Week 10 – Security

Protecting against SQL injections

To protect your system against SQL injections use a prepared statement.

```
// Create and prepare query
$query = "INSERT INTO users (firstname, lastname, email, password, salt)
VALUES (:first_name, :last_name, :email, :password, :salt)";

// Prepare the statement to prevent SQL injections
$stmt = $database->prepare($query);
```

Protecting against Cross Site Scripting (XSS)

To protect your site against XSS, use **htmlspecialchars** wherever you output the user's input. An example of XSS is when a user submits a text post containing JS code inside of <script></script> tags. A site that is unprotected would execute it as HTML and would run the code. In one of your php files, add this code and run it. You should redirect to Facebook once the page loads. If a user submits a comment or post to your site and you don't protect it, then they could have all of your users redirect to their site which could be a phishing site.

```
$text = "<script>window.location.href = 'http://www.facebook.com';</script>";
echo $text;
```

Try passing the text through **htmlspecialchars** and running the page again. You should see that it comes up as text now.

```
$text = "<script>window.location.href = 'http://www.facebook.com';</script>";
echo htmlspecialchars($text);
```

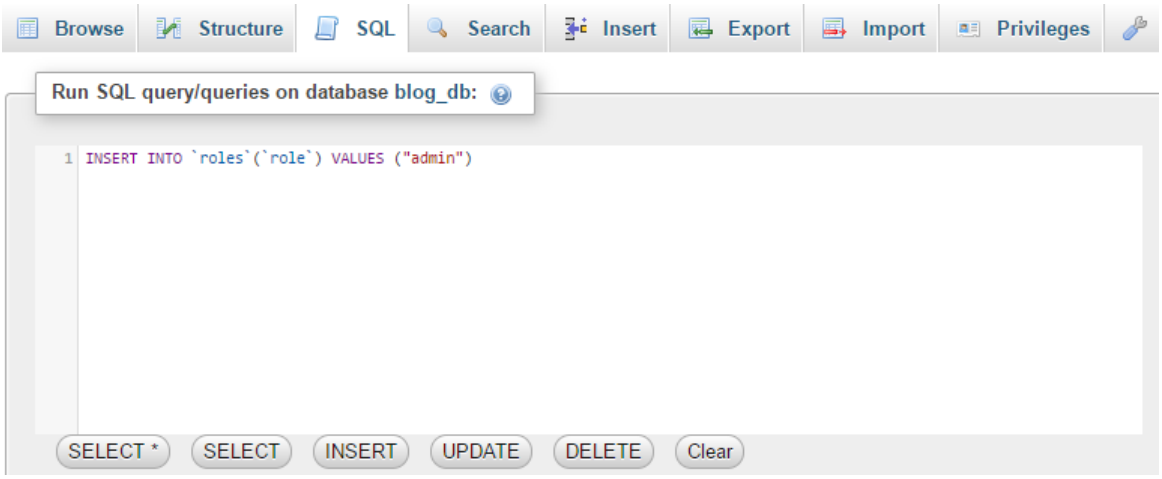
Remember, you should really only call **htmlspecialchars** whenever you want to output the user's input, and not before submitting it to the database. This gives users the ability to update their original input before it gets serialised.

User Roles

User roles help enforce what level of access each user account has within a system. Typically small scale websites will have 2 roles: **Admin** and **User**; where the Admin role grants total access to the system and the user has more limited access. Other roles can be defined as "Deactivated", "Banned", "Moderator", etc. which further defines the level of authorisation the account has.

Take a look at **blog_db**. Inside of **blog_db** you have four tables: **users**, **roles**, **posts**, and **user_in_roles**. For this exercise you'll be working with **users**, **roles**, and **user_in_roles**.

The roles table has 2 columns: **id** and **role**. Insert 2 new records for "**admin**" and "**user**" in this table like so:



Admin should have an ID of 1 and user should have an ID of 2.

Your **user_in_roles** table assigns a role to an existing user account. **User_in_roles** has two columns: **user_id** and **role_id**; both are foreign keys for the users and roles tables respectively. So if the user with ID 1 has a role_id of 1 (1 being admin) that means the user as admin level access.

Ok, so how do we check who has which role? By using an INNER JOIN statement like so:

'SELECT 1 FROM users AS U INNER JOIN user_in_roles AS UIR ON U.id = UIR.user_id WHERE UIR.role_id = 1 AND U.id = :user_id'

What the statement above does is joins two tables together so we can use the data for selecting and comparing.

- **SELECT 1** just means we want a true/false response rather than any data.
- **"users AS U"** means we're giving the users table an alias of **U** to shorthand it. Same goes for **user_in_roles**.
- **ON U.id = UIR.user_id** means we want to join records from both tables where the **id** from **users** is the same as **user_id** in **user_in_roles**, so two records are merged together as one.

You can use INNER JOIN statements to select specific kinds of data from either table like so:

'SELECT U.id, U.username, P.title, P.post, P.date_time FROM users AS U INNER JOIN posts AS P ON U.id = P.user_id WHERE U.id = :user_id ORDER BY P.id DESC LIMIT 25'

The above query will return the User's ID and Username and all posts' titles, text, and creation date-times and orders it in descending order by the Post's ID so it will show the most recent ones made, and will only show the top 25 results.

Sessions

Because HTTP is stateless, in order to associate a request to any other request, you need a way to store user data between HTTP requests. Cookies and URL parameters are suitable ways to transport data between requests, however they are accessible and editable on the client side, which is a major security risk. The solution is to use a web server session.

Sessions allow us to store data on the server, give it an ID, and allow the client only know that ID.

Sessions are always used for logging into an account. Whenever you log into a website a new session is started which allows you to access the site.

To create a new session we need to first initialise it with **session_start().** After that you can create new session value using $_SESSION global variable. Sessions in PHP act similar to associative arrays and take a key/value pair.

```php
// Init sessions
session_start();
// User is now logged in
$_SESSION['loggedIn'] = true;
```

Above we started a started a new session and created a new variable **loggedIn** which now records the user as logged in. We can use this global variable across the site to enforce authorisation by checking whether or not if the user is logged in. If they aren't, they're redirected to the login page.

```php
if (!$_SESSION['loggedIn']) {
    // Redirect to login page if not logged in
    header("location: http:{$_SERVER['HTTP_HOST']}/login.php");
}
```

To destroy all of the sessions we use **session_destroy**. But first you will need to call **session_start**, otherwise **session_destroy** will not work.

```php
// Init sessions
session_start();
// Destory all session values for the user
session_destroy();
```

Exercise 1 – Assign user roles

Upon user registration a user must be given the role of “user” by default. In **includes/functions.inc** create two functions called **get_role($role_name)** and **assign_user_role($userId, $roleId)**. The function **get_role** should return the role ID from the table “roles” based off the role name (i.e admin). The function **assign_user_role** takes the user’s ID and the role ID and creates a new record in the **user_in_roles** table.

Call these functions in **scripts/registration.inc** like so:

```php
// Execute the query
$stmt->execute();

if ($stmt->rowCount() > 0) {
    // Get newly created account ID
    $userId = $database->lastInsertId();

    $roleId = get_role($role);
    assign_user_role($userId, $roleId);

    // Redirect to index.php
    header("location: http://localhost/blog/index.php");
}
```

In your database remove all of the current users in your users table and restart the WAMP server. This will reset the ID auto-increment counter. Create a new account with the role of “admin”. Create another account with the role of “user”.

Exercise 2 – Authorisation

In **includes/functions.inc** add this function:

```php
function enforce_login() {
    if (!$_SESSION['loggedIn']) {
        header("location: http://{$_SERVER['HTTP_HOST']}/login.php");
    }
}
```

Call this function in **post.php** under the included files. This will redirect you back to **login.php**.

In **header.inc** alter the nav block to show the links to show the login and register page when the user is logged out, and show the logout link when they are logged in.

```php
<nav id="accessMenu">
    <?php if (!isset($_SESSION['loggedIn'])) { ?>
    <a href="./register.php">Register</a>
    |
    <a href="./login.php">Login</a>
    <?php } else { ?>
    <a href="./logout.php">Logout</a>
    <?php } ?>
</nav>
```

Now go to **scripts/login.inc** and add **$_SESSION['loggedIn'] = true** inside of the if statement on line 13 before redirecting to **index.php**. Do the same thing for **register.inc.**

Exercise 3 – User role authorisation

Inside of **functions.inc** create a function called **is_admin($userId)**. This function will check whether or not the user has the role of admin when logging in and return a boolean value.

If the query returns true then assign true to the session variable, otherwise assign false. Use the INNER JOIN statement from before to perform this query.

Call this function in **scripts/login.inc** before the **loggedIn** session and assign it to a session variable **$_SESSION['isAdmin'].**

Inside of **about.php** there is an anchor tag that contains an event handler to show a form to allow the admin to update the blog's about page. Wrap the anchor tag inside of an if statement to check if the **isAdmin** session has be created and if the user is an admin.

```php
<?php if (isset($_SESSION['isAdmin']) && $_SESSION['isAdmin']) { ?>
<a id="editAbout" onclick="toggleForm();" class="action-trigger">
    Edit
</a>
<?php } ?>
```

Exercise 4 – Logging out

Create a new file called **logout.php** in your root folder. The logout file must destroy all created sessions and redirect the user back to the login page, otherwise they will be stuck on a blank page.

Create the file and try logging out. Check the header for the login/register links and try accessing **post.php**. Login again as an admin and look through all of the pages. Logout once more.