6005 CEM CW2

Gurvinder Nagra

SID: 10248274

Contents

[Introduction 2](#_Toc119481225)

[**Systems Design/Implementation** 2](#_Toc119481226)

[**User accounts** 2](#_Toc119481227)

[**Creation** 2](#_Toc119481228)

[**Reviews** 4](#_Toc119481229)

[**Logging and analytics** 4](#_Toc119481230)

[**Implementation** 4](#_Toc119481231)

[**Conclusion** 5](#_Toc119481232)

[References 5](#_Toc119481233)

# Introduction

When creating a website precautions must be taken to ensure that both the user and the website itself is secure from any vulnerabilities, this can vary from many things to protecting from malicious files, preventing data exploits, or allowing malicious users access to a system they shouldn’t have access to. Here we will be exploring such vulnerabilities and showcase how they can be dealt with and properly prevented/circumvented by rewriting and adding security features to code.

What we are working with is a bookshop website in which we must ensure supports User Accounts (customer and admin), Reviews, Logging, and analytics as well as the ability to view and purchase products. We will be looking at implementing this as well as adding certain aspects of security features alongside it all.

# **Systems Design/Implementation**

## **User accounts**

# **Creation**

User accounts are to have many different levels to them, starting with account creation this will be necessary in order for the user to access the site and buy and review products. In order to prevent multiple accounts from spamming the system the user will be required to use emails in order to get conformation for accounts. This will help prevent attacker from creating multiple accounts to try and target the website system.

Whilst creating accounts there is also a level of password security, this is to ensure that passwords have requirements of more than 6 characters as well as ensuring that they use a mixture of numerical, lowercase, uppercase and special characters to lower the risk of an attacker brute forcing a password. There is also the use of a captcha system to prevent bots from creating new accounts as well as making it harder for malicious users.

Another thing is that in order to secure the passwords, we make sure to encrypt them before storing them in a database, here we use MD5, but since MD5 alone isn’t a completely secure way to save password we are also sure to salt them, this means “even in the instance of two users choosing the same passwords” (Arias, 2021) both passwords will have a random bit added to it meaning that even if one password is unsecure another user with the same password won’t be at risk.

**Login**

In order for the user to log into the website it will require a form in which the user shall input their details, and this will be hashed and compared to the hash within the database of the website, this will ensure only a hash is sent and means less unnecessary data will be required and prevent any passwords and emails being stored within the source code should that be breached. There will also need to be forms of input sanitation since there are text boxes, this will be explained further on. As well as that the user settings where users can change password details will need a similar implementation.

Furthermore, the login will also need further protection, we already have this in the form of hashing and salting the password, however, to add onto this we can add another captcha system for logins to prevent bots from trying to login with brute force attempts. As well as that we can also have it so that the login attempts are recorded for each session meaning that the user will only have a limited number of attempts before they can’t login, this is done through checking sessions in the code but can also be done via logging an IP address for each attempted login “These logs can be used to determine the source of attacks, the pattern of attacks and to provide and early warning about such attacks” (Rasch, 2018). This will help against constant brute forcing and will also help alert there is a possible risk of an attack so that such a threat can be properly dealt with.

password **=** Salt**+**flask**.**request**.**form**.**get**(**"password"**)** *#Salting the password*

encrypt **=** hashlib**.**md5**(**password**.**encode**()).**hexdigest**()** *#encryption for the password after it is salted*

Declaring limited login attempts to prevent malicious users from brute forcing.

LoginAttempt**=**session**[**"Attempt"**]**

Checking whether the user is an administrator and if they are using the correct password to login (password is in encrypted form)

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

Incrementing the failed login attempts after each failed attempt from the user

**else:**

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

LoginAttempt **=** session**.**get**(**'Attempt'**)**

LoginAttempt **=** LoginAttempt **+** **1**

After a certain amount of attempts the user will not be able to log in anymore.

**elif(**LoginAttempt **==** **3):**

flask**.**flash**(**"This is your final login attempt"**)**

**else:**

flask**.**flash**(**"You cannot login in anymore, an administrator will review your account and contact you via email to sort this"**)**

Settings page with added encryption

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

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

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

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

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

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

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

**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"**)**

**Admin Account**

For admin accounts, they will function differently from user accounts, having the ability to both view user accounts fully as well as change stock detail. These accounts would need to be secured better, one way is by making sure the system is properly able to identify which account is an admin account and ensuring that no SQL injection can let this trip up in the database. Also, for the admin webpages I will also need to ensure that the user must be logged in as an admin and can’t simply just add /adminsettings to the URL and gain access, which is done by having each admin route check if the user is an admin.

As well as that admin accounts will have steeper requirements for passwords to make sure that they are even less vulnerable with higher requirements and even a phone number to ensure they are harder to access with phone authentication. What we could further do is have admin IP addresses saved so that it gets logged in the system if the admin’s login from an unknown location or so that the admin accounts can only be accessed from certain locations/networks making them more secure.

There is also the fact that admin’s will be able to promote users to admin, this will require a level of authentication as well, to prevent anyone who may have gained access to an admin account from not using it to change user levels but also to ensure that an account doesn’t get a new level due to a mis click and instant change.

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

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

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

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**)**

Admin settings to change user 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(**"Successfull"**)**

*#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*

## **Reviews**

On this site the users will be able to review the products they buy, for privacy and as such as security reasons I have made it so users have the option to remain anonymous whilst reviewing so no one who may not like a review can trace it back and bother a particular user via email. As well as that reviews will be stored in an entirely separate database so that leaves no vulnerability to the database containing the passwords and user levels to any SQL injection. Also, to be safe the reviews are limited to 500 words this will ensure that when data enters through the review to the website it is less vulnerable to cross site scripting which works by “works by manipulating a vulnerable web site so that it returns malicious JavaScript to users. When the malicious code executes inside a victim's browser, the attacker can fully compromise their interaction with the application” (PortSwigger, N/A) and in this case the review section can allow that. Though the password database is kept separate to aid this as well.

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

## **Logging and analytics**

When creating the website, we also had to take account into logging what happens on the website in order to check for any malicious activity. To do this I have created a log file which will log every request that happens on the website from the moment it is brought online. As well as that the log file will contain data about dates and time of any request as well as the IP address it originated from. This is to make it easier to track down a particular user should they have done something malicious to the site, this does contain data that is considered PII, to keep compliant with GDPR there is a clear warning on the login page as “EU citizens now have more rights and policies in place to ensure their data is kept securely. Organisations must have a lawful purpose to store and process the sensitive personal data of a data subject. There must be a document of explicit consent from the individual in regards to what data is being collected, for how long, and what it is being used for.” (Rooney, 2018) All the data kept is necessary for security purposes as it is just storing requests that go through the website and the IP and the data and time, they come from, so it doesn’t have any other sensitive information which we do not need.

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

Text

Description automatically generated

# **Implementation**

**SQL Injection**

Since a lot of things are stored within a database, we will need to be sure that the data stored within them is not prone to an SQL injection as this could allow users to bypass the login and access admin accounts which would be a serious breach though “It all depends on the capabilities of the attacker, but the exploitation of an SQL injection vulnerability can even lead to a complete takeover of the database” (Invicti, 2022). To do safeguard against SQL injection throughout the code I have been sure to use APIs within python libraries that will perform parameterized queries on most of the database as it can. Furthermore, I have been sure to watch out and make sure there isn’t any/minimal SQL within the templates or source code to ensure no one can simply inspect element and take advantage of any vulnerabilities there.

**Cross Site scripting**

Cross site scripting is also another vulnerability we must look out for with the code as it can allow users with malicious intent to inject HTML and JavaScript into the context of a website. To prevent against this, I have made sure to properly escape text so it doesn’t include any HTML tags that are not needed as well as that since this website is built on Flask, Jinja2 which when in use with Flask will be configured to automatically escape all values unless told otherwise, meaning there shouldn’t be any XSS problems in templates.

# **Conclusion**

Overall, we can see there are quite a few security vulnerabilities that have been considered whilst creating these website as well as exploits that had to be managed in order to ensure proper security. Though for the most part the precautions set should be enough, considering that website itself doesn’t have many things that are particularly worth hacking for. Though attacker may want to take passwords which could be used to access accounts on something else a user may have, however since passwords are encrypted this should be much of an issue, the same with many other vulnerabilities.

# Bibliography

Arias, D. (2021, February 25th). *Adding Salt to Hashing: A Better Way to Store Passwords*. Retrieved from autho0 blog: https://auth0.com/blog/adding-salt-to-hashing-a-better-way-to-store-passwords/

Invicti. (2022, N/A N/A). *What is SQL Injection*. Retrieved from Invicti: https://www.invicti.com/blog/web-security/sql-injection-vulnerability/

PortSwigger. (N/A, N/A N/A). *Cross site scripting*. Retrieved from PortSwigger: https://portswigger.net/web-security/cross-site-scripting

Rasch, M. (2018, September 6th). *Is it Unlawful to Collect or Store TCP/IP Log Data for Security Purposes?* Retrieved from securityboulevard: https://securityboulevard.com/2018/09/is-it-unlawful-to-collect-or-store-tcp-ip-log-data-for-security-purposes/

Rooney, N. (2018, December 12th). *groundlabs*. Retrieved from What is PII for GDPR: https://www.groundlabs.com/blog/what-is-pii-for-gdpr/