SQL injection attacks and countermeasures in PHP, and Cross-Site Request Forgery

Alexander Sabelström
Abstract

Websites have become a vital tool for any establishment's success. However, all users do not enter websites with good intentions. Two commonly seen website functionalities are a login system and the possibility for users to leave and read reviews. To evaluate the vulnerability of login systems to injection attacks, two SQL injection attacks bypassing authentication and one error-based SQL injection attack were executed. Three different countermeasures in PHP against these attacks were then evaluated, which were not allowing quotes as input, escaped statements and parameterized queries.

In addition, a malicious review was written to initiate a potential Cross-Site Request Forgery attack. Client-side input sanitizing was evaluated as a countermeasure which only allowed letters, numbers, spaces and periods.

The attacks and countermeasures were implemented and evaluated by locally hosting a self-made website.

We found that parameterized queries were the most effective in preventing SQL injection attacks. Additionally, the study revealed that initiating Cross-Site Request Forgery attacks was possible through the review functionality by redirecting users to a malicious website. Client-side input sanitizing was found to be an ineffective countermeasure against CSRF attacks.
1 Introduction

In today’s digitalized world, websites are a prominent tool in the success of all kinds of enterprises. A website can work as a main communication channel between the enterprise and its customers, an online shop or even a presentation of what products or services are offered. Successful businesses will have a user-friendly website making it easy to sell their products to anyone having access to the internet. Unfortunately, not everyone who accesses the website has good intentions and some may attempt to attack it.

When developing a website and all its different functionalities, making it secure should be as prioritised as making it user friendly. This thesis investigated two website functionalities, from a security point of view. Firstly, the login functionality should enable users to log in to a website through secure input fields. PHP [1], HTML [2] and SQL [3] were the programming languages used for the login functionality. The scenario of an attacker having access to the login form will be investigated, along with questions such as:

- How is the implementation vulnerable to SQL injection attacks? [4] Chapter 4.3
- How can that vulnerability be exploited by attackers?
- What is the best way of defending against SQL injection attacks in PHP?

The other functionality investigated was the ability for users to leave and read reviews. The possibility of initiating a CSRF [5] attack by writing a review for other users will be tested. In addition, client-side [6] input sanitizing will be evaluated as a countermeasure against CSRF attacks. Besides the programming languages used for the login functionality, JavaScript will also be used in the review functionality. The scenario of an attacker trying to write a review with an embedded link in it will be investigated.

On a self-made website, a locally hosted connection between a PHP script and a MySQL [3] database will be implemented. The connection will be implemented using XAMPP [7], which provides a free and easy PHP environment. Although other solutions providing live connection, as opposed to localhost, also could be suitable, for this thesis, the questions can still be answered using only a locally hosted website.

1.1 Structure

This thesis begins by introducing the login and review functionality in Section 2. In Section 3 the implementations of the functionalities, SQL injections and CSRF attacks are described, along with a discussion of their relevance today. The countermeasures against these attacks are also introduced. Section 4 shows the results of how effective each countermeasure were. Section 5 discusses the
results regarding the countermeasures evaluated. Finally, the thesis concludes with a conclusion and future works in section 6.
2 Website functionalities

A complete website is a collection of numerous website functionalities working together. For a website to be secure, every functionality needs to be inspected separately for security flaws. Vulnerabilities are weaknesses that can allow harm to occur [4, chapter 1.4]. When each functionality has been tested, modules of several functionalities working together should be tested for security vulnerabilities. This process will eventually end up testing the whole website, and will be much easier if done from the start of development. When developing a website, it is important to reflect on the possible vulnerabilities the planned implementations may have.

This section introduces two website functionalities: login and review writing.

2.1 Login

Having the possibility to create an account which later will be used to login, is a common functionality for any website. This is a functionality where security most often is prioritised, due to the well known vulnerabilities login functionalities have. A login functionality consists mainly of two parts. Firstly, it is what the user sees, which are the two input fields asking for a username/mail and password. This part of the functionality is often vulnerable to attacks such as SQL injection attacks [8, chapter 1]. The second one is the database saving the login information. This part may be vulnerable to attacks taking advantage of cryptographic failures, such as rainbow tables [9]. This thesis will focus on the first part.

2.2 Review writing

The review writing functionality gives users the chance to leave and read reviews. This implementation will once again require an input field for the user to write in, which should always be developed with caution. This implementation will also require some kind of database, saving all the different reviews. From a security point of view, one major aspect separates this functionality from the login functionality. The reviews will be available for any user entering the website, as opposed to the login functionality where an attacker’s input can not be seen by any other user. A review could include a link redirecting the user clicking on it to a malicious website, which makes this functionality vulnerable to attacks such as CSRF [5].
3 Methodology

This section begins by introducing how the login functionality was implemented. The SQL injection attacks executed to attack the login functionality will be introduced as well, along with the countermeasures against them. Additionally, the implementation of the review functionality will be introduced. An initiation of a CSRF attack will be implemented along with client-side input sanitizing as a countermeasure against it.

3.1 Implementation of a login functionality

There are several different ways to defend against SQL injection attacks, and in order to test them, a minimalistic website was implemented. The website contained a form with two input fields, one each for the username and password. This was set up using PHP, HTML and a connection with a MySQL database through XAMPP. The database contained one table with one user’s login information.

To connect to the database, the following code was used:

```php
<?php
$host = "localhost";
$username = "root";
$password = "";
$db = "logindb";
$con = mysqli_connect($host, $username, $password, $db);
```

The code states that it is hosted locally, followed by the default username and password for XAMPP. The database was named "logindb", which then is connected using

`mysqli_connect()`

Consider the following PHP code that was used to create the login form:

```html
<html>
<body>
<form action="logincheck.php" method="POST">
  username: <input type="text" name="username"><br>
  password: <input type="password" name="password"><br>
  <input type="submit" name = "submit">
</form>
</body>
</html>
```

Figure 1 displays how it looks on the website.
It exists two input fields, one each for entering the username and password. Lastly there is a submit button, which sends the value entered in the fields to another PHP script for verification. In order to check whether the input provided matches any user’s credentials, the following code was used:

```php
<?php
// include database connection
include_once 'includes/dbh.php';
// Retrieve user's input
$uname_input = $_POST['username'];
$pword_input = $_POST['password'];
// SQL query to find matching credentials
$sql = "SELECT * FROM users WHERE username = '$uname_input' AND password = '$pword_input';";
$result = mysqli_query($con, $sql);
$resultcheck = mysqli_num_rows($result);
if($resultcheck > 0){ // row with matching credentials
    echo 'Log in accepted';
} else{
    echo 'Invalid username or password';
}
```

This code first retrieves a user’s input. A SQL query is then created with the values provided by the user written in the form, and checks whether it found a match or not. In either case it prints on the screen if it was successful or not, whereas in a real life example, the user would be authenticated and logged in accordingly. SQL or Structured Query Language is a data sublanguage for access to relational databases that are managed by relational database management systems (RDBMS) [10, p. 4]. SQL is used for various applications, such as login functionalities. By entering values in the two input fields of the login form, SQL will be used to look up the entered credentials in the database. This is a great example of how the implementation of a login check shouldn’t be done, because a SQL injection attack is easily executed.

For further testing purposes, a single table including one user with the username "user1" and password "password123" was inserted into the database. Figure 1: Login form
displays that "Log in accepted" is echoed [11], after entering the correct login information.

![Log in accepted](localhost/textSQL/logincheck.php)

**Log in accepted**

Figure 2: Result page of entering the correct login information

Similarly to Figure 2, Figure 3 displays that "Invalid username or password" is echoed after entering any other credentials.

![Invalid username or password](localhost/textSQL/logincheck.php)

**Invalid username or password**

Figure 3: Result page of entering the incorrect login information

### 3.1.1 SQL query

SQL queries is requests given to databases to manipulate or retrieve data in it [12]. When a user attempts to login, a query will be sent to the database searching if the given username and password match any of the users stored in the database. A SQL query could look something like this:

```sql
SELECT * FROM Users WHERE username = '{$POST["username"]}'
AND password = '{$POST["password"]}';
```

where

- `$POST["username"]`' and
- `$POST["password"]`'

will be the username and password entered by the user trying to login. To understand how SQL queries works, it is easiest to split up the example query just mentioned.

```sql
SELECT * FROM Users
```

This first part selects every column from a table named Users. Specific columns can be chosen by naming them, but in this case where a '*' is written, every column will be selected. The next step is to filter records in the database that fulfills some criteria. The criteria in this query is to have matching credentials for any user in the database. The next part of the query looks like this:

```sql
WHERE username = '{$POST["username"]}'
AND password = '{$POST["password"]}';
```
This part specifies that SQL should query the database for a user that matches with the entered credentials. If that username exists and the correct password was entered, the user will be logged in.

SQL queries can be much more complex, but the idea for SQL injection attacks remain the same.

### 3.2 How SQL injection attacks work

SQL injection attacks only work on input fields where the input will be used in a SQL query. Login functionalities are therefore often vulnerable to SQL injection attacks. There are several different ways to execute SQL injection attacks. In order to demonstrate different SQL injection attacks, the SQL query introduced in Section 3.1.1 will be used.

```
SELECT * FROM Users WHERE username = '${$_POST["username"]}'
AND password = '${$_POST["password"]}';
```

Three attacks will be demonstrated in this section.

#### 3.2.1 SQL injection attacks

SQL injection refers to a class of code-injection attacks in which data provided by the user is included in a SQL query in such a way that part of the user’s input is treated as SQL code [13]. The idea is to ”inject” SQL queries or part of them via the input data to the application, hence the name SQL injection. When a user’s input could be treated as SQL code, severe damage can be made, and attackers will take advantage of this. The damage that can be done could range from no damage at all to highly severe [13], depending on the goal of the attacker. If authentication does not work as intended, other users may be able to login as another. This can lead to the attacker getting access to sensitive user information stored in the database. Depending on what user the attacker managed to login as, several new privileges could be handed out to the attacker. Managing to login as admin could therefore have damaging consequences.

#### 3.2.2 SQL injection today

Despite several different injection attacks being known for a long time, injection attacks still ranks in the top 3 of OWASP’s top 10 web application security risks for 2021 [14]. Out of all the different injection attacks, SQL injections are mentioned first as one of the most common.

#### 3.2.3 Attack 1 - Bypass authentication

In the first attack, an attacker will log in as another user by knowing its username. In order for a login attempt to succeed, the two checks

```
username = '${$_POST["username"]}';
```
and

\[
password = '$_POST["password"]'
\]

needs to be fulfilled simultaneously. Usernames are often public knowledge whereas the passwords are secret. It should therefore be assumed that attackers know all usernames. The difficult part of logging in as another user is to pass the password check. In this attack, the password check will be removed. In order to make the following check disappear:

\[
\text{AND password} = '$_POST["password"]'
\]

an attacker needs to take advantage of that SQL can create comments in the code. By entering:

\[
\text{ValidUsername’ --}
\]

in the username input field, the SQL query for checking the login attempt will now look like this:

\[
\text{SELECT * FROM Users WHERE username = 'ValidUsername'}
\]

In this query, ”ValidUsername” is any valid username registered in the database. By removing the password check, an attacker can log in as ”ValidUsername”. This works because the first quotation mark entered by the attacker will close the the first quotation mark in the code for checking the username. This will end up avoiding any errors a left out quotation mark would result in. In SQL, a valid query needs an opening and corresponding closing quotation mark for delimited identifiers [15, p. 41] and string literals. If the code does not consists of valid quotation mark pairs for each case, the query may produce an error. This is avoided by inserting one quotation mark after ”ValidUsername” and commenting out the original one. The last part is the double hyphen, which is how comments are created in SQL [16]. Everything following this double hyphen will be treated as a comment instead of code. In this case, the last quotation mark and the entire password check from the query is removed.

3.2.4 Attack 2 - Bypass authentication

This attack will log in as the first user in the database. To login as the first user in the database, the query requires an immediate match of the input with the stored credentials. The problem is that the credentials are unknown. This attack will log in as the first user, even if the username for it is unknown. This is done by entering the following in the username input field:

\[
' \text{ OR true --}
\]

By entering this in the username field instead of a valid username, an attacker will end up logging in as the first user in the database. This is how the query would look like by entering this:

\[
\text{SELECT * FROM Users WHERE username = ' OR true}
\]
By adding the double hyphen last, the password check will be completely removed as it will be seen as a comment instead of code. This time the username check will always end up as true which will be the only check as the password check was removed. This is because the boolean expression:

```
username == '' OR true
```

will always be true.

### 3.2.5 Attack 3 - Error-based

When a user can enter input which produces an error message, it is highly likely that SQL injection attacks are possible. This is because the input contained a character which would not be interpreted as plain text and is the root cause of why executing SQL injection attacks are possible. Error-based SQL injection attacks aim to gain as much information about the database as possible through error messages. Error-based SQL injections can be used efficiently when negative input validation is the only source of defence. Characters that are not validated may also have a different meaning to the database. For example can the “backslash” character be used to escape the closing quote for the username input, creating a SQL syntax error. By entering the “backslash” character in the username input field, the following error message is shown:

```
Fatal error: Uncaught mysqli_sql_exception: You have an error in your SQL syntax; check the manual that corresponds to your MariaDB server version for the right syntax to use near ''\' AND password = ''' at line 2 in C:\xampp\htdocs\testSQL\logincheck.php:10
Stack trace:
#0 C:\xampp\htdocs\testSQL\logincheck.php(10): mysqli_query(Object(mysqli), 'SELECT * FROM u...
#1 {main}
```

Because the environment used in this report is locally hosted, this error message will probably give more information than what an attacker might receive from a live website. But if an attacker saw this error message, it would provide crucial information for further attacks, and some of them are: the database used is MySQL, the input query is not escaped and information about the query used in handling the input is revealed.

### 3.3 Countermeasures against SQL injection attacks

One of the most widely known defences against SQL injection attacks is input validation. Positive validation seeks to only accept a certain class of inputs, whereas negative validation seeks to deny a certain set of characters. One of the main reasons positive validation is preferred over negative validation is that a developer may not know every kind of character that could be used in a SQL injection attack. Another well known way to defend against SQL injection attacks is input sanitization. Input validation only accepts certain characters, whereas input sanitization escapes certain characters in order to make an input
valid. By escaping characters, the SQL parser will not interpret them as SQL commands\(^\text{18}\). Because injection attacks still are so common, built-in functions in various programming languages has been implemented to make it easier to write secure code.

The code in Section \[3.1\] will be used as a base for the login check. Other versions of it will be implemented in order to test which version would protect against SQL injections the best.

3.3.1 Version 2 - Deny the quotation character

The second version implemented a check to see if the input contained any quotes, and if it did, it would deny that input. The code for that looked like this:

```php
if(strpos($uname_input, "'")){
    echo "Quotes are not allowed";
}
```

which echoed an error message if an input contained a quote.

3.3.2 Version 3 - Escaped statements

The third version used a built-in function that escapes several characters from an input. Consider the following PHP code:

```php
<?php
    // include database connection
    include_once 'includes/dbh.php';
    // Retrieve user’s input
    $uname_input = $_POST["username"];
    $pword_input = $_POST["password"];  
    // SQL query to find matching credentials
    $sql = sprintf("SELECT * FROM users WHERE username='%s' AND password = '%s'",
                   mysql_real_escape_string($con, $uname_input),
                   mysql_real_escape_string($con, $pword_input));
    $result = mysqli_query($con, $sql);
    $resultcheck = mysqli_num_rows($result);
    if($resultcheck > 0){  // row with matching credentials
        echo 'Log in accepted';
    }else{
        echo 'Invalid username or password';
    }
```

The query is built with the escaped input made from the built-in function `mysqli_real_escape_string`, which sanitisises the input\(^\text{1}\). Characters that are escaped are ASCII 0, \n, \r, \', " and Control-Z\(^\text{20}\).

\(^1\)Note that `mysqli_real_escape_string` is used instead of `mysql_real_escape_string`, since `mysql_real_escape_string` was deprecated in PHP 5.5.0, and removed in PHP 7.0.0\(^\text{19}\).
3.3.3 Version 4 - Parameterized queries

The fourth version uses parameterized queries in order to make it possible for the database to distinguish between SQL commands and input provided by a user. By using parameterized queries, the input from a user will be treated as data instead of a command. OWASP’s recommendation for PHP is to use strongly typed parameterized queries using bind_param() [21]. Consider the following PHP code:

```php
<?php
include_once 'includes/dbh.php'; // include database connection
// Retrieve user's input
$uname_input = $_POST['username'];
$pword_input = $_POST['password'];
// SQL query to find matching credentials
$sql = $con->prepare("SELECT * FROM users WHERE username = ? AND password = ?");
// bind variables
$sql->bind_param('ss', $uname_input, $pword_input);
$sql->execute();
$result = $sql->get_result();
$resultcheck = mysqli_num_rows($result);
if($resultcheck > 0){ // row with matching credentials
    echo 'Log in accepted';
} else{
    echo 'Invalid username or password';
}
```

This is an example of how a parameterized query is made, and is a way to make it impossible to perform a SQL injection attack [21]. This is because all input provided by a user will be treated as plain text instead of characters that may have a different meaning to the database. The function prepare() prepares a statement that the execute() function later executes. The question mark characters will be substituted with values defined by bind_param() as the statement is executed. The bind_param() function defines the data types of the parameters, which in this case both will be strings. Before the query is done, the variables that will be bound to the question mark placeholders will be defined.
3.4 CSRF in the review functionality

In order to evaluate the possibilities of executing a CSRF attack in the review functionality, a simple review functionality was implemented. This section introduces the implementation made, as well as the CSRF attack.

Similar to the login functionality, the database connection was made with the following code:

```php
<?php
$host = "localhost";
$username = "root";
$password = "";
$db = "reviews";
$con = mysqli_connect($host, $username, $password, $db);
```

The code states that it is hosted locally, followed by the default username and password for XAMPP. The database was named "reviews", which then is connected using `mysqli_connect()`

3.4.1 Implementation of a review functionality

A minimalistic website containing the ability to write and delete reviews was implemented, which is shown in Figure 4:

![Website offering a chance to write and delete reviews](image)

Figure 4: Website offering a chance to write and delete reviews

The website contains one input field for the review writing along with a submit button. All available reviews in the database will be listed under the submit button. At the bottom, there is a delete button, which will delete all the reviews. The code for this starts with the input field and the submit button, which looked like this:

```html
<html>
  <body>
    <form action="insertreview.php" method="POST">
      review: <input type="text" name="review"><br>
      <input type="submit" >
    </form>
  </body>
</html>
```
When the submit button is clicked, information will be sent to insertreview.php, which will insert the review into the database, and then switch page back to CSRF.php which was the original page. One review needs to be able to be inserted, which will be given ID 1. The code for inserting reviews looks like this:

```php
<?php
$review_input = $_POST['review'];
$sql = "INSERT INTO reviews (ID, review) VALUES (1, '$review_input');";
$result = mysqli_query($con, $sql);
header("Location: CSRF.php");
die();

In order to write all the reviews on the screen, the following code was implemented:

```php
<?php
echo "REVIEWS:";
$sql = "SELECT * FROM reviews"; // retrieve all reviews
$result = mysqli_query($con, $sql);
while ($row = mysqli_fetch_assoc($result)){
    echo $row['review'];
}
?>
```

The code selects all reviews from the database containing the reviews, and then echoes them.

In order to delete a review, this button was implemented:

```html
<html>
<body>
<form action="deletereview.php" method="POST">
    <input type="submit" value = "Delete review" name="deletereview"><br>
</form>
</body>
</html>
```

When this button is clicked, deletereview.php is ran, which will delete all reviews. The code for deletereview.php looks like this:

```php
<?php
$review_input = $_POST['deletereview'];
$sql = "DELETE FROM reviews WHERE ID = 1;";
$result = mysqli_query($con, $sql);
header("Location: CSRF.php");
die();
```
The page after the review "Great service" was submitted is displayed in Figure 5:

Figure 5: Page with the review "Great Service"

From this figure, "Great service" can be seen as the only review available.

3.4.2 CSRF

Cross-Site Request Forgery or CSRF, is a frequently seen attack on web applications. A CSRF attack occurs when a malicious website causes a user’s web browser to perform an unwanted action on a trusted site [5]. These kinds of attacks are often hard for websites to detect and defend against, because they appear normal from the website’s perspective. CSRF attacks can have severe consequences because the attacker will be able to forge requests on a website disguised as another user. This is made possible due to HTTP protocols sending session cookies for each request to a server [5]. Cookies consists of data stored in the browser that are sent with every request to the server [22]. Cookies are used for various functionalities, such as acquiring information for use in subsequent browser-server communication without asking for the same information [23]. They are used to confirm that two requests were sent from the same browser [24]. If an attacker receives this session cookie, the attacker can be authenticated as a different user. While authenticated as another user, that user can be forced to execute unwanted actions. This can be done without the user being aware that any of this is happening. In order to understand how CSRF attacks work, consider Figure 6.
In step 1, a user authenticates itself to a website, in this case called "Abc.com". In step 2, a session cookie is generated and stored in the user’s web browser. In step 3, an attacker embeds a forged request into a hyperlink. The goal for the attacker is then to trick the user into clicking on the hyperlink. This is often done with the help of social engineering, such as sending the hyperlink via email [25]. In step 4, the user clicks on the link. In step 5, the forged request is sent to "Abc.com", with the user’s session cookie. In step 6, "Abc.com" will perform the forged request, and depending on the goal of the attacker, various things can occur, such as transferring money to the attacker.

When posting a review on the website, a HTTP POST request will be made in order for the website to store the review. GET and POST are two methods of form submissions, where information in the form fields are sent to the server [5]. GET requests are less secure and visible to everyone in the URL [26]. GET requests are therefore more vulnerable to attacks, such as CSRF attacks. One solution to this may seem as easy as using POST requests instead, but POST requests does not stop CSRF attacks either [5]. Cookies are sent in HTTP requests, which this attack will take advantage of. The most valuable information of the cookie for an attacker is the authenticated identity of a user. This is because having access to it enables various attacks to be executed.

3.4.3 CSRF today

In the OWASP’s top 10 web application security risks from 2021, the number one security risk was broken access control [14]. CSRF attacks are one of Notable Common Weakness Enumerations (CWEs) [27] that make this the number one security risk.
3.4.4 CSRF attack

This section will explain how a CSRF attack could be initiated by writing a review for other users to see. To be more exact, it will be evaluated if step 4 in Figure 6 can be executed by writing a review. Instead of writing a normal review, Figure 7 displays how an alternative review could look like.

![Figure 7: Page with a malicious review written on it](image)

This review seems to be encouraging users to click on a link, and is an example of how social engineering is used to trick people, enabling an attacker to execute a CSRF attack. It appears as if the link named ”MyReview” will redirect a user to the full review. The link can be made without showing the actual URL with basic HTML knowledge. If it does not exist any input sanitation, the attacker can write HTML code in the review. This enables the attacker to hide the URL with the text ”MyReview”, by using the ”href” attribute. This is what the attacker’s review actually looked like before posting it on the website:

```html
<a href="maliciousWebsite.com">MyReview</a>
```

The link destination ”maliciousWebsite.com” is defined, which could be the URL of the attacker’s website. Another way of posting the malicious link is to do it with an image tag. As soon as a user clicks on the link, a CSRF attack could be executed. When a user clicks on the link, they will be redirected to the attacker’s malicious website. Various things can happen at this point, all depending on the attacker’s goal, the user’s privileges on the website and so on. On the malicious website, the attacker could include a real review in order to not raise any suspicion that any attack is happening. As the user is exploring the malicious website, the CSRF attack has already happened in the background, without the user having any idea. As mentioned earlier, the session cookie is always sent with HTTP requests. This also means that when the user visits the malicious website created by the attacker, the user’s cookies is sent to that website as well. The only thing required from the user for the attack to work is that the user has to be authenticated on the previous website with the real reviews. In order for the website to identify users, a valid session cookie will be granted when logging in.
3.5 Countermeasure against CSRF attacks

In order to prevent CSRF attacks, it has to be a way to notice the forged requests made by an attacker, which same origin policy does not achieve \[30\]. This can be a tough challenge since the requests from a legitimate user and an attacker will be the same as seen from the website’s point of view. The complexity of defending against CSRF attacks can be noticed by broken access control still being the number one security risk in OWASP’s top 10 web application security risks from 2021 \[14\]. An initial defence against CSRF attacks could be client-side input sanitizing. By restricting the use of HTML tags \[31\] in the review, especially the `<` and `>` characters, tricking a user to enter a malicious website will be more difficult. If the attacker can not hide the original URL, as in the CSRF attack shown, the user will hopefully be more cautious.

3.5.1 Input sanitizing, accepting a-z,A-Z,0-9, space and period character

There are several ways of implementing client-side input sanitizing on HTML forms. The one presented in this section uses a JavaScript function to only allow the characters a-z, A-Z, 0-9, space and the period character character. The original form asking the user to enter the review needs to be updated, which now will look like this:

```
<html>
<body>
<form action="insertreview.php" method="POST">
  review: <input type="text" name="review" onkeyup = "restrictCharacters(this)"><br>
  <input type="submit" >
</form>
</body>
</html>
```

The new form will call a function named ”restrictCharacters” every time the user releases a key, this is done with JavaScript’s ”onkeyup” \[32\] event. The goal of ”restrictCharacters” is to only allow the characters mentioned and nothing else. The function looks like this:

```
<script>
  function restrictCharacters(input)
  {
    var userInput = /[^a-z0-9 .]/gi;
    input.value = input.value.replace(userInput,"");
  }
</script>
```

The script tags is used to let HTML know that this is a JavaScript code block. The function creates a regular expression that matches any character that is not a letter, number, space or period.
The final row then replaces any character that matches the regular expression with an empty string.

```
input.value = input.value.replace(userInput, 
```

By replacing any characters that are not allowed to an empty string, client-side sanitizing has been implemented.
4 Results

In this section, how well the different versions of the login implementation defended against different SQL injection attacks will be evaluated. The different types of SQL injection attacks that will be tested are the different attacks mentioned in the methodology section. The countermeasure against the CSRF attack will also be evaluated.

4.1 SQL injection countermeasures

4.1.1 Version 1 - No security

The first version had no extra security implemented, which of course will lead to every SQL injection being possible to execute. Taking Attack 1 from the section about SQL injections as an example (in this case entering user1'– ), will lead to a successful login attempt. On this setup it was crucial to enter a space after the double hyphen, otherwise it would leave an error message instead. By having no countermeasures against SQL injection attacks, the input can be treated as SQL commands, which makes it impossible to differentiate between the commands and the user's input.

Attack 1 worked by having a successful login attempt.
Attack 2 worked by having a successful login attempt.
Attack 3 worked by successfully creating an error.

4.1.2 Version 2 - Deny the quotation character

This implementation will defend against some SQL injections attempts, but also displays why it is not recommended to implement negative validation. This is because more characters than just the quote character can cause harm. This check will prevent attacks like Attack 1, but will not protect against several other attacks. Attack 3 as seen from the methodology section, will be effective on Version 2, because the quote character is the only character that is not allowed.

Attack 1 did not work and the message that quotes are not allowed was returned.
Attack 2 did not work and the message that quotes are not allowed was returned.
Attack 3 worked by successfully creating an error.

4.1.3 Version 3 - Escaped statements

In Version 3, mysqli_real_escape_string was used. This will be a better option than version 2, because it escapes several characters instead of only accepting input without the quote character. This will defend against attacks such as attack 3, because the "backslash" character will be escaped. Because databases usually have several different characters that could be interpreted as commands instead of plain text, escaping characters should not be the only source of defence against SQL injection attacks. Using this built-in function will defend against various SQL injection attacks, but complex attacks overcoming this defence will
be discussed later.
Attack 1 did not work because the quote was escaped.
Attack 1 did not work because the quote was escaped.
Attack 3 did not work because the "backslash" character was escaped.

4.1.4 Version 4 - Parameterized queries

In version 4, parameterized queries was used. It managed to defend against all
the attacks tried here as well.
Attack 1 did not work
Attack 1 did not work
Attack 3 did not work

4.1.5 Overview

<table>
<thead>
<tr>
<th>Version</th>
<th>Attack 1</th>
<th>Attack 2</th>
<th>Attack 3</th>
</tr>
</thead>
<tbody>
<tr>
<td>No safety</td>
<td>x</td>
<td>x</td>
<td>x</td>
</tr>
<tr>
<td>No quotes</td>
<td>✓</td>
<td>✓</td>
<td>x</td>
</tr>
<tr>
<td>Escaped statements</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Parameterized statements</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
</tbody>
</table>

Table 1: Showcasing how well the different versions managed to defend against
certain SQL injection attacks; ✓ if defended successfully, x otherwise
4.2 CSRF in review functionality

Initiating a CSRF attack through review writing was shown possible. This is due to it being possible to embed a link when writing the review, that could then be used to redirect a user to a malicious website carrying out the attack.

The input sanitizing showcased a clear difference in the defence against CSRF attacks. Executing the same CSRF attack as previously shown in the CSRF attack section, but now with the input sanitizing in place, the review looks like this:

![Figure 8: Page with a failed initiation of a CSRF attack](image)

Because special characters were not allowed, the original URL is now clearly visible, and the clickable link is now only text. Comparing with the output of the review when no input sanitizing was done, this is a better option. But a user could still enter the malicious website. Input sanitizing on the client-side will not be effective against an experienced attacker, which is discussed in Section 5.
5 Discussion

The complexity of securing a website is showcased by injection attacks such as SQL injection attacks remain among the top 3 security risks of 2021. Despite being known for many years, SQL injection attacks and CSRF attacks are still common and pose a threat to many web applications. The following are some key insights based on the results.

5.1 Countermeasures against SQL injection attacks

In most cases, escaped statements and parameterized queries offer comparable protection against various SQL injection attacks. However, there are complicated attacks that could potentially bypass escaped statements but not parameterized queries. One such attack are encoded SQL injection attacks, where the input will be encoded and then translated in the database to the original characters [18]. Another reason why escaped statements is not enough is that it does not escape the wildcards "\" and "\%" used in SQL. These can be exploited in a query consisting of the LIKE statement, which could produce similar effects to the WHERE statement. The key point is that input validation/sanitation and escaped statements can add depth but should not be the only defence against SQL injection attacks.

5.2 CSRF in the review functionality

When writing a review, initiating a CSRF attack was shown to be possible by embedding a link with HTML code. This link can redirect users to a malicious website that can execute several attacks depending on the goal of the attacker. Extensive efforts are required to defend against CSRF attacks, and client-side input sanitizing was tested as an initial step to defend against them. Client-side input sanitizing will on the other hand not be effective at all to an experienced attacker. The solution presented in this thesis has limitations, one of which is its client-side implementation. This enables various possibilities for an attacker to overcome this countermeasure, such as disabling JavaScript [33]. Since client side processes typically are executed on the user’s web browser, an attacker could disable JavaScript in it. This could potentially deactivate the "restrictCharacters" function shown in Section 3.5 allowing all characters to be accepted in the reviews again. The next step for a more advanced countermeasure would be token-based mitigation [25], but is beyond the scope of this project and could be explored in future work.

5.3 Safe today, safe tomorrow?

If a system meets some certain criteria for being safe today, will it be safe forever? The answer is no, and there are several reasons why.
Security systems should always have a documentation containing what the system is supposed to achieve. This can clarify which threats it is supposed to defend against, what it is not defending against and so on. Without a stated goal it is nearly impossible to analyse and test the security of the system, since the expectations of it is unknown. A secure system will constantly be tested for any vulnerabilities it may posses and updated accordingly. Although the countermeasures presented today may be effective, they may not be tomorrow. New technologies and more clever attacks will most certainly and should be expected to be invented. Identifying every threat against the system and ensuring that none of its vulnerabilities can be exploited by any of them is difficult.
6 Conclusion and future work

This thesis investigates SQL injection attacks and countermeasures in PHP, as well the possibilities of initiating a CSRF attack through review writing and how one might try to prevent it.

For the login functionality, various implementations of the login form were tested to evaluate their effectiveness against SQL injection attacks. Escaped statements and parameterized queries were shown to be the most effective countermeasures. Although they performed similarly for tests done here, escaped statements could potentially still be bypassed by other methods as discussed. The conclusion is therefore to use parameterized queries in PHP as a countermeasure against SQL injection attacks.

The review functionality was shown to be vulnerable to CSRF attacks, due to the possibilities of embedding a link in a review. This link could then be used to redirect the user who clicks on it to a malicious website, where the CSRF attack then could be executed. Client-side input sanitizing against CSRF attacks was evaluated as ineffective for many reasons, the primary one being that it was implemented on the client side, which allows an experienced attacker to easily overcome it. The conclusion is therefore that client-side input sanitizing is not enough to defend against an initiation of a CSRF attack.

This thesis opens up several possibilities for further research, such as identifying more security threats the functionalities face, implementing the functionalities on a live website and testing various countermeasures that could be used to prevent CSRF attacks, such as token based mitigation. A natural extension to this thesis would be to execute a complete CSRF attack, using the initiation demonstrated here.
References


