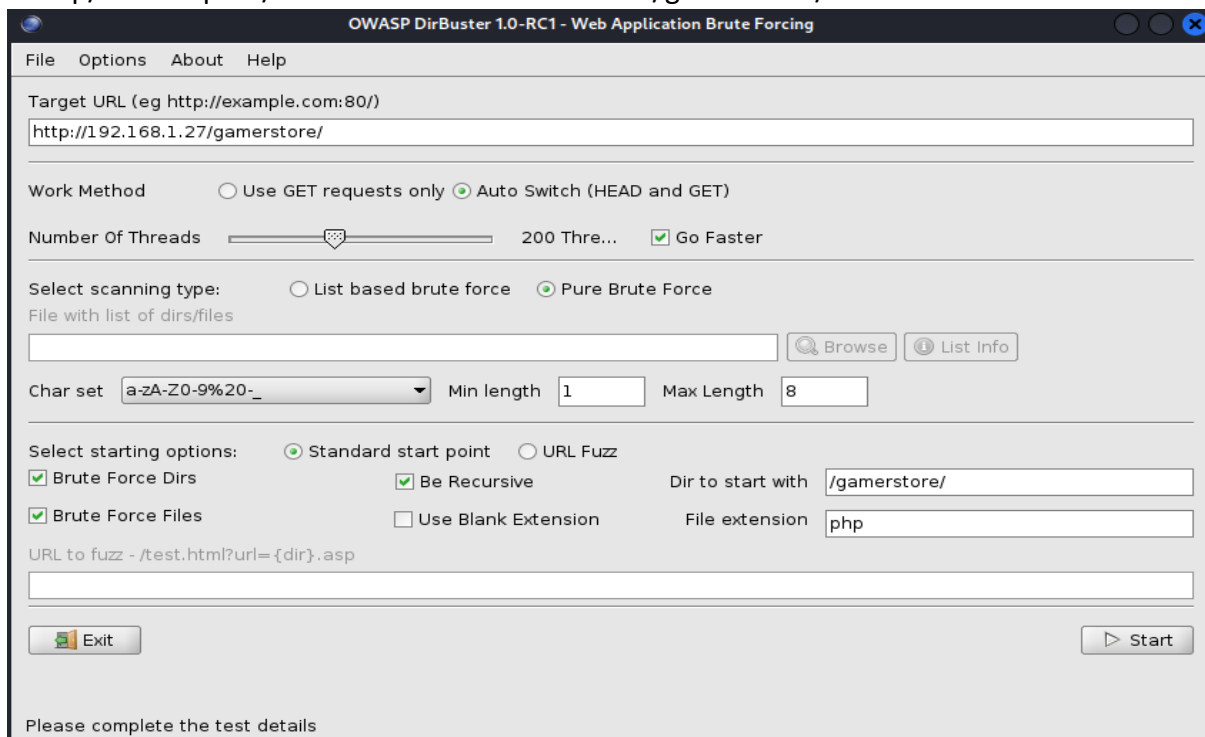


Figure 21 DirBuster tool

As shown in above figure 21 we can choose to use the method of the send request either GET or HEAD request. In addition to that, we can specify the number of threads the tool can use to perform the attack and give it the list to use it or it can do pure brute force. We also specify the file extension to look for which is php files then click start.

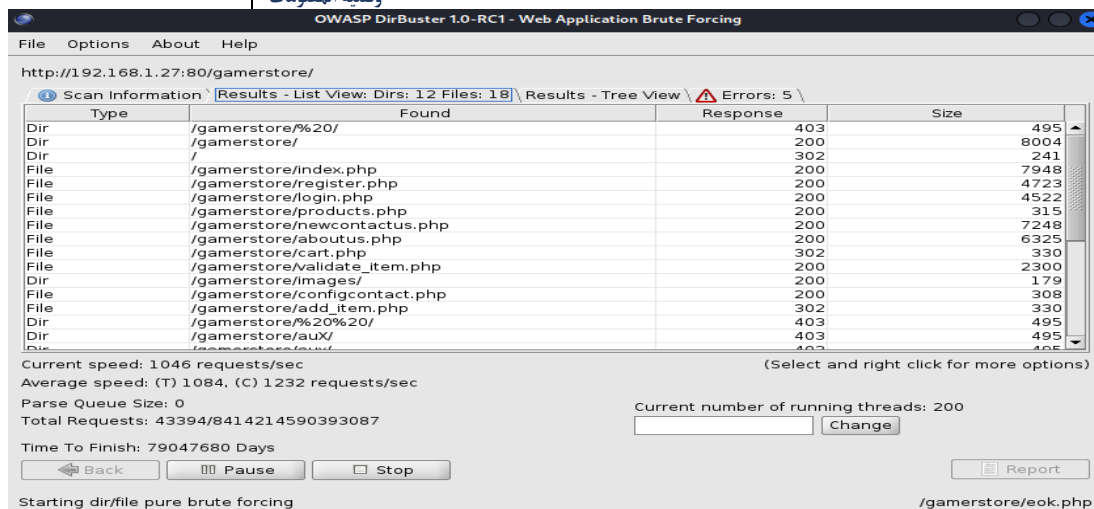


Figure 22 Brute-force results

As we can see from the result in the above figure 22 the tool manages to get almost all the php files we had on the server in just a few minutes. Using this technique could be dangerous if we had unprotected secret files on the server. We will discuss how we can protect the server against this type of attack.

• COUNTERMEASURES:

There are several ways to protect against directories brute-force attacks. We will discuss two of them.

First use authentication and authorization: in this way, we can lock the secret files and request the user trying to access it to log in first, then check if he has the privilege to allow him to view these files. Even if the attacker gets the path of the secret files, he will not be able to access them without the correct credentials. In our case, we have done this for the cart page. No one can access it without logging in to the website. So, if the attacker writes the path of the cart page, he will be redirected to the login page using this code:

```
if(!isset($_SESSION['user'])) {
    header('location:login.php');
    exit();
}
```

Isset is used to check whether the user has logged in or not. If not, the header function will work and redirect him to the login page.

The second way is by implementing a firewall and limiting the number of requests an IP address can send to the server. Since the directory attacks using brute force have to send thousands of requests in just a few seconds, we can easily determine the IP address of the attacker with the firewall and block it.



4. References

- [1] owasp, "owasp," [Online]. Available: https://owasp.org/www-community/attacks/SQL_Injection.
- [2] invicti, "invicti," SQL Injection Cheat Sheet, [Online]. Available: <https://www.invicti.com/blog/web-security/sql-injection-cheat-sheet/>.
- [3] Invicti, "Types of SQL Injection (SQLi)," Acunetix, 2023. [Online]. Available: <https://www.acunetix.com/websitesecurity/sql-injection2/>.
- [4] "SQL injection," OWASP, [Online]. Available: https://owasp.org/www-community/attacks/SQL_Injection.
- [5] "SQL MAP," [Online]. Available: <https://www.kali.org/tools/sqlmap/>. [Accessed 19 05 2023].
- [6] "Dirbuster," Kali, May 2023. [Online]. Available: <https://www.kali.org/tools/dirbuster/>.