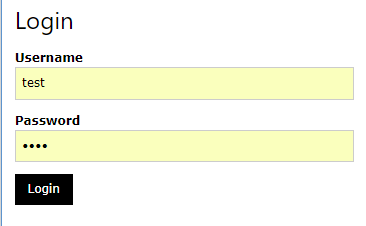
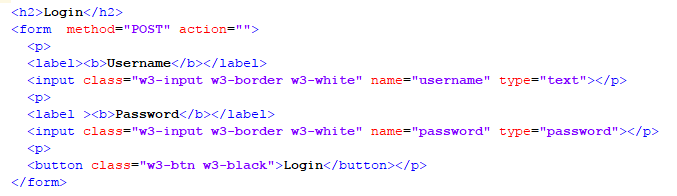
Task 1: Create a simple form containing a text input type, e.g.





The form needs to have the method set to what you would expect, e.g. **POST** for a login form.

# Task 2. Test the following using more than one browser:

Enter:

<script>document.writeline("testing testing");</script>

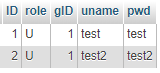
into the text box, e.g. Username and press submit.

What happens?

# Task 3. Add Authentication

To add authentication to your web page, you need a database to store user login details. Then you need a script to query the database and a method to pass the results back to a view.

Create a table in your database, e.g. users:

For the users table shown, the ID is an auto incrementing integer, used also as the unique key and index.

## SQL query

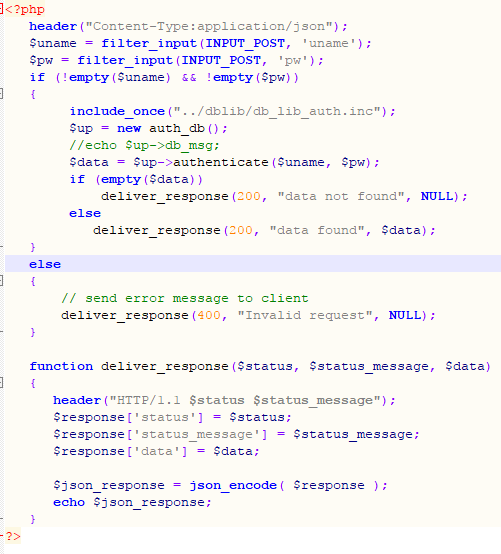
Create the queries needed to extract the data you wish to work with. For my example, I used the following:

SELECT CONCAT(ID, \":\", role, \":\", gID) As uProfile

FROM users WHERE uname = "test" and pwd = "test";

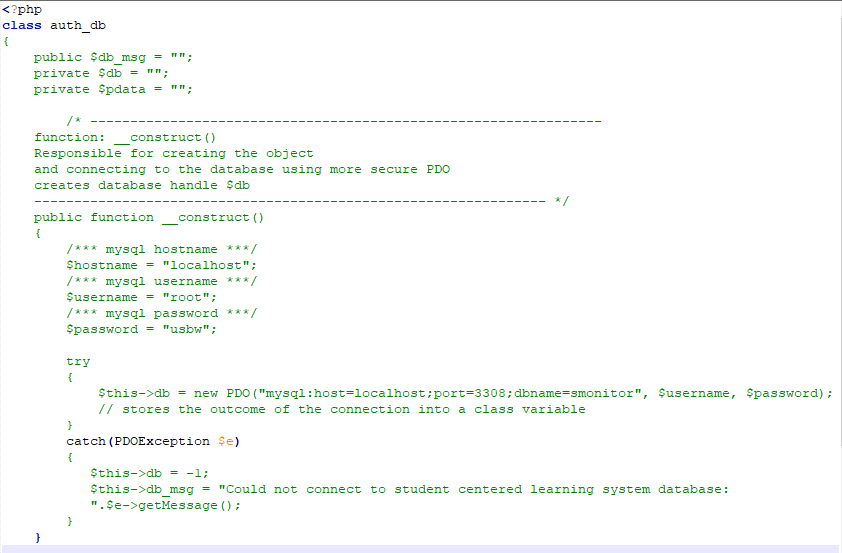
The above will provide a string of data (e.g. 1:U:1) to the calling function/program. It is entirely up to you how you return data, but this seemed to be more ambiguous to a hacker than an array or well- formed string.

With this working, a web API with associated database functionality can be created to perform the query.

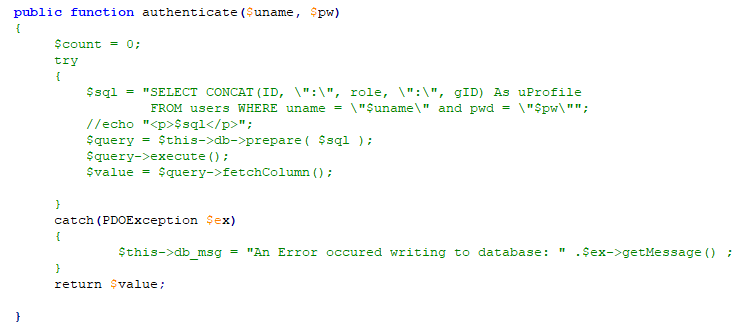


You will see as you read through the code that the **filter\_input()** command is used to extract POSTed data. A database library named: db\_lib\_auth.inc is then included and used to authenticate the user.

The following DB library will be improved going forward to deal with SQL injection, etc., but for now, it has been created without such mechanisms in place (for testing):



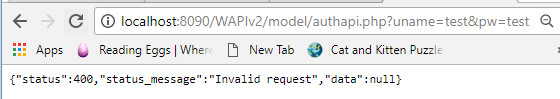
The above shows the constructor that creates a connection to the database using PDO.



Remember to add the final **}** for the class.

Testing so far:

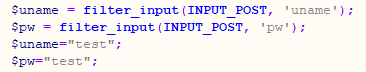
The Web API cannot be tested via the URL this time, as it is expected POSTed data.



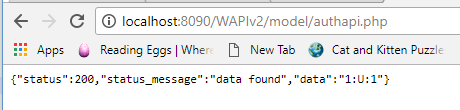
As you will see, you cannot pass the username and password into the URL. Therefore, as a start, you can manually change the following:



Comment them out and set $uname=”test” and $pw=”test”:



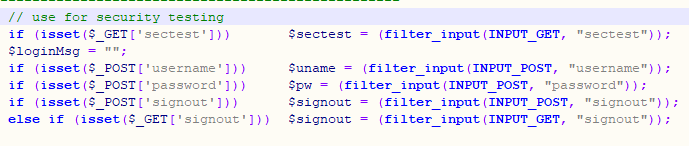
You should now be able to test the new Web API:



The next stage is to add controller code, e.g. to connect to the URL via a script.

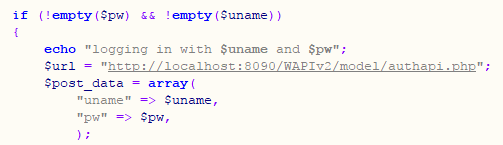
Create a file named login\_controller.php (or .inc) and store it in the controller folder.

Add the following variables to store form data:

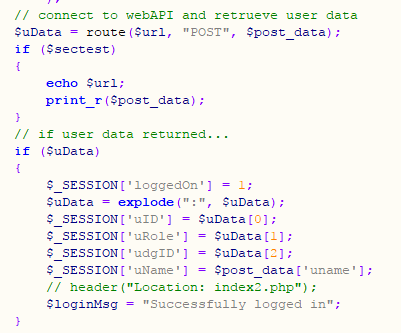


The first variable $sectest will be used to display data only if it is set. The other variables store form data.

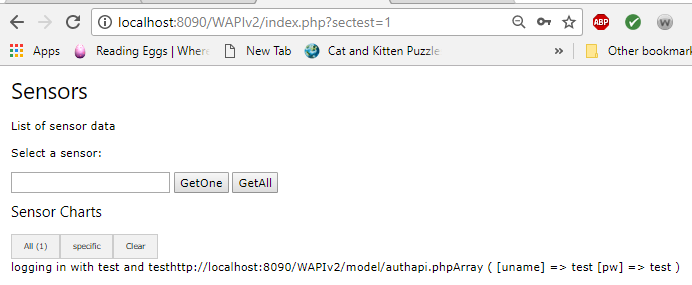
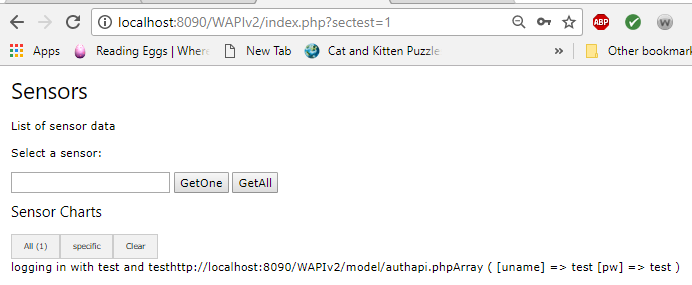
The next thing to do is to check whether the username and password were delivered via the HTML form. If the password variable and the username variable are not empty, then the URL can be set to connect to the authentication Web API and the POST data generated using the contents of the variables.



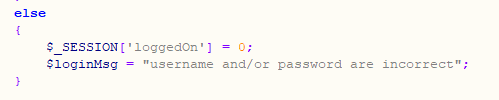
Next, the connection to the URL can be made. This is achieved in the example, via the route(…) function (passing the URL, method type: POST and post data to it). The **route()** function will be shown shortly. This has been added to a separate file and included by index.php.



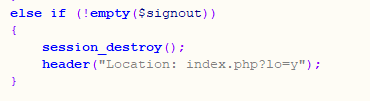
The above code shows that is data was returned from the route() function, session variables will be set (that relate to the user data as well as a variable to represent that the user is logged on). The “if ($sectest)” check prints data on the screen if the URL contains ?sectest=1:



If no data was returned an error message is set and the **loggedOn** session variable is set to 0.



A further statement is also included to deal with signing out of the system. This can be achieved via a GET or POST request. Signing out will cause the user’s session to be destroyed.



After sign out the index.php file is called again to refresh the web page without session variables set. And the login\_view.php file is shown.



The **login\_view.php** shows either the login form or the signout form / hyperlink:



The final code for the authentication system are the **route()** function and the include statements in the **index.php** file (which is in the root directory.



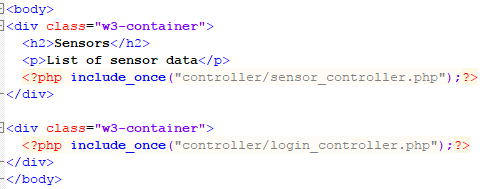
The **routing.php** file contains a function that connects to the web API. This function is used for the main sensor Web API and the authentication Web API.



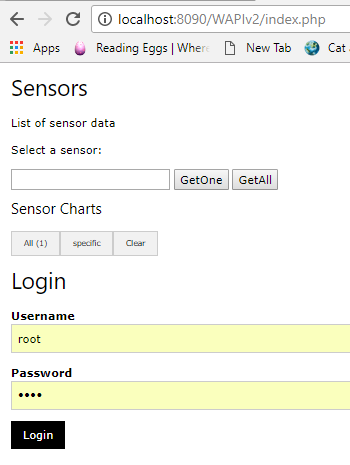
As can be seen the POST method sends data as POST data and the GET method ensures data is appended to the URL string. Several options are then set, e.g. to ensure JSON is used and to set the URL to connect to.

After receiving the data from the URL the connection is closed.

The body of the main **index.php** file is:

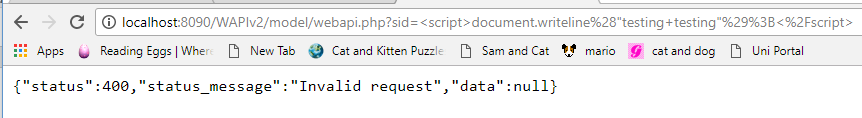


The above code can be run via:

When you first navigate to the demonstration code, you see two forms that you can now use to learn more about web security.

Task 3 Enter JavaScript into the sensor text box. This will result in the following query being sent to the Web API – shown as a separate test.

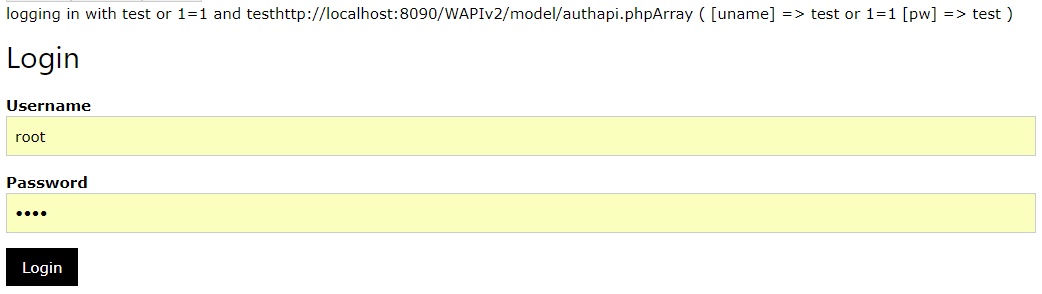
When a script is sent to the web API, the following is retrieved:



Another test to perform is the SQL injection check, where you not only add, say a sensor id, but also additional code. See example below:

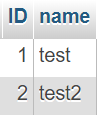
select ID, uname from users where id=1 or 1=1

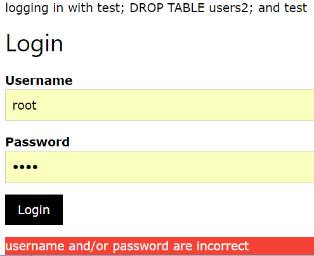
Running the above results in the following, when run with **?sectest=1** appended to the url:



# Task 4

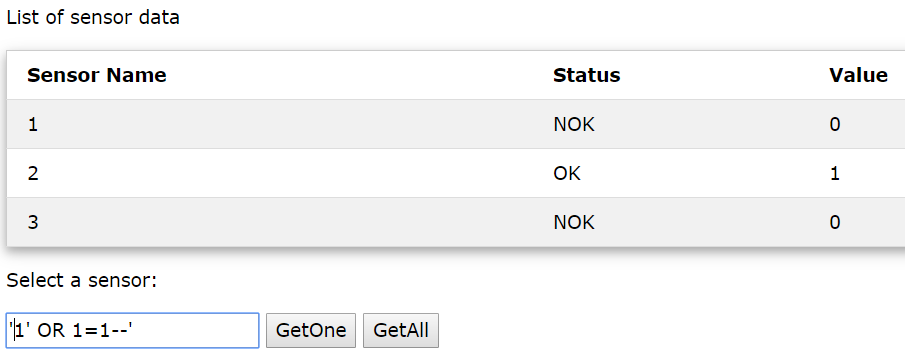
You might also wish to enter an SQL query to the end of a string, e.g. enter the following into the username field:

test; DROP TABLE users2;

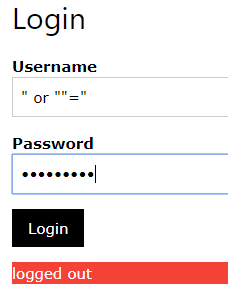
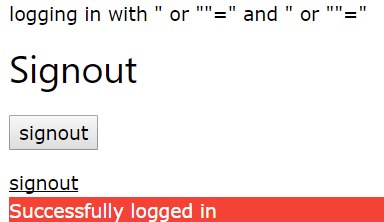
The output from trying this:

As can be seen, the user was unable to log in. You would also observe that the users2 table was not deleted from the database.

Task 5. In the case of the sensors form, adding the wrong data (e.g. ‘1’ OR 1=1—‘ resulted in all sensors being selected (see below). This might not be desirable, e.g. if certain users were restricted as to what data they could see.



Now try: adding **" or ""="** to the username and password boxes as shown:

The output is the following (logged in)

The resulting SQL is:

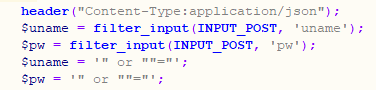
SELECT CONCAT(ID, ":", role, ":", gID) As uProfile

FROM users WHERE uname = "" or ""="" and pwd = "" or ""=""

As can be seen, the user is logged in. This is because all user values were returned and the db\_auth\_lib.php code returns one row, i.e. the first row encountered. Anyone can log in to the system without improvements being made to the code, e.g. sanitising HTML form elements.

Task 6. **Test the code with further SQL injection statements**, via the HTML form and see what results are returned.

In some cases, to see the outcome, the data needed to change to GET requests or hard coded, e.g. in the authapi.php script, the form data was hard coded:



Results in:

SELECT CONCAT(ID, ":", role, ":", gID) As uProfile

FROM users WHERE uname = "" or ""="" and pwd = "" or ""=""

You would need to test via the API and the browser, as well as to echo the SQL statement in the db\_lib\_auth.inc file.

This results in all data being returned from the users table.

## Another test:

SELECT CONCAT(ID, ":", role, ":", gID) As uProfile

FROM users WHERE uname = "" or ""="" and pwd = "" or ID = 2 or"x"=""

### To test this via the HTML form:

Username: " or ""="

Password: " or ID = 2 or"x"="

With this you will be logged on. This could be used to try to gain access via an administration account.

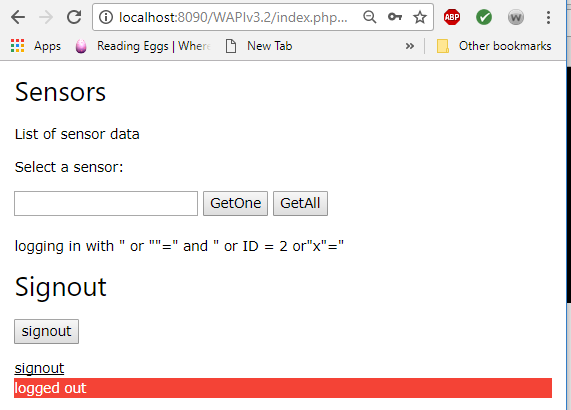
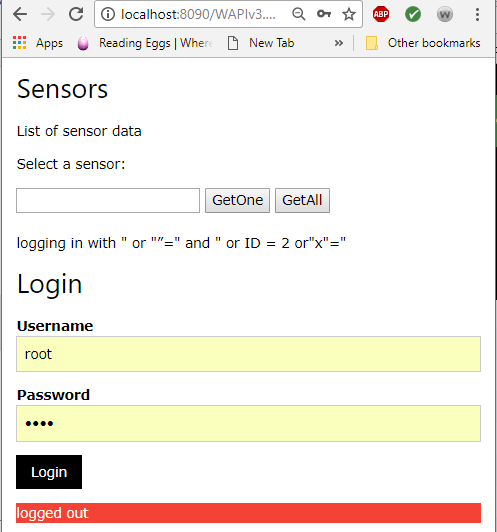
Explanation:   
  
The default query tries to return a single account where the username equals something and the password equals something. If we were to use a “ or “”=” on both fields however, this would return multiple results, which isn’t what the system should do; we need a single row returned.

With this SQL injection attack, we take the attention away from the password field comparison and focus on the ID field. As website administrators are usually the first or second accounts created on a website, focusing on the ID is likely to find us an administrator account.

This isn’t the end of the task however, as we need to remember that there is a text comparison that we need to supplement. For this, we simply make a statement that will always fail i.e “x” = “ (the trailing quote will be added during execution by the original comparison)

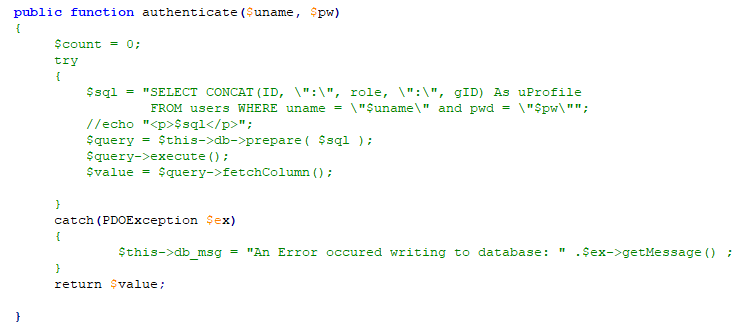
Now you can try some more tests (before the code is fixed).

Task 7. Investigate sanitisation methods for the code and attempt to fix it.

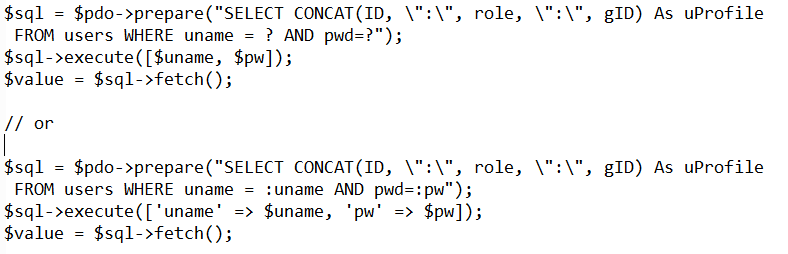


**filter\_var($\_POST['uname'], FILTER\_SANITIZE\_STRING);**

In order to secure queries that are being sent to the MySQL server, one tactic is to use the PDO object. By default all parameters that are passed into the PDO class are filtered for any suspicious data. Unfortunately this example below doesn’t incorporate this:



To improve this, we could do either of the following:



These two examples differ from the original provided in that they use the PDO parameters. When PDO parameters are provided they are automatically filtered. If you don’t use parameters, the PDO class assumes that the SQL string is secure and doesn’t have any potentially dangerous data.