Software Engineering 2 Report

100126186 and 100121797 and 100186938

1 Introduction

The Music Room web app aims to deliver a fully functional micro-blog with an emphasis on security and privacy without compromising on user experience and control. The application maintains tried and tested protection against: Cross-Site Scripting, SQL Injection, Account Enumeration, Cross-Site Request Forgery, Man-in-the-Middle, Database Overload, and Database Leaks. In order to ensure these protections function as intended future developers must ensure that they follow all procedures described within this documentation prior to developing new functionality or accessing the database. Furthermore, users still maintain the ability to format their posts and edit their account while the admin is capable of banning and unbanning users when necessary.

2 Coding Conventions and Architecture

The current architecture in place can be considered an MVC. Object classes include User, Post and DBController while static classes such as TokenHandler and EmailHandler contain a collection of specific related functions. The database was developed using SQLite3 and contains the tables “Users” and “Posts” with the union using the “UserName” foreign key.

Future developers should incorporate similar practices to those used to develop the software in its current state, dependent on the size of team maintaining the software. Agile development practices such as SCRUM and Test-Driven Development were put into use over the course of development while BitBucket was used to host the git repository for version control. Smaller teams should consider the advantages of paired programming which were extremely useful during the early stages of development when learning the flask library and in latter stages while attempting to eliminate bugs.

Furthermore, there is a strong emphasis on self-documentation of code in the current iteration of the software. This policy should be continued by future developers to ensure maintainability of the software. Readability of code is ensured through the use of comments where necessary and specific descriptive variable names. This is to make certain that there is little ambiguity to the purpose of a variable and its scope within the application. It is often tempting to use short variable names or acronyms in the pursuit of expediting work-flow however these will often lead to un-maintainable code and large delays in the future. For example, a variable which represents the hash of a newly entered and validated password may be named using acronyms however in order to follow the conventions currently in place the ideal name would be “hashedValidatedNewPassword” which offers great insight into the purpose of the variable.

3 Security

Security was a top priority in the implementation of The Music Room web app and should continue to be treated as such by future developers.

3.1 Cross-Site Scripting (XSS)

Cross-site scripting is one of the most significant vulnerabilities currently on the web. It facilitates a significant amount of successful theft of user data due to vulnerable web apps. Like most security vulnerabilities, if it can affect the users of a website then it will affect the website owner as the privacy and security of the users is the owner's responsibility. Furthermore persistent XSS, such as attacks made in comment boxes can break down the functionality of a website and make it behave in ways unintended by the developers.

3.1.1 How does it work?

HTML does not automatically distinguish between a less-than sign entered in text form and the same symbol used to denote an angled bracket used in tags such as "$< script >$". This means that when an attacker enters a block of code between the tags "$< script >$" and "$< /script >$" into an unfiltered form-entry/comment box the HTML will interpret it as a script instead of replacing it with text. One example attack would be entering the following into a reflective form entry field:

```
<script>alert(test)</script>
```

On a vulnerable website this will result in an alert box appearing on the website with the word "test", revaling the website's vulnerability.

3.1.2 How to defend against it?

There are several methods to defend against XSS. The first is a blacklist which doesn't allow users to enter specific inputs that are considered dangerous e.g "<". However, this also prevents users from using those characters for legitimate reasons, for example on a coding forum where someone would like to paste code to be read in text form or in a maths equation. It is better to have a filter in place which replaces the characters with their UTF-8 code so that it can still be displayed as text without being executed. In this case the user would enter "<" and it would be replaced by "&*lt*" in the HTML (OWASP, 2018a).

However, it's nearly impossible to blacklist every threat and only one threat needs to be missed for the filter to be vulnerable. Additionally, escaping all instances of "<" will prevent the use of harmless tags which users may want to use to format their posts.

Furthermore, HTML can also be interpreted when written in ASCII or hexadecimal. If a plain HTML attack does not work, it is as simple as copy-and-pasting the code into an ASCII or hexadecimal converter and trying again with the converted code. It is best practice to encode user entries in UTF-8 in order to avoid malicious code being disguised in this form.

3.1.3 Using the XSS Protection System

The Music Room Web App provides inbuilt protection against cross-site scripting. The protocol detailed within this documentation must be followed by future developers in order to guarantee the app stays protected. Currently there are 18 text entry fields available (including those available to the admin). These are located on the login/registration, newsfeed and profile pages and currently are protected. Any addition of a form entry must follow the same protocol as is followed by these fields.

Upon reading the user input from a form the data must be considered unsafe. Currently this is denoted by a comment next to the variable containing the data from the form which tags it as "UNSAFE". This must then be passed through the input validation and then overwritten by the return value of input validation. Only once overwritten by this function can the value be considered safe and should be tagged as such to avoid unintentionally passing the unsafe variable through any other parts of the application. At this point the value can be stored in the database or reflected back to the web page.

It is imperative that the inputValidation() function in the XSSProtection.py class is not edited unless there is a clear understanding of cross site scripting prevention and necessary reason to do so. This is due to the fact that the cross site scripting protection of the app is entirely dependant on this function and any vulnerability exposed within will affect the entirety of the application. The purpose of the input validation function is to remove the HTML tag "<" and instead replace it with the UTF-escaped version "&*lt*". However to ensure maximum functionality of the posting system a whitelist of certain tags has been implemented to allow users to format their text using safe HTML functionality. These tags are:

```
<b> <s> <strong> <i>
```

The tags have been deemed safe to use as they do not allow the insertion of any scripts or executable code. Our initial testing showed that this method protected against all forms of plain text XSS however was vulnerable to US-ASCII format XSS. This was later fixed by ensuring all user entered data is encoded to UTF-8 (whitelisting plain-text) and then escaped. Upon addition of the UTF-8 encoding, US-ASCII along with other forms of encoding such as hexadecimal were shown to be safe after further testing.

3.1.4 Testing

The current cross-site scripting resistance of the website has been tested heavily and has been deemed to be safe. The testing methodology, however, cannot guarantee that this method will always be safe as it is incapable of running all scenarios which may damage the website. By testing as many potential forms of attack as possible we can guarantee that the website is safe against all we have tested however can not guarantee that there is no way to expose vulnerabilities completely as there is no way of testing all possible attacks. The testing conducted, however, should enable future developers to be reasonably confident in the app's security.

The test methodology was as follows:

1. Verify every form entry has its variables passed through sanitisation.
2. Select a form entry field (the post form was chosen).
3. Enter all form entry examples given by "The XSS Filter Evasion Cheat Sheet" (OWASP, 2018b) noting any that performed anything other than simply printing the input as text.

Table 1 lists the results of a few significant tests performed (many more tests were also performed and deemed to not be a threat)

US-ASCII: When a US-ASCII encode attack was attempted it led to to the page rendering an error. This was determined to be due to the encoding used by the entry being recognised however not supported by the python version run on the website. While not performing the intended XSS action and producing an alert box, it was still a major vulnerability when used in a persistent XSS attack. If input into the post box on the newsfeed the result was that the newsfeed would serve up an error to all users and essentially break the website. The input validation function was revisited to encode all input to UTF-8 - whitelisting only plain text. Upon retesting it was deemed that this vulnerability had been removed.

3.2 SQL Injection

SQL has an inherent vulnerability which allows any string input to be misinterpreted as SQL logic when certain special characters such as a single quote ($'$) or double hyphens ($--$) are in

Table 1: XSS Testing

Test	Input	Expected Outcome	Actual Outcome	Fix
Plain text scripting	(See table notes)[a]	Prints the input in text form	Prints the input in text form	No Fix Needed
Escaping JavasScript escapes	(See table notes)[b]	Prints the input in text form	Prints the input in text form	No Fix Needed
US-ASCII encoding	(See table notes)[c]	Prints the input in text form	Internal server error	Applied UTF-8 encoding to all inputs

[a] `<script>alert("XSS")</script>`

[b] `\"alert('XSS');//`

[c] `<script>alert('XSS')</script> - (US-ASCII encoded)`

use. This allows for malicious code to be input into forms to fool an SQL query into performing an action that the developers did not intend such as gathering sensitive data, logging in without an account or even the destruction of the database.

3.2.1 How Does it Work?

The Most common way to exploit SQL is escaping the query string using a single quote (′) at the beginning of the input. Now whatever is typed is interpreted as actual SQL code. An attacker can use this method to create their own SQL queries and simply comment out the remaining code that would follow using double hyphens (−−). An especially devastating example being

```
" ' UNION DROP Users; --"
```

which, if typed into the correct form entry, could destroy the database by removing the User table entirely.

3.2.2 How to Defend Against it?

Paramaterised Queries: The main protection against SQL injection is the use of parameterized queries, however this method could not be used in the development of The Music Room web app. Parameterized queries pass raw data as opposed to strings in order to avoid inputs being misinterpreted as SQL code. Future developers should consider this method if it is allowed by the design brief.

Whitelisting/Sanitizing User Input: The Music Room Web app incorporates a whitelist sanitisation method to protect against SQL Injection. Alphanumeric characters have been deemed to be safe and are passed through to the database as strings however anything else such as a special character is converted into its respective hexadecimal value and delimited by curly braces ({ and }) which are used for decoding the data when read from the database. This is done by stripping away the delimiters and converting the hexadecimal value between the stripped delimiters back into its original character value.

Query Delays: By measuring the time taken to return a query attackers are capable of discerning the structure of the database which can then be used to launch more sophisticated attacks. By implementing a delay in query returns the application can negate the ability of attackers measuring specific query return times.

3.2.3 Using The SQL Injection Protection System

The Music Room Web App provides inbuilt protection against SQL Injection. Prior to storing or reading data stored in the database one must ensure that the data has been validated using the XSS protection protocol detailed above. The nature of SQL injection protection is multi-faceted and complicated so it is imperative that unless there is a full understanding of the protections necessary developers do no attempt to query the database unless using the specific methods developed with the intention of safe database querying. These are:

selectFromTable() - Allows a safe query to be executed which selects data from a table where a certain condition is met. Arguments taken are:

- cursor - the cursor created to access an instance of the database
- table - the table which is being selected from
- getVariableName - the column name which contains the variable being selected
- condition - the variable which is being checked in the condition
- conditionValue - the value that the condition variable must equal to be true

updateTable()- Allows a safe query to be executed which updates data in a table where a certain condition is met. Arguments taken are:

- updateVariableName - The name of the column which contains the variable to be updated
- updateVariableValue - The value that the variable will be changed to
- condition, conditionValue

deleteFromTable()- Allows a safe query to be executed which deletes data from a table where a certain condition is met. Arguments taken are:

- cursor, table, getVariableName, condition, conditionValue

These functions implement a delay which wraps the query execution as well as performing any necessary hexadecimal conversions for characters which are not white-listed. They should allow all forms of real-world queries to be performed easily and with the peace of mind that they do no expose the database to SQL Injection.

Should a developer need to make a custom query function it is imperative that the function performs all necessary protection measures found in the SQLInjectionProtection static class. Firstly the query must be wrapped in a delay. This is done by getting the time in seconds before a query executes using the function getCurrentTimeInSeconds() and storing it in a variable called preQueryTime. After the query has executed and before returning the result the function delayQueryOutput(preQueryTime) must be called. Additionally, prior to passing user input data to the database, it must be overwritten by calling the querySanisise(userInput) function.

Table 2: SQL Testing

Test	Input	Expected Outcome	Actual Outcome	Fix
Bypass login form	(See table notes)[a]	No Access Granted	No Access Granted	No Fix Needed
Vulnerability to sqlite3 Commands	(See table notes)[b]	No Change to Page	No Change to Page	No Fix Needed

[a] `OR 1=1;--`

[b] `UNION sqlite3_sleep(10);--`

3.2.4 Testing

The Testing methodolgy used to ensure protection against SQL Injection is as follows:

1. Ensure all database access incorporates the necessary functions in the SQLInjectionProtection class.
2. Attempt SQL injection on a field which selects from the database (the login fields) using a condition that is always true for example 1=1.
3. Attempt A Union to edit the database.
4. Repeat all steps using the SQL Injection cheat sheet (netsparker, 2018) to attempt less common attacks.

Using this methodology all SQL Injection attempts used were deemed to be nullified. As with all testing, however, this is not a complete guarantee against every possible attack. Table 2 contains a couple examples of the tests run.

3.3 Account Enumeration

Account Enumeration is the unintentional exposing of private user information through poorly thought-out error messages. For example a web app which produces an error message such as: "This email is already taken" exposes two vital pieces of information to an attacker:

- The owner of the email has an account with the website (information they may not wish to be exposed).
- The email is a valid email to send spam and phishing attacks.

To avoid this, error messages must be carefully considered to not expose anything that could be used to infer the validity of private information. Additionally a method of confirming user emails anonymously must be developed.

3.3.1 Protection

The Music Room web app ensures that any time a user wishes to create an account a confirmation email is sent to the address which is given. If the user is new they can click the link in the email and confirm their account. If the user already has an account (for example in the case that another user accidentally enters the wrong email) they can simply ignore the link which will have no effect if clicked. This means that error messages regarding the validity of private user information are not required making the web app protected against account enumeration attacks. Furthermore, the query delays - developed for the SQL injection protection - are dual purpose as they ensure attackers can not determine the validity of an input based on the query return time.

3.4 Cross-Site Request Forgery (CSRF)

CSRF is a vulnerability in which an attacker can send a user of a website malicious HTML which is edited slightly to execute an action automatically on loading. If the user session is logged in the vulnerability will potentially allow the attacker to force the user to unintentionally post comments or change crucial data such as their passwords.

3.4.1 Protection

The Music Room web app ensures that any action performed by the user when logged in is reliant on a specific user token which is passed from the last template the user was on.

This is done by rendering the current template with a token parameter generated by the setCSRFtoken() function. When an action is performed one of the inputs is the token in the template which is then verified using the verifyCSRFtoken() function. It is imperative that any future actions added to the web app follow this protocol and are wrapped by the process of setting the CSRF token and then verifying it prior to performing an action.

This system enables a user to execute an action such as changing their password while on the profile page as the token will be passed from the profile page template to the action. However, if sent malicious HTML which performs the action automatically the token will not be verified as correct - as the attacker has no way of knowing the tokens of other users - and the action will not be performed.

3.5 Man in the Middle

Man in the middle attacks are the ability of an attacker to intercept user traffic through methods such as setting up fake public WiFi networks. This can then be used to supply an unsuspecting user with an altered form of a website and steal private data. This is only one of many methods attackers may use to perform man-in-the-middle attacks. There are many potential ways a website can be exposed to this vulnerability such as using client-side sessions or an un-encrypted protocol such as HTTP.

3.5.1 Protection

The Music Room Web App currently does not support the encrypted HTTPS protocol due to the lack of a certificate however future developers must purchase a certificate and the enable HTTPS using the code and instructions contained within the app.py file. This is highly important prior to any future live implementation.

Furthermore, the current sessions used are contained server side and therefore do not expose themselves to client-side attacks such as man-in-the-middle. Attempting to decrypt the client side session, as was done in the testing of the app using an online tool (Petherbridge, n.d.), reveals no usable information. This will ensure that user sessions can not be hijacked by an attacker.

3.6 Database Overload (Spam/Registration Bots)

Spam bots are a risk to the user experience of the website as they greatly diminish the quality of the newsfeed. While not necessarily exposing a risk to user data and privacy they can result in the damaging of the brand of the web app. Additionally, they pose a risk of overloading the database which could either create a slow user experience or lead to the database running out of storage space. In order to conduct an attack like this all an attacker would have to do is run a script which posts repeatedly. Similarly an attacker could also create a script which repeatedly registers users in an attempt to overload the database.

3.6.1 Protection

There is currently an algorithm in place on the newsfeed page which detects spam-like activity. This is defined as a user account which creates more that 100 posts in 10 seconds. If this is detected as true, the user is immediately banned and cannot continue posting. Admins also maintain the ability to manually ban users. Furthermore, in order to prevent registration bots affecting the website any user accounts that are created are not committed to the database until they have been confirmed by clicking the link in the confirmation email.

3.7 Database Leaks

Sensitive user information such as passwords should not be stored in plain text. This is both in the interest of privacy, as user passwords should not be visible to the application administrator, and to protect user information in the event of a data breach. In order to ensure the safety of sensitive user data it must be hashed using an algorithm. However, simply hashing a password alone tends to not be much more safe than simply storing a password in plain text. This is due to the existence of "Rainbow Table" attacks (Ronacher, 2018). For a more secure method of storing passwords they must be hashed with the addition of a salt. Furthermore, the salt should not simply be concatenated but instead use HMAC.

3.7.1 Protection

The Music Room developers have understood hashing to be a highly complicated procedure so have opted to use the imported salted hash method in the werkzeug.security library. As soon as the password data is input it is hashed prior to being used for verification purposes or stored in the database. This is to ensure that at no point do the administrators of the application have the ability to see user passwords or handle them insecurely.

It is imperative that if future functionality requires the processing of password data the policy is maintained by using the following protocol:

If a user must input a password:

1. Read in user input data
2. Create a new variable by calling werzeug .pw_hash
3. Use the new variable for any password processing or storage

If the password must be read from the database:

1. Read the data from the database
2. To check if the hashed database password matches an input password call check_password() method passing through the hash and the plain text input

Table 3: Group K Testing

Test	Vulnerabilities Confirmed?	Level of vulnerability	Fixing Priority
XSS	No	None	None
SQL Injection	No	None	None
Account Enumeration	Yes	Low	Low
CSRF	Yes	High	High
Man-in-the-middle	Yes	Medium	Medium
Spam Bot	Yes	High	Low
Account Registration Spam	No	None	None

4 Testing Group K Web App (Elixir)

Table 3 is a brief summary of the security vulnerabilities of the elixir web app. The testing methodology used and the details of the vulnerabilities are expanded upon below.

4.1 XSS

The XSS prevention methods implemented have been verified as safe using the same testing methodology as was used in the testing of The Music Room Web App. The technique implemented successfully defends against all forms of XSS mentioned in the XSS Filter Cheat Sheet. However, a minor functionality oversight was discovered when attempting to format posts using harmless tags such as bold ($< b >$) which instead printed the tag. This plays no role in security. Overall, the vulnerabilities to XSS can be considered nullified by the protections in place.

4.2 SQL Injection

The implementation of SQL injection has been verified as safe through testing key areas which are known to be failure points for SQL injection protection algorithms. The same testing methodology as was used to verify The Music Room Web App was incorporated. No input resulted in a security flaw and there was an equal delay for all types of input in the fields tested. The elixir application can be considered safe by the testing standards of The Music Room.

4.3 Account Enumeration

Account Enumeration has been tested and verified as having only very minor vulnerabilities which may not need to be addressed. Upon entering a username which is already present in the database the login page serves an error message “Username Already Exists” - potentially exposing the identity of certain users who choose to use a username related to their identity. It is of note, however, that this is only a minor offence as the username is generally considered the public-facing part of a user’s account and there is no implied secrecy to a username upon registering. Had this error occurred upon entering an email which already exists then the issue would have been far more serious as private user information would be exposed.

Additionally, a minor functionality bug related to the username was discovered: when entering a username that already exists but in capitals, it still displays that the username exists when it technically doesn’t due to the capitalisation. Otherwise, Account Enumeration protection is good as the current handling of email confirmations seems to be ideal and there is no real way to determine if a specific email is registered on the app. When an already registered email is used the confirmation link is sent by default and no error messages appear.

4.4 CSRF

CSRF has been revealed to be a high-priority vulnerabilty that needs fixing immediately. Unfortunately, while it appears that CSRF protection is in place it functions based on a severe misunderstanding of CSRF prevention methods. The token used for CSRF prevention is defined specifically within the HTML and identical for all users. This can be exposed using the "inspect element" function of google chrome making it visible client-side for anyone using the web app. In order for CSRF tokens to function correctly each user must have a unique token. This means the app is not actually protected against CSRF as malicious HTML can be sent to a user using the CSRF token which is identical to all other users.

Furthermore, even if the issue of exposing the CSRF token in the HTML is resolved there is the additional issue of the web app using client-side sessions. While the sessions themselves are encrypted they can be easily decrypted using an online tool (Petherbridge, n.d.). This exposes the user ID and CSRF token and can be intercepted by an attacker using man-in-the-middle techniques. This allows an attacker to hijack a session and have access to the user's account.

4.5 Database Overload/Spam Bots

Due to only having a minor impact on security the ability to create spam bots is considered a low priority vulnerability, however should be resolved to not compromise user experience. Using the firefox extension "iMacros" a user can create unlimited spam posts in immediate succession. While this is not a risk to user privacy the user experience can be diminished greatly - especially through a coordinated attack. It appears that there is no ability to ban a user that is spamming nor any form of spam detection or prevention leaving the app highly vulnerable to this type of attack. Prior to going live, functionality to prevent this type of attack should be incorporated.

5 Conclusion

Overall, the development of The Music Room web app and the testing of the elixir app has revealed that even when extremely confident in the security of one's software there is still a very high likelihood that an oversight somewhere has led to a vulnerability. The high quality of the elixir web app and the effort put into development is clear yet vulnerabilities still exist. The Music Room web app has been tested within the abilities of the developers and confirmed as secure. However, as shown by the outcome of this report, third-party testing is invaluable and likely to reveal some vulnerabilities.

References

netsparker. (2018). *Sql injection cheat sheet.* https://www.netsparker.com/blog/web-security/sql-injection-cheat-sheet/. Author.

OWASP. (2018a). *Xss (cross site scripting) prevention cheat sheet.* https://www.owasp.org/index.php/XSS$_(Crosssite_Scripting)_Prevention_Cheat_Sheet.Author.$

OWASP. (2018b). *Xss filter evasion cheat sheet.* https://www.owasp.org/index.php/XSS$_Filter_Evasion_Cheat_Sheet.Author.$

Petherbridge, N. (n.d.). *Flask session cookie decoder.* https://www.kirsle.net/wizards/flask-session.cgi. kirsle.net.

Ronacher, A. (2018). *Flask snippets - salted passwords.* http://flask.pocoo.org/snippets/54/. werkzeug.