



SECURING WEB APPLICATIONS

Assignment 3

Luke Fox

20076173

CONTENTS

Useful Information3

Section 14

Section 25

Section 37

Section 48

Section 59

USEFUL INFORMATION

Index:

- 1.a) login_POST.php
- 1.a) welcome_POST.php
- 1.b) login_POST.php
- 1.b) welcome_POST.php
- 2) access_DB.php
- 2) databaseLogin.php
- 3) html_security.php
- 4) validation.php
- 5) cookies.php

XSS Script Used:

```
<script type="text/javascript">window.location = 'http://facebook.com/';</script>
```

SQL Injections Used:

- 1) junk' or '1'='1
- 2) '; DROP TABLE loginDetails; --

SECTION 1

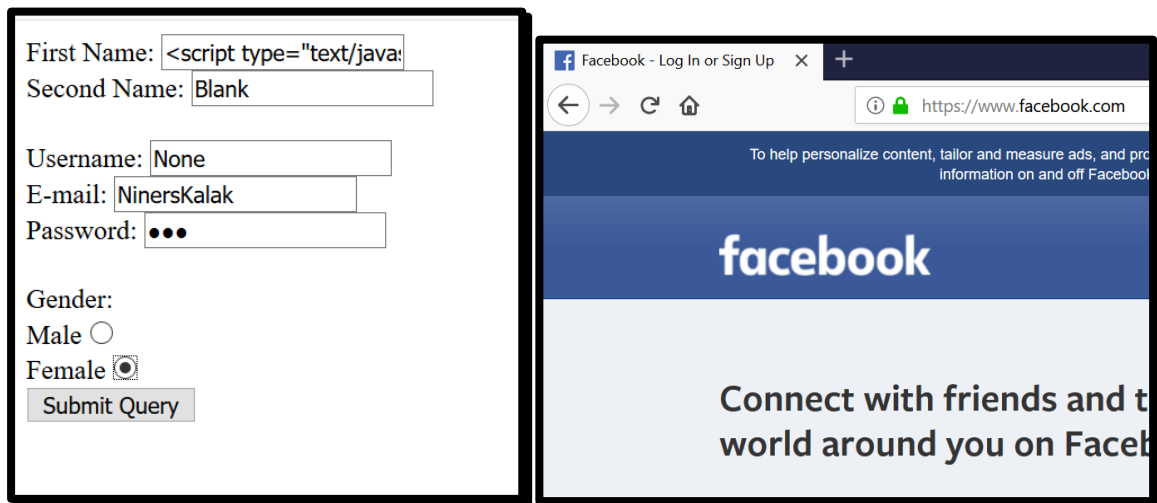
For section 1 of the assignment, I prepared 2 PHP scripts. Both scripts were identical, being account creation pages asking the user for details like their name, email, password and gender. Once the user submitted this data, they would be redirected to a welcome page that would display some of their entered information. The only difference with these two scripts was the method they used. In this case, one script used the POST method, and the other used the GET method. Regardless of the method used, the outcome in the welcome page was the same. However, are some differences in using GET and POST that should be noted.

POST: When using POST, the information that is submitted will not appear in the URL. Pages with the POST method also cannot be bookmarked, as these pages are not cached. POST can also pass much more information, as variables can have any length.

GET: GET is used to retrieve data, unlike POST which inserts/updates data. It is also limited in the information it can pass, as the variables are stored in the URL, and the URL must have a finite length.

POST and GET are similar methods used in HTML forms, but as seen above, share some key difference. GET is primarily used to retrieve data, and POST to insert/update data. Depending on the task at hand, either method could be more useful than the other. But for simple submitting data from a form, POST is the more practical option as it allows unlimited variable lengths.

The final step to section one was to demonstrate cross-site scripting. To achieve this, I used a simple script (Cross-site Scripting.txt in the scripts folder of assignment) that, when entered into the form, would ignore the form action of redirecting to the welcome page and instead direct them to a specified URL within the script. This is shown in the screenshots below:

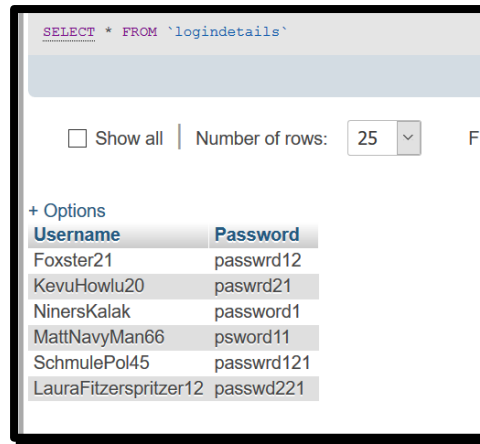


The image consists of two side-by-side screenshots. The left screenshot shows a web form with the following fields: 'First Name:' with the value '<script type="text/javascript">', 'Second Name:' with the value 'Blank', 'Username:' with the value 'None', 'E-mail:' with the value 'NinersKalak', 'Password:' with three dots, and 'Gender:' with 'Male' selected. A 'Submit Query' button is at the bottom. The right screenshot shows a browser window with the Facebook login page. The address bar shows 'https://www.facebook.com'. The page has the Facebook logo and the text 'Connect with friends and the world around you on Facebook'.

*Note, this was demonstrated in Firefox as Chrome protects against XSS.

SECTION 2

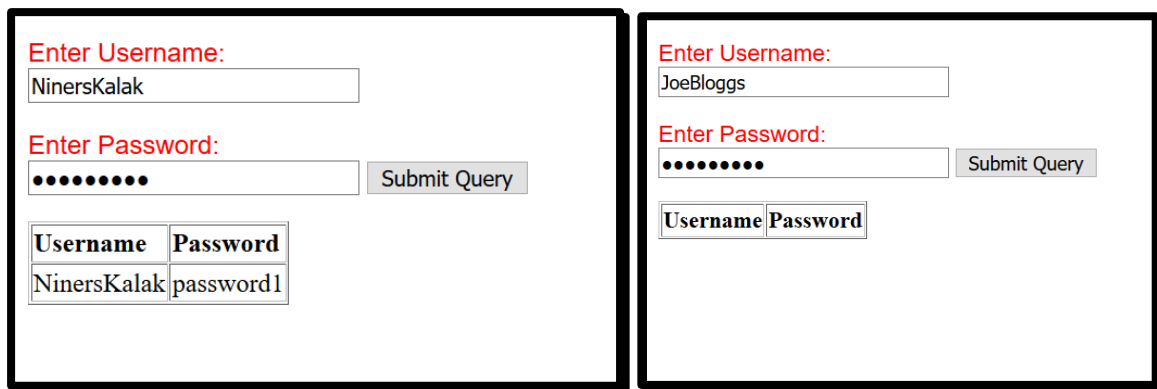
In section 2 of the assignment I created a database using phpMyAdmin. The next task was to create a HTML form that could query this database. This database is shown below:



The screenshot shows the phpMyAdmin interface with a SQL query executed: `SELECT * FROM `loginDetails``. The results are displayed in a table with two columns: Username and Password. The table contains seven rows of data.

Username	Password
Foxster21	passwr12
KevuHowlu20	paswr121
NinersKalak	password1
MattNavyMan66	psword11
SchmulePol45	passwr121
LauraFitzerspritzer12	passwd221

This form allowed the user to enter a username and password. If the username AND password matched an entry in the database, that entry would be returned in a table. Else, the table would return blank. This is shown below:



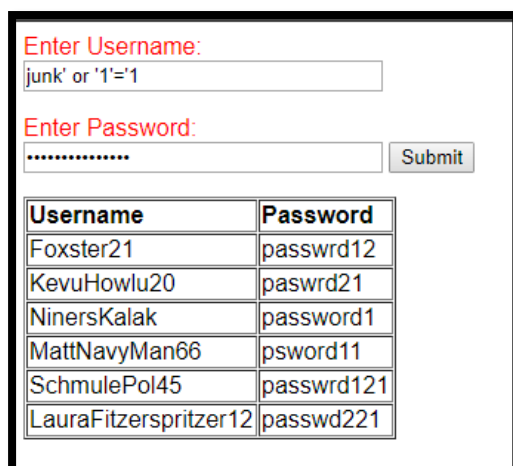
The left screenshot shows a successful login. The username 'NinersKalak' and password 'password1' are entered, and the 'Submit Query' button is clicked. The resulting table shows the login details for 'NinersKalak'.

The right screenshot shows a failed login. The username 'JoeBloggs' and a password (represented by dots) are entered, and the 'Submit Query' button is clicked. The resulting table is empty.

Username	Password
NinersKalak	password1

The second part of this section required an SQL Injection to be performed. In the case of this form, simply entering "junk" or "1='1'" into both input boxes would display all the tables within the database. The second injection was achieved by entering ";" DROP TABLE loginDetails; --", which would delete the table containing this data.

INJECTION 1



The screenshot shows the login form with the SQL injection payload "junk' or '1='1'" entered in the username field. The password field is empty. The 'Submit' button is clicked. The resulting table displays all the data from the loginDetails table.

Username	Password
Foxster21	passwr12
KevuHowlu20	paswr121
NinersKalak	password1
MattNavyMan66	psword11
SchmulePol45	passwr121
LauraFitzerspritzer12	passwd221

INJECTION 2

The diagram illustrates a two-step process. On the left, a login form with fields for 'Enter Username:' and 'Enter Password:', a 'Submit' button, and a table with columns 'Username' and 'Password'. A red arrow points to the right, where the same login form is shown. In this second state, the 'Enter Username:' field contains 'NinersKalak' and the 'Enter Password:' field contains '*****'. Below the form, an error message is displayed: 'ERROR: SQLSTATE[42S02]: Base table or view not found: 1146 Table 'fox_credentials.logindetails' doesn't exist'.

Enter Username:

`[' DROP TABLE loginDetails; --`

Enter Password:

Submit

Username

Password

Enter Username:

NinersKalak

Enter Password:

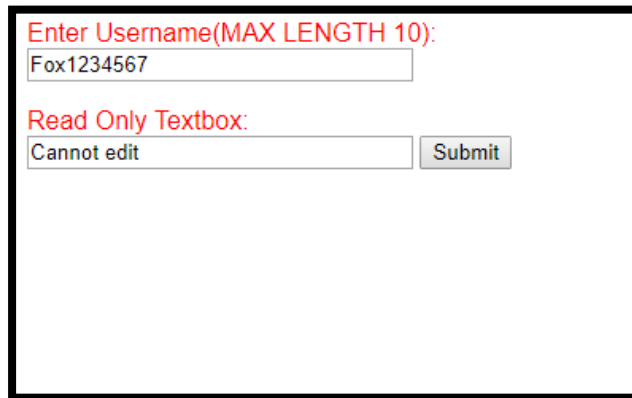
Submit

ERROR: SQLSTATE[42S02]: Base table or view not found: 1146 Table 'fox_credentials.logindetails' doesn't exist

As seen in the first injection, the table is displayed with all entries. In the second injection, the drop table SQL query deleted the database, and entering a previously matching entry as seen in the second picture, returns an error saying the table no longer exists.

SECTION 3

Section 3 required a demonstration of some of the basic forms of protection that HTML can offer to protect against attacks. For this, I again prepared a basic form page alongside two HTML form restrictions, 'maxlength' and 'readonly'. The use of these form restrictions is shown below:

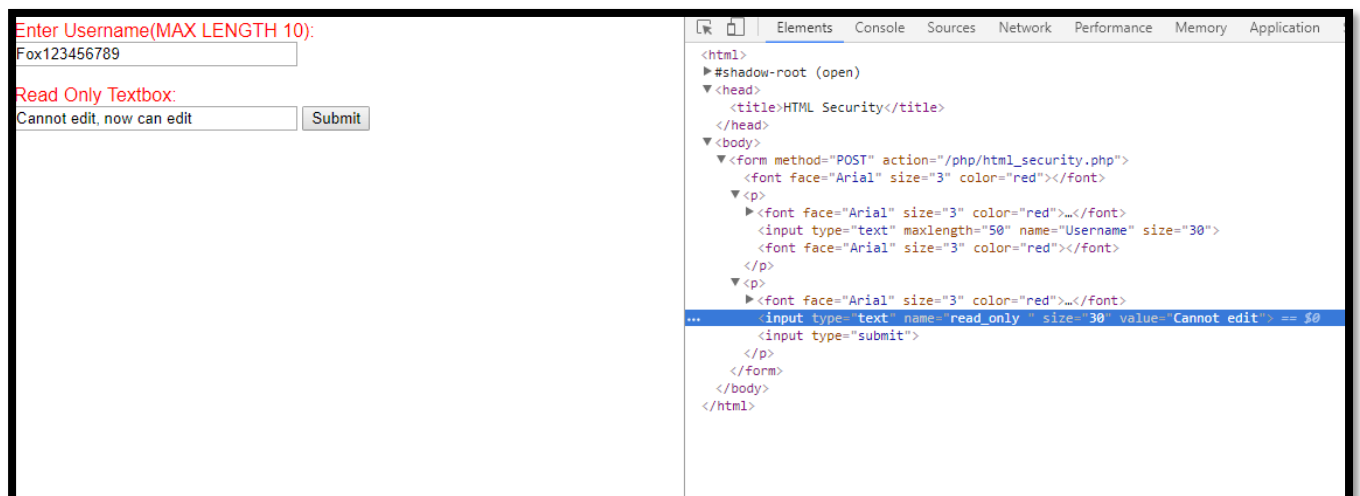


Enter Username(MAX LENGTH 10):
Fox1234567

Read Only Textbox:
Cannot edit

Submit

As mentioned, these are only basic forms of protection and offer no real form of protection against hackers. 'Maxlength', for example, can easily be bypassed in browsers like chrome by right clicking and selecting 'inspect element'. From here you can change the 'maxlength' attribute. It is also possible to bypass this restriction via the form's 'action' URL. This is the same case with 'readonly', as the HTML source can be edited within the browser to remove the attribute.



Enter Username(MAX LENGTH 10):
Fox123456789

Read Only Textbox:
Cannot edit, now can edit

Submit

Elements Console Sources Network Performance Memory Application

```
<html>
  <#shadow-root (open)>
    <head>
      <title>HTML Security</title>
    </head>
    <body>
      <form method="POST" action="/php/html_security.php">
        <font face="Arial" size="3" color="red"></font>
        <p>
          <font face="Arial" size="3" color="red"></font>
          <input type="text" maxlength="50" name="Username" size="30">
          <font face="Arial" size="3" color="red"></font>
        </p>
        <p>
          <font face="Arial" size="3" color="red"></font>
          <input type="text" name="read_only" size="30" value="Cannot edit">
          <input type="submit">
        </p>
      </form>
    </body>
  </html>
```

SECTION 4

Section 4 involved the use of 4 different PHP functions for the purposes of validation. To demonstrate this, I created a form asking for a first & second name, a preferred search engine URL, and a valid student number using a regular expression. The functions I used were:

Empty() – to determine if an input box is empty or not.

Preg_match() – used along with the regular expression to search for particular patterns in a string.

Filter_input() – in my script, was used to determine if the URL entered was a valid URL.

Htmlspecialchars() – used to convert predefined characters into HTML entities.

Below are examples of incorrect entries and correct entries:

Incorrect

First Name: * Name is required

Second Name: * Second name is required

Preferred Search Engine: * Invalid URL format

Student Number: * Number MUST be 8 digits

Input Details:

Please fill in all fields.

Correct

First Name: *

Second Name: *

Preferred Search Engine: *

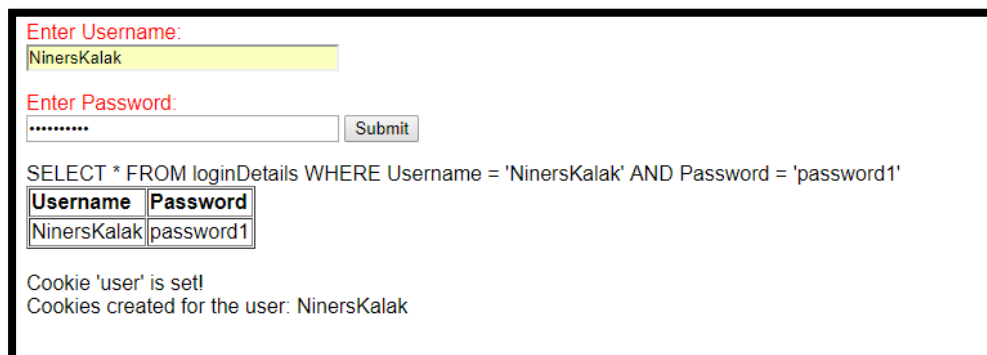
Student Number: *

Input Details:

Luke
Fox
https://www.google.ie/
20076173

SECTION 5

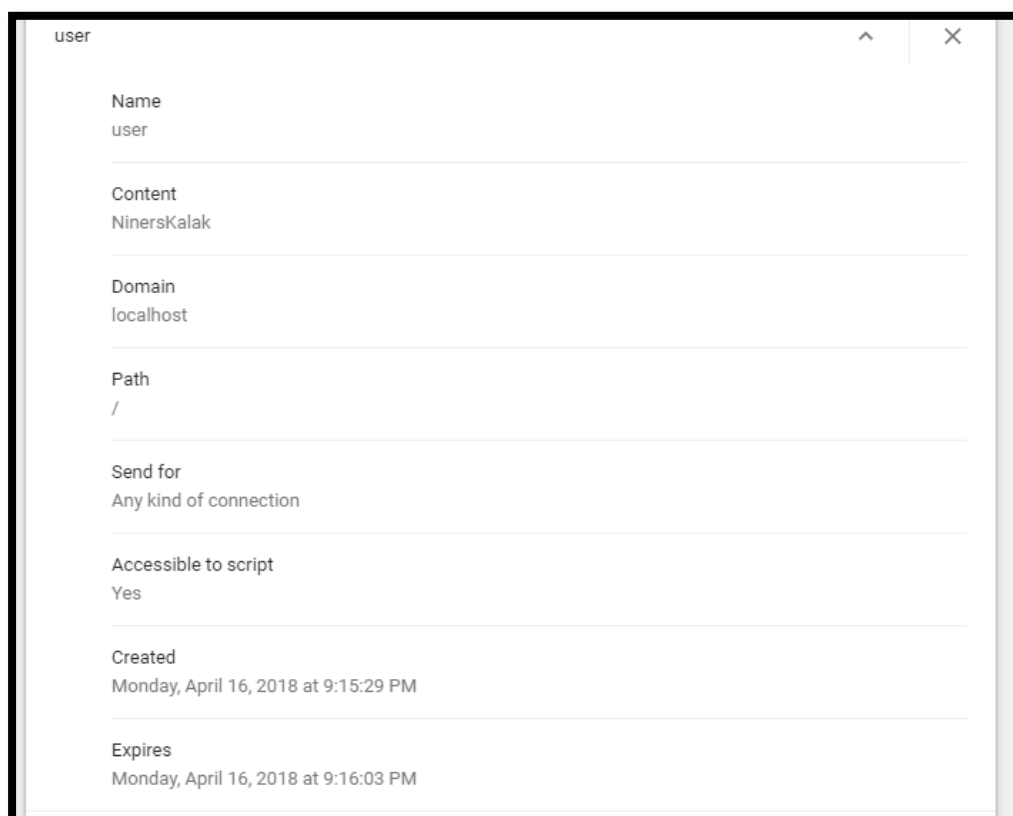
In section 5, I was required to create a script showing how cookies work. This script was presented in a login form similar to section 2, using the same database also. If the user entered a username and password matching an entry in the database, a cookie would be created with the value of the cookie being the matching username from the database. This cookie could be seen upon refreshing the login page, right up until it expires.



The screenshot shows a login form with two input fields: "Enter Username:" and "Enter Password:". The username field contains "NinersKalak" and the password field contains "password1". A "Submit" button is next to the password field. Below the form, the SQL query "SELECT * FROM loginDetails WHERE Username = 'NinersKalak' AND Password = 'password1'" is displayed. Underneath the query is a table with two columns, "Username" and "Password", containing the values "NinersKalak" and "password1" respectively. At the bottom, the text "Cookie 'user' is set!" and "Cookies created for the user: NinersKalak" is shown.

Username	Password
NinersKalak	password1

Cookie as seen in Chrome's listed cookies:



The screenshot shows a window titled "user" with a close button. It displays the details of a cookie named "user". The fields and their values are: Name: user, Content: NinersKalak, Domain: localhost, Path: /, Send for: Any kind of connection, Accessible to script: Yes, Created: Monday, April 16, 2018 at 9:15:29 PM, and Expires: Monday, April 16, 2018 at 9:16:03 PM.

Name	Content	Domain	Path	Send for	Accessible to script	Created	Expires
user	NinersKalak	localhost	/	Any kind of connection	Yes	Monday, April 16, 2018 at 9:15:29 PM	Monday, April 16, 2018 at 9:16:03 PM

There are a few options the user has at their disposal when keeping these cookies secure. For one, the user can specify the path that the cookie can exist in. This means the user can only send cookies to specific parts of their application. A '/' denotes the cookie can be requested anywhere throughout the application, where as changing it to, for example, '/login' means the cookie is limited to this specific page. Another method of keeping cookies secure is using the 'httponly' flag. This parameter ensures that JavaScript is unable to access the contents of the cookie – which can prevent exploits such as XSS as seen above.