**6005 CEM Security Coursework** – 1. Audit

Ernest Zimus

– SID: 10135033 –

Contents

[Introduction 1](#_Toc119682874)

[Audit methodology 1](#_Toc119682875)

[Vulnerabilities 1](#_Toc119682876)

[Detailed description of chosen vulnerability 6](#_Toc119682877)

[Conclusions 7](#_Toc119682878)

[References 7](#_Toc119682879)

# Introduction

A security audit is a must-have thing when developing a website, it helps to detect code vulnerabilities that could be exploited. I am using a bookstore website built using the flask framework that is retrieved from the learning environment. I will go through it to find its code vulnerabilities and try to explain them there including code snippets, for that I will use different audit methods, and perhaps I will talk about a chosen vulnerability in detail.

# Audit methodology

The purpose of an audit is to find code vulnerabilities and explain them. To find vulnerabilities I will go through the code manually and will use an automatic tool called HCL Appscan CodeSweep, it supports many programming languages including Python and can try various exploits on the code and display where exactly the vulnerable code part is. (HCL AppScan CodeSweep - Visual Studio Marketplace, n.d.)

# Vulnerabilities

**Vulnerable code lines are marked in red colour**

The vulnerabilities I have found manually so far are in the:

* Account creation
* Login
* Reviews
* Settings update

The first to talk about I want to is account creation, the data that is from HTML form that is saved into the database when creating an account is not encrypted, in means of password is missing encryption, it is best to store passwords encrypted as only what gets encoded, can’t be decoded back if it is compromised. With this code, if the database would be compromised, the passwords that are saved in it are plain text which can be read by a human being. Which makes it high severity.

**@app.route(**"/user/create"**,** methods**=[**"GET"**,**"POST"**])**

**def** create**():**

*""" Create a new account,*

*we will redirect to a homepage here*

*"""*

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

**return** flask**.**render\_template**(**"create\_account.html"**)**

*#Get the form data*

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

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

*#Sanity check do we have a name, email and password*

**if** **not** email **or** **not** password**:**

flask**.**flash**(**"Not all info supplied"**)**

**return** flask**.**render\_template**(**"create\_account.html"**,**

email **=** email**)**

*#Otherwise we can add the user*

theQry **=** "Select \* FROM User WHERE email = '{0}'"**.**format**(**email**)**

userQry **=** query\_db**(**theQry**,** one**=**True**)**

**if** userQry**:**

flask**.**flash**(**"A User with that Email Exists"**)**

**return** flask**.**render\_template**(**"create\_account.html"**,**

name **=** name**,**

email **=** email**)**

**else:**

*#Crate the user*

app**.**logger**.**info**(**"Create New User"**)**

theQry **=** f"INSERT INTO user (id, email, password) VALUES (NULL, '{email}', '{password}')"

userQry **=** write\_db**(**theQry**)**

flask**.**flash**(**"Account Created, you can now Login"**)**

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

Moving on to log in, the code lacks a captcha or 2-factor authentication, perhaps it also lacks limited login times for user sessions. Captcha would be necessary in case to avoid brute force attacks, as brute force is mainly performed with username and password, a script can not read image captchas, which would mean that login itself has to be performed manually with limited login attempts, even if username and password were compromised, it will not be possible to get into the account with just with the username and password. 2-factor authentication usually comes in an SMS message or generator code variants, both prove the account holder (KENTON, 2022). The code snippet is for reference.

**@app.route(**"/user/login"**,** methods**=[**"GET"**,** "POST"**])**

**def** login**():**

*"""*

*Login Page*

*"""*

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

*#Get data*

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

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

app**.**logger**.**info**(**"Attempt to login as %s:%s"**,** user**,** password**)**

theQry **=** "Select \* FROM User WHERE email = '{0}'"**.**format**(**user**)**

userQry **=** query\_db**(**theQry**,** one**=**True**)**

**if** userQry **is** None**:**

flask**.**flash**(**"No Such User"**)**

**else:**

app**.**logger**.**info**(**"User is Ok"**)**

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

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

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

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

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

**else:**

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

**return** flask**.**render\_template**(**"login.html"**)**

For reviews, there is a possibility to perform a cross-site scripting attack. A cross-site scripting attack or XSS is placing a malicious part of code into the website to try and steal other users' sessions (PortSwigger, n.d.). If changing the session id to another user bypasses most of the security and you could impersonate someone else and even do tasks under their name. High severity.

**if** reviewId**:**

*#Update an existing oe*

app**.**logger**.**info**(**"Update Existing"**)**

theSQL **=** f"""

UPDATE review

SET stars = {reviewStars},

review = '{reviewComment}'

WHERE

id = {reviewId}"""

app**.**logger**.**debug**(**"%s"**,** theSQL**)**

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

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

**else:**

app**.**logger**.**info**(**"New Review"**)**

theSQL **=** f"""

INSERT INTO review (userId, productId, stars, review)

VALUES ({userId}, {itemId}, {reviewStars}, '{reviewComment}');

"""

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

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

flask**.**flash**(**"Review Made"**)**

*#Otherwise get the review*

theQry **=** f"SELECT \* FROM product WHERE id = {itemId};"

item **=** query\_db**(**theQry**,** one**=**True**)**

theQry **=** f"SELECT \* FROM review WHERE userID = {userId} AND productID = {itemId};"

review **=** query\_db**(**theQry**,** one**=**True**)**

app**.**logger**.**debug**(**"Review Exists %s"**,** review**)**

**return** flask**.**render\_template**(**"reviewItem.html"**,**

item **=** item**,**

review **=** review**,**

**)**

Also in the writing a review form, the vulnerability is that the review can contain an unlimited amount of characters, which could just ruin the website. Medium severity.

**<form** method="POST"**>**

**<div** class="row"**>**

**<div** class="input-field col s6"**>**

**<p** class="range-field"**>**

**<input** type="range" id="rating" name="rating" min="0" max="5" value="{{review.stars}}"**/>**

**</p>**

**<label** for="rating"**>**Rating**</label>**

**</div>**

**<div** class="row"**>**

**<div** class="input-field col s12"**>**

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

**<label** for="textarea1"**>**Feedback**</label>**

**</div>**

**</div>**

**<input** type="hidden" name="reviewId" value="{{review.id}}"**>**

**<button** class="waves-effect waves-light btn"**>**Submit**</button>**

**</form>**

When updating the password in settings, there is no password hashing.

**@app.route(**"/user/<userId>/update"**,** methods**=[**"GET"**,**"POST"**])**

**def** updateUser**(**userId**):**

*"""*

*Process any chances from the user settings page*

*"""*

theQry **=** "Select \* FROM User WHERE id = '{0}'"**.**format**(**userId**)**

thisUser **=** query\_db**(**theQry**,** one**=**True**)**

**if** **not** thisUser**:**

flask**.**flash**(**"No Such User"**)**

**return** flask**.**redirect**(**flask\_url\_for**(**"index"**))**

*#otherwise we want to do the checks*

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

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

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

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

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

**if** current**:**

**if** current **==** thisUser**[**"password"**]:**

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

*#Update the Password*

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

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

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

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

**else:**

app**.**logger**.**info**(**"Mismatch"**)**

flask**.**flash**(**"Current Password is incorrect"**)**

**return** flask**.**redirect**(**flask**.**url\_for**(**"settings"**,**

userId **=** thisUser**[**'id'**]))**

flask**.**flash**(**"Update Error"**)**

**return** flask**.**redirect**(**flask**.**url\_for**(**"settings"**,** userId**=**userId**))**

Perhaps there is another issue but with the majority of HTML forms, the input id, name and label are reused, which could be revealed when inspecting the source code on the client version. It makes it easier for an attacker to perform an SQL injection to reveal the database when knowing the table name and other surnames, the ones used in the database are very primitive and easy to be guessed.

**<input** id="password" name="password" type="password"**>**

**<label** for="password"**>**Password**<label>**

Automated testing with HCL app scan found potential SQL injection vulnerabilities

Text

Description automatically generated

The vulnerable code for it is:

Line 90:

theQry **=** "Select \* FROM User WHERE email = '{0}'"**.**format**(**user**)**

Line 129:

theQry **=** "Select \* FROM User WHERE email = '{0}'"**.**format**(**email**)**

Line 155:

theQry **=** "Select \* FROM User WHERE id = '{0}'"**.**format**(**userId**)**

Line 197:

theQry **=** "Select \* FROM User WHERE id = '{0}'"**.**format**(**userId**)**

Line 351:

theQry **=** "Select \* FROM User WHERE id = {0}"**.**format**(**flask**.**session**[**"user"**])**

SQL injection allows the attacker to make SQL queries with malicious code to see the backend database which could compromise, edit/delete/view data (Acunetix, 2022)

# Detailed description of chosen vulnerability

Not having password encryption Implemented in the website, makes it not GDPR compliant, and also puts a lot of pressure and responsibility when there is a data leak. If the database gets compromised, the passwords could be read in plain text.

Having a hashed password stored in the database makes it non-readable to anyone, the password holder only knows the password.

One of the simplest encryption types that could be used is MD5, giving an instance password is “admin”, this would be encrypted into that: 21232f297a57a5a743894a0e4a801fc3

If we would encrypt word admin, it would be the same again: 21232f297a57a5a743894a0e4a801fc3

MD5 is case-sensitive, it will hash a word differently if we start changing letters to uppercase.

When hashing the same word with MD5 it will give the same hash, if we change one of the letters to uppercase, the hash will be different, perhaps it provides security until the word that is hashed is well known by others, in means of there are decryption tools that store a lot of basic words and various variations of them, and if the hash of that word matches – it is the only way the MD5 hashed word could be reversed.

In addition, comes the password requirements, like having at least one special symbol in the password or uppercase letters and a minimum character limit. This is most likely that this password will not be available in any MD5 hash-reversing databases. As said the only one who will know the password is the user.

Nowadays hashing a password with MD5 is not recommended to use, because it was designed for fast hashing and was proven to be insecure (Shacklett & Loshin, 2021). MD5 was used here to explain hashing easily. It is now recommended to use the B-Crypt encryption algorithm and it is considered secure (Raman, 2021).

# Conclusions

The website lacks password encryption in account creation and when updating the password, additional protective measures should be added to the login page such as limited login attempts. Reviews must be in limited word count, and overall, the input to the database where possible has to be checked to make the SQL injections not possible. As seen, the automatic security audit method was not fully reliable as it only showed possible SQL injections, going through the code manually truly revealed more flaws in the code.

# References

Acunetix. (2022, July 21). *What is SQL Injection (SQLi) and How to Prevent Attacks*. https://www.acunetix.com/websitesecurity/sql-injection/

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

*HCL AppScan CodeSweep - Visual Studio Marketplace*. (n.d.). https://marketplace.visualstudio.com/items?itemName=HCLTechnologies.hclappscancodesweep

KENTON, W. (2022, September 8). *What Is Two-Factor Authentication (2FA)? How It Works and Example*. Investopedia. https://www.investopedia.com/terms/t/twofactor-authentication-2fa.asp

*OWASP Risk Rating Methodology | OWASP Foundation*. (n.d.). https://owasp.org/www-community/OWASP\_Risk\_Rating\_Methodology

PortSwigger. (n.d.). *What is cross-site scripting (XSS) and how to prevent it? | Web Security Academy*. https://portswigger.net/web-security/cross-site-scripting

Raman, R. (2021, November 2). *The Ultimate Guide to BCrypt and Authentication Protocols*. Clerk. https://clerk.dev/blog/bcrypt-hashing-authentication-encryption

Shacklett, M. E., & Loshin, P. (2021, August 23). *MD5*. SearchSecurity. https://www.techtarget.com/searchsecurity/definition/MD5

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