6005 CEM CW1

Gurvinder Nagra

SID: 10248274

Contents

[**Introduction** 2](#_Toc119509083)

[**Audit Methods** 2](#_Toc119509084)

[**Audit Results** 2](#_Toc119509085)

[**Automated testing** 2](#_Toc119509086)

[**Manual testing** 3](#_Toc119509087)

[**Chosen Vulnerability** 3](#_Toc119509088)

[**Conclusion** 3](#_Toc119509089)

## **Introduction**

In this security audit I plan to discover and evaluate any flaws in the black hat bookstore website, to do this I will be using different audit methods to find vulnerabilities. I will also be sure to detail how the website can be secured as well as how to mitigate a specific attack, ensuring that the website is fixed and not vulnerable to exploits.

## **Audit Methods**

In this audit I will be using automatic as well as manual audit methods. The reasons I am doing this is because an automated test will be quick to help discovering any vulnerabilities at a surface level however by also doing a manual audit, I will be able to go into further depth with the code allowing me to further test and check the code at a more detailed level. Since “Testing a large website manually is a very tedious task for manual security tester as they have to test one by one URL. Automation tools can help the tester to find out basic vulnerabilities quickly and they can focus their time on findings business logic and other security issues which tools cannot find.” AeoLogic. (2020) Which is why it should be standard practice to do so here.

The code (insert link to github) will be tested on a coding platform known as codio which uses a ubuntu based terminal.

The tool I will be using for the automated testing will be app scan, these are a security linter which looks through the code of a website and discovers vulnerabilities and other issues with the code. This will bring up a decent number of vulnerabilities at a base level and may point out other things which may not be directly visible to the user.

For the manual testing I will be doing a code review, meaning I will go through the source code of a website since an automated tool may not be considered smart enough to check through everything properly like a human would. I will also be sure to test the vulnerabilities myself with the webpage up and see if there is anything that is especially exploitable.

## **Audit Results**

# **Automated testing**

From appscan I had received a few vulnerabilities, mainly being SQL injection and multiple cases of it, though all of the cases were either the same or very similar SQL code. I had also discovered hardcoded credentials in python code.

Text

Description automatically generated



These were all considered to be high risk vulnerabilities throughout the code meaning they would have a high impact on security if exploited by a malicious user.

SQL injection is an exploit that allows someone to insert an SQL query through input data, an SQL attack itself can vary from just being able to view confidential data from a data base or even be able to perform database functions meaning the database could be at risk of being rewritten. A way to deal with these exploits could be to “Literally, you assemble your string with placeholders for the data to be inserted, and apply the data in the same sequence as the placeholders.” (Invicti, 2022)

itemQry = query\_db(f"SELECT \* FROM product WHERE id = ?",[theItem], one=True)

*#And Associated Reviews*

*#reviewQry = query\_db("SELECT \* FROM review WHERE productID = ?", [theItem])*

theSQL = f"""

SELECT \*

FROM review

INNER JOIN user ON review.userID = user.id

WHERE review.productID = {itemQry['id']};

"""

reviewQry = query\_db(theSQL)

*#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"))

INNER JOIN user ON review.userID = user.id

WHERE review.productID = {itemQry['id']};

"""

reviewQry = query\_db(theSQL)

*#Purchases*

theSQL = f"Select \* FROM purchase WHERE userID = {userId}"

purchaces = query\_db(theSQL)

theSQL = """

SELECT productId, date, product.name

FROM purchase

INNER JOIN product ON purchase.productID = product.id

WHERE userID = {0};

""".format(userId)

purchaces = query\_db(theSQL)

# **Manual testing**

My manual testing involved looking through the code. Whilst I was looking through, I found a few vulnerabilities.  
  
The passwords aren’t hashed or salted or encrypted in anyway making them easier to crack/brute force. As well as that if they are taken from the database, they are in the simplest form meaning attackers won’t need to take time decrypting or dealing with salts etc.

There is no timeout or limited attempts for logins, this makes the entire website prone to brute force logins since a malicious user could simply set up a bot to try out combinations of passwords/password list and emails until it gets one right and logs in.

@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")

When creating user accounts, there is no requirements for passwords, by having simple passwords it would allow for the account to be easily breached with a common password list.

@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"))

There is no proper authentication in this website, this can be seen in many different places such as creation, login, payment, updating settings. Whilst having a secure password maybe good having 2 factor authentication

When creating a new review there is potential for an XSS attack in the body section.

@app.route("/review/<userId>/<itemId>", methods=["GET", "POST"])

**def** reviewItem(userId, itemId):

*"""Add a Review"""*

*#Handle input*

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

reviewStars = flask.request.form.get("rating")

reviewComment = flask.request.form.get("review")

*#Clean up review whitespace*

reviewComment = reviewComment.strip()

reviewId = flask.request.form.get("reviewId")

app.logger.info("Review Made **%s**", reviewId)

app.logger.info("Rating **%s** Text **%s**", reviewStars, reviewComment)

**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,

)

## **Chosen Vulnerability**

The vulnerability that I have specifically chosen to talk about here will be the brute forcing password exploit, since they are quite prone to brute force attacks throughout this website. As well as that it is one of the biggest vulnerabilities out there especially with the fact that there is a great number of bots created to automate such a task. “Brute-force password guessing means using a random approach by trying different passwords and hoping that one work Some logic can be applied by trying passwords related to the person’s name, job title, hobbies or similar items.” In which it is also shown to be a very common attack that can be used by attackers. (Melnick, 2018)

In terms of how the attack can be done brute forcing an attack is where you get a single computer or perhaps more to quickly test combinations of passwords as well as usernames. This can be done by a variety of different methods such as using a password list that contains common or leaked passwords. You can also use an auto generator for this as well which will create all the different combinations and keep trying them until one works. This makes them a big concern because without proper encryption and security measures they will bypass and allow attackers full access.

There are many ways to prevent and mitigate such an exploit one being having a limited amount of login attempts; this can be done by logging an IP address and checking if it has tried a large number of login attempts within a short period of time. “Data analytics on such log data can not only identify sources of attacks, but also can be used to prevent future attacks—both at that institution and others.” (Rasch, 2018) This can then be flagged to the system and prevent that IP from logging in until it is resolved as to whether it was an attempted brute force attack.

Another method would be to use a captcha system which gives tests each time you try and login, as such “CAPTCHA offers protection from remote digital entry by making sure only a human being with the right password can access your account. CAPTCHA works because computers can create a distorted image and process a response, but they can't read or solve the problem the way a human must pass the test.” (Google, N/A) . Beyond this you can also implement two factor authentication having the user require a phone or email address which can have a code sent too it in order.

The final thing that could be improved is the password requirements themselves, this could be done by increasing the character limit for the password which will increase the amount of password variations that will need to be tried with alphabetic characters. To add to this, we can also increase the requirements for the type of characters, requiring the user to have upper and lowercase characters as well as numbers and special characters. This will only further increase the number of combinations that are needed and will be a good way to help mitigate brute force attacks.

In terms of which method would be the best to use here, it would be ideal to just use all of these together since they don’t interfere but rather help each other when all used. Though ultimately if a hacker was to manage to get a correct password, none of this would stop them from accessing the website. As well as that all these methods do have weakness such as changing IP addresses to regain ability to login, creating bots that can get through captcha etc. But regardless using such components to mitigate/prevent attacks is still the best practice.

## **Conclusion**

After looking at the website, I have identified quite a few vulnerabilities which do need patching in order to properly secure the website. Looking at it generally however a lot of things can simply be fixed by proper coding practices such as sanitising user input to prevent SQL or XSS attacks from happening. Though other glaring issues such as the issue with brute forcing will require more work in order to properly sort and should be dealt with first since they it is one of the biggest exploits on this website.

# Bibliography

AeoLogic. (2020, February 4th). *Automated Vs Manual Web Application Security Testing.* Retrieved from AeoLogic: https://www.aeologic.com/blog/automated-vs-manual-web-application-security-testing/#:~:text=Testing%20a%20large%20website%20manually,issues%20whi

Google. (N/A, N/A N/A). *What is CAPTCHA*. Retrieved from Google Workspace Admin Help: https://support.google.com/a/answer/1217728?hl=en#:~:text=CAPTCHA%20offers%20protection%20from%20remote,must%20to%20pass%20the%20test.

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

Melnick, J. (2018, May 15th). *Top 10 Most Common Types of Cyber Attacks*. Retrieved from netwrix: https://blog.netwrix.com/2018/05/15/top-10-most-common-types-of-cyber-attacks/

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/