TEXAS TECH UNIVERSITY

INTRODUCTION TO INFORMATION AND COMPUTER SECURITY

CS-5340

Instructor: Abdul Serwadda

LAB-2 Hands-On Project

Name: Manvitha Aathmuri
RNumber: R11847781

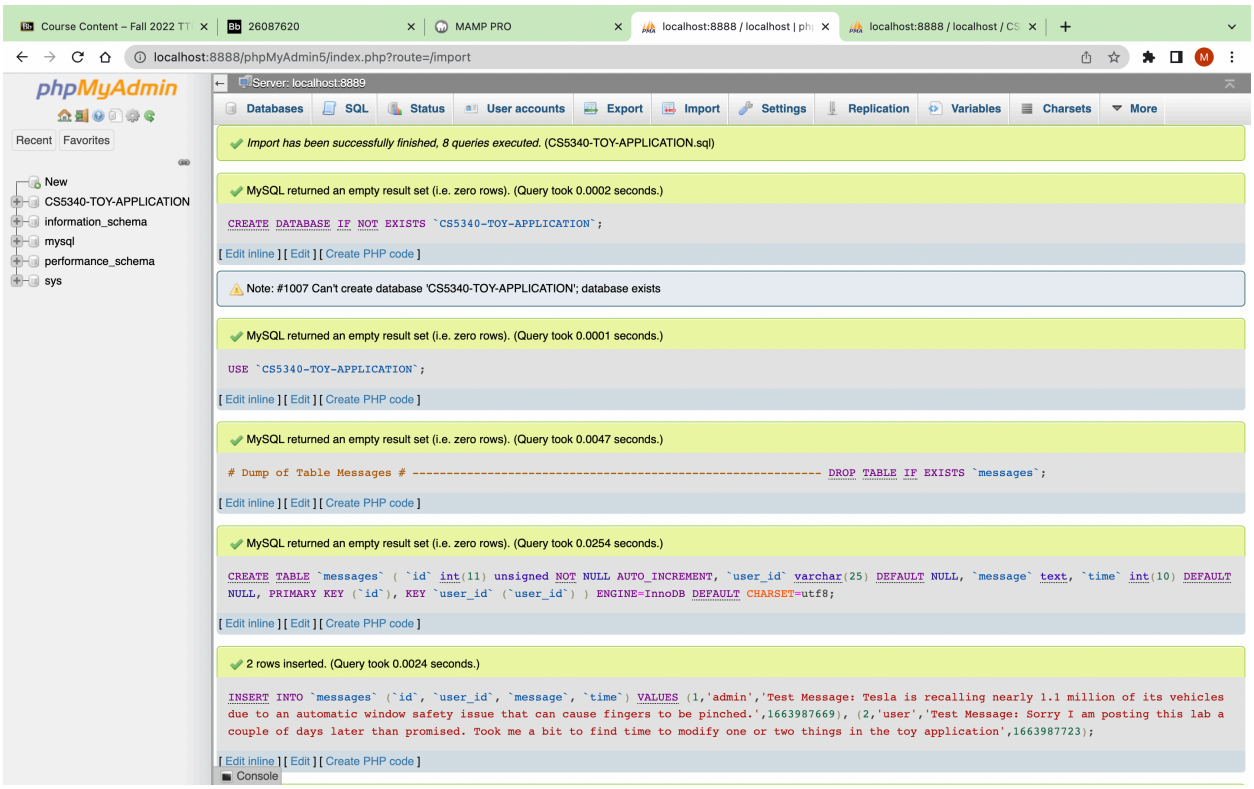
HANDS-ON PROJECT LAB ON WEB ATTACKS

LAB SETUP:

The server setup MAMP has been downloaded using https://www.mamp.info/en/
Once the given RAR file is downloaded, the project folder named “CS5340-TOY-APPLICATION-2022” is copy pasted in the localhost folder of the MAMP.
In order to run the given project in MAC, there are few modifications to be done as described below in the *functions.php* file.

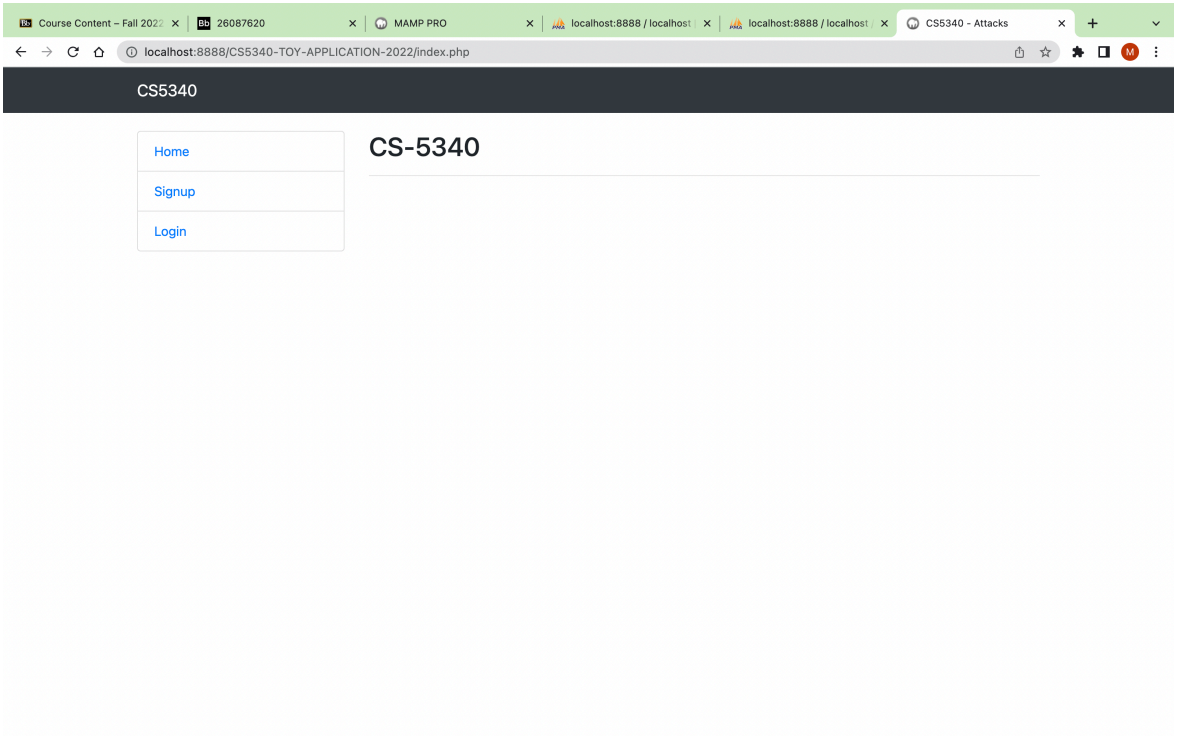
- Since the host extension in MAC is setup as localhost, $dbhost has been changed to **localhost** from **127.0.0.1**.
- MAMP server has a login password. In order to build the connection and run the application, all the four parameters such as host, name, user and password should match. Thus, the password ‘**root**’ has been added to the original PHP file.

Once the above-mentioned file is modified, the SQL file named CS5340-TOY-APPLICATION.sql has been imported on the MYSQL server.



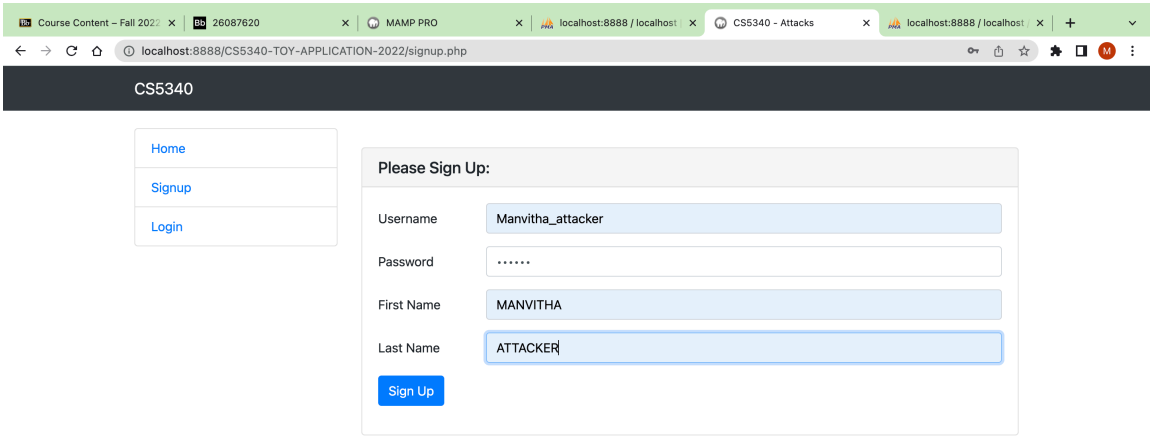
Browse http://localhost:8888/CS5340-TOY-APPLICATION-2022/ to run the web application.

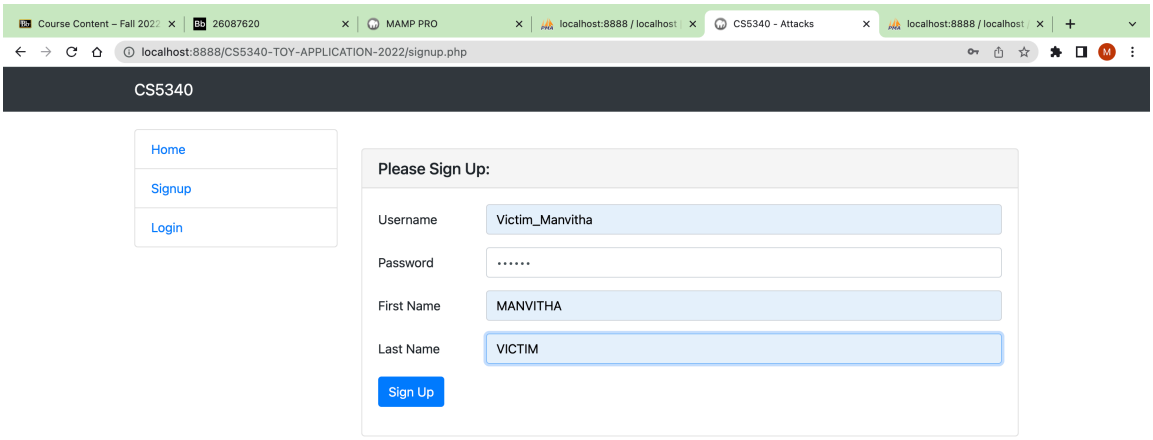
Below is the screen once the application is up and running.



Now sign up on two accounts, one for the attacker and the other for the victim to perform the tasks which reveals the vulnerabilities of this web application.

In this project, below are the users for attacker and victim
Attacker: Manvitha_attacker
Victim: Victim_Manvitha





SECTION 1 – XSS ATTACKS

Cross-Site Scripting (XSS) attacks are a type of injection, in which malicious scripts are injected into trusted websites. XSS attacks occur when an attacker uses a web application to send malicious code, generally in the form of a browser side script to a different end user.

Non-Persistent XSS:

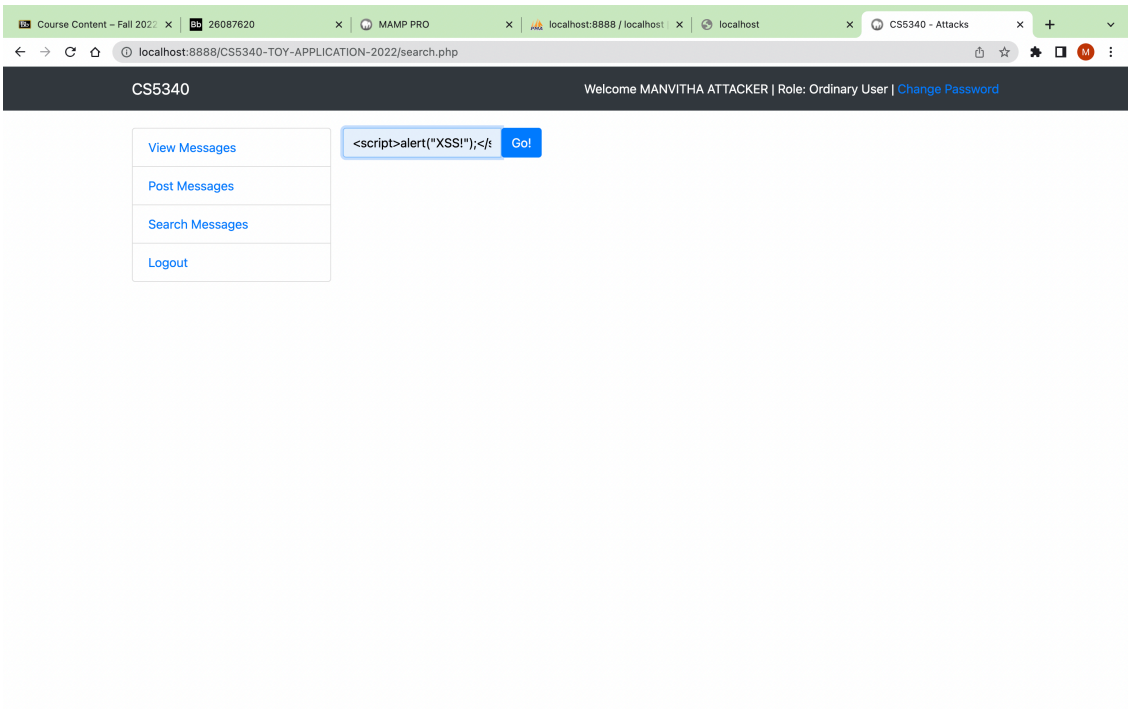
Non-Persistent cross-site scripting or non-persistent XSS is also known as Reflected XSS. It is one of the major categories of XSS attacks where malicious code is executed by the victim's browser, and the payload is not stored anywhere. Instead, it is returned as part of the HTML response that the server sends. Therefore, the victim is being tricked into sending malicious code to the vulnerable web application, which is then reflected to the victim's browser where the XSS payload executes.

In the similar way we are going to check one of the simplest XSS attack which displays alert window.

Task 1: XSS Vulnerability Check:

- Log into the application with any one of the usernames and click on Search Messages.
- Once the page is opened, enter the below piece of script in the search box and hit Go!

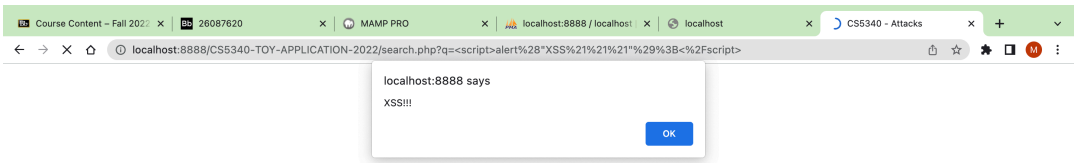
<script>alert("XSS!!!");</script>



Observation:

The above piece of script displays an alert window with the text mentioned inside the parenthesis.

This is because the input given in the search box is being sent as a HTTP request to the server using GET parameter. Once the server processes the information, it returns a response. The browser detects the response as HTML text according to the content-type mentioned and provides the result. So, when the alert script is given in the search box, browser detects it as HTML code and runs which in return executes the java script code. Thus, an alert pops out.



Task 2: Defacing a webpage using JavaScript DOM API

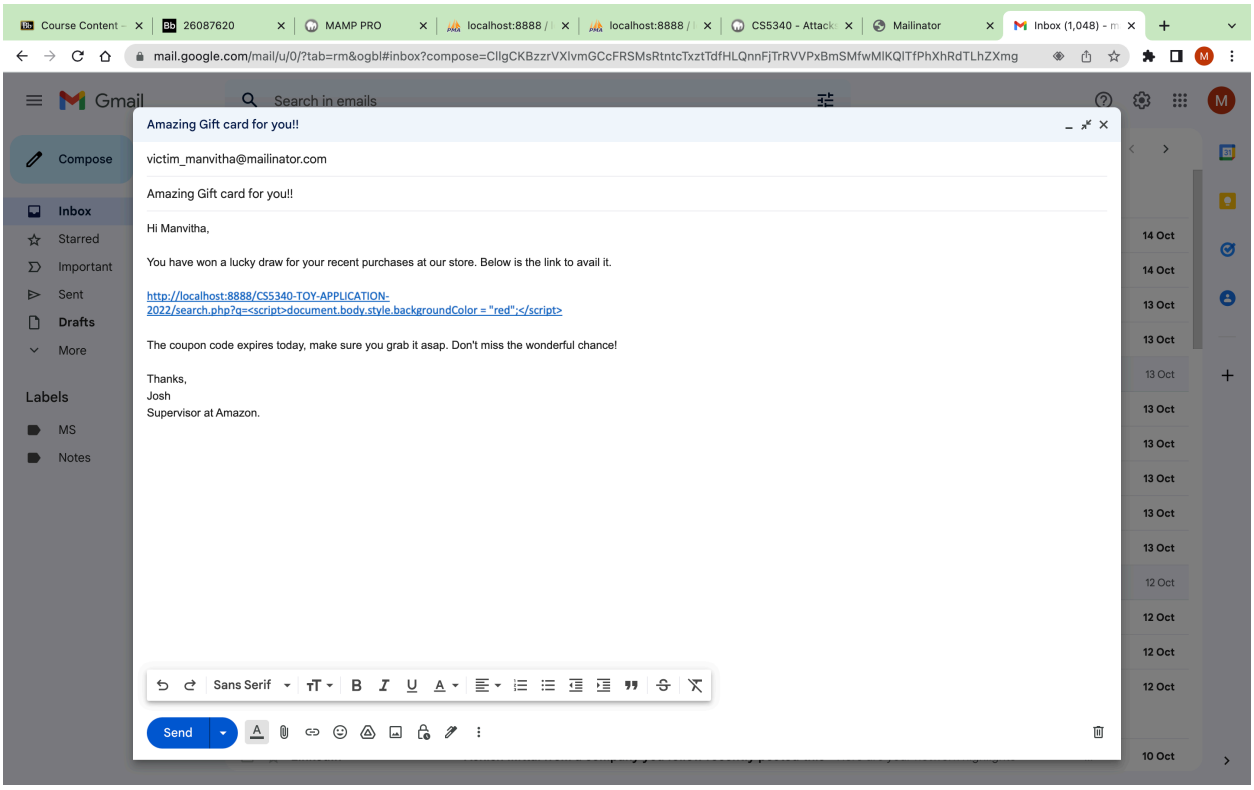
1) Background color change operation

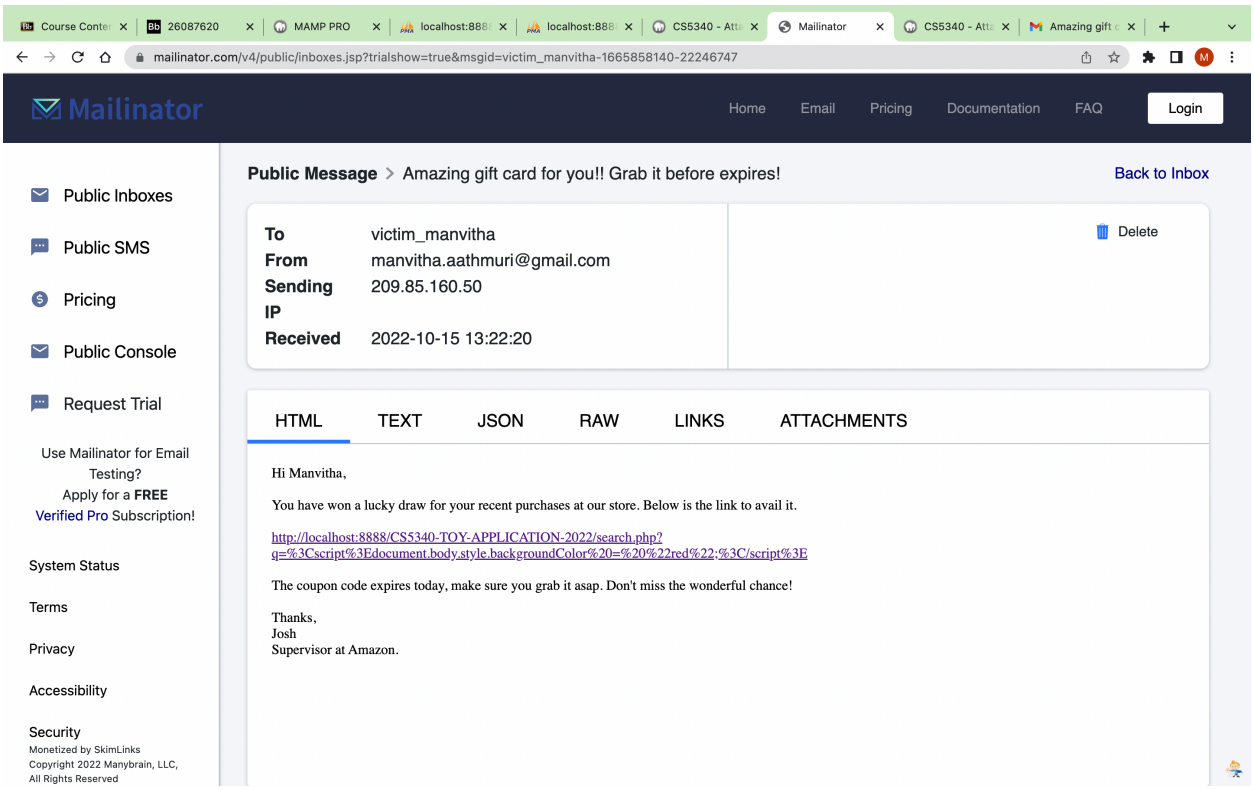
Given format of the link- http://localhost/CS5340-TOY-APPLICATION-2022/search.php?q=<script>unknown1=”unknown-2”;</script>

In order to change the background color of the web page, the pace holders unknown 1 and 2 are replaced by following script,
Unknown1: document.body.style.backgroundColor
Unknown2: red

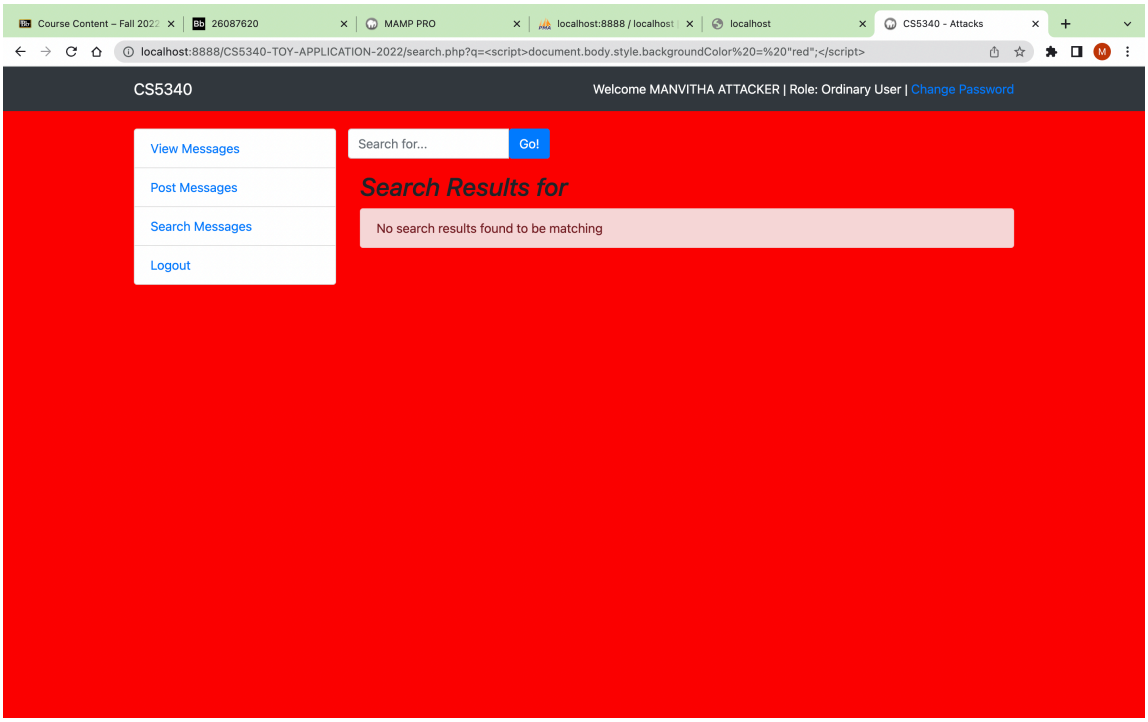
http://localhost:8888/CS5340-TOY-APPLICATION-2022/search.php?q=<script>document.body.style.backgroundColor = "red";</script>

This malicious link is being sent by the attacker to the victim via email.





Once the victim clicks on the link, the web page turns out to Red in color.



2) Background picture change operation

Given format of the link- http://localhost/CS5340-TOY-APPLICATION-2022/search.php?q=<script>unknown1=”unknown-2”;</script>

Let the image from the below link be used as the background image of the web page.

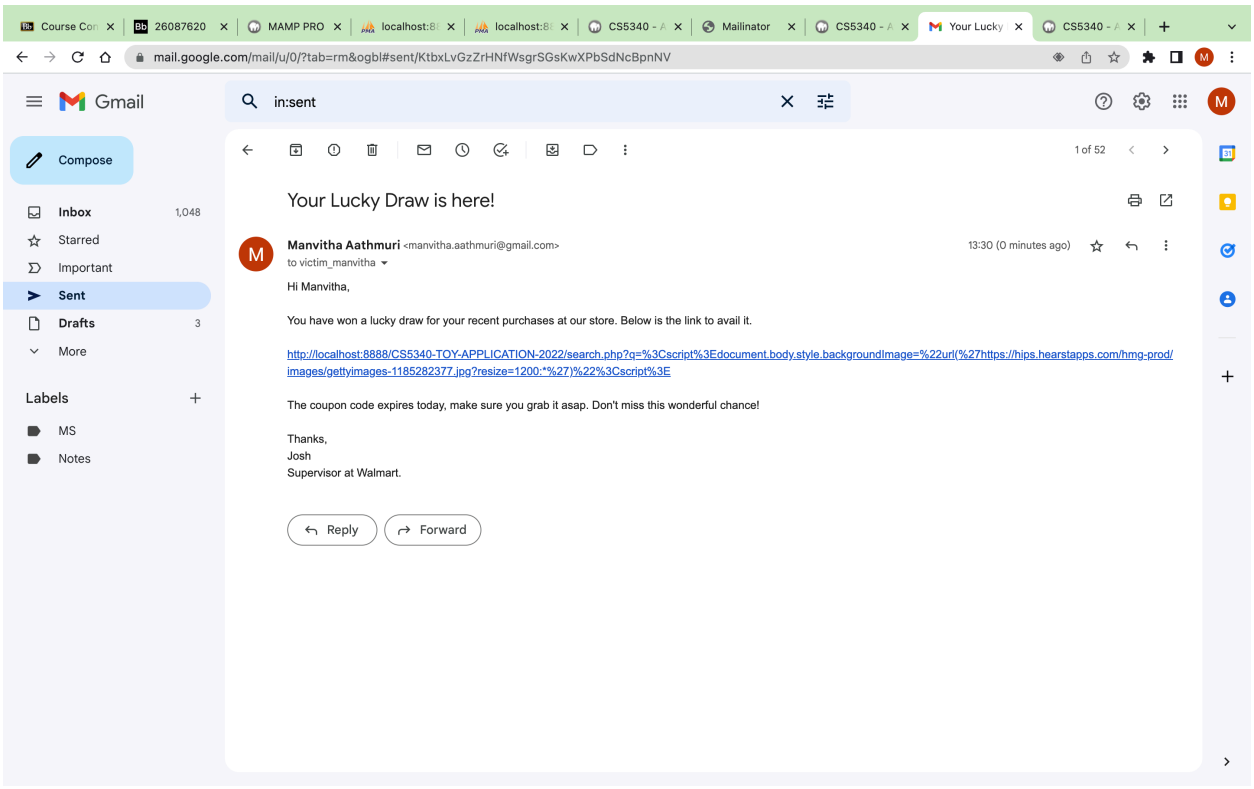
https://hips.hearstapps.com/hmg-prod/images/gettyimages-1185282377.jpg?resize=1200:*

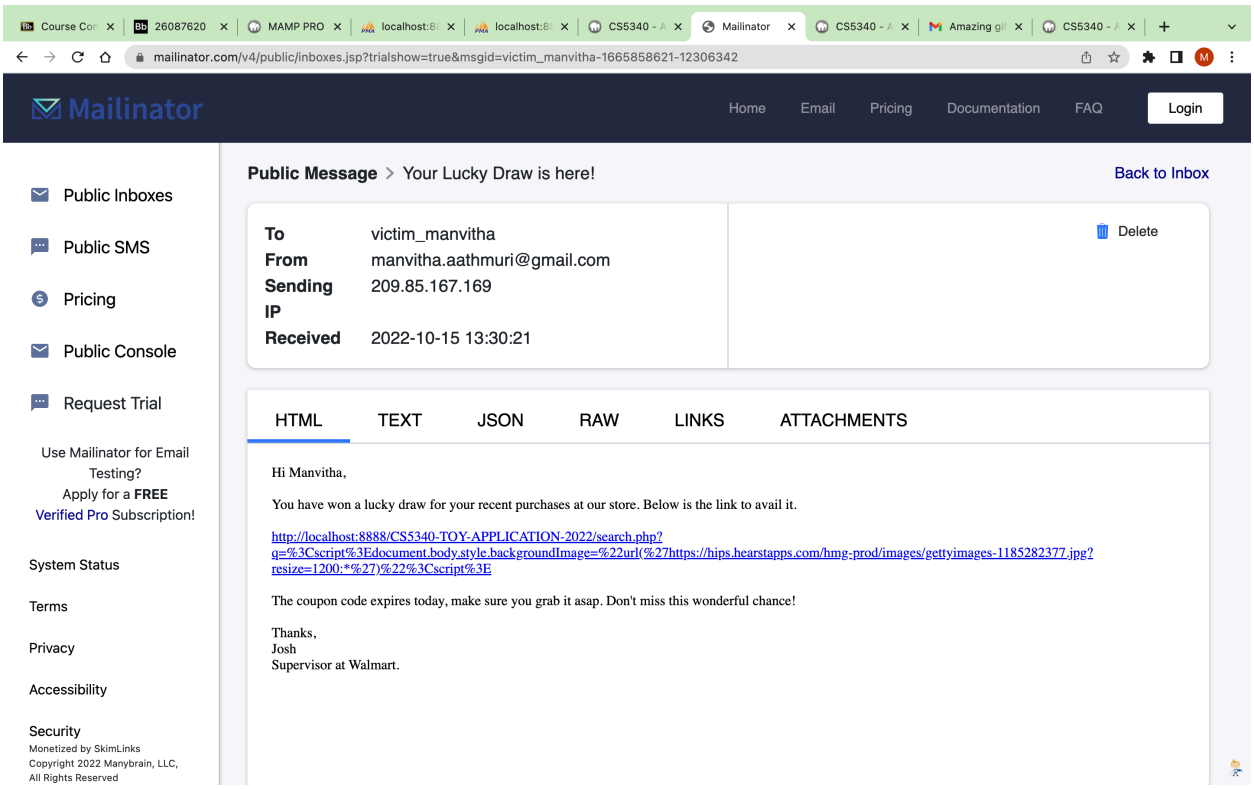
In order to change the background image of the web page, the place holders unknown 1 and 2 are replaced by following script,

Unknown1: document.body.style.backgroundImage
Unknown2: “url(‘https://hips.hearstapps.com/hmg-prod/images/gettyimages-1185282377.jpg?resize=1200:*’)”

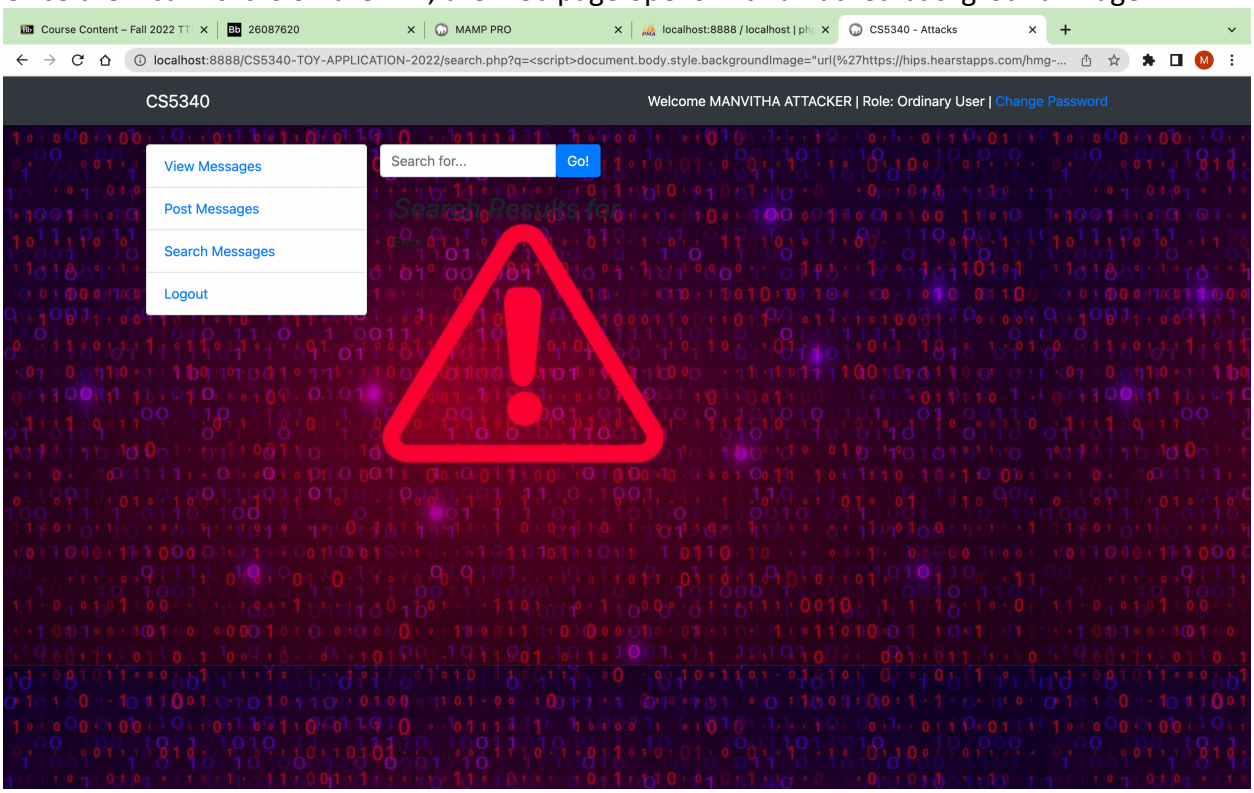
http://localhost:8888/CS5340-TOY-APPLICATION-2022/search.php?q=<script>document.body.style.backgroundImage="url(%27https://hips.hearstapps.com/hmg-prod/images/gettyimages-1185282377.jpg?resize=1200:*%27)"<script>

The above malicious link is being sent to victim by the attacker via email.





Once the victim clicks on the link, the web page opens with a hacked background image.

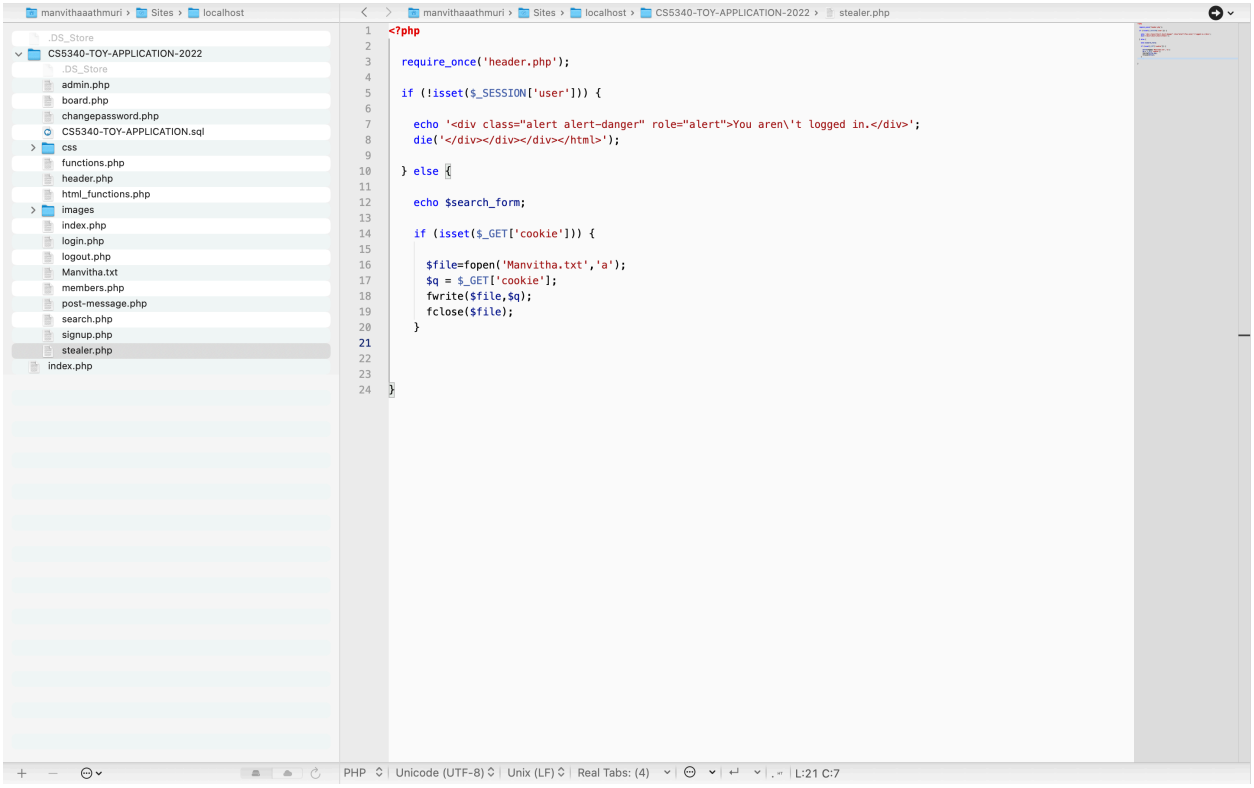


Persistent XSS:

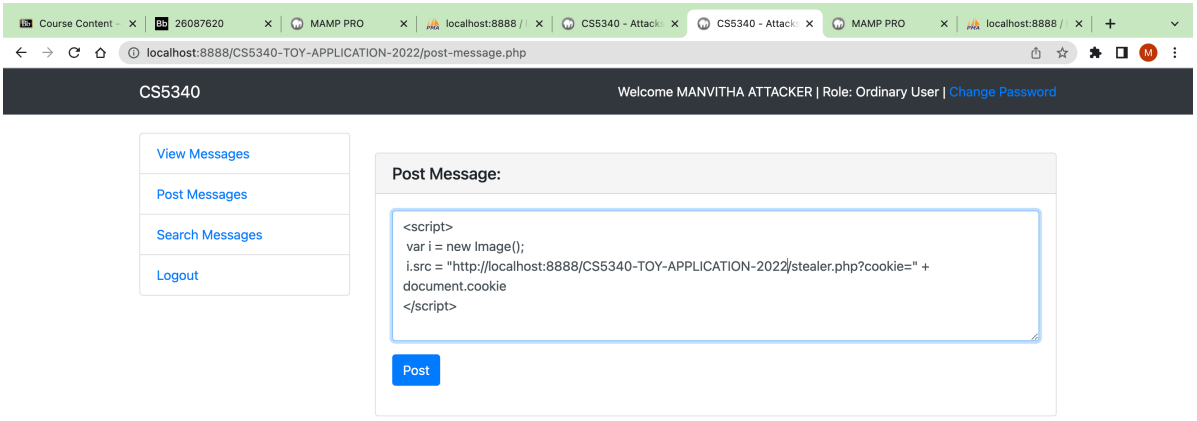
A Persistent XSS attack is possible when a website or web application stores user input and later serves it to other users. An application is vulnerable if it does not validate user input before storing content and embedding it into HTML response pages. Attackers use vulnerable web pages to inject malicious code and have it stored on the web server for later use. The payload is automatically served to users who browse web pages and executed in their context. Thus, the victims do not need to click on a malicious link to run the payload (as in the case of Non-Persistent XSS). All they must do is visit a vulnerable web page.

Task 1: Stealing user's credentials e.g., cookies

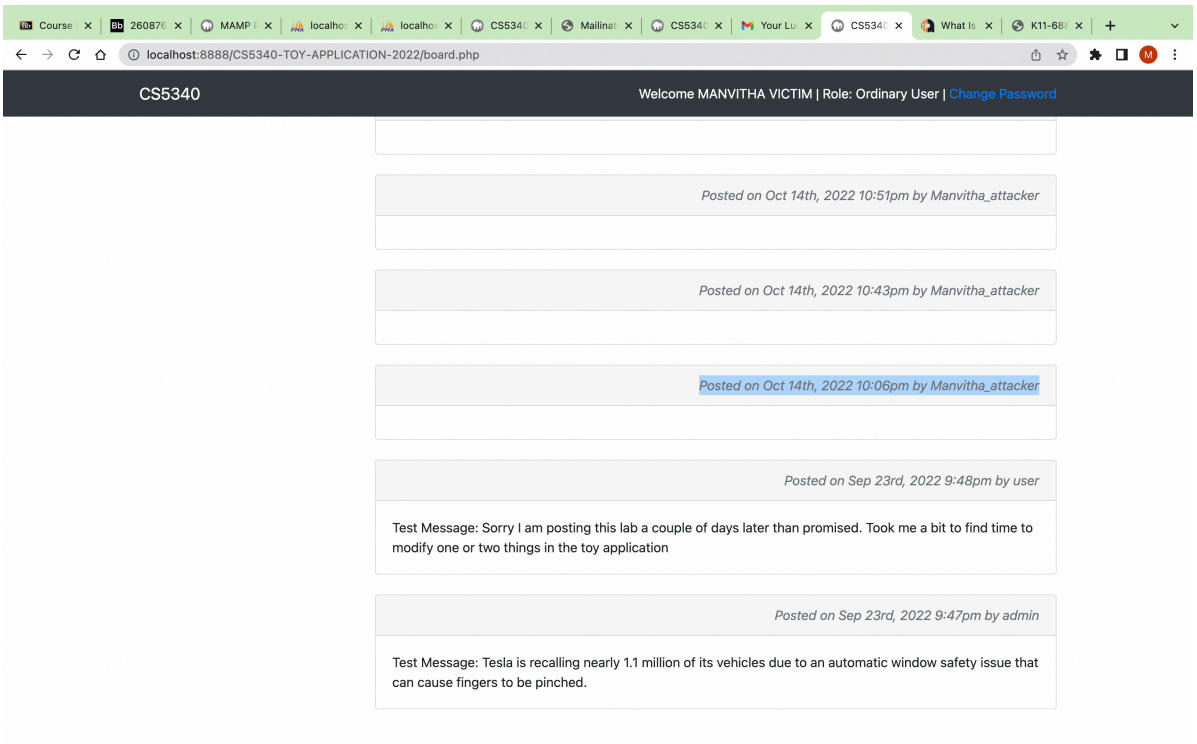
In order to steal the user's credentials, a PHP file named *stealer.php* is created to access the cookies which uses the HTTP GET variable.



Once the above file is uploaded to server, Log into the web application as Attacker and post the malicious script shown in the below screenshot at Post message input box.



Log into the application as Victim and navigate to View messages page. It is seen that an unknown message has been posted by the attacker.



This creates an open text file at the file directory of the web application with the credentials of Victim such as **user id** and the **password hash** that is being created for password.

Manvitha.txt

```
user_details=Manvitha_attacker; PHPSESSID=grc24o8odj8fh9k593utphnb91
```

Observation:

The attacker has stolen the credentials in the form of cookies. Thus, every time someone logins into the web application, the credentials are getting displayed in the same text file as shown below.

Manvitha.txt

```
PHPSESSID=iemjve4mra8fke204rutt4talauser_details=Victim_Manvitha;
PHPSESSID=iemjve4mra8fke204rutt4talauser_details=Victim_Manvitha;
PHPSESSID=iemjve4mra8fke204rutt4talauser_details=Victim_Manvitha;
PHPSESSID=iemjve4mra8fke204rutt4talauser_details=Victim_Manvitha;
PHPSESSID=iemjve4mra8fke204rutt4talauser_details=Victim_Manvitha;
PHPSESSID=iemjve4mra8fke204rutt4talauser_details=Victim_Manvitha;
PHPSESSID=iemjve4mra8fke204rutt4talauser_details=Victim_Manvitha;
PHPSESSID=iemjve4mra8fke204rutt4talauser_details=Victim_Manvitha;
PHPSESSID=iemjve4mra8fke204rutt4talauser_details=Victim_Manvitha;
PHPSESSID=iemjve4mra8fke204rutt4talauser_details=Victim_Manvitha;
PHPSESSID=iemjve4mra8fke204rutt4talauser_details=Victim_Manvitha;
PHPSESSID=iemjve4mra8fke204rutt4talauser_details=Manvitha_attacker;
PHPSESSID=45r3ef6qjokiqi81jhtu6tgsiauser_details=Manvitha_attacker;
PHPSESSID=45r3ef6qjokiqi81jhtu6tgsiauser_details=Manvitha_attacker;
PHPSESSID=45r3ef6qjokiqi81jhtu6tgsiauser_details=Victim_Manvitha;
PHPSESSID=iemjve4mra8fke204rutt4talauser_details=Victim_Manvitha;
PHPSESSID=iemjve4mra8fke204rutt4talauser_details=Victim_Manvitha;
PHPSESSID=iemjve4mra8fke204rutt4talauser_details=Victim_Manvitha;
PHPSESSID=iemjve4mra8fke204rutt4talauser_details=Victim_Manvitha;
PHPSESSID=iemjve4mra8fke204rutt4talauser_details=Victim_Manvitha;
PHPSESSID=iemjve4mra8fke204rutt4talauser_details=Victim_Manvitha;
PHPSESSID=iemjve4mra8fke204rutt4talauser_details=Victim_Manvitha;
PHPSESSID=iemjve4mra8fke204rutt4talauser_details=Victim_Manvitha;
PHPSESSID=iemjve4mra8fke204rutt4talauser_details=Victim_Manvitha;
PHPSESSID=iemjve4mra8fke204rutt4talauser_details=Victim_Manvitha;
PHPSESSID=iemjve4mra8fke204rutt4talauser_details=Manvitha_attacker;
PHPSESSID=sbgb97ds261euj51p5g1ur3qu4user_details=Victim_Manvitha;
PHPSESSID=r27mu4qadt3ae2clssjb3ur9dkuser_details=Manvitha_attacker;
PHPSESSID=dmfjuho52o3qlul0t31uks78h3user_details=Victim_Manvitha;
```

Task 2: Illegitimately posting messages on behalf of the user

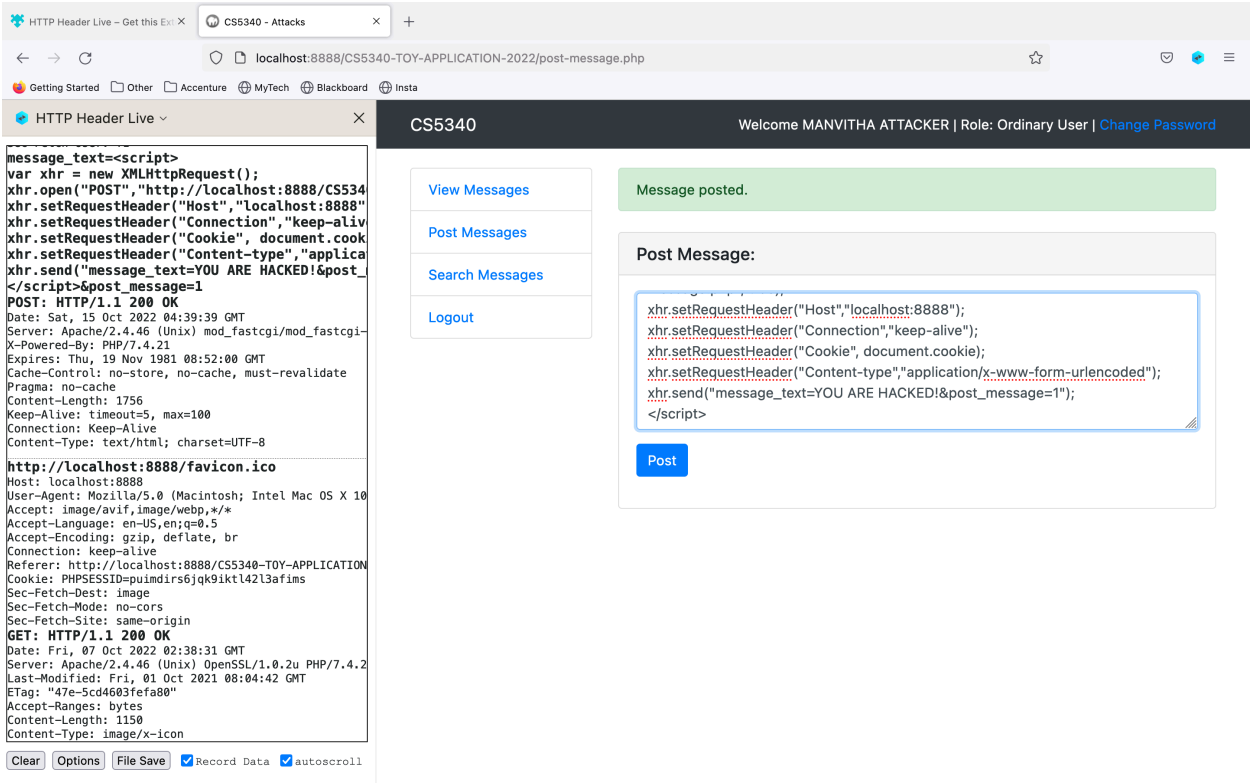
To post the messages on behalf of user without their consent, below steps are to be followed. Using the link: https://addons.mozilla.org/en-US/firefox/addon/http-header-live/ install the LiveHTTPHeaders extension to the Firefox browser.

Once done, Login as attacker and post any message on Post messages web page.
Now expand the POST in the LiveHTTPHeader session and collect the data such as Host, Connection, Content-type.

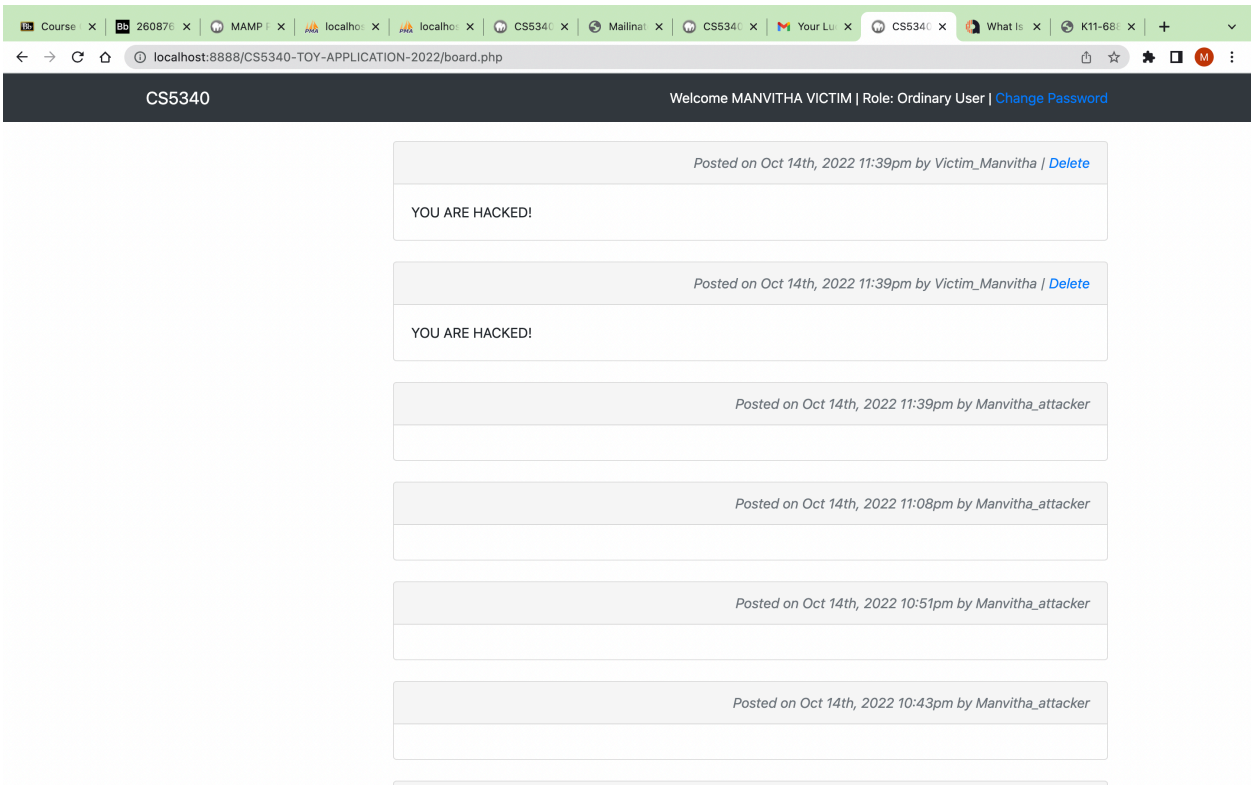
Now to post the illegitimate message, post the below malicious script in the Post message tab.

```
<script>
var xhr = new XMLHttpRequest();
xhr.open("POST","unknown-1", true);
xhr.setRequestHeader("Host","unknown-2");
xhr.setRequestHeader("Connection","unknown-3");
xhr.setRequestHeader("Cookie", document.cookie);
xhr.setRequestHeader("Content-type","unknown-4");
xhr.send("message_text=unknown-5&post_message=1");
</script>
```

The placeholders unknown-1,2,3,4,5 are taken from the above LiveHTTPHeader and replaced as shown below,
Unknown-1: http://localhost/CS5340-TOY-APPLICATION-2022/post-message.php
Unknown-2: localhost
Unknown-3: keep-alive
Unknown-4: application/x-www-form-urlencoded
Unknown-5: YOU ARE HACKED!

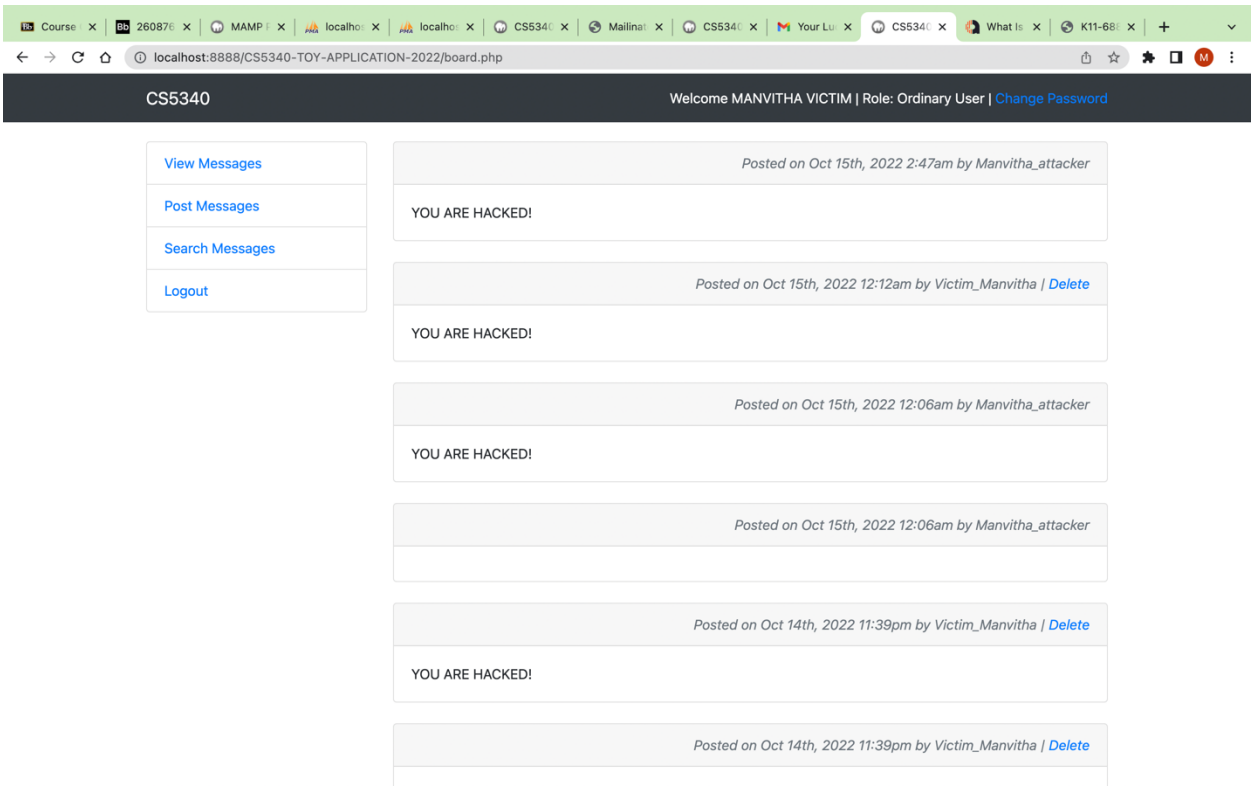


Now Login to the web application as Victim and click on View message and Boom! an unknown message is being posted without the consent of user.



Observation:

Every time the user clicks on View message, the illegitimate message pops up as shown below.

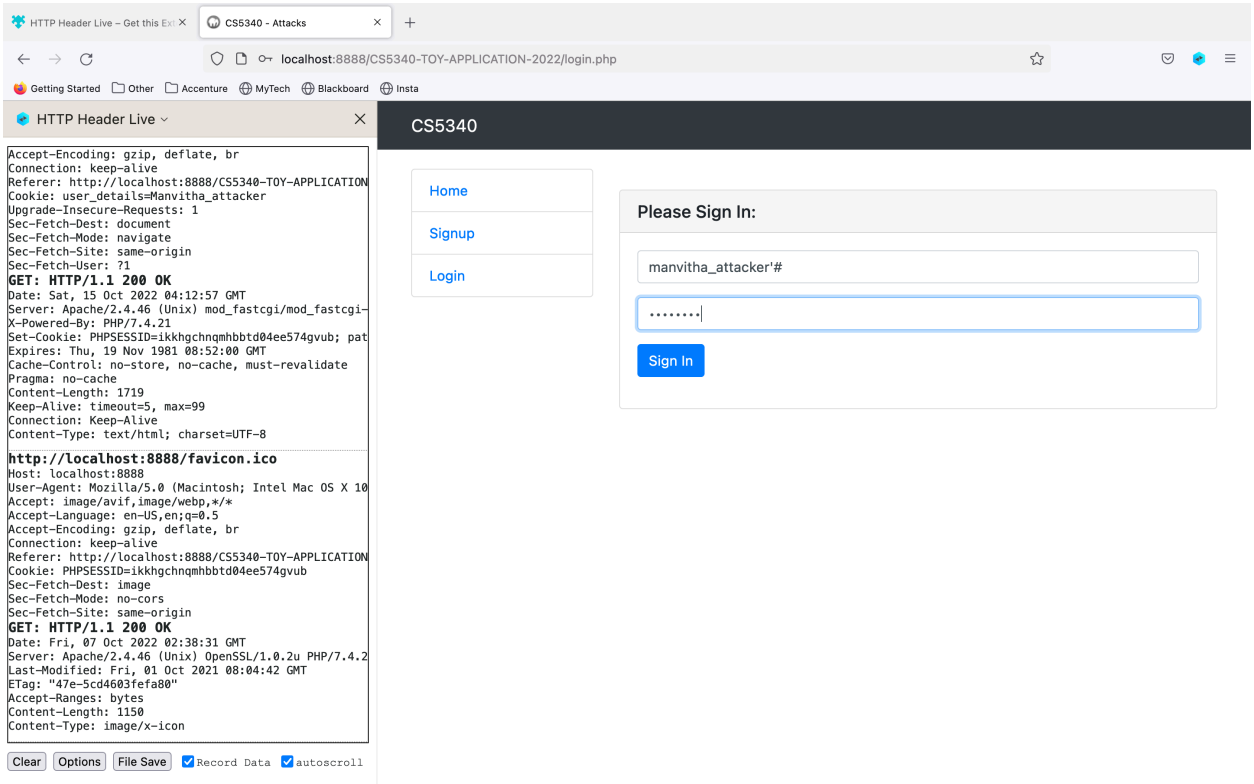


SECTION 2 – SQL INJECTION ATTACKS

SQL injection (SQLi) is a web security vulnerability that allows an attacker to interfere with the queries that an application makes to its database. It generally allows an attacker to view data that they are not normally able to retrieve. This might include data belonging to other users, or any other data that the application itself is able to access. In many cases, an attacker can modify or delete this data, causing persistent changes to the application's content or behavior.

Task 1: Log into the web application without correct password

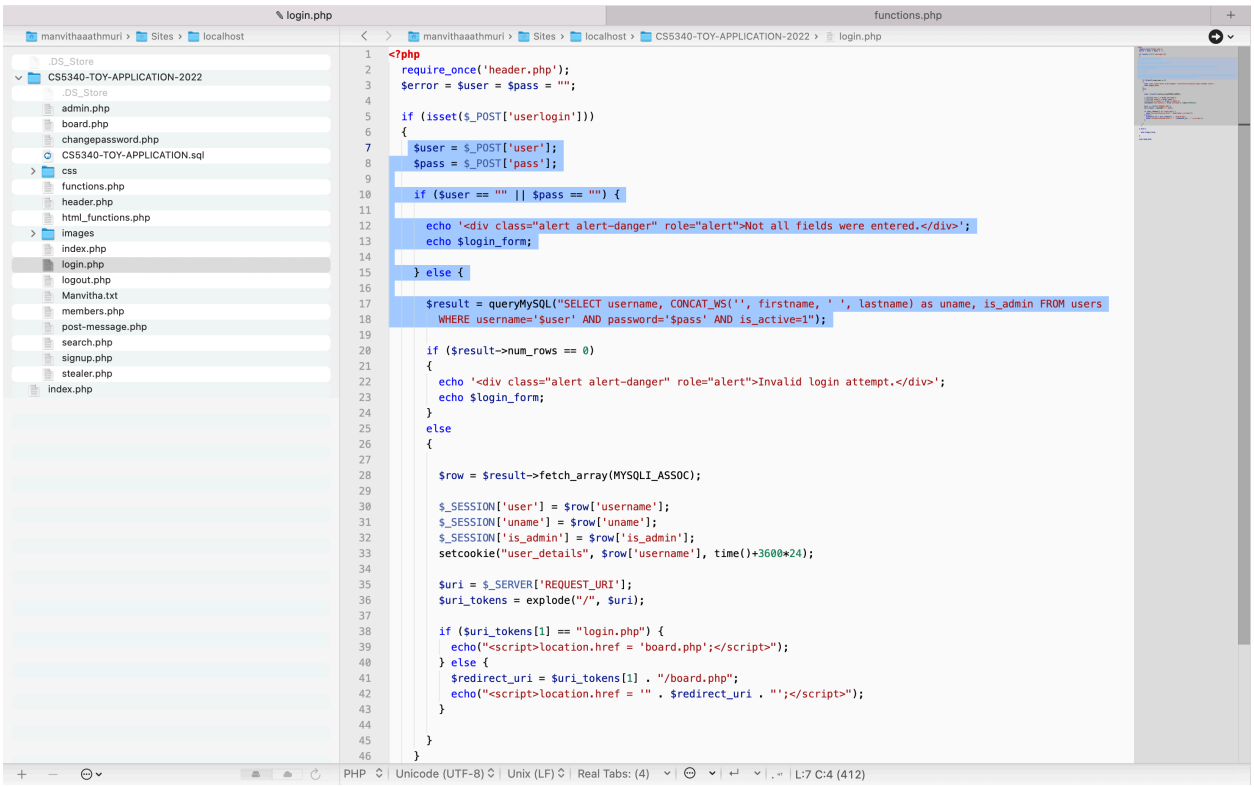
Login the web application as the attacker such that add the string ***'#*** to the username and sign in with any random password.



This allows the attacker to login the application with incorrect password i.e, any random password with the SQL injection that we have done.

The *login.php* file checks for the user and password but there is no defense check parameter to protect from SQL injection.

The code to fix this hack has been discussed below in the Defence mechanism section.

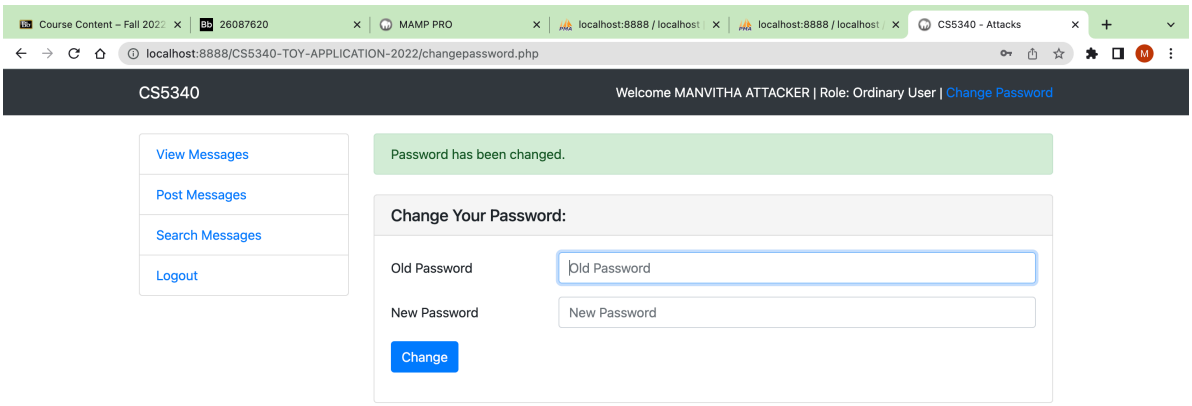


Task 2: Elevate user's account privileges

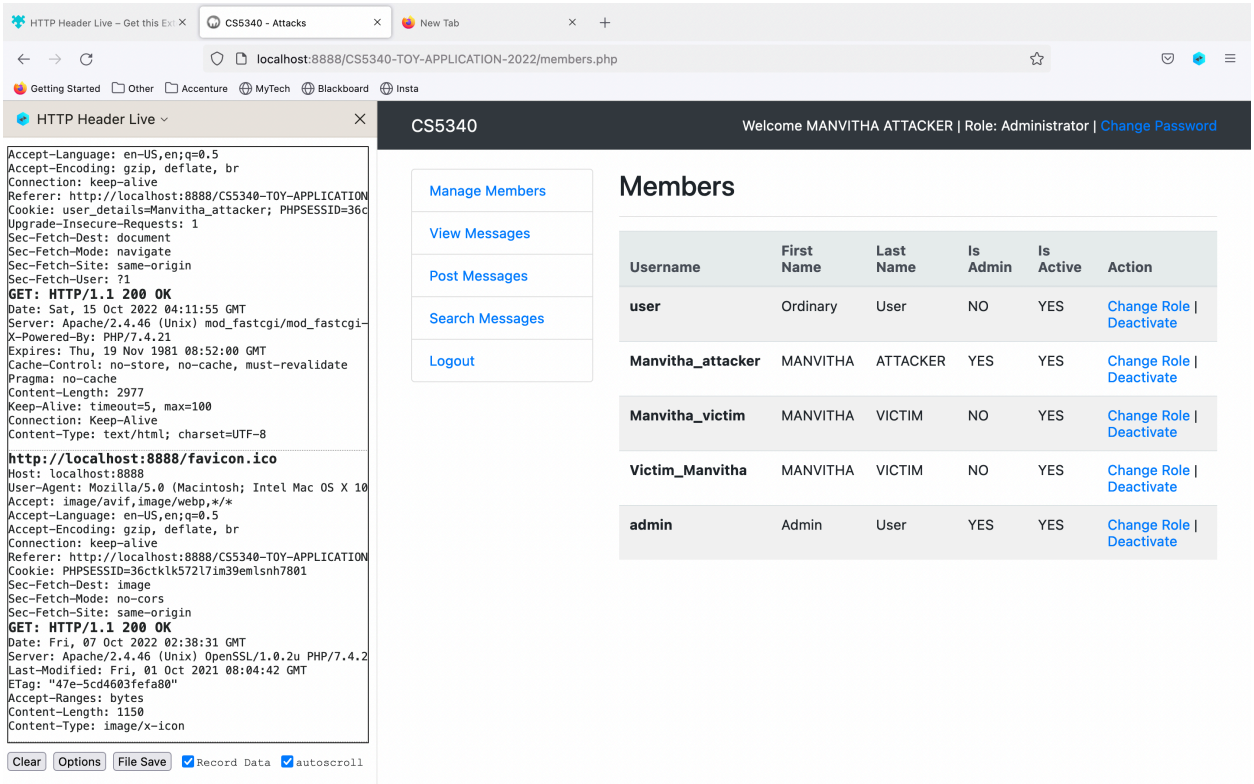
This SQL injection is a critical one where the attacker gets the privilege of Admin access.

Login as any ordinary user, here I have logged in with Attacker credentials since it is an ordinary user for now.

Now click on Change password and give the old password. For the new password, enter any new password and concatenate with the string ***',is_admin='1***

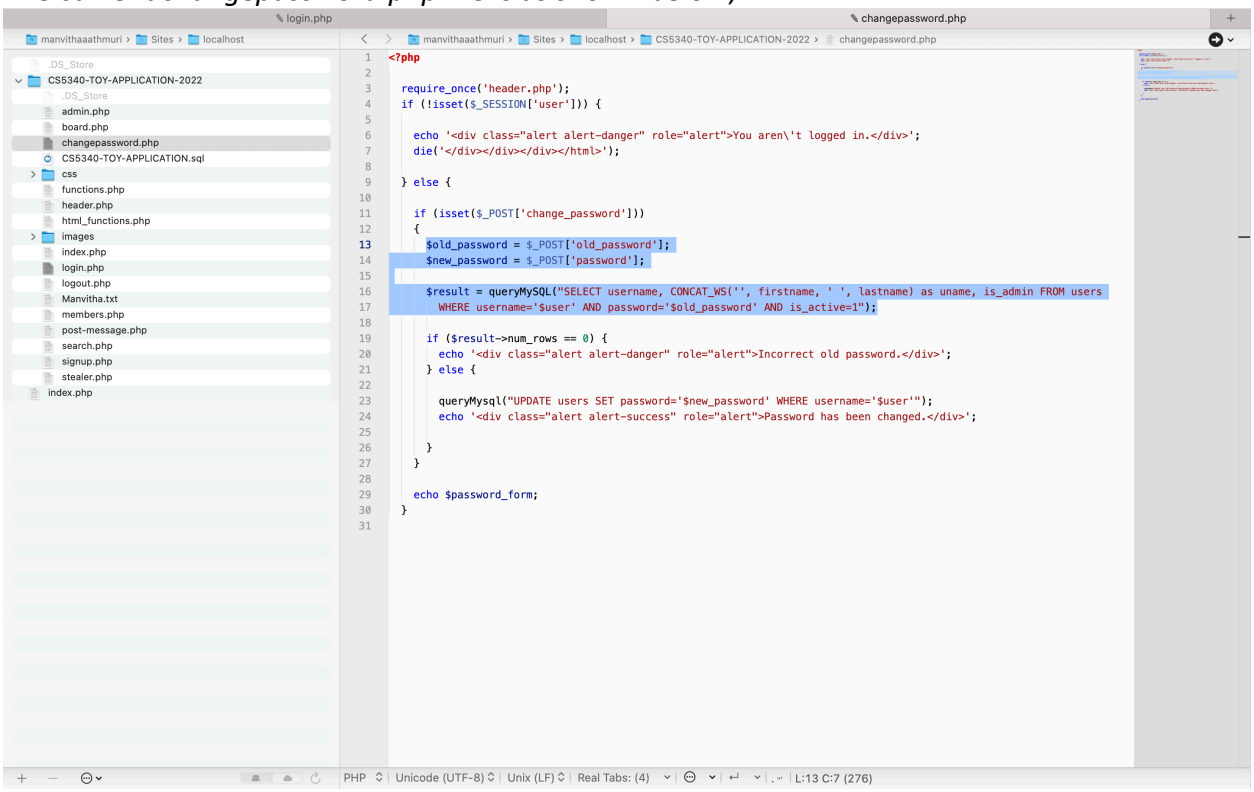


Once password is changed, logout and login with the new password created without string concatenation and Tada! The attacker's role has been changed from ordinary user to Administrator where he can manage the access of all the other members of this application.



The attacker with Admin access can perform functionalities like deactivate/ change the roles of other members of application.

The current *changepassword.php* file is as shown below,



The code fix on this vulnerability has been explained in the below Defence mechanism section.

SECTION 3 – DEFENCE MECHANISMS

a)

i. Provide a description of the various defence mechanisms that can help mitigate the attacks showcased in this lab. Describe in detail the core ideas behind each of the defense mechanisms you describe.

One of the crucial logics that was missing in the present web application was Input Validation and SQL parameter check. These fixes are discussed in detail with relevant attack mitigation screenshots going further.

ii. Why do these attacks continue to plague today's web applications despite the existence of the defense mechanisms that you describe above?

For many companies' security has no priority until they encounter a breach. The primary functionality to be taken care of a company's security is Authentication and Authorization.

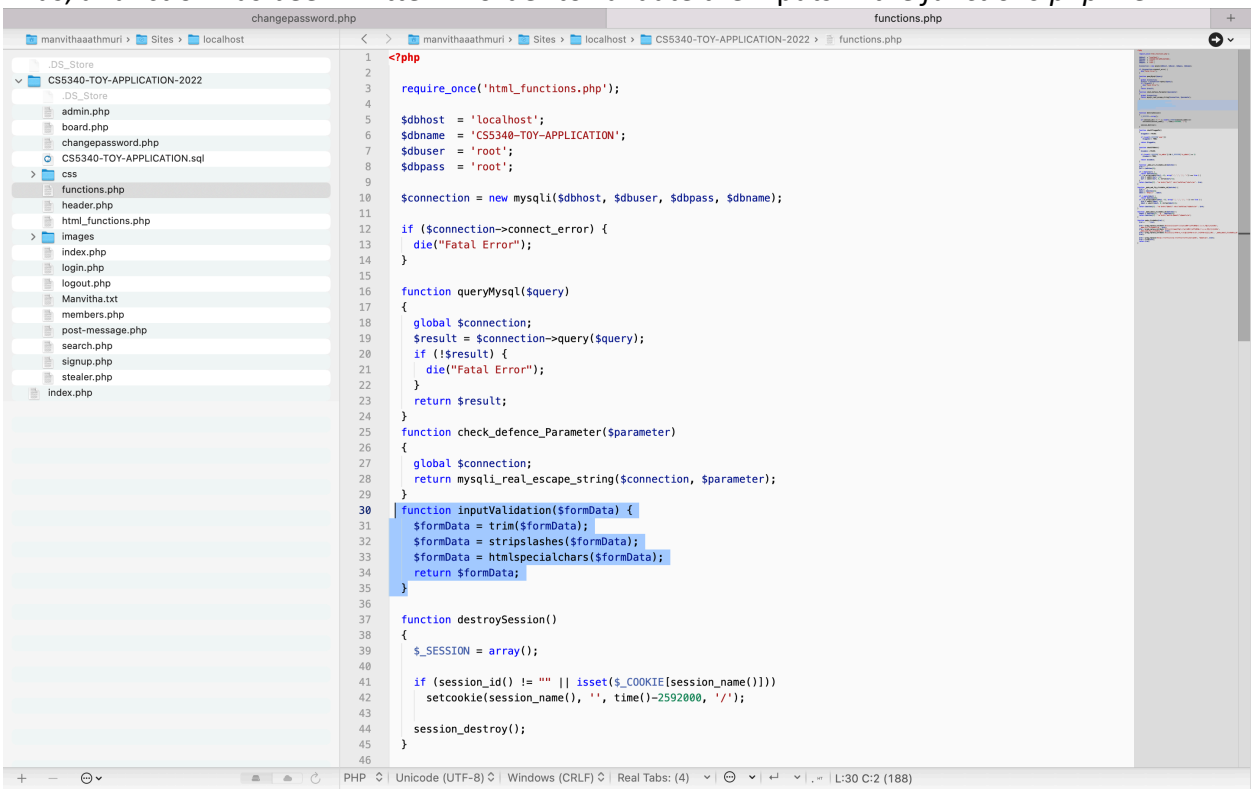
- Authentication means verifying that a user is the person they say they are.
- Authorization is nothing but providing permission or granting access to a specific resource to perform a particular action.

Anything that our application receives from an untrusted source must be filtered.

b) Modify the application to ensure that it is immune to these attacks. Using screen shots as appropriate, demonstrate that the application cannot fall to these attacks.

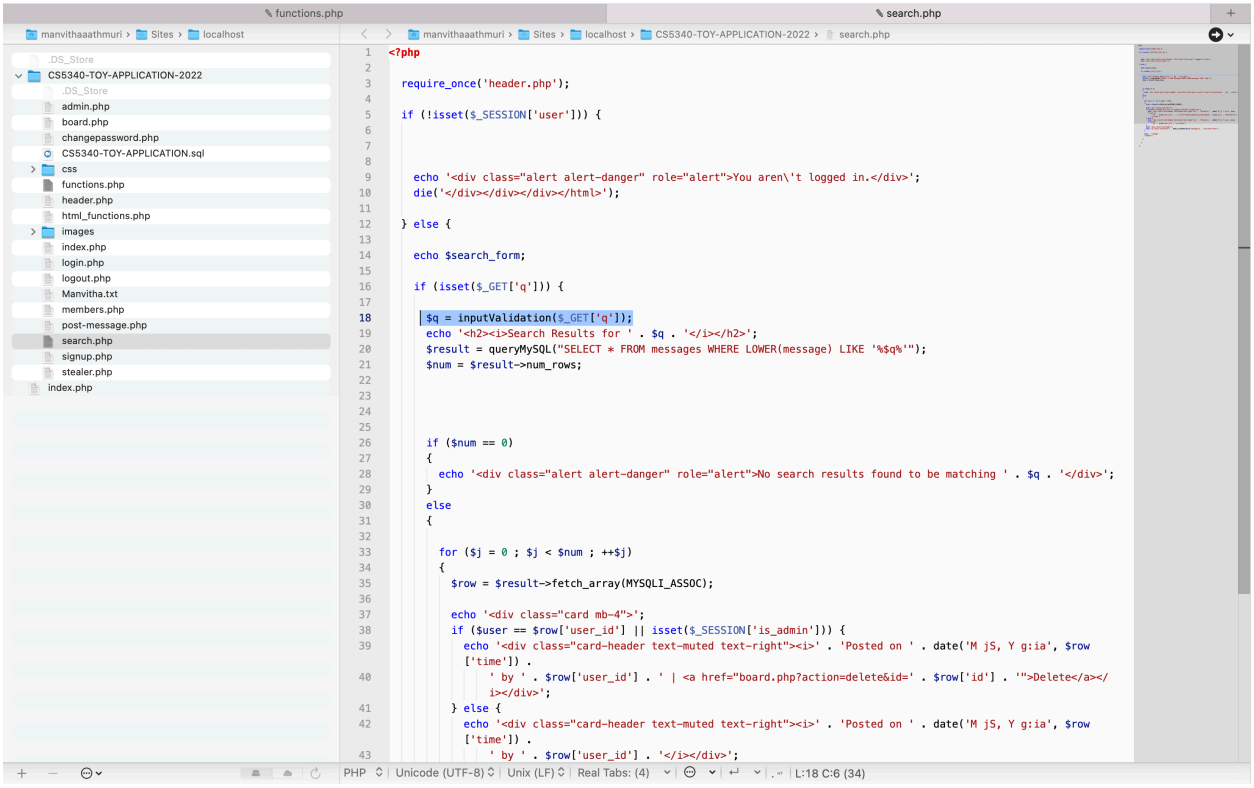
Prevention from an XSS attack or HTML injection is by not returning the HTML tags to the client. As there was no input field validation, the server processes information and provides the response. According to the mentioned content-type, the browser detects the input and executes the server's response i.e, either script or HTML tag (i.e, image injection etc) without proper validation.

Thus, a function has been written in order to validate the inputs in the *functions.php* file.

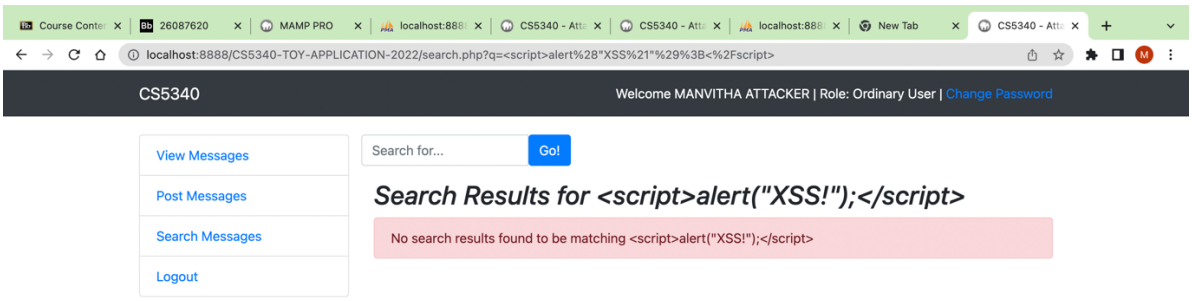


This function **inputValidation** has been called in *search.php* file to validate the inputs so that it strips away HTML tags or expressions such as **<** and **>**. But it is dangerous because some

browsers may not interpret broken HTML so, we are converting all the characters to their escaped counterparts. Thus, any malicious script would not be executed.



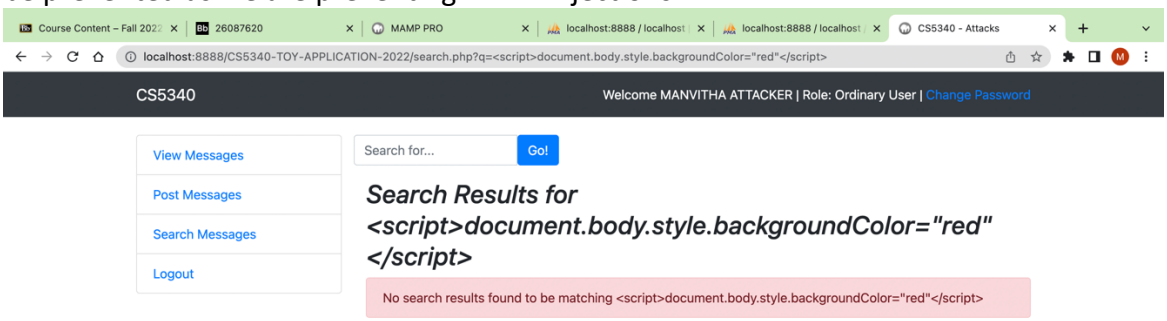
Let us test the task 1 where we have carried out an XSS alert.

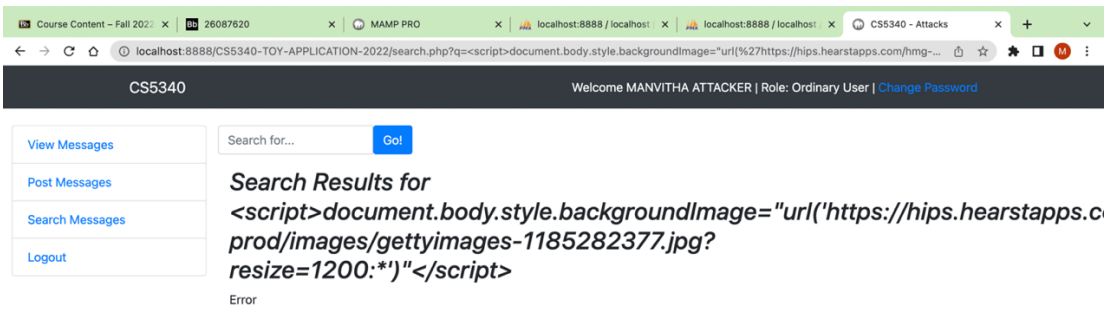


Observation:

As explained above, The **inputValidation** function validates every input by trimming the data, striping the data and by converting the HTML special characters (for example: the ***<script>*** tags is converted to ***<script&gt***) so that the browser detects it as a string.

Similarly, defacing the web page by changing background color and background image can also be prevented as we are preventing HTML injections.



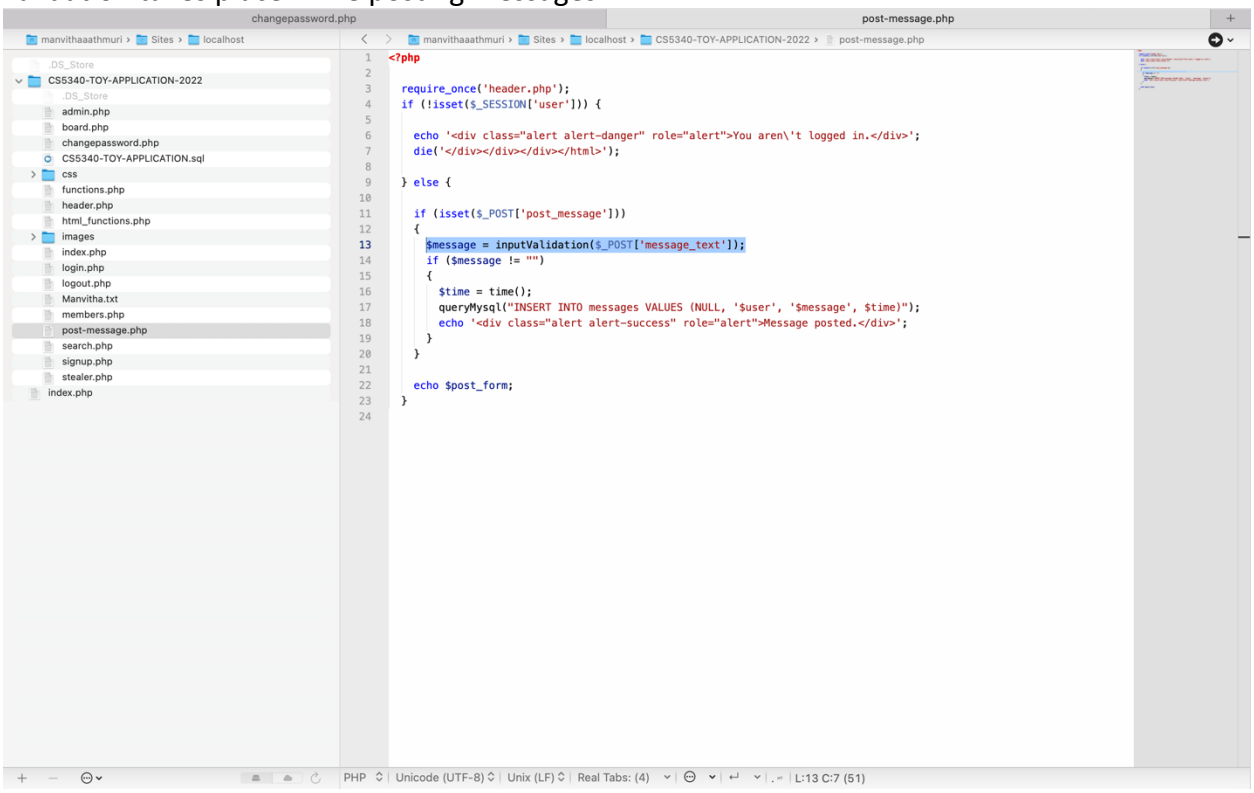


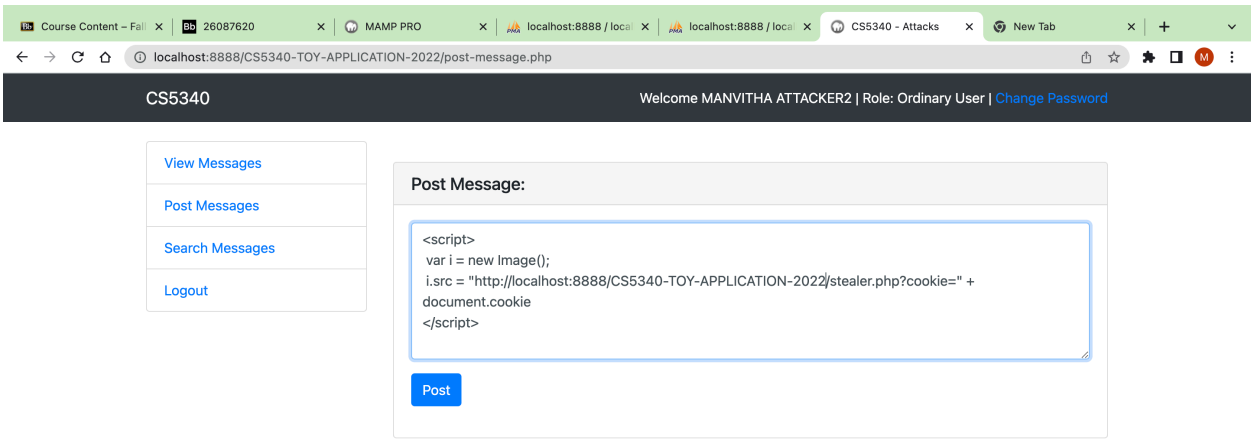
Stealing cookies prevention:

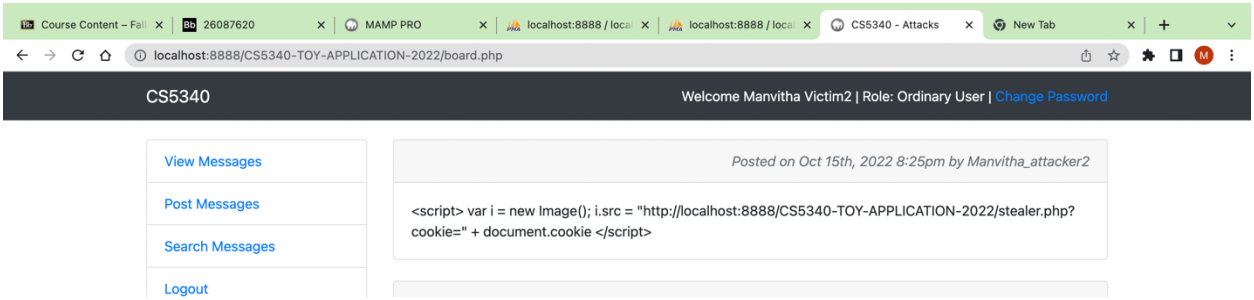
This web security vulnerability is about resource protection. Sensitive data such as user credentials should always be encrypted including in transit and at rest without any exceptions. We know that the passwords are being hashed using either crypto or hashing algorithms where they must not be weak. Cookies with sensitive data should always have a ‘secure’ flag on.

The simple way to reduce the exposure to this vulnerability is by virtually shredding the cookies if the sensitive data is not needed anymore.

Here we are using the same inputValidation function in the *post-message.php* file so that validation takes place while posting messages.







Observation:

As explained above, the function **inputValidation** trims, strips and convert the HTML special characters so that the browser reads the input as string and post the message without executing the java script.

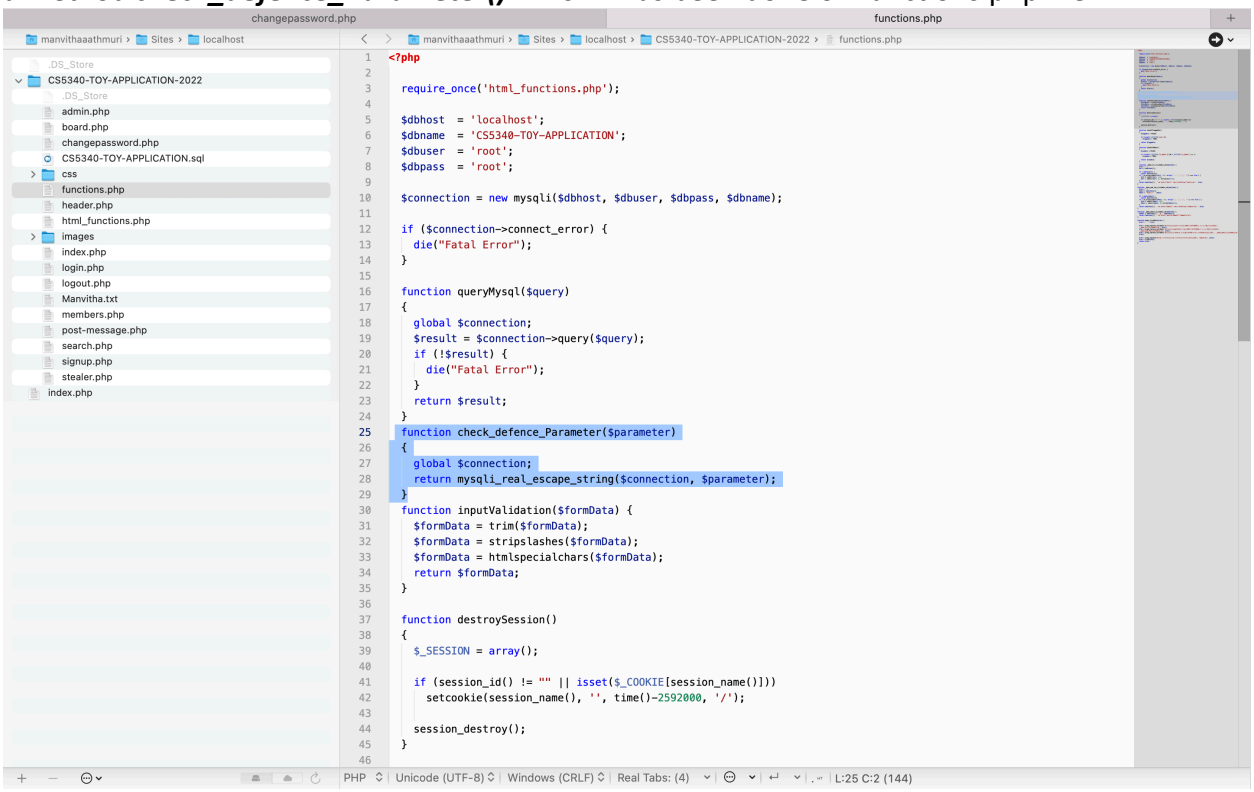
SQL injection mitigations:

Injections flaws can happen when we pass unfiltered data to the SQL server.

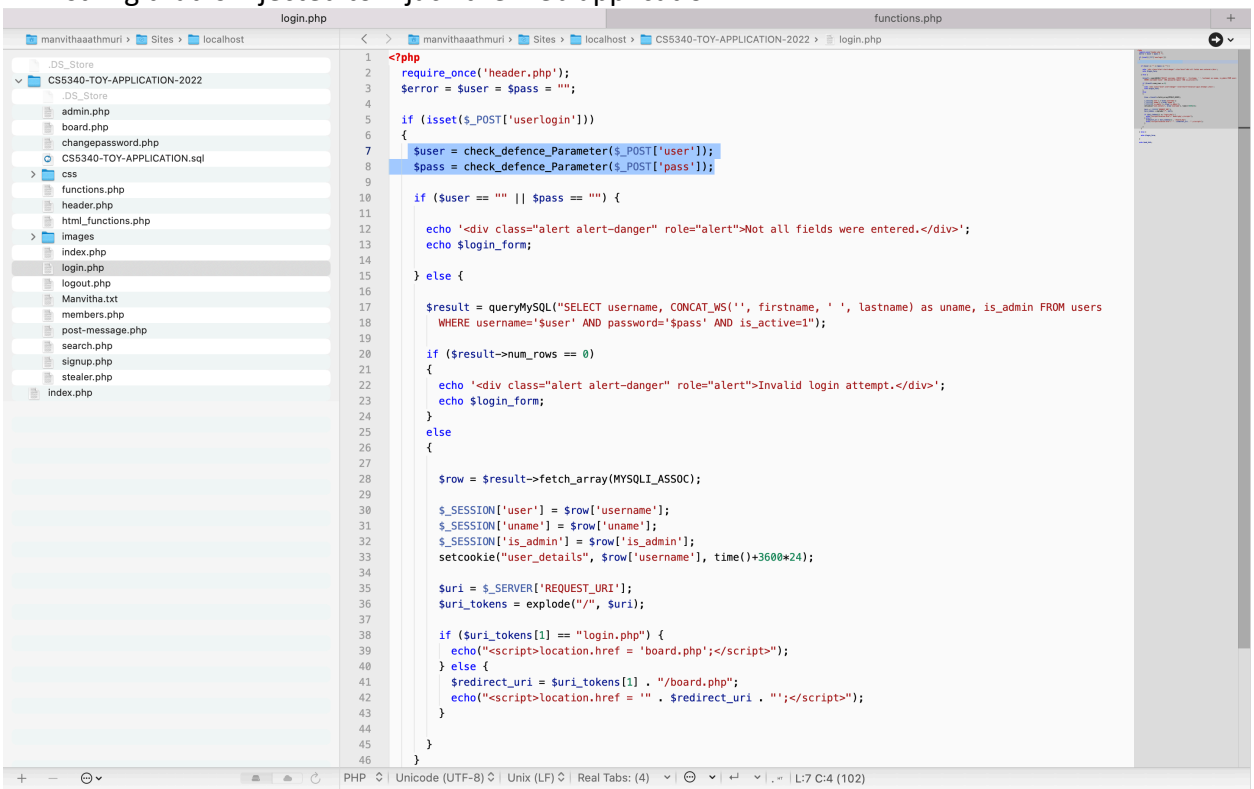
Protecting against injection is simply a matter of filtering the input and considering which senders can be trusted.

Below is the code fix done to mitigate the task 1 from section 2.

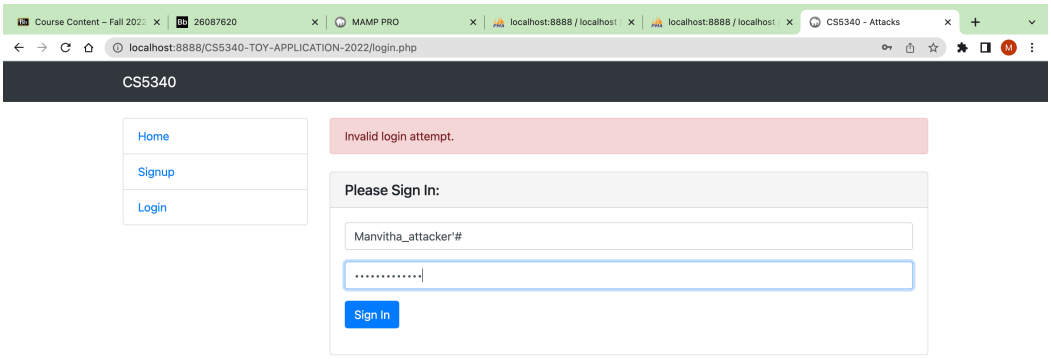
The function ***mysqli_real_escape_string()*** escapes the characters in a string which is written in a method ***check_defence_Parameter()***. This fix has been done on functions.php file.



Now the ***check_defence_Parameter()*** method is used in *login.php* file to escape the characters ***'#*** in string that is injected to hijack the web application.

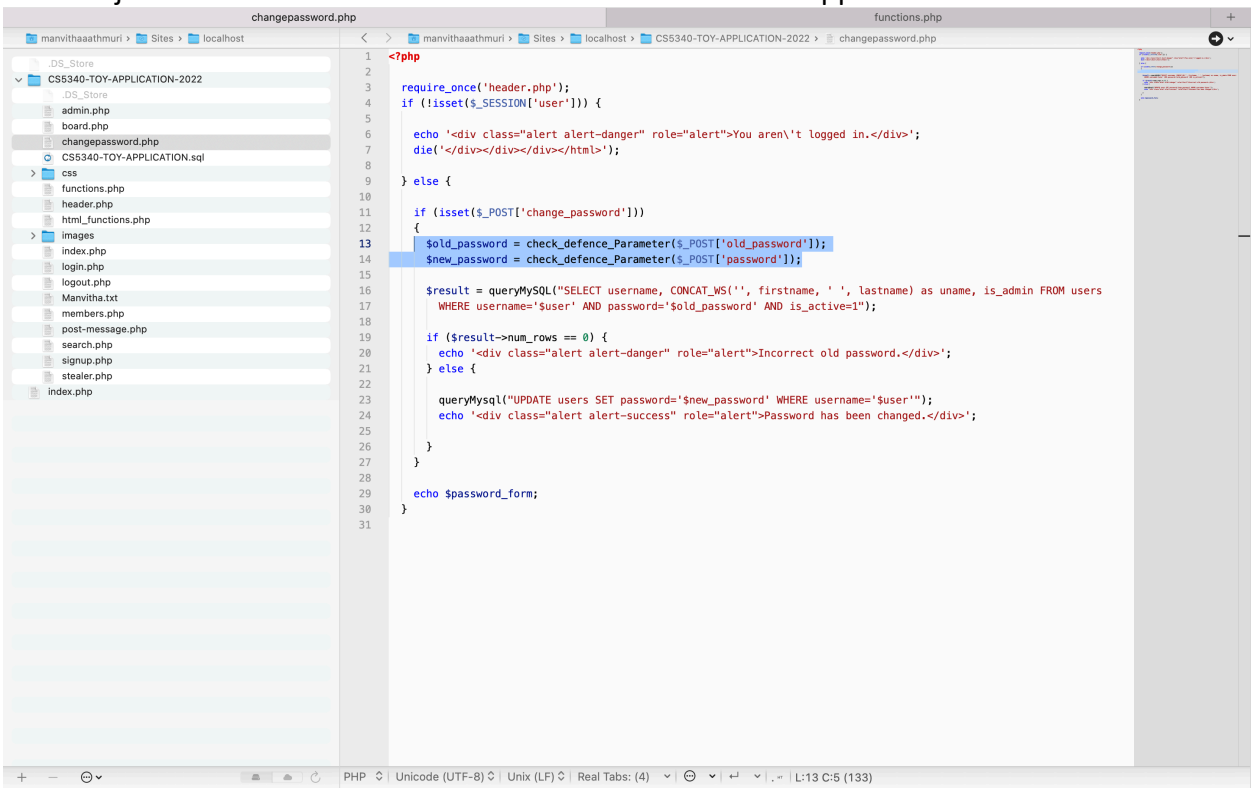


Result:

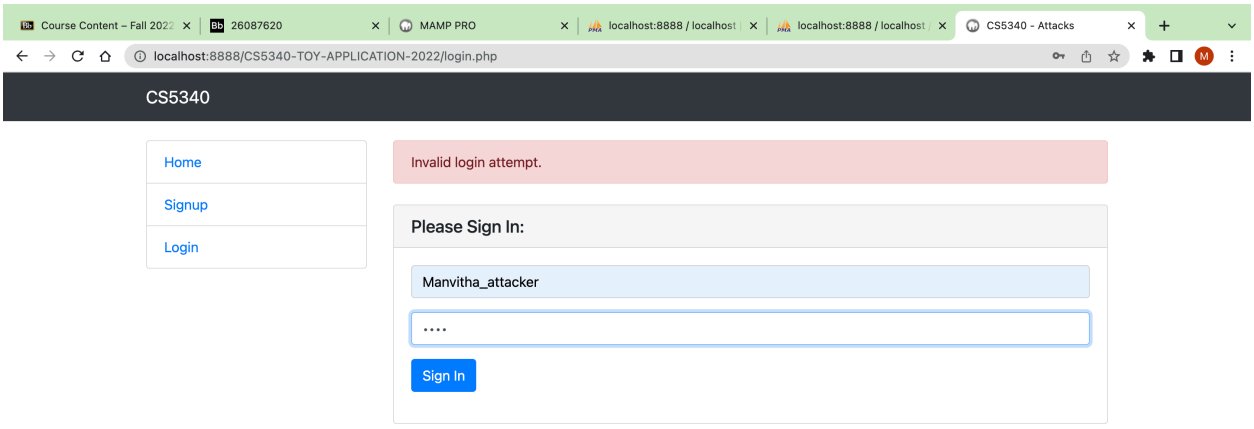


Prevention of Admin access to attacker:

The same method ***check_defence_Parameter()*** containing function ***mysqli_real_escape_string()*** checks for the characters and escapes them in string ***',is_admin='1*** that is injected to obtain the Administrator access of the web application.



Result:



c) Two other attacks discussed in class but not included in this lab are clickjacking and the CSRF attack. Discuss defense mechanisms for these attacks.

Clickjacking:

Clickjacking is an attack that tricks user into clicking a webpage element that is either invisible or shown as another element. This can cause users to download malware, visit malicious web pages, provide credentials or sensitive information, transfer money, or purchase products online without their consent.

This can be mitigated by

- Sending proper Content Security Policy (CSP) frame-ancestors directive response headers which instructs the browser to not allow framing from other domains.
- Defending with SameSite cookies.
- Employing defensive code in the UI to ensure that the current frame is the top-level window.

Cross-site Request Forgery (CSRF):

This is an attack that tricks a web browser into executing an unwanted action in an application to which a user is logged in. This results in unauthorized fund transfers, changed passwords and data theft including stolen session cookies.

To mitigate CSRF attacks,

- Log off web applications when they are not in use
- Do not allow browsers to remember passwords
- Avoid parallel browsing when an application is logged in.
- Double submission of cookies also blocks CSRF
    - It is like using unique tokens. Random tokens are assigned to both a cookie and a request parameter. The server then verifies that the tokens match before granting access to the application.