# Secure version explanations

To make our web application safer, we improved our code in several different ways. First of all, in the class “authentication” (and in all files containing some external input gained from a post action), we used **mysqli\_real\_escape\_string** method to escape input and avoid a sql injection

$username = **mysqli\_real\_escape\_string**($db, $\_POST['username']).

Then we forced the result of the login to be equal 1, so that the entry MUST coincide to only one login

if (**mysqli\_num\_rows**($results) == 1)

Lastly, we used the **md5** method in order to encrypt the password, which was previously visible while typing.

The second big change concerns the login. In fact, whereas in the insecure version the login is not maintained through all the pages, in the secure version we used the **$\_SESSION** function of php instead of cookies. Sessions are more secure compared to cookies, as they save data in encrypted form. On the contrary, Cookies are not secure, as data is stored in a text file, and if any unauthorized user gets access to our system, they can temper the data. This change can be observed in the “authentication” class.

Moreover, another way in which we avoided sql injections, was through prepared statements. These are useful, because escaping parameter values is not needed. The values will, later, be transmitted using a different protocol. In this way, an SQL injection is not possible.

For instance, a snippet of code (belonging to the “authentication” class of the insecure version is:

**$user\_check\_query** = "SELECT \* FROM authentication WHERE username='$username'";

**$result** = **mysqli\_query**( $db, $user\_check\_query);

**$user** = **mysqli\_fetch\_assoc**($result);

While the secure version is:

**$stmt** = $**db** -> prepare("SELECT \* FROM authentication WHERE username=? AND password=?");

**$stmt**->bind\_param("ss", $username, $password);

**$stmt**->execute();

With this solution, there is no way to perform a sql injection.

In order to avoid XSS, we made use of the function **strip\_tags**, so that if anyone tries to insert some malicious code, it is not read as code because of the lack of tags. This can be seen in the “home” class.

Example of use in the class “home”:

if ( isset(**$\_POST**['question'] ) && !empty(**$\_POST**['question']))

**$question** = **strip\_tags**($\_POST['question']);

In alternative, the function **html\_specialchars** can be used.

In real usage, csrf.php is another website, reached via a link that, in our case, could be posted using the answer box. When the user clicks, a hacker posts a mysterious question in the database without the user’s knowledge. To avoid these type of attacks, hidden tokens can be used. These tokens are linked to the session. Each time a user logs in, and whenever a request made, the program checks if the newly generated token corresponds to the original one; it this is not the case, the app does not let the user access it.

if($\_SERVER[‘REQUEST\_METHOD’] === ‘POST’){

if(!isset($\_POST[‘token’]) //($\_POST[‘token’])){

die(‘Invalid CSRF token’);

}

}

$\_SESSION[‘\_token’]= bin2hex(openssl\_random\_pseudo\_bytes(16));

<input type= “hidden” name= “\_token” value= “<?php echo $\_SESSION[‘\_token’];?>”>