SQL Injection Safety on a Self-implemented Web-interface

Ester Aguilera, Jingru Dou, Kelsey Knowlson, Victoria Lien, Bryce Palmer, Emely Seheon, Lasair Servilla

May 2024

Abstract

SQL is a ubiquitous database management system for most online sites and many systems requiring data storage. Unfortunately, there are SQL vulnerabilities, called SQL injections, that can be exploited by malicious parties. The purpose of this project is to demonstrate though implementation a faux ticket sales website that utilizes a mySQL database, open-source relational database management system, the security risks as well as the preventative techniques that can be taken to block threats. A safe and unsafe state has been added to the site for experimentation purposes such that the effectiveness of password hashing with SHA-256, input sanitation and validation, and parameterized queries may be compared to a state lacking these security measures.

I Problem Statement

SQL is a ubiquitous back-end database management language that is used across a plethora of platforms to organize and store a variety of confidential information. This type of database management, while useful, carries the risk of a security breech by malicious parties using a method called SQL injection.

Due to SQL's pervasive use across a diverse range of industries from consumer shopping to student and employee databases, it is crucial to uphold robust security features to prevent possible exploitation.

The specific challenge at hand is to implementing an instance of a faux website utilizing a SQL database using mySQL for the purpose of testing the effectiveness of security measures such as password hashing with SHA-256, input sanitation and validation, and parameterized queries.

II Motivation

As of now, SQL injection is still a reliable and strong attack route that can compromise sensitive data by taking advantage of vulnerabilities in database-driven systems. This concern is raised as organizations increasingly rely on databases to store sensitive information; the exploitation of SQL injection vulnerabilities poses a significant risk to data integrity and system security. Because so many organizations rely on database systems to handle vital data, a successful SQL injection attack might have disastrous repercussions, from system failures to data leaks.

Through the implementation of strong input validation procedures and a thorough understanding of SQL injection vulnerabilities, this project aims to further the more general objective of improving cybersecurity safeguards in database-driven environments. Furthermore, with new advancements in this area, the potential impact of SQL injection attacks has increased, making it more important than ever for cybersecurity efforts to stay ahead of sophisticated threats.

III Contributions

Our goal was to design a realistic, yet simulated website, with queries to a SQL database for the purpose of testing safety measures that prevent against SQL injection. Our contributions towards this goal are as follows:

- We created a fully operational front-end of a website designed to sell event tickets.
- Our website features many of the typical pages one would expect to see in a live website such as a homepage, a contact us page, a FAQ page,

and a find tickets page. Additionally, functional login and create user pages were created.

- Using mySQL as our SQL platform, we successfully created and linked our SQL database to the website allowing us to store both user information and retrieve pre-stored ticket information though the use of the website's user interface.
- We created a safe and unsafe mode that can be toggled on and off throughout the session to compare and contrast the effects of SQL injections attempts with and without the implemented safety measures.
- We were able to secure the website using input validation, parameterized inputs, sanitizing inputs, and password hashing.
- Our website is able to accurately query for tickets using the 'find ticket' feature, as well as remove unavailable tickets, and track each user's purchases on their their account page.
- Through our testing to prove safety during safe mode, no SQL injection attacks were able to access secure information nor cause any unexpected errors in our websites functioning.
- When testing on unsafe mode, we realized that mySQL, the SQL database we used, has embedded security features that prevent against SQL injections. However, without the additional safety code that we added on the back-end of our website, numerous error messages were discovered and caused unexpected functioning of our website that was not present in safe mode. Due to this, our testing and premise are still valid since additional safety features blocked unwanted actions due to SQL injections, even if the database itself was not manipulated.

IV Methodology

In order to test for vulnerabilities in a legal manner we have implemented a simplified version of a web interface connected to SQL database that specifically utilizes MySQL. We have created a safe and unsafe version of the site to illustrate the effectiveness of standard safe guards as compared to an unprotected system. The standard safeguards include input sanitation, parameterized inputs, and password hashing.

To simulate a logical real world use of a SQL database in conjunction with a web interface we have implemented a ticket selling site that requires a SQL database to store available tickets to various fictitious shows, their price, location, and availability. Additionally the site retains user information such as username, password, user email, and previously purchased tickets. This approach leaves many possible attack points and methods that reflects the same risks active sites face. As the site was a planned attack surface we built in a reset ability to restore the MySQL database to the same start state. This reset is triggered anytime the user goes to the landing page 4.

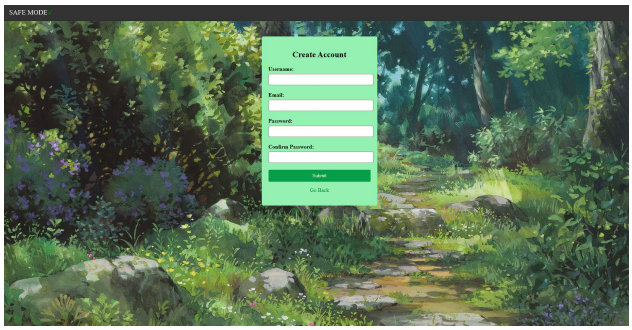


Figure 1: User Account Creation Page

There are two parts to our implementation: a back-end database supported by MySQL and a front-end way for users to interact with the website/database. The front-end section includes an account creation 1, user login 2, and a ticket purchasing method 3.

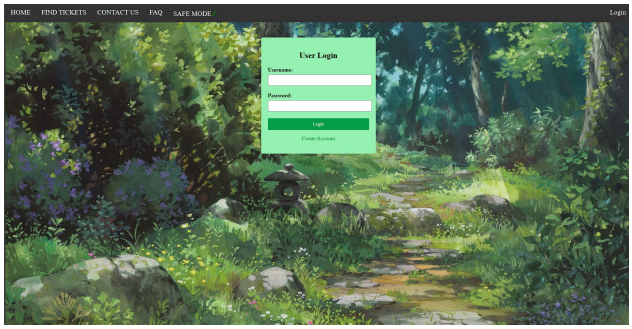


Figure 2: User Login Page

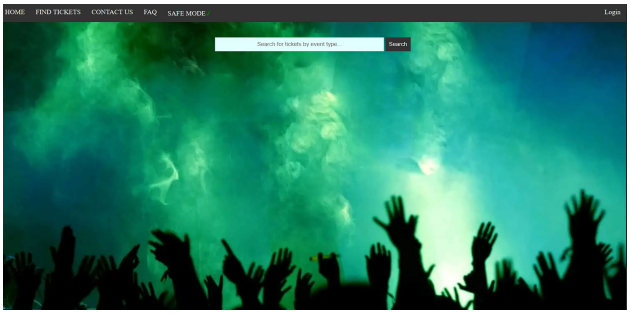


Figure 3: Search Tickets Page

A Setup

In order to use the website created to test SQL injections you will need both the flask application and the MySQL server. If desired use the following instructions for setting-up and running of the project.

A.1 MySQL Download and Setup

The following download instructions are for Ubuntu Linux systems, further instructions for other operating systems can be found at the URL: `https://dev.mysql.com/downloads/`

1. First download the MySQL APT repository at `https://dev.mysql.com/downloads/repo/apt/` selecting the correct package for your Linux distribution.

2. Install the downloaded package using the following terminal command:

```
$ sudo dpkg -i /PATH/version-specific-package-name.deb
```

3. Once the repository has been installed update the package information with the following command:

```
$ sudo apt-get update
```

4. Install MySQL with APT using the following command:

```
$ sudo apt-get install mysql-server
```

5. During installation you will need to provide a root user password. It is important that you either use our supplied password, **KIL0b!te!**, or change the 'password_personal' value in the 'connect_and_create.py' file in the flask application to your input password. If the passwords in both location do not match the flask application will not be able to connect to the MySQL server.

6. It is recommended that during installation you set up the MySQL workbench as it provides a much more user friendly environment for working with MySQL.

7. Further information on the download and installation process can be found at: `https://dev.mysql.com/doc/mysql-apt-repo-quick-guide/en/#apt-repo-fresh-install`

A.2 Flask Application

The flask application can be found in the zip file under our final submission, initiate setup by unzipping the files. The application needs multiple packages in order to run properly, known packages are listed below along with installation instructions.

- **pandas**

```
$ pip install pandas
```

- **mysql.connector**

```
$ pip install mysql-connector-python
```

- **flask**

```
$ pip install Flask
```

The above instructions for needed packages should keep the same package names if using a conda environment, however it is recommended that you double check. Also depending on your current environment setup you may need addition packages.

Once the needed packages have been installed run the flask application through your IDE by running 'main.py'. It can be run through your terminal as well, however we have encountered the occasional error running it this way. To access the website in your browser go to `http://127.0.0.1:5000/`. This will bring you to the landing page 4 for the project, click the link highlighted in blue on the page to proceed to the home page of the ticket purchasing site.

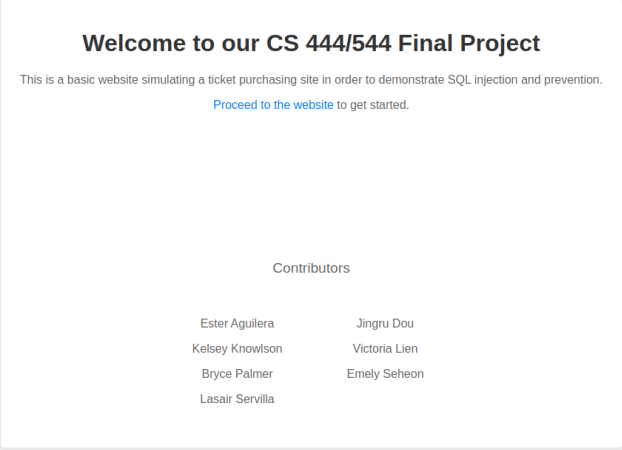


Figure 4: Landing page for the site

B Website Creation

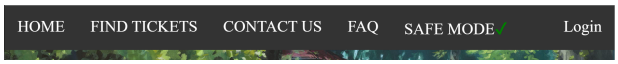
Creating a dynamic and user-friendly web-page involves several components working together in a seamless fashion. In this section we delve into each specific component. Each element plays a critical role in enhancing the website's functionality.

B.1 Navigation Tabs

The first HTML template that was coded included only the first four tabs with limited functionality. Later on, one of our developers implemented the Login utility along with it's corresponding tab. A simple HTML script for style and body was implemented to generate them. Navigation tabs and their utility include the following:

- HOME - This tab takes the user to the Home page, offering a serene welcome with familiar visuals and a brief overview of our website.
- FIND TICKETS - This tab navigates the user to the Find Tickets page where a new background is rendered along with the search bar.
- CONTACT US - This tab brings the user to a gateway for reaching out and connecting with our support team. Through a form submission, this space facilitates seamless communication for inquiries, feedback, or assistance.
- FAQ - This tab delve into the realm of Frequently Asked Questions by selecting this tab. Here, users can find answers to common queries, providing clarity and guidance on various aspects of our site's operations and services.
- Login - This tab navigates the user to the Login Page where a new background is rendered along with fields and buttons allowing the user to log in or create an account.

Although SAFE MODE is on the navigation bar, it does not navigate the user to a new page and has it's own separate functionality. SAFE MODE is discussed in detail in subsection E. A figure of the navigation bar is shown below.



B.2 Contact Us

The **front-end** portion of the contact form mechanism is straightforward and only contains an HTML form with a user information input field and a submit button. Users can enter their contact information into the input field and submit the form by clicking the "Submit" button. If there's an error message (for example, the invalid input format), it will be displayed on the screen. The request being used is shown in image below.

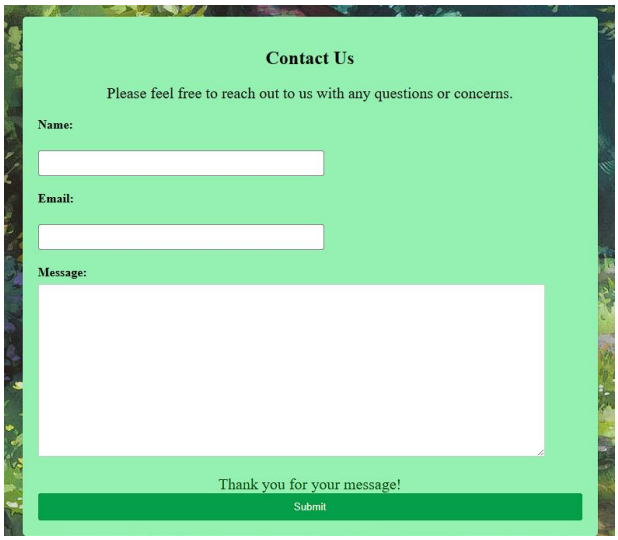


Figure 5: User Input Field

The *route-handling* within the **back-end** portion of the contacting mechanism, managed by contact.html, operates as follows:

1. When the user fills out the contact form and clicks the "Submit" button, the browser sends a POST request to the server with the form data (name, email, message).
2. In the Flask back-end, there should be a route defined to handle the POST request from the contact form. This route is typically defined in the Flask application code.

3. The Flask route associated with the contact form submission should specify the HTTP method as POST, and it should have a unique endpoint name.

4. Within the route handler function, the back-end code retrieves the form data sent by the client. This data includes the name, email, and message provided by the user.

5. Once the form data is retrieved, the back-end code can perform various actions, such as sending an email notification to the website owner, saving the message to a database for later review, or any other relevant processing.

6. After processing the form data, the back-end typically sends a response back to the client. This response can indicate whether the submission was successful or if there was an error. In the case of success, the response may include a "thank you" message or a redirect to a confirmation page.

7. On the client side, JavaScript may be used to handle form submission events, prevent the default form submission behavior (to avoid page reload), and provide visual feedback to the user, by clearing the form fields and displaying a "thank you" message.

8. Both on the client and server sides, error handling is implemented to handle cases where the form submission fails or encounters issues during processing. Error messages can be displayed to the user to inform them of any problems encountered.

B.3 Frequently Asked Questions

The **front-end** portion of the FAQ page is straightforward and only contains an HTML form. Users can basically just scroll up and down to check each FAQ boxes if needed. There shouldn't be any error generated in this section since all the data is uploaded already in the back-end, and We the developers are the only group who can change and edit all the FAQ questions and answers. The image of the FAQ page is shown below:

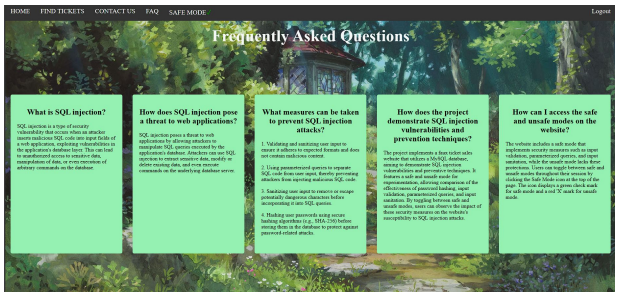


Figure 6: FAQ Example Pages

The *route-handling* within the **back-end** portion of the FAQ page editing, managed by faq.html, operates as follows:

1. The faq.html file contains the structure of the FAQ page, including the navigation bar, search bar, and FAQ content section. The FAQ items are dynamically generated based on the data provided by the backend.

2. The navigation bar contains links to different sections of the website, including the FAQ page. Additionally, the login/logout button is adjusted based on the user's session status.

3. The search bar allows users to search for specific FAQ items or tickets. However, the functionality for processing search queries and displaying search results is implemented on the backend.

4. The FAQ content section is where dynamically generated FAQ items are displayed. These FAQ items may be retrieved from a database or generated dynamically based on predefined questions and answers.

5. The JavaScript code at the bottom of the faq.html file handles the toggling of safe mode. When the user clicks the "SAFE MODE" button, an asynchronous XMLHttpRequest (XHR) is sent to the server to toggle the safe mode setting.

6. The XHR sends a POST request to the /toggle_safe_mode endpoint on the server. The server-side code processes this request and updates the safe mode setting accordingly.

7. Upon receiving a response from the server, the client-side JavaScript updates the appearance of the safe mode status indicator (safeModeStatus) based on the updated safe mode setting.

8. When the page loads (window.onload), the JavaScript code retrieves the initial safe mode

setting from the session and updates the appearance of the safe mode status indicator accordingly.

B.4 Search Bar

The **front-end** portion of the search bar mechanism is straightforward and only contains an HTML form with a search input field and a submit button. Users can enter their search query (based on "event type" such as "circus" or "soccergame") into the input field and submit the form by clicking the "Search" button. If there's an error message (for example, if no search term is provided or if the search query is unsafe), it will be displayed above the search bar. Search Bar being used is shown in image below.

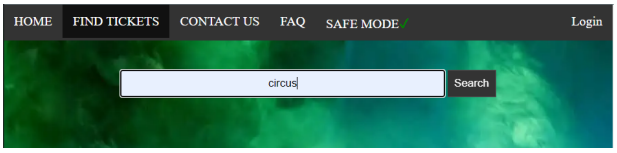


Figure 7: Search for **circus** events

The *route-handling* within the **back-end** portion of the search bar mechanism , managed by main.py, operates as follows:

1. When a user submits the form, the Flask route /findTickets handles the request and it distinguishes between GET and POST methods to determine which should be executed.

2. If the request's content type is JSON, it parses the JSON data for the search query and other event details.

3. It validates the search query, ensuring it's safe (strips unsafe characters) if SAFE MODE is active. It also checks for missing search terms or errors.

4. If the search query is valid, it calls the search_tickets function to retrieve matching tickets from the database.

5. If no tickets are found, an error message is set.

The *search functionality* within the **back-end** portion of the search bar mechanism, managed by query.py, operates as follows:

1. The search_tickets function connects to the database and executes a SELECT query to find events matching the search term.

2. The query filters events whose names contain the search term.

3. If SAFE MODE is enabled, the search term is sanitized to prevent SQL injection.

4. Finally, the function returns the matching tickets.

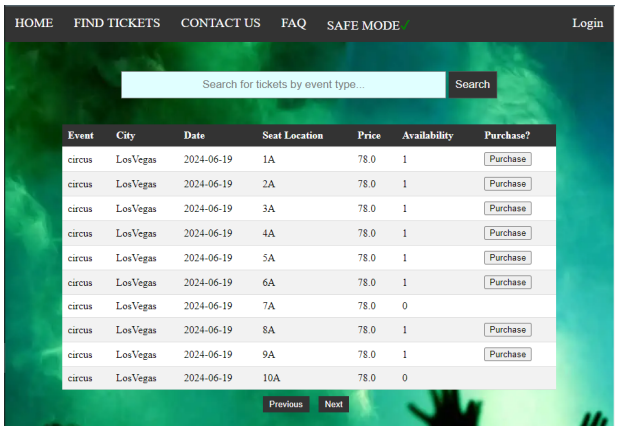


Figure 8: Search Results for **circus** event

B.5 Search Results Table

The **front-end** portion of the search results mechanism is designed using HTML and JavaScript. The HTML template includes pagination controls and dynamically displays search results within a table structure. JavaScript functions manage pagination, allowing users to move between pages of results seamlessly. The *loadResults()* function dynamically loads and displays results based on the current page, ensuring a smooth and response user experience. Pagination buttons are initially disabled or enabled based on the user's position within the result set, providing intuitive navigation cues. In summary:

- The pagination section includes buttons for navigating between pages of search results.

- JavaScript functions *prevPage()* and *nextPage()* handle the functionality of showing the previous and next pages, respectively.

- Pagination buttons are initially disabled/enabled based on the current page and total pages.

- Results are displayed in a table structure within the HTML template.

- JavaScript dynamically loads and displays results based on the current page.

- The *loadResults()* function calculates the start and end indices of results for the current page and adjusts the display accordingly.
- Rows of the table are hidden initially and then shown selectively based on the current page.

On the **back-end**, Flask handles the retrieval and processing of search results from the database. Upon receiving a request for search results, Flask routes the request to the appropriate endpoints, where it executes queries to fetch matching tickets based on the user's search criteria. The back-end ensures that the retrieved results are accurate and relevant to the user's query, filtering and formatting data as necessary before sending it to the front-end for display. Additionally, error handling mechanisms are in place to manage unexpected issues.

B.6 Purchase Button

The purchase button is used to integrate user interaction with the website's backend, allowing users to acquire tickets. Upon viewing the search results, users can initiate the purchase process by clicking the purchase button. The purchase button is only shown on tickets that are available for purchase, so that a user can not purchase a ticket that is not available. Once a user has purchased a ticket, the availability of the ticket becomes zero and the purchase button disappears. After purchasing the ticket, the user is taken to their home page, where they can view all purchased tickets, including the ticket they have just purchased. If a user tries to purchase a ticket while they are not logged in, they are taken to the login page where they can create an account or simply log in.

The **front-end** aspect of the purchase button is designed using HTML and JavaScript. On clicking the purchase button, a JavaScript function is triggered, which communicates with the server, facilitating the transaction without the need for page reloads. If the user is logged in, the button sends a POST request to the Flask route '/purchase', providing essential ticket details in JSON format. These details include the event name, city, date, and ticket location. If the user is not logged in, they are redirected to the login page.

In the **back-end**, Flask handles the purchase request, executing necessary updates to the database and the user's ticket inventory. The purchase function in main.py verifies the request's authenticity, confirming that the ticket data is provided and that the user is authenticated. Upon confirmation, it calls the update availability function in query.py to mark the selected tickets as unavailable and adds them to the user's ticket inventory. This process ensures data integrity and transactional consistency, providing users with accurate ticket availability information, and finally confirming a successful purchase.

In summary, the general flow of the purchase button is:

1. User Interaction: The user navigates to the Find Tickets page, searches for a ticket, and upon finding the desired ticket, the user clicks the Purchase button, which is only visible on tickets that are available for purchase.
2. When the user clicks the HTML button, a JavaSctipt purchase() function is triggered.
3. User Validation: The purchase() function first checks if the user is logged in. If they are not logged in, they are redirected to the login page to authenticate.
4. POST Request: If the user is logged in, the purchase() function constructs a JSON object containing essential ticket details including the event name, city, date, and ticket location. It sends a POST request to the Flask route '/purchase' with the JSON payload.
5. Flask handles the POST request sent from the client side. The '/purchase' route verifies the request, ensuring that the required data is provided and the user is authenticated.
6. Database Update: The update availability function is query.py is called, which updates the database, marking the selected ticket as unavailable.
7. User Ticket Inventory Updated: The purchased ticket is added to the user's ticket inventory by calling the add user ticket function in query.py.
8. Transaction Confirmation: The user is redirected to the home page, confirming the successful purchase.

C SQL Vulnerabilities

SQL vulnerabilities can include any user interface that allows the user to make requests of the SQL database. If proper precautions are not taken, the system is left vulnerable to attacker that attempt to manipulate queries and potentially gain unauthorized access to sensitive and confidential information. SQL injection attacks occur when an attacker injects malicious content or SQL queries through user input

into an application. These attacks have the potential of extracting sensitive data, manipulating or deleting existing information, and in some instances execute commands to the operating system.[6]

Some common approaches of detecting SQL injection vulnerabilities include:

- Testing with boolean conditions such as `OR 1=1`. Exploits in logical conditions are common SQL injection techniques.
- Testing with a single quote ' character. A vulnerable application may exhibit errors when a single quote is passed in user input.
- Testing with SQL specific syntax. SQL syntax, when executed, should return the same value as the original input. If it evaluates to a different value, it may indicate a vulnerable system.[8]

D MySQL

We have employed MySQL for constructing our SQL database. MySQL is an open-source relational database management system. Databases are data repositories for software that securely store information, ensuring the data can be accessed in subsequent instances with convenience. A relational database stores data into distinct tables rather than consolidating all information in a single table. This approach is optimized for speed and efficiency.

MySQL is cross-platform compatible, making it so that development and deployment are flexible and well-suited for diverse computing environments. It is also scalable, accommodating applications ranging from small to large. MySQL is optimized for performance, achieved by utilizing query optimization and caching mechanisms, ensuring the efficiency of database operations. It also incorporates security features to protect the confidentiality of data through features such as data encryption and user authentication protocols. Moreover, MySQL supports a plethora of programming languages, including C, Python, C#, and Java. This versatility will enable individuals to contribute effectively to the project despite having different programming backgrounds.[3]

E Safe Mode

A safe mode was created to allow for testing and comparison between code that is robust to SQL injection and code that does not have all the safe precautions implemented. The session cookie was set using a secret key and the status of the session's safety setting is stored in the session object, which is referred before each input. This method allows the safety status to be maintained across all pages of the site throughout the users session.The default setting upon opening the page is set to safe but the user may change the setting to unsafe at any time by clicking the Safe Mode icon at the top of the page. This icon is present and functional on all pages of the site9

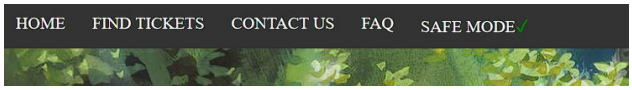


Figure 9: Safe Mode On

When the safety mode is turned off the user is alerted by a change in the icon of the Safe Mode button. When safe a green check will be displayed and when unsafe a red 'X' mark will be seen10. This status can be toggled on or off at any juncture during the user's session.

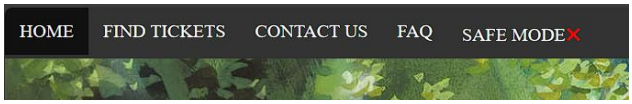


Figure 10: Safe Mode Off

When the safety mode is turned off no safety measures are in place, leaving the site vulnerable to SQL injection attacks. When the safety mode is turned to on, each input box will be protected from attack by input validation, parameterized queries, and sanitation of inputs.

F Prevention Strategies

In obstructing potential exploits against our SQL database, we have four preventive techniques: input validation, parameterized queries, input sanitation, and password hashing. The implementation of these strategies is crucial for ensuring the robustness of the security of our database and the potential mitigation of risks from SQL injection attacks.

F.1 Input Validation

Validating input is a necessary step. Preventing erroneous data from accessing the database and maybe causing issues with deeper-level components is its main goal. Improving security necessitates thwarting potential attacks as soon as user requests are handled. Input validation is a helpful method for detecting unauthorized input before any data is further processed by the software.

```
cursor.execute("SELECT * FROM users WHERE
    username= \%s", (username,) )
```

Figure 12: Parameterized Query Example

```
cursor.execute("SELECT * FROM users WHERE
    username = '" + username + "'")
```

Figure 13: Non-Parameterized Query Example

Syntactic and semantic input validation are necessary when it comes to validation procedures. The primary objective of syntactic validation is to ensure that structured input fields have the correct syntax, such as verifying formats of phone numbers and social security numbers. Conversely, semantic validation verifies that values are correct within the parameters of the specific business requirements. Verifying that a price falls inside the anticipated range or that the start date occurs before the end date are two examples of this.[12]

Specifically, this code must verify email address format so that it follows the standard `emailadress@serviceprovider.com`. To ensure this, regular expression are used in combination with pattern matching from the python library 're' 11.

```
r'^[a-zA-Z0-9._%+-]+@[a-zA-Z0-9.-]+\.[a-zA-Z
    ]{2,}$'
```

Figure 11: Email Pattern with Regular Expressions

F.2 Parameterized Queries

Parameterized queries are a security measure utilized to prevent SQL injection attacks in databases. In parameterized queries, parameters are utilized to denote user input instead of directly incorporating user input into query strings as in traditional SQL queries[13], and parameter values are supplied during execution.[11] By avoiding the direct incorporation of user input into the query, this allows for enhanced security and reduces risks of malicious attacks. This enables the database to distinguish between code and data, ensuring that the purpose of a query remains unchanged even if SQL commands are inserted by an attacker.[9] Furthermore, parameterized queries offer the advantage of being reusable and adaptable with dynamic data, where values are unknown until the statement is executed.[10]

Below is an example of a parameterized query that is utilized in username entry during safe mode 12.
In the example below, the unsafe alternative of the same code is displayed.

F.3 Sanitizing Inputs

Input sanitation involves disallowing certain characters and removing them before they are allowed to enter the SQL commands. Specifically, disallowed characters are stripped from the input via regular expressions before the string can be used within the a parameterized query. The characters that are disallowed are those with known functions in SQL such as `#` or `--`. [2] Specifically, the characters `"", , \, <>, [], #, -, / =` are not allowed for use in usernames or passwords when signing up and will be sanitized from user inputs in other text boxes. Additionally, inputs are checked for common SQL commands that would be unsafe for query such as DROP TABLE or DELETE.

```
    re.sub(r'[\^a-zA-Z0-9$\s$]', '', search\
    _query
```

Figure 14: Regular Expression Sanitizing

F.4 Password Hashing in MySQL

Password hashing is an essential step for securely storing user credentials. Since MySQL version 8.0, passwords may be stored stored in a hashed format using the SHA-256 algorithm with a plugin. This plugin allows for enhanced security as passwords are stored in a non-reversible format, making it difficult for unauthorized users to decipher passwords even if the database is compromised.

G Vulnerability Assessment

An emulated malicious attack and series of probing is carried out. The MySQL server is installed and run locally on a computer running MAC-OS Sonoma 14.4.1. The primary adversarial tools utilized in the attack are the cURL command library and Burp Suite developed by PortSwigger. The cURL command library which belongs to the Linux environment provides functionality allowing the transfer of data with URLs over various protocols including: HTTP, HTTPS, FTP, SCP, and SFTP. The prominent feature cURL provided in the assessment is the ability to modify GET and POST calls that would not be otherwise possible through a standard web browser.

Burp Suite is a comprehensive cybersecurity tool primarily used in web application testing. Burp Suite acts as a proxy between the user's browser and the target web application, allowing the user to intercept and modify web traffic. This is useful for analyzing HTTP requests and responses, identifying vulnerabilities, and testing security controls.

G.1 Scanning

The initial phase entails using cURL commands to manually send HTTP requests to the web application's endpoints. These commands are set up to contain particular information, consisting of the HTTP method (GET or POST), headers, parameters, and any other necessary information needed for the interaction to occur.

The responses which the web server generated in response to these requests are then recorded and carefully examined. This investigation includes a review of several elements, such as the HTTP status codes, response headers, and response body content. Finding any anomalies or signs of potential vulnerabilities is the goal here.In addition, a manual examination of the responses is part of the inspection process to find common vulnerabilities like directory traversal, SQL injection, and cross-site scripting (XSS) even if these vulnerabilities were not the primary focus of the project.

Utilizing a custom web browser by PortSwigger, interception is enabled within Burp Suite's "Proxy" tab to effectively analyze and manipulate HTTP traffic. The target web application is specified for focus scanning efforts. The spidering process is initiated within Burp Suite's "Target" tab to identify and catalog all discovered URLs. Progress monitoring of the spidering process is conducted within the "Spider" tab, displaying all subdomain URLs.

During browsing, Burp Suite automatically intercepts and logs all GET and POST requests for analysis as seen in Figure 15. Upon completion of the spidering process, scanner settings are configured within Burp Suite's "Scanner" tab. The scanning process is initiated, and progress is monitored within the "Scanner" tab. Upon completion of the scan, identified vulnerabilities are reviewed, and a detailed report summarizing the findings is generated using Burp Suite's reporting functionalities.

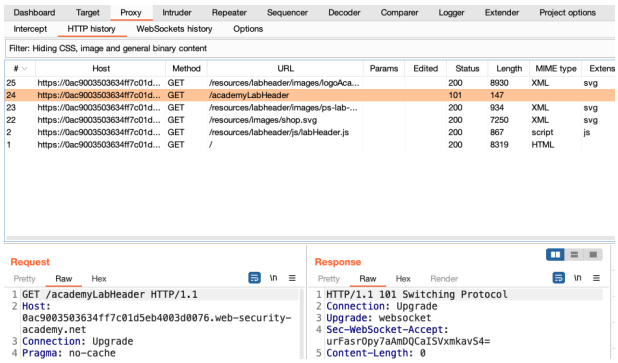


Figure 15: Burp Suite Proxy HTTP Records

G.2 Request Manipulation

All HTTP requests and responses sent back and forth between the browser and the web application are intercepted by Burp Suite and recorded. Burp Suite's Proxy tab displays these intercepted requests for inspection, enabling a thorough examination of the application's communication patterns.

Identifying target requests involves pinpointing specific HTTP requests associated with functionalities targeted for potential unauthorized access. These could be requests that deal with sensitive data operations, policies for access control, or authentication mechanisms. The parameters in these requests may change after the target requests are identified. Transferring intercepted requests to the Repeater or Intruder tools in Burp Suite allows for additional manipulation and analysis. For example, settings including user IDs or authentication tokens can be changed to try and gain illegal access to resources that are protected.

After the request parameters have been altered, Burp Suite sends the modified requests back to the web application. The response of the application to these modified requests is then monitored, looking for evidence of successful unauthorized access in particular. This might show up as unexpected application responses, circumvention of authentication procedures, or access to restricted resources.

The process is iterated for different functionalities or areas of the web application, enabling thorough testing of potential vulnerabilities. A comprehensive evaluation of the application's security posture is carried out by modifying request parameters and examining the application's responses in a variety of scenarios. This facilitates the identification and of potential risks.

G.3 Session Hijacking

Burp Suite is initially configured to operate as a proxy server. All requests and answers made to the web application are recorded by Burp Suite. A particular focus is on recognizing session cookies, which frequently have session identifiers like "sessionid" or "PHPSESSID." Once session cookies have been identified, Burp Suite is used to intercept and edit them. This could entail creating a new session ID entirely or changing the session ID value to a legitimate session ID that was obtained from another user.

The altered requests which now contain the tampered session cookies are then sent to the online application. Burp Suite makes it much simpler for these requests to be sent to the server, which may permit access while posing as a legitimate user. If the attempt to hijack an unauthorized session on the web application is successful, this will provide the opportunity to use the corresponding hijacked user's privileges to carry out actions inside the application.[5]

The use of stealth methods such as regularly deleting cookies from the custom web browser or using other forms of concealment can delay discovery allowing extended use of the hijacked account. It is important to survey the hijacked session to find any sensitive actions or data that could be leveraged. This culminates in the ability to take unapproved actions, extract private data, or elevate privileges within the application, depending on the level of access obtained.

G.4 SQL Injection

First, the input fields that are vulnerable to SQL injection are identified. These include a range of user-input areas that interface with the application's database, consisting of search bars and username login areas. A thorough examination of the input fields is carried out to determine how the application handles user input. Determining the presence of input sanitization or validation mechanisms intended to reduce SQL injection risk is a focus.

Customized SQL injection payloads are carefully constructed to target the particular input fields that were previously identified. The methods used can include using time-based or Boolean-based blind injection strategies, or appending SQL statements to input parameters.[6] The efficacy of the SQL injection payloads is evaluated by conducting them on relevant fields, such as the username field for attempts to bypass authentication or search fields for data manipulation attempts.

Particular focus is kept on how the application reacts to the injected payloads during this testing phase. The viability of the exploit is evaluated by closely examining signs of successful injection, such as error messages or unexpected application behavior. In order to guarantee a comprehensive assessment of the vulnerability, the testing procedure is repeated, continuously improving the SQL injection payloads and investigating a variety of scenarios. Several SQL injection methods and iterations are tried in order to maximize the probability of success.

Finally, all successful SQL injection exploits are carefully documented, including the injected payloads and how they impact the functionality of the application. To validate the results and present thorough documentation of the vulnerabilities found, supporting documentation in the form of screenshots and logs are included.

V Findings and Results

To our surprise, the unsafe mode of the site was not as simple to manipulate as previously expected. The first issue we found was the inability to pass the quote character ' into the input box without an error occurring. The following error page was produced when attempting the aforementioned:

```
ProgrammingError
mysql.connector.errors.ProgrammingError:
1064 (42000): You have an error in your
SQL syntax; check the manual that
corresponds to your MySQL server version
for the right syntax to use near
''root''' at line 1
```

Figure 16: Programming Error when Using Single Quote

This appeared to be an issue with the SQL database we utilized for the implementation of our website. The following code determined the MySQL version we were using.

```
import mysql.connector

# Connect to the MySQL server
connection = mysql.connector.connect(
    host='127.0.0.1',
    user="root",
    password="password",
    database="TestDatabase"
)

# Create a cursor object
cursor = connection.cursor()

# Execute a query to retrieve the MySQL
    server version
cursor.execute("SELECT VERSION()")

# Fetch the result
result = cursor.fetchone()

# Print the MySQL server version
print("MySQL Server Version:", result[0])

# Close the cursor and connection
cursor.close()
connection.close()
```

Figure 17: Code to Determine MySQL Version

We were able to determine that our database was running on MySQL version 8.3.0. MySQL disallows for single quotations to be utilized as input. A potential workaround to this issue is utilizing double quotes or escaping the single quote character using a backslash \. Unfortunately, we were still unable to break into the site's unsafe mode using this escape sequence. If input in the username field on the user login page or search field on the ticket search page, the backslash or double quote is considered part of the query string, and the query will not work.

Curiously enough, we found that we were able to manipulate the user input fields on the account creation page, and we were able to use the single quote character in the fields. We determined that this is likely due to the way the Python code is processing queries in the non-safe mode:

```
    "INSERT INTO users (username, email,
    password) VALUES ('{}', '{}', '{}')".
    format(username, email, password))
```

Figure 18: String Formatting Using .format()

This string formatting inserts user input values into the string query, allowing for special characters to be used, whereas for string queries for user sign in or searching a ticket, it uses string formatting such as the following in non-safe mode:

```
    "SELECT * FROM users WHERE username = '"
    + username + "'"
```

Figure 19: String Formatting Using Concatenation

This line of code directly concatenates a variable, which is a captured by user input, into the SQL query string. If the username variable contains a single quote character ', it will result in a syntax error in the SQL query, as the single quote character is not accepted by MySQL.

To circumvent the issue of an unaccepted single quote character, we attempted to insert encoded text in place of the apostrophe. We attempted methods such as utilizing URL encoding (`%27`), hexadecimal encoding (`0x27`, `\x27`), double URL encoding (`%2527`) in order to bypass input and validation filters. We also attempted to use characters similar to the apostrophe, such as backticks `, double quotes ", and a typographic single quote ‘ instead of the standard ASCII single quote. Unfortunately, none of these attempts worked in bypassing the string concatenation query.

This distinction in string formatting results in different processing of the single quote character. This single quote character indicates the end of a field in SQL, and it is utilized in many SQL injection attacks. Thus, due to how the fields were read in for the user account creation, we were able to perform SQL injection attacks on this website page, allowing us to manipulate the database connected to the website.

Through the account creation page on non-safe mode, we were able to manipulate tables within the database. However, we were unable to alter data within specific tables, such as adding new entries, modifying existing entries, and deleting entries.

The following are a few input that successfully demonstrated SQL injection on our website (all commands were utilized in the username field, although not a necessary requirement for a SQL injection attack on the site):

- `admin'; DROP TABLE events; --`

This creates a username of `admin`, and the single quote terminates the SQL string created from the username input. The semicolon denotes the end of

the SQL statement, and `DROP TABLE events` is another SQL query to delete a table of name `events` from the database. The two hyphens indicate a comment in SQL, which effectively comments out the rest of the query, meaning the email and password fields are not parsed in the query and a new account will not be created from this query.

- `admin'; CREATE TABLE hack (id INT AUTO_INCREMENT PRIMARY KEY, user VARCHAR(255) NOT NULL); --`

This creates a table of name `hack` into our database with columns `id` and `user`. The datatypes of these fields are specified as well. Similar to the previous command, the rest of the query following this portion of the string are commented out, and a new user will not be created.

These commands were successful in modifying tables in the database despite receiving an error page (see Figure 20) after submitting the form.

```
DatabaseError
mysql.connector.errors.DatabaseError:
2014 (HY000): Commands out of sync;
you can't run this command now
```

Figure 20: Error Page After Attempting SQL Injection

The following queries were unsuccessful SQL injection attack attempts:

- `root', --`

This was attempted in the user login page and inserted into the username field. Our database contains a default "root" user. The intent of this query is to log into the root user's account without a password. However, as mentioned above, MySQL does not allow for the single quote character with the string concatenation query. Variations of this, including obfuscation and character encoding, were unsuccessful.

- `admin'; INSERT INTO events (index, event, city, date, ticketLocation, price, availability) VALUES (253, 'h@ck', 'Albuquerque', '2024-05-02', '66A', 0.0, 1); --`

This was attempted on the user creation page in the username field. The intent of this query was to insert a ticket entry into the `events` table in our database. However, we found that we were unable to modify specific table entries and only the table itself, such as deleting an entire table or inserting a new table.

- `admin'; DELETE FROM users WHERE username='root'; --`

This was attempted on the user creation page in the username field. Like the previous query, this query was an attempt to modify specific entries with the database. This attempts to delete the `root` user from the `users` table. However, similar to the previous attempt, this was also unsuccessful, showing us that we were simply unable to modify table entries using SQL injection attacks.

The above SQL injection attempts were also attempted on the site's safe mode. Due to the usage of input sanitation and rejection of specific special characters using regular expressions in safe mode (see Figure 14), including the single quote character, these requests were deemed inappropriate or incorrect. For example, passwords rejected certain special characters. Inputting a single quote character in the username field would simply lead to the single quote being read as part of the username, and no error would occur. However, it could not be used as part of a SQL injection attack.

Due to the multiple failed SQL injection attacks, we tried other methods of accessing the site through dishonest means. Using PortSwigger's Burp Suite, we were able to monitor `POST` and `GET` requests when clicking through website pages. See Figure 21 for a visual representation of the information and interface provided by Burp Suite.

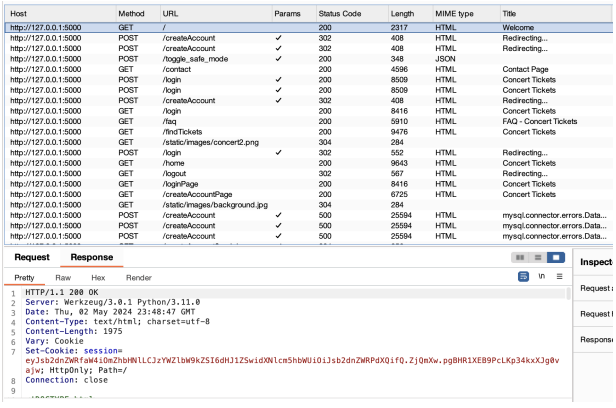


Figure 21: Burp Suite Interface

The "Request" tab provided us with information regarding the session cookie, and it displayed us

information about user input fields in plaintext. In other words, if a user were to log into their account, we would be able to view their password in plaintext (see Figure 22). The "Response" tab let us view the HTML of the site pages (Figure 23). The HTML allowed us to view parts of code from the backend.

Figure 22: View Session Cookie and User Input

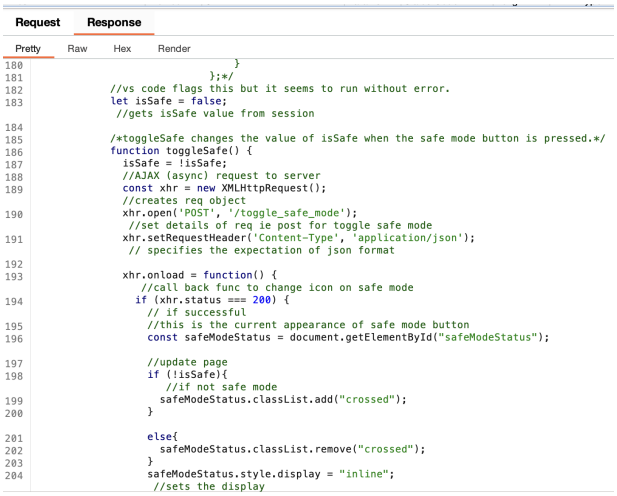


Figure 23: HTML in Response

Although this project focused on SQL injection attacks and preventative measures against injection techniques, the above issues led us to surmise that there are many vulnerabilities beyond SQL injection attacks that need addressing, even in the site's safe mode. Weaknesses include:

1. **Exposure of sensitive data:** Visible user input in plaintext suggests a broader issue of inappropriate handling of sensitive input, making it so that sensitive data, including passwords, is exposed and making users vulnerable.

2. **Cross-Site Scipting (XSS):** User input fields are not properly sanitized, making it so that bad actors can insert malicious code and scripts in the input fields other than SQL injection attacks. This can lead to the theft of session cookies and session hijacking.[1]

3. **Insecure Direct Object References (IDOR):** "Insecure Direct Object Reference (IDOR) is a vulnerability that arises when attackers can access or modify objects by manipulating identifiers used in a web application's URLs or parameters. It occurs due to missing access control checks, which fail to verify whether a user should be allowed to access specific data."[4] Because of the ability to view some backend code from the HTML, attackers may attempt to manipulate data and resources, leading to unauthorized access attempts.[7]

These concerns imply flaws with authentication and authorization. Attackers may be able to bypass authentication or gain unauthorized access to functionalities of the site.

To demonstrate some of these security risks on both safe mode and nonsafe mode, we firstly attempted to use cURL commands from the terminal to gain access to the site. Figure 24 shows a command that utilized some SQL injection to bypass the password field by commenting it out, but due to the setup of the website, it was unable to find one of the variables needed and I was unable to successfully get this and variations of the command to get through to the site. This resulted in `KeyError: 'safemode'`, with safemode being a boolean variable indicating if the site was site to safe mode or non-safe mode.

```
curl -X POST -d "username=root' OR 1=1 --
&password=pass" http://127.0.0.1:5000/login
```

Figure 24: cURL Attempt with SQL Injection

Our next attempt involved a user account we created called Tommy, and we attempted to login using help from Burp Suite's "copy as curl command (bash)" from a login attempt on Tommy's account. This creates a `POST` request. This was to simply test if we could log into an account using a cURL command.

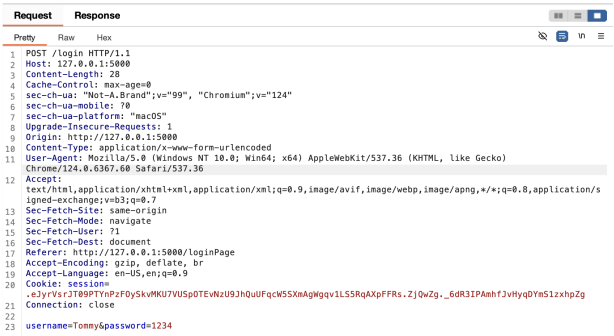


Figure 25: Request to Log Into Tommy's Account

Figure 26: cURL Attempt to Log Into Tommy's Account

This generated a cookie for Tommy's account and indicated it was possible to use cURL commands to access this website without any errors occurring.

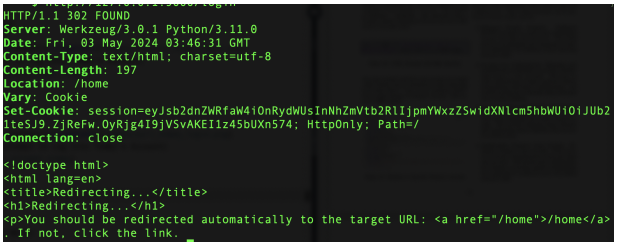


Figure 27: Get Tommy's Cookie

This led to our last attempt with cURL, which was to determine if we could log into an account without knowing credentials. Given that a user's session cookie can be accessed from a GET request, we attempted to log into Tommy's account simply using his cookie. When we visited Tommy's account on the homepage, we were able to obtain his session cookie (see Figure 28). Using Burp Suite, we were able to acquire a cURL command from the GET request to his homepage as seen in Figure 29.

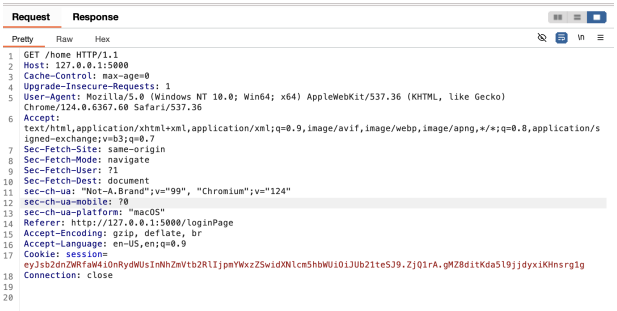


Figure 28: Access Tommy's Cookie

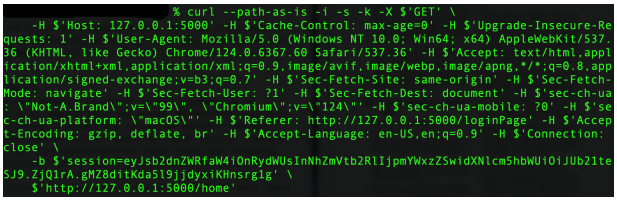


Figure 29: cURL from Burp Suite Request

Figure 30 shows that we were able to successfully access Tommy's homepage without logging in with his credentials and using a session cookie instead. This demonstrates that if an adversarial attacker were to gain access to a user's session cookie, they are able to access their account without using credentials to log into their account.

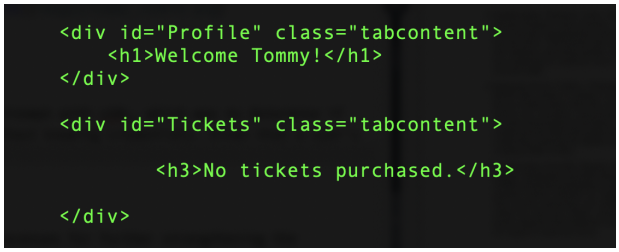


Figure 30: Successful Access without Credentials

VI Future Work

This section outlines potential avenues for further strengthening the security posture of our website. While the current implementation focuses on mitigating SQL injection vulnerabilities through the website's safe mode, there exist several opportunities to improve the overall security framework of the site. We have found several security issues in our site, including that it is not protected against man in the middle attacks, packets, POSTs, and requests can be observed, the submission fields are sent in plain text, and the cookie information allows users to log in as a different user. These things and more can be protected in future works, such as:

- Enhanced Security measures: Implement HTTPS encryption to secure data transmission between the client and the server. This would prevent eavesdropping and tampering of sensitive information by malicious actors observing network traffic.

- Securing User Authentication: Enhancing user authentication mechanisms to prevent unauthorized access to user accounts. This may include implementing stronger password policies, incorporating multi-factor authentication, and securely storing user credentials using techniques such as salted hashing.

- Protecting Session Management: Implementing secure session management techniques to prevent session hijacking and session fixation attacks. This includes enforcing session timeouts

and regenerating session identifiers after successful authentication.

- Implementing Transport Layer Security: Deploying Transport Layer Security (TLS) protocol to encrypt communication between the client and the server. TLS ensures data confidentiality and integrity, protecting against eavesdropping and tampering by malicious actors intercepting network traffic.
- Enforcing Secure Cookies: Utilizing secure and HTTP-only cookies to enhance cookie security. Secure cookies ensure that they are only transmitted over encrypted connections, mitigating the risk of interception by attackers. HTTP-only cookies prevent client-side scripts from accessing cookie data, reducing the likelihood of cross-site scripting (XSS) attacks.
- Implementing Cross-Site Request Forgery Protection: Integrating Cross-Site Request Forgery (CSRF) protection mechanisms to prevent unauthorized actions initiated by forged requests. This involves generating and validating CSRF tokens for each user session, ensuring that requests originate from legitimate sources and are not forged by malicious actors.

VII Group Member Contributions

A Project Lead: ***Lasair Servilla***

Areas of contribution:

- Written portion of updates: For the written portion of updates I worked on the methodology section as well as my own contribution section.
- Written portion of final report: I read over and fixed any typos in the final report as well as added my contribution. I also wrote the setup guide under methodology.
- Presentation: I worked on the methodology section of the presentation alongside another team member. I specifically worked on the slides: Methodology, Website Layout, both Walk-through slides, and all three Behind the Scenes slides.
- MySQL setup: I determined what MySQL service we were going to use for this project and ran through the initial set up to get it working locally on my computer. After I guided my team and informed them on what they would need to do in order to set it up for themselves.
- GitHub: Created the GitHub repository for the project and shared it with team members.
- Website initial setup: I created the base flask site we were going to use for the SQL injection attacks. This includes creating the file 'main.py' with a empty home page and setting up the file structure so that flask is able to correctly find any HTML files and image files needed for the site.
- Website connection to database: With the basic site set up I added functionality to connect to the MySQL database. As a part of this I created CSV files that would remain static and could be used to create the tables within the database each time the user returned to the landing page of our site. I set it up in this way so that any damage done to the MySQL database during our attacks could be easily reversed. The tables set up at this time were the users and events.
- Website user account creation: The next contribution I made to the site was adding in user account creation functionality. This included the front-end page that the user sees when creating an account as well as the back-end that connects to and edits the database accordingly. At this point it only had parameterize user input and password hashing for safety measures as adding other measures fell to another team member.
- Website user login: As a part of the user system I also added the login system. This again includes both the front-end page that the user sees and the back-end work that connects to and checks the database. This also includes just parameterized user input as a starting point.
- Password hashing: I created the functionality to use password hashing as a safety measure.
- User session management: I added user session tracking to the functionality of the website. This means that once the user is logged in, it is known by the system that there is a user logged in and who the user is. It also allows the user to logout when desired.
- User ticket purchases: I added the functionality to the ticket purchases that adds the ticket to the current users record. This way the user purchasing the ticket has that ticket associated with them in the database, and it will remain apart of their record until the system in reset.

- User home page: I created the front-end and related back-end of the home page that is shown when there is a user logged in. This displays a welcome message with the username under the 'profile' tab and a table showing all the tickets the user has purchased under the 'your tickets' tab. If the user has not bought any tickets yet it also says so.

- Group meeting: Lead an in person learning session where I reviewed the code written for the user account creation and login. This allowed me to explain how it all worked between the flask application and the MySQL platform, and answer any questions that team members had.

- General Meetings: Participated in group chat and contributed to online team meetings.

- Troubleshooting team questions: Given I am the only team member with extensive experience in this area I tried to answer questions that came up during other team members contributions. Including but not limited to explaining the structure of the project setup, to helping with issues connecting to the MySQL database.

- Setting up team goals: I played a strong role in setting up team goals and determining what the final site would look like. As mentioned I have a history working on this type of project, so it is due to this that I did so. It was agreed on as a group that I would act as a kind of leader in the technical aspects of this project.

B Full-stack Developer: ***Ester Aguilera***

Areas of contribution:

- **Written portion of Updates**: I worked on the problem statement (for first update) and motivation (for second update). For the first update, I wrote the roles of each team member on our document based on what we decided as a group. Additionally for the second update, I specified my contributions.

- **Written portion of Final Report**: I wrote the following topics within the Web Creation subsection of the Methodology section: Search Bar, Search Results Table and the majority of Navigation Tabs. Additionally, I wrote the introductory paragraph of the Web Creation subsection of Methodology. I also specified my own contributions to the project.

- **Presentation Slides**: I worked on the Motivation slides and Search Functionality slides within Methodology section.

- Researched and learned basics of SQL injection vulnerability through the use of research papers and cybersecurity web articles.

- Learned the basics of web development through free online tutorials, concentrating primarily on the functionality of website user interfaces by learning HTML, CSS, and JavaScript, with a secondary focus on server-side components.

- **Initial HTML Template**: I wrote and designed our initial HTML template with the following **navigation tabs**: HOME, FIND TICKETS, CONTACT, and FAQ. Note that I only wrote the content of the FIND TICKETS page. Other developers wrote the content of HOME, CONTACT and FAQ.

- **Search Bar**: I wrote the front and back-end portions of the search bar mechanism. Specifics are described within the Web Creation subsection of Methodology.

- **Search Results Table**: I wrote the front and back-end portions of the search results mechanism. Specifics are described within the Web Creation subsection of Methodology.

- **Team Organization**: At the beginning of the project I setup a discord group chat for our team to communicate and a one-drive folder for our team to store relevant project documents. Throughout the project, I coordinated and coled the majority of our meetings usually with Lasair leading the discussion on the technical side of things while I focused on leading the discussion on task delegation and recording agreed-upon goals and decisions for the team to reference. Outside of meetings, I played an active role in initiating and participating in discussions for setting goals, assigning tasks and generally just staying on track.

C Front-end Developer: ***Jingru Dou***

Contributions towards Paper and Presentation:

- Developed detailed descriptions for the navigation tabs on the website, focusing on FAQ, Contact, and Home pages.

- Utilized HTML and CSS to design visually appealing and intuitive navigation elements, optimizing usability and accessibility across different devices and screen sizes.
- Provided insights into the achievements and areas of improvement identified through the first update, offering recommendations for future iterations.
- Delivered a comprehensive overview of the project's conclusion during the presentation, emphasizing the major accomplishments, insights gained, and lessons learned throughout the development lifecycle.
- Conducted thorough research and analysis to gather pertinent information and insights to support the arguments or claims presented in Section B including B.2 and B.3.
- Designed visually appealing and informative slides that effectively communicate the diverse skill sets, expertise, and efforts of each team member in contributing to the project's success.

Research contributions:

- Conducted an in-depth investigation into various techniques and best practices for securing a website against SQL injection attacks, which are among the most prevalent and damaging security vulnerabilities.
- Delved into the fundamentals of HTML, the standard markup language used to create and structure web pages, to contribute effectively to front-end development efforts.
- Learned the core HTML elements, attributes, and syntax conventions for creating semantic and accessible web content, ensuring compatibility across different browsers and devices.
- Conducted a deep dive into MySQL, exploring its functionalities and intricacies to gain a comprehensive understanding.
- Acquired knowledge and comprehension of web design principles, gaining an understanding of the essential elements of web development methodologies. Explored the interplay between front-end and back-end technologies, covering a diverse range of tools and programming languages such as HTML, JavaScript, Python, Flask, and others.

Coding contributions:

- Designed and developed the FAQ.HTML pages to present the collected information in a clear, organized, and user-friendly manner, ensuring easy access and comprehension for website visitors.
- Developed the Contact.HTML pages to include essential contact details, email addresses and contact forms, allowing users to submit inquiries, feedback, or support requests conveniently.
- Collaborated with team members to troubleshoot and resolve a coding issue related to MySQL passwords, leveraging expertise in database management and security protocols.
- Organized the FAQ content into logical categories or sections, making it easy for users to navigate and locate specific information based on their interests or needs.
- Developed engaging and informative content to populate the Contact pages, providing users with comprehensive details on how to get in touch with the website administrators or support team.
- Enhanced the functionality of the main.py file by incorporating code to handle HTTP requests and responses for serving HTML pages dynamically to users.

D Safety Tester/Hacker: ***Victoria Lien***

Contributions towards Paper and Presentation:

- Contributed to and wrote "Findings and Results" sections for all updates and final report. Certain sections incorporated in "Methodology" were findings taken from previous reports with some altered text. Included citations from resources utilized.
- Created slides related to "Findings and Results" in presentation.
- Revised formatting and fixed errors in LaTeX documents.

Research:

- Researched SQL injection and common SQL injection techniques, SQL injection prevention strategies (input validation, parameterized queries, etc.), SQL databases (mainly MySQL), and password encryption in MySQL to create findings sections across both updates.
- Several SQL injection lab and training websites were investigated in advance of the project web

application's penetration testing, including Altoro Mutual, OWASP WebGoat, OWASP Juice Shop, PortSwigger Web Security Academy, and PentesterLab.

- Researched HTTP requests in Python.
- Researched web vulnerabilities, such as XSS and IDOR, other than SQL injection to determine potential website weaknesses.
- Researched tools to assist with web development and security testing, including cURL and Burp Suite.

Project Contributions:

- Designed and carried out comprehensive penetration testing on web application and MySQL server. Includes initial scanning, HTTP request analysis, request manipulation, session hijacking, and SQL injection. Utilized software such as cURL and Burp Suite to ensure thorough testing coverage. Focused on exploiting and identifying SQL injection vulnerabilities to gain access to MySQL server, but additional focus was placed on XSS, CSRF, and IDOR vulnerabilities. Identified areas in which code contributed to or prevented SQL injection attacks.
- Text log outputs, HTTP query logs, and screenshots taken while interacting with the web application through a browser during penetration testing.
- Penetration testing identified flaws in the system's code, prompting comprehensive enhancements to bolster security. During this process, vulnerabilities identified in the testing, such as non-parameterized queries and unencrypted HTTP Post requests were pinpointed and addressed. Through these code revisions, the system security was increased vastly, safeguarding both the web application and server data against potential threats.

E Full-stack Developer: ***Kelsey Knowlson***

Contributions towards Paper and Presentation:

- Wrote Abstract
- Wrote Problem statement on final paper and previous all previous updates
- Set up overleaf project for papers
- Wrote Safe mode section in methodology
- Revised and contributed to user account creation, login, and search page sections in methodology.
- Revised SQL vulnerabilities section in methodology
- Revised and contributed to input validation section in methodology
- Revised and contributed to parameterized input section in methodology
- Wrote sanitizing input section in methodology
- Wrote contributions section of paper
- Stylized slides
- Wrote Problem statement slide
- Wrote Behind the scenes-User login slide
- Wrote Safe mode slides 1-3 titled: (Safe Mode, Safe Mode Features, and Safe Mode Applications)

Research contributions:

- Researched SQL safe measures to secure site
- Researched how to use and query with SQL and mySQL
- Researched html to be able to contribute to front end and stylistic choices of the site
- Researched methods used in SQL injections to prevent against such approaches

Coding contributions:

- Researched and implemented safety methods to protect from malicious attempts to gain private information
- Helped fellow team members debug a coding issue involving mySQL passwords and streamlined existing code by setting up absolute paths and a global password.
- Created the Safe Mode by storing and recording session data. The safety mode involved adding the button to the user interface, creating icons for the safety button, and coding a toggle function that switches between the two icons and changes the boolean value of the safety mode within the session when pressed.

- I added the safety parameters for all categories such as parameterized inputs, sanitized inputs, and input validation. This involved a check to the session safe mode boolean before moving forward on any user input, including username and password on the Login page and search inputs on the Find Ticket page.
- On the Create Account page, username, email, and both passwords were not only parameterized, sanitized, and validated by me but I also placed limitations on character use was implemented to avoid creating usernames or passwords with dangerous side effects.
- I created a plethora of user messages to inform the user of their error or which safety mechanism they had triggered should they use a disallowed word or character. These messages were created at each user input field and only show during while the Safety mode is turned on.
- I did not implement the SHA-256 password hashing but did utilize it within my own code and changed the manner in which it was called.
- To contrast the safe mode, I implemented an unsafe mode that can be used for testing and as a control for how effective my safety strategies proved to be. I also created the unsafe version for storing emails, passwords, usernames, and making all queries.
- While it did not fall directly under my tasks, I also added and stylized the Find Ticket page but did not handle the functionality of the searches done on the page.

F Safety Tester/Hacker: ***Bryce Palmer***

Contributions towards Paper and Presentation:

- Contributed to and wrote "Findings and Results" sections for all updates and final report. Additionally, wrote "Vulnerability Assessment" subsection in "Methodology". Prior sections incorporated in "Methodology" were findings taken from previous reports with some altered text.
- Added to "Methodology" slides in presentation related to vulnerability assessment and contributed to "Findings and Results" slides.
- Identified several notable and well respected resources in the field of cybersecurity and amended them to the bibliographies of all updates and final paper.

Research Contributions:

- Researched SQL vulnerabilities, countermeasures, and SQL servers providing "findings" section in all preceding updates.
- In preparation for penetration testing of the project web application several SQL injection lab and training websites were explored including: Altoro Mutual, OWASP WebGoat, OWASP Juice Shop, PortSwigger Web Security Academy, PentesterLab
- Investigated preventive measures, such as the use of parameterized queries and input sanitation protocols, and surveyed the plethora of SQL injection techniques. The goal of the thorough investigation was to comprehend the subtleties of SQL injection vulnerabilities and come up with a comprehensive penetration test for the web application and corresponding server.

Project Contributions:

- Designed and carried out comprehensive penetration testing on web application and MySQL server. Includes initial scanning, HTTP request analysis, request manipulation, session hijacking, and SQL injection.
- Recorded results of penetration testing in form of text log outputs, HTTP query logs, and screen shots interacting with the web application from a browser.
- Extended the purview of the penetration testing by utilizing a wide array of tools including cURL, WireShark, PortSwigger's Burp Suite, SQLMap, and OWASP ZAP. While several of these provided redundant features, each of which was a learning experience.
- After conducting penetration testing, which revealed vulnerabilities within the system's code, thorough revisions were made to enhance its security. This process involved identifying and addressing specific weaknesses discovered during penetration testing including non-parameterized queries and unencrypted HTTP Post requests. By implementing these code revisions, the system was made robust removing these vulnerabilities thus safeguarding the web application and server data.

G Full-stack Developer: ***Emely Seheon***

Reports Contributions:

- Section IV B.6: Purchase Button on the final report
- Section VII: Future Work on the final report
- Methodology: Purchase Button Functionality: Front-End slide of the final presentation
- Methodology: Purchase Button Functionality: Back-End slide of the final presentation
- Future Works two slides of the final presentation
- Section 4: Objectives of project proposal
- Section 6: Expected Results of update one
- Section 5: Expected Results of update two

Research Contributions:

- I conducted a deep dive into MySQL, exploring its functionalities and intricacies to gain a comprehensive understanding.
- Gathered Information on SQL Injection and Similar Attacks on Web Servers. Undertook a rigorous exploration of security vulnerabilities, particularly focusing on SQL injection and other potential threats to web servers. This involved extensive research to comprehend the nuances of SQL injection attacks, as well as similar exploits like the notorious Heart Bleed vulnerability.
- Learned and Understood Web Design Practices. Learned the fundamental aspects of web development practices, which include the relationship between front-end and back-end technologies. This encompassed a broad spectrum of tools and languages, including HTML, JavaScript, Python, PostgreSQL, Flask, and more.

Code Contributions:

- Designed and developed the HTML structure and styling of the purchase button, ensuring it aligns with the overall theme and user interface of the website.
- Implemented client-side JavaScript functions responsible for handling user interactions upon clicking the purchase button, including validation checks and asynchronous communication with the server.
- Created JavaScript functions to construct JSON payloads containing essential ticket details such as event name, city, date, and ticket location, ensuring accurate data transmission to the server.
- Integrated user authentication checks within the purchase button functionality, verifying user login status before processing purchases.
- Programmed the purchase Flask back-end route to handle POST requests from the client side, including route validation and request parsing to extract ticket data from JSON payloads.
- Developed server-side validation logic to verify the authenticity of purchase requests, ensuring that only authenticated users can initiate transactions and that all required ticket details are provided.
- Implemented database update operations within the purchase route handler, utilizing SQL queries to update ticket availability status and insert purchased tickets into the user's ticket inventory.
- Incorporated error handling mechanisms to manage exceptions and edge cases during the purchase process, providing informative feedback to users and ensuring graceful degradation in case of errors.
- Collaborated with front-end and back-end team members to integrate the purchase button functionality seamlessly into the overall website architecture, ensuring consistency and coherence across all user interactions.

References

[1] KirstenS. *Cross Site Scripting (XSS)*. 2024. URL: https://owasp.org/www-community/attacks/xss/ (visited on 05/02/2024).

[2] Mihai Neagu. "Sanitizing inputs: Avoiding security usability disasters". In: *SQL Shack* (2019). URL: https://www.sqlshack.com/sanitizing-inputs-avoiding-security-usability-disasters/.

[3] Oracle. *What is MySQL?* 2024. URL: https://www.oracle.com/mysql/what-is-mysql/ (visited on 02/25/2024).

[4] OWASP. *Insecure Direct Object Reference Prevention Cheat Sheet*. 2024. URL: https://cheatsheetseries.owasp.org/cheatsheets/Insecure_Direct_Object_Reference_Prevention_Cheat_Sheet.html (visited on 05/02/2024).

[5] OWASP. *Session hijacking attack*. 2024. URL: https://owasp.org/www-community/attacks/Session_hijacking_attack (visited on 05/02/2024).

[6] kingthorin OWASP. *SQL Injection*. 2024. URL: https://owasp.org/www-community/attacks/SQL_Injection (visited on 02/25/2024).

[7] PortSwigger. *Insecure direct object references (IDOR)*. 2024. URL: https://portswigger.net/web-security/access-control/idor (visited on 05/02/2024).

[8] PortSwigger. *SQL Injection*. 2024. URL: https://portswigger.net/web-security/sql-injection (visited on 02/25/2024).

[9] OWASP Cheat Sheet Series. *SQL Injection Prevention Cheat Sheet*. 2024. URL: https://cheatsheetseries.owasp.org/cheatsheets/SQL_Injection_Prevention_Cheat_Sheet.html (visited on 02/25/2024).

[10] GE Digital Solutions. *Parameterized SQL Queries*. 2024. URL: https://www.ge.com/digital/documentation/historian/version72/c_parameterized_sql_queries.html#:~:text=The%20benefit%20of%20parameterized%20SQL,SQL%20queries%20for%20each%20case. (visited on 02/25/2024).

[11] SQL-Server-Team. *How and Why to Use Parameterized Queries*. 2019. URL: https://techcommunity.microsoft.com/t5/sql-server-blog/how-and-why-to-use-parameterized-queries/ba-p/383483 (visited on 02/25/2024).

[12] Cheat Sheets Series Team. *Input Validation Cheat Sheet*. 2024. URL: https://cheatsheetseries.owasp.org/cheatsheets/SQL_Injection_Prevention_Cheat_Sheet.html (visited on 02/25/2024).

[13] LUKAS VILEIKIS. *Parameterized Queries in SQL – A Guide*. 2023. URL: https://www.dbvis.com/thetable/parameterized-queries-in-sql-a-guide/ (visited on 02/25/2024).