**6005 CEM Security Coursework** – 2. Template

Ernest Zimus

– SID: 10135033 –

# Introduction

Security implementations into the website are important, they allow the website owners to preserve data, keep operations running and provide trust to the user’s viewpoint. The template that was worked on is a bookstore written in Flask. Firstly, it had been looking abstract with barely any functionality and zero security. This paper is about reworking it by means of adding a few new features and trying to improve its security.

# Design

**Home**

Homepage design did remain unchanged in terms of there being the same copy made of the landing page when logging in as admin. On the copy version, the heading says “Black hat books – admin”, it is done for the user to understand that they are logged in as an admin. Perhaps the new books that are added to the database are shown on the home page and they can be scrolled through.

**Products**

The product landing page fully shows the name and description of the products that are saved in the database including images. They can be clicked on and added to the cart for further checkout.

Products can only be added to the cart when the user is logged in.

**Creating account**

Contains an HTML form to retrieve and save user data to the database. The data that is being saved in the database is the user's email and an encrypted password. Also, there is a Boolean value that is assigned to the user in the database “adm”, it determines whether the user is an admin, which would be equal to 1 or they are an ordinary user which is 0, that will be explained in detail in further reading. The password must be entered twice for the user does not make a mistake when creating an account. Also, there is a checkmark for the user to agree with the terms and services. There were security measures implemented into that page - password encryption and limited login attempts.

Limited login attempts are designed to tackle brute-force attacks (Balkhi, 2021). Login attempts are limited to 5. It works in a way that the browser session in the code is recorded and it counts each time a user attempts a login, each incorrect login gives +1 to the variable. On the last attempt, the user is notified that this is their last login attempt. There was also a possibility to implement a captcha or 2-factor authentication.

The main privacy concern for users is passwords, if the password in the database would be saved in plain text, that would be a disaster for the organisation that would host this website and perhaps would put a lot of responsibility on them, for that case the password encryption was implemented to the website, it allows the password that is entered into the account creation form in plain text to be encrypted in MD5+salt. For more information, password encryption makes the password non-readable to anyone. If the database would be compromised, the passwords would be in an encrypted form... I tried to implement bcrypt because it is considered secure according to (Rothenberger, 2022), but I kept getting sqlite3 syntax error, the passwords were successfully encrypted with it but were unable to store themselves in the database, decided to go with MD5+salt, even though MD5 itself is not secure, it’s a bit more secure with salt (What Is MD5 Salt and How to Use It?, n.d.). Nevertheless, having some sort of encryption is better than nothing. Having a non-working bcrypt broke login/account creation, I had to comment it out.

**Login**

The login landing page contains an HTML form that takes an email and password in plain text, the email matches the user’s row in the database and afterwards, the password that was provided in the form is encrypted in MD5+salt and then compared to the encrypted password in the database if both values match the user is redirected to the home page. Also to add, the user’s adm Boolean value is checked whether it is 1 or 0. 1 would mean the user is an administrator and redirects them to the admin home page, which is a copy of the home page itself. If it is 0, then it is considered that a basic user is logging in and they are redirected to the main page.

In short, the implementations are: checking if a user is an administrator and checking an encrypted password

**Settings**

Settings are available to only logged-in users. For a basic user it shows their ID, email, and purchase history through which they can leave a review and they can update their password. The password update form does not store plain text passwords. It takes a current and desired password in plain text form and encrypts both in md5+salt, if the current password is matched in the database with the encrypted form password, the current password is overwritten with the desired password in an encrypted form.

For an admin the settings are called “ADMIN” in the selection menu, it is the same copy of the user settings page, except the administrator has 2 extra buttons: “Edit users” and “Edit stock”

“Edit users” redirects to a page that shows a table containing all user ids, emails, encrypted passwords, and their “adm” status. There is a form on the page that allows an admin to grant admin privileges to another user and also revoke them, as input it takes the user’s id value and for admin 0 or 1. Also, an extra button to go to the home page for navigation.

“Edit stock” redirects to a page that shows a table containing product id, name, description, price and image. Also Has a button to return which returns to the home page. And at the end of the page, there is a form that allows you to put new books into the store, it takes the ID, Name, Description, and Price of a book, and the image selection to upload is only able to retrieve full image name and save it into the database, I tried to implement image upload with the flask .save() method, all seemed well but I have kept getting a permission denied error which I was unable to fix, only for this design the image upload must be done manually into the uploads folder of the project.

**Reviews**

Exposes email addresses to non-logged-in users. Logged-in users that purchased a certain book can leave a text review about it and select 1-5 stars. The review section is a direct SQL query with no protection at all, which is prone to a SQL injection/cross-scripting attack.

# Implementation

FULL Source code available here: https://github.coventry.ac.uk/zimuse/6005\_10135033

Added Daniel Goldsmith as a collaborator in the repository.

The code that is posted here is only for reference, and may not be full code, but gives a brief example, if needed please relate to the GitHub repository.

You will have to unzip the compressed folder 6005-CW-Teplate-main/app/views.py is the file with main code.

**Admin home page**

One of the If statements in login determines whether an user is an admin, redirects an admin to admin home page. Also logs that admin logged in.

**if** userQry**[**"adm"**]** **==** **1** **and** userQry**[**"password"**]** **==** encrypt**:**

app**.**logger**.**info**(**"ADMIN:Login as %s Success"**,** userQry**[**"email"**])**

flask**.**session**[**"user"**]** **=** userQry**[**"id"**]**

*#flask.session["admin"] = userQry["adm"]*

*#admin=+1*

flask**.**flash**(**"Login Successful"**)**

attempt **=** **0**

session**[**'attempt'**]** **=** attempt

**return** **(**flask**.**redirect**(**flask**.**url\_for**(**"admin"**)))**

Admin home page

**@app.route(**"/admin"**)**

**def** admin**():**

rows **=** query\_db**(**"SELECT \* FROM product"**)**

app**.**logger**.**info**(**rows**)**

**return** flask**.**render\_template**(**"admindex.html"**,** bookList **=** rows**)**

**Products**

<n/a>

**Creating account**

email **=** flask**.**request**.**form**.**get**(**"email"**)**

password **=** salt**+**flask**.**request**.**form**.**get**(**"password"**)**

encrypt **=** hashlib**.**md5**(**password**.**encode**()).**hexdigest**()** *#ENCRYPT*

*#encrypt = bcrypt.hashpw(password.encode(), salt)*

As said contains password encryption using md5+salt, tried to implement bcrypt and was unsuccessful…

Also, it is not possible to create an account on the same email.

**Login**

Encrypting the password from the form

password **=** salt**+**flask**.**request**.**form**.**get**(**"password"**)**

encrypt **=** hashlib**.**md5**(**password**.**encode**()).**hexdigest**()** *#ENCRYPT*

*#encrypt = bcrypt.hashpw(password.encode(), salt)*

Declaring limited login attempts

attempt**=**session**[**"attempt"**]**

Checking whether the user is an administrator and if an encrypted password is correct

**if** userQry**[**"adm"**]** **==** **1** **and** userQry**[**"password"**]** **==** encrypt**:**

Adding failed login attempts after each failed attempt

**else:**

flask**.**flash**(**"Password is Incorrect"**)**

attempt **=** session**.**get**(**'attempt'**)**

attempt **=** attempt **+** **1**

When the login attempt is 4 (count from one) (python counts from 0), the user is notified that this is their last attempt to log in.

**elif(**attempt **==** **3):**

flask**.**flash**(**"This is your last attempt to log in"**)**

**else:**

flask**.**flash**(**"You can't log in anymore."**)**

**Settings**

Contains the same password encryption as for account creation. Takes 2 plain text passwords, encrypts both, compares one to the one in a database and replaces it with the second one.

**if** flask**.**request**.**method **==** "POST"**:**

current **=** salt**+**flask**.**request**.**form**.**get**(**"current"**)**

encryptcurrent **=** hashlib**.**md5**(**current**.**encode**()).**hexdigest**()** *#ENCRYPT*

*#encryptcurrent = bcrypt.hashpw(current.encode(), salt)*

password **=** salt**+**flask**.**request**.**form**.**get**(**"password"**)**

encrypt **=** hashlib**.**md5**(**password**.**encode**()).**hexdigest**()** *#ENCRYPT*

*#encrypt = bcrypt.hashpw(password.encode(), salt)*

app**.**logger**.**info**(**"Attempt password update for %s from %s to %s"**,** userId**,** encryptcurrent**,** encrypt**)**

app**.**logger**.**info**(**"%s == %s"**,** encryptcurrent**,** thisUser**[**"password"**])**

*#matched = bcrypt.checkpw(userQry["password"], encryptcurrent)*

**if** encryptcurrent**:**

**if** encryptcurrent **==** userQry**[**"password"**]:**

app**.**logger**.**info**(**"Password OK, update"**)**

*#Update the Password*

encrypt **=** salt**+**hashlib**.**md5**(**password**.**encode**()).**hexdigest**()** *#ENCRYPT*

theSQL **=** f"UPDATE user SET password = '{encrypt}' WHERE id = {userId}"

app**.**logger**.**info**(**"SQL %s"**,** theSQL**)**

write\_db**(**theSQL**)**

flask**.**flash**(**"Password Updated"**)**

I also added user privilege modifications below.

**Reviews**

Limited character counts on reviews in HTML form to prevent long texts.

**<textarea** id="textarea1" name="review" class="materialize-textarea" rows=10 maxlength="50"**>**{{review.review}}**</textarea>**

**Logging**

logging**.**basicConfig**(**filename**=**"log.txt"**)**

Logging was done in that way everything saves into a text file, which provides plenty of information on what is happening on the server, it helps to determine whether there is an attack being performed on the website (Konov, 2022).

Sample:

A picture containing text

Description automatically generated

**View users as admin and edit their admin privileges:**

**@app.route(**'/userview'**,** methods**=[**'GET'**,** 'POST'**])** *#route to view users and set admin 1 or 0*

**def** userView**():**

**if** flask**.**request**.**method **==** 'POST'**:**

*#return insertNewStock(, r)*

a**=**request**.**form**[**'aidi'**]**

n**=**request**.**form**[**'admp'**]**

con **=** sqlite3**.**connect**(**'database.db'**)**

con**.**execute**(**f"UPDATE user SET adm = '{n}' WHERE id = '{a}'"**)**

con**.**commit**()**

con**.**close**()**

**print(**"i am success"**)**

*#return showusers();*

**print(**"main function"**)**

con **=** sqlite3**.**connect**(**'database.db'**)** *#connection to database*

con**.**row\_factory **=** sqlite3**.**Row

cur **=** con**.**cursor**()**

cur**.**execute**(**"SELECT \* from user"**)**

rows **=** cur**.**fetchall**()**

con**.**close**()**

**return** flask**.**render\_template**(**"userview.html"**,**rows **=** rows**)** *#prints out all of the user*

Takes 2 values from an HTML table, id and adm (1 or 0) and changes it accordingly.

I would also write about the stock replenishment feature but it does not relate to security, it rather contributes to the functionality that was required.

# Summary

The website is fully functional, so far functionality it has:

* Store/match/update encrypted passwords
* Working login/register/password update
* Log what is happening on the website into text file
* Display all products
* Add more products
* View users, grant/revoke admin privileges
* Leave reviews
* See purchase history in settings
* Pseudo-working checkout page

Most of the flaws were found manually in the code and potential SQL injections were found automatically using an HCL AppScan tool.

What was done in terms of security is password hashing with MD5+salt, limited login attempts, checking whether the user is an administrator or not, logging what is happening on the website, ability to grant/revoke admin permission, and limited character reviews.

# References

Balkhi, S. (2021, October 30). *How and Why You Should Limit Login Attempts in WordPress*. WPBeginner. https://www.wpbeginner.com/plugins/how-and-why-you-should-limit-login-attempts-in-your-wordpress/

*Free Citation Generator | APA, MLA, Harvard*. (2022, November 8). Scribbr. https://www.scribbr.com/citation/generator/

Konov, K. (2022, October 25). *The Importance of Website & Server Logs*. Sucuri Blog. https://blog.sucuri.net/2022/08/importance-of-website-logs-for-security.html

Rothenberger, R. (2022, January 14). *More Secure Passwords in Bcrypt — Beating the 72 Bytes Limitation*. https://www.monterail.com/blog/more-secure-passwords-bcrypt

*Source code beautifier / syntax highlighter – convert code snippets to HTML « hilite.me*. (n.d.). http://hilite.me/

*What is MD5 Salt and How to Use It?* (n.d.). InfosecScout. https://infosecscout.com/md5-salt-hash/