COS20031 Individual Portfolio

Evaluation and Enhancement of Database Security for Online Property Marketplace

Name: Audrey Vanessa Chee Wan Tai

Table of Contents

Table of Figures

Introduction

Project Background

Recently, due to the innovations in the use of advanced communication technology, property markets have made it easier for users to make property purchases, sales or rentals through the internet. The Online Property Marketplace is a website intended to perform these functions, offering clients the opportunity to manage property in a more convenient manner. Ther registration and login module has been developed by a local software development company which is essential for account security and user access management. Before it integrates to other developer modules, the company intends to check the level of security of this specific module especially on SQL injection attacks that are among the common and most dangerous threats to web applications. This project specifically deals with aspect of registration and login module with a view to establishing the gap in security that leads to vulnerability in compromising database security.

Problem Statement

SQL injection attacks remain the most threatening attack in the list of web application attacks, according to the OWASP studies. These attacks are most used in web applications as attackers can alter the queries in the SQL and gain control over all the users' usernames and passwords, including other personal data. In the case of the Online Property Marketplace, this type of vulnerability may result to serious violation, including erasure of users' information, negative impacts on the confidence of the users hence affecting the company's legal and financial responsibilities. Since the registration and the login module deals with user credentials as well as overseeing the access to the system, protecting the system against SQL injection is crucial to protecting both the user data and the overall structure of the website.

Project Requirements and Expectations

The purpose of this project as a cybersecurity contractor is to analyse the registration and login module of the program and specifically search for SQL injection attacks. This project will involve:

- Execution of vulnerability study to enhance the identification of SQL injection and other security weaknesses.
- Strengthening security measures, input validation check, parameterized queries and other good industry practice recommendations.
- Keeping records, especially, the basic tests, employed tools, discovered weakness and applied remedies.
- Confirmation, after testing, that security has been enhanced after the fix has been done.

The expected result of this project is a secure registration and login module which will guard the database against SQL injection attacks and similar risks to give the basis for future implementation of the Online Property Marketplace application.

Literature Review

Preventing SQL injection attacks in web applications, more precisely during login and registration is a crucial issue in protecting online environments. SQL Injection attack on databases is aimed at web front ends containing faulty data validation on web residing components such as CGI scripts. These loopholes have within the last few months been exploited by attackers to post messages on e-commerce sites, and capture usernames and passwords by phishing, among other things, populating sensitive information like credit card information (Balasundaram & Ramaraj 2011). This review looks at the traditional methods of defending against SQL injection attacks in terms of ways of protecting data and data access points such as authentication measures.

SQL injection attack occurs when an attacker injects legitimate SQL statements into the vulnerable fields of the website such as form fields. These attacks are most successful when the input validation and sanitization mechanisms in the web applications are not effective, thus enabling the attackers to exploit the authentication mechanism to make them perform arbitrary database operations (Balasundaram & Ramaraj 2011). This systematic view has been highlighted by Balasundaram and Ramaraj (2011) who have pointed out that the major mitigation measure is enhancing the authentication procedures using the encryptions, via Advanced Encryption Standard (AES) to encrypt the username and the password. This technique guarantees that, even in case an intruder gains a direct access to the database, encrypted credentials are effectively unusable since the true credentials stored in the database are encrypted, thus minimizing the effects of the SQL Injection attack by securing the authentication data that attackers try to inject.

Besides the use of encrypted credentials, input validation remains part of the best practice to avert SQL Injection. This requires meticulously scanning the inputs coming from the end user for any form of pattern or character that might pose a threat before the information is processed by SQL query. Evaluating the characterization of prepared statements and parameterized queries, the user inputs and values passed to the SQL queries are treated as data and not as instructions; thus, resisting the risk of SQL Injection (Halfond et al. 2006).

Another potential solution is that of ensuring creation of secure authentication plugins for example the Secure Login and Registration Plugin for WordPress. Awad and Dhabi (2024) explain that it uses One-Time Password (OTP) and machine learning towards strengthening the mechanisms of log in and registration. The OTP system that employs PHPMailer guarantees that each login attempt is verified via the email, even if the login credentials get into the wrong hands. Furthermore, the plugin conducts automatic locking for 24 hours of a particular user's account after the second failure in the login authentication process. This lockout mechanism also helps to prevent brute force attack and notify administrators about these activities.

Furthermore, the plugin uses PHP-ML to employ machine learning to strengthen security features as threats are detected, thereby using a flexible method. Data mining techniques also help in selection of classes based on outliers in log-in attempts that helps the system to better cope-up with the ever-changing techniques of attacks. This dynamic approach when supported by the plugin's auto generated security headers as well as improved input sanitisation offers a layered defence against both S SQL Injection and Cross-Site Scripting (XSS) attacks which strengthens the extent of the login process.

Methodology

Before Implementing Fixes

Manual SQL Injection

1. Entering **random' OR 1=1 -- '** into the email parameter and a **dummy password**

Figure 2: Inserting random' OR 1=1 -- ' with a dummy password into login.php

Figure 1: Successful login into dashboard.php

When **random' OR 1=1 -- '** is inputted in the email field, it alters the SQL query. Particularly, the injected part of the query transforms it into a regular condition which is always TRUE (OR 1=1). In SQL, -- is used to comment out rest statements of the query which means that everything after it is not taken into consideration. The query will be seen as:

```
SELECT * FROM userlist WHERE email='random' OR 1=1 -- ' AND pass='dummypassword';
```

As 1=1 is always true, the query selects all the records from the 'userlist' table and omits the check for the password. For instance, the attacker can log into the system without having to know the real password and this makes them impersonate the first user in the user list database (or any user depending on how the session is managed).

2. Entering **audrey@gmail.com' -- '** into the email parameter and a **dummy password**

Figure 4: Entering audrey@gmail.com' -- ' and a adummy password into login.php

Figure 3: Successful login into dashboard.php

In this case, when audrey@gmail.com' -- ' is entered, the injected part (--) comments the rest of the query in the SQL query like the previous method. The query becomes:

```
SELECT * FROM userlist WHERE email='audrey@gmail.com' -- ' AND pass='dummypassword';
```

This comment allows the password check to be bypassed which essentially means that if the email audrey@gmail.com is in the system, the query will bring back the user record without checking the password. Hence, the attacker can log in with any password if the media exists in the database, which makes it quite easy to go around the authentication system.

3. Testing with another email, **megan@gmail.com' -- '** into the email parameter with a **dummy password**.

Figure 6: Injecting megan@gmail.com' -- ' with dummy password in login.php

Figure 5: Successful login to dashboard.php

By injecting megan@gmail.com' -- ', the query becomes:

```
SELECT * FROM userlist WHERE email='megan@gmail.com' -- ' AND pass='dummy';
```

The -- comment syntax guarantees that everything that follows is not parsed by the database and, therefore, the password check is skipped all together. If the email megan@gmail.com is in the database, the query will list the record of that user, and the attacker can log in without inputs of password. This is a result of sanitation and validation of the user inputs which results in accessing the application unauthorized.

Automated SQL Injection with ZAP

Step 1: Include the POST:login.php()(email,login_button,password) in a new context and rename the context to OPM Login.

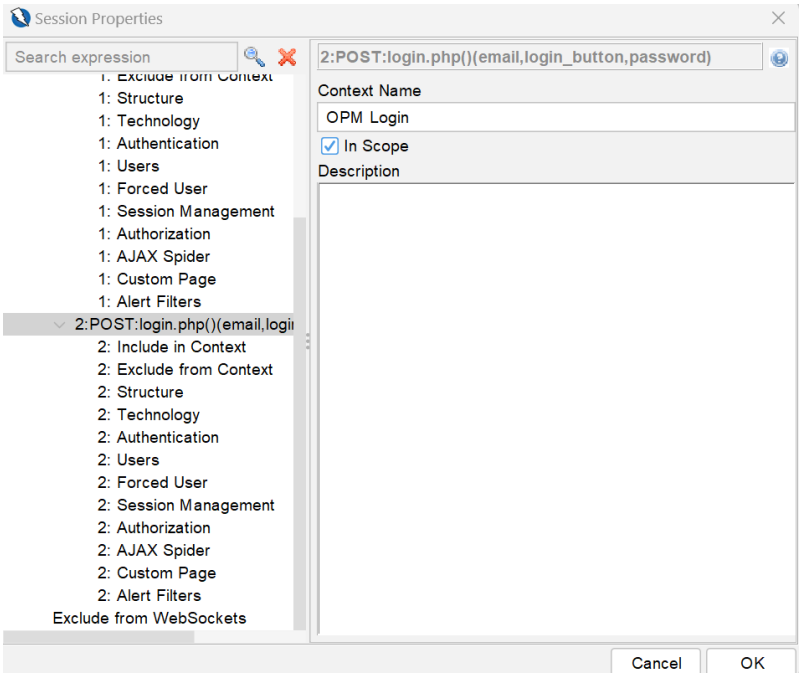


Figure 7: Including and Renaming Context

Step 2: Go to Authentication section and select Form-based Authentication and fill the rest of the following according to the image below.

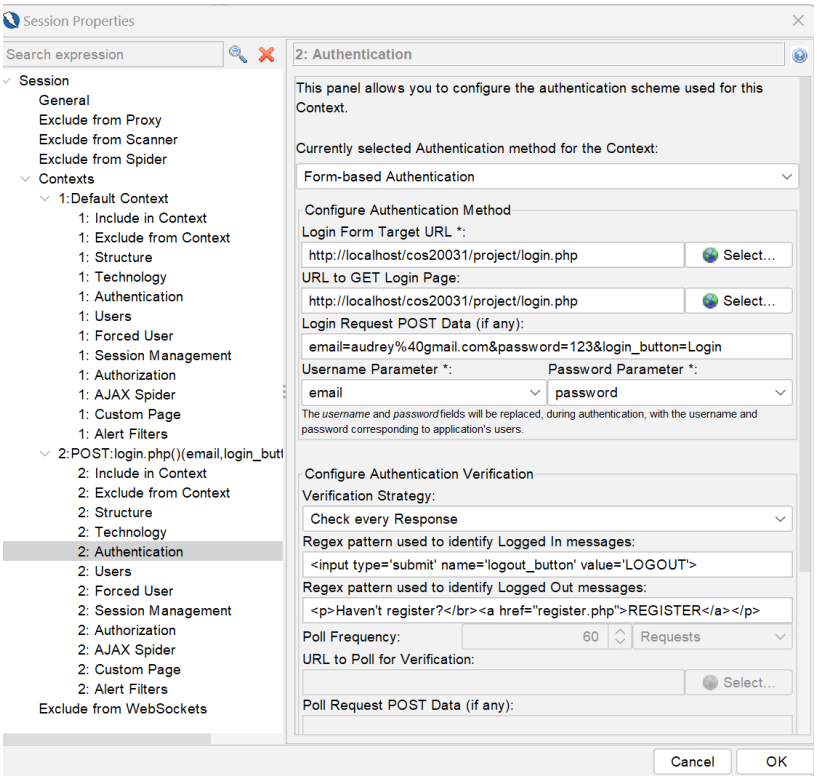


Figure 8: Filling Authentication section

Step 3: Add a new user at the Users section based on the registered username and password on the web application.

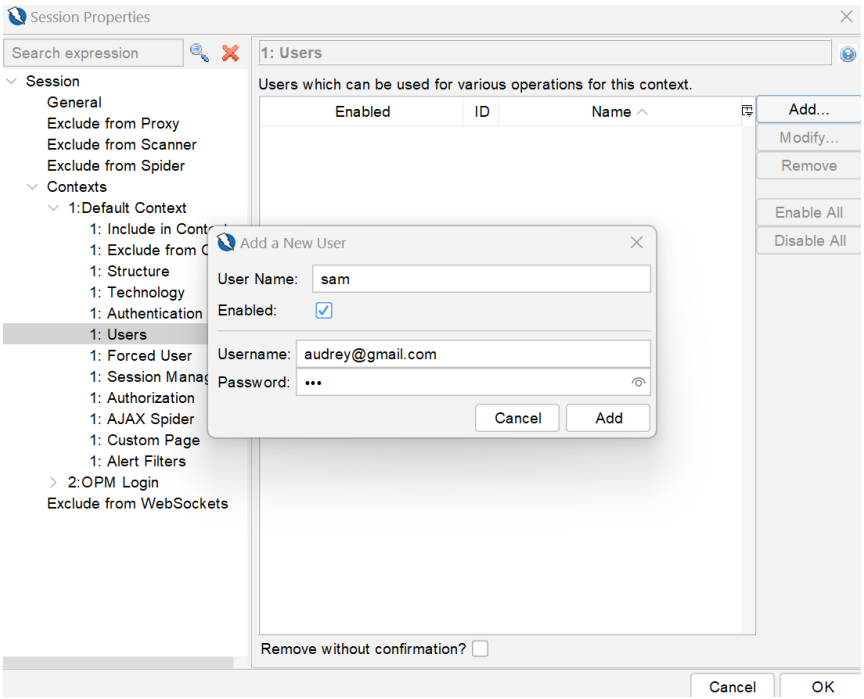


Figure 9: Entering a registered user and password

Step 4: After flagging OPM Login: Form-based Auth Login Request, include OPM Login in context for the base URL which is project.

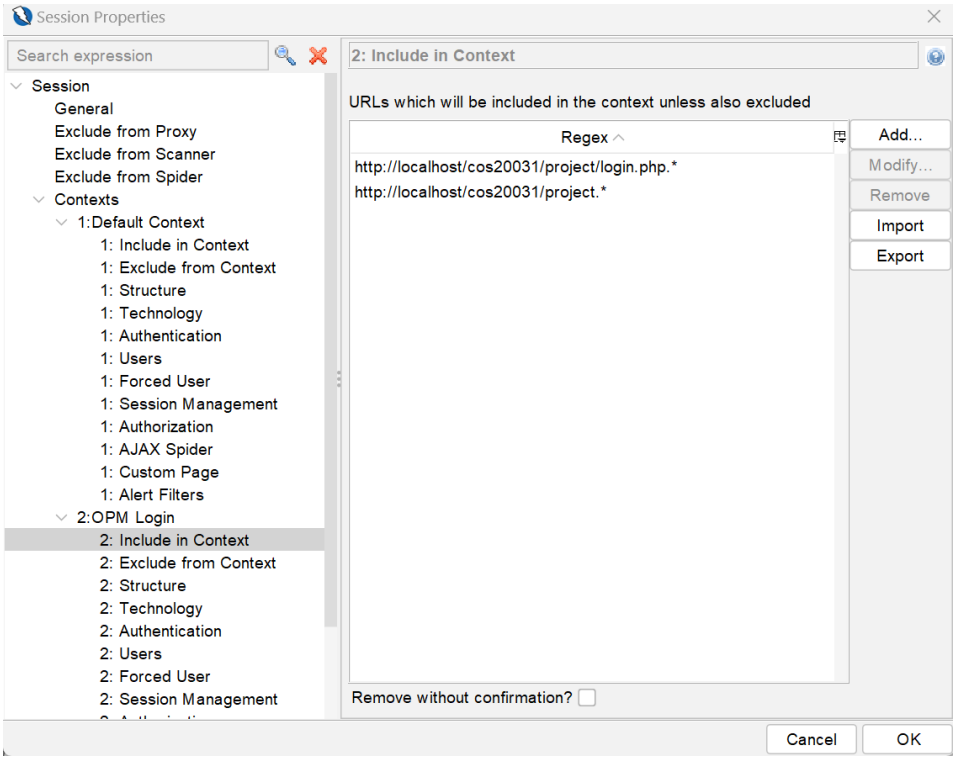


Figure 10: Including context into the base URL

Step 5: Persist the session and enable Forced User Mode.

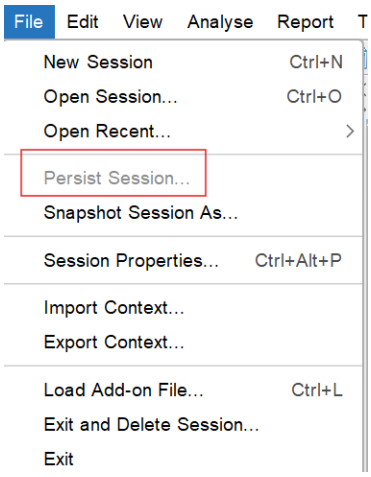


Figure 12: Persist the session

Figure 11: Enable Forced User Mode

Step 6: Start the attack on the base URL (project) by selecting Active Scan. Here, 14 SQL Injection alerts were found as shown in the image below.

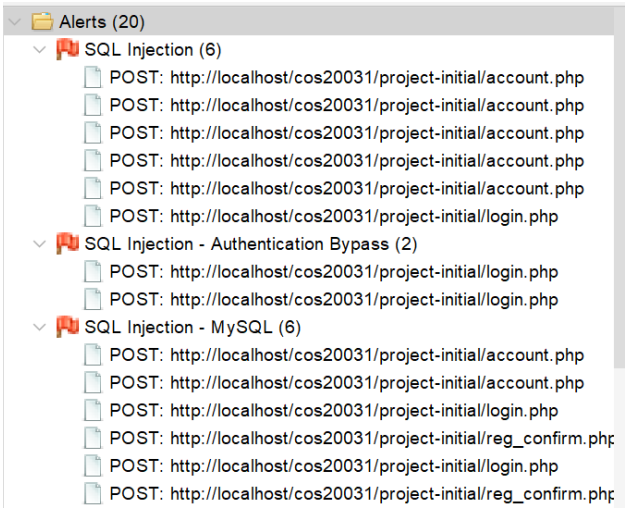


Figure 13: Alerts Found in OWASP ZAP

Explanations of the alerts:

1. SQL Injection in the fname and lname parameter

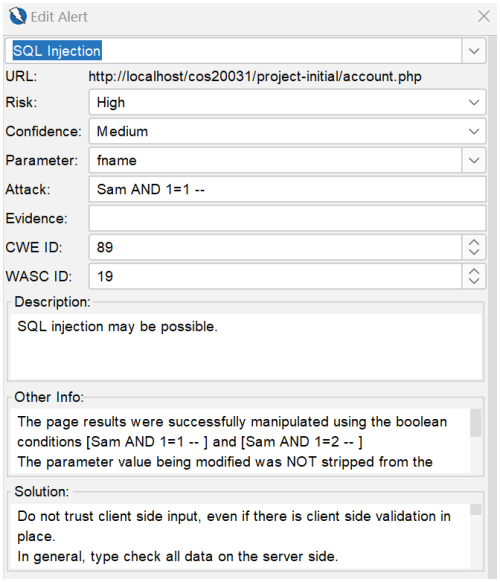


Figure 15: Payload in the fname parameter

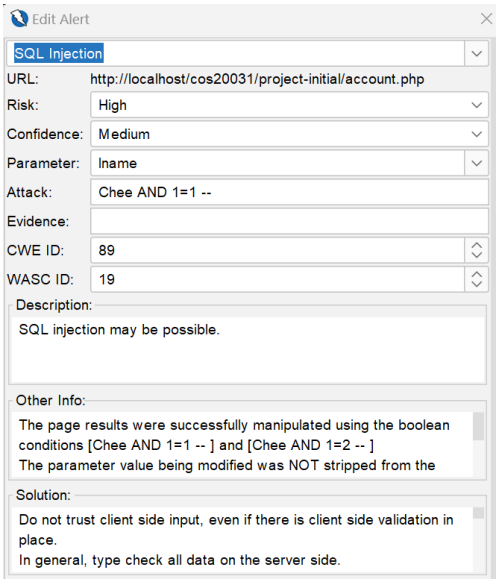


Figure 14: Payload in the lname parameter

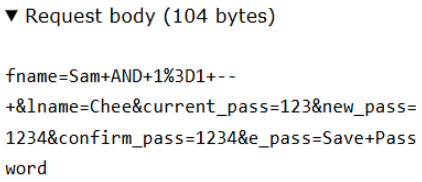


Figure 16: Request body from fname

These are two similar alerts whereby the first name and the last names are injected with the payload AND 1=1 -- and OR 1=1 -- after the first and last names. In the request body, Sam OR 1=1 -- is injected into the first name (fname) parameter when the user changes the first name in account.php. Similarly, the last name (lname) which is Chee AND 1=1 -- is also injected with payload.

In the payloads `AND 1=1 --` and `OR 1=1 —` which are Boolean expressions that are imposed into the structure of the SQL query and override the planned logic of the query. When the parameter is appended with `AND 1=1 --`, the expression `1=1` means that it is always TRUE and `--` is known as comment delimiter in SQL, making SQL ignore the remaining of the query. This can change the way that query runs to exclude conditions or constraints. Likewise, 'OR 1=1 --' can be used to avoid authentication where clause of the query will always return TRUE despite conditions put in place to lock down the entries.

2. SQL Injection in the current_pass and new_pass parameters

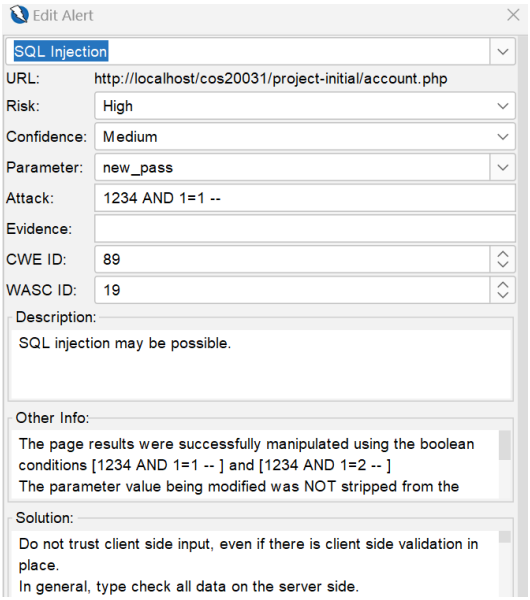


Figure 18: Payload in the new_pass parameter

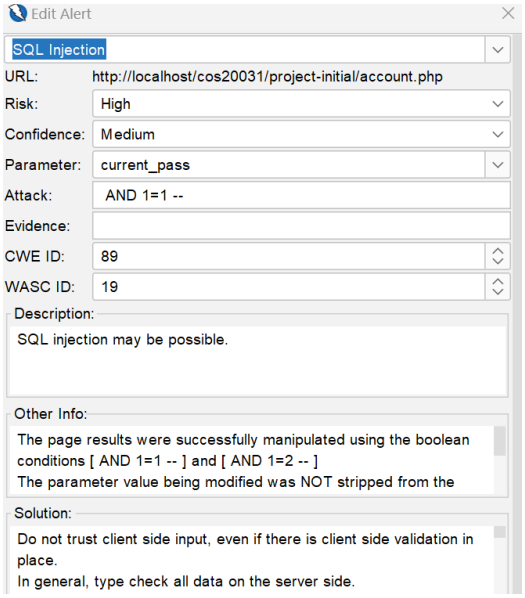


Figure 17: Payload in the current_pass parameter

The payloads AND 1=1 -- injected into the new_pass and current_pass parameters affect SQL query logic the same way as in the fname and lname parameters. Before the use of parameterized queries and input validation, 1=1 makes the database interpret the statement in a way that will always be decoded as true. The -- allows the rest of the SQL to be interpreted as a string or a comment, enabling intruders to gain access or alter data.

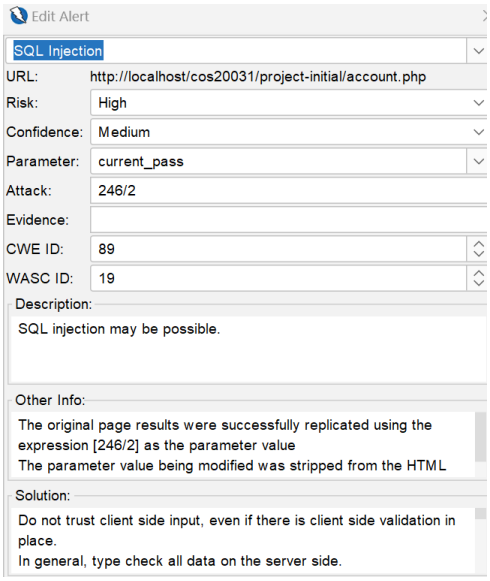


Figure 19: Mathematical payload in the current_pass parameter

On the other hand, current_pass is also exploited with the attack of 246/2 which is a mathematical formula. Whenever unsanitised input is incorporated into a SQL statement, mathematical expressions such as '246/2' can be executed by the database as input. For instance, if the query tries to match a password with `246/2` the database engine will, for example, calculate the result which is `123` and then compare to the password.

3. SQL Injection on the login_button parameter

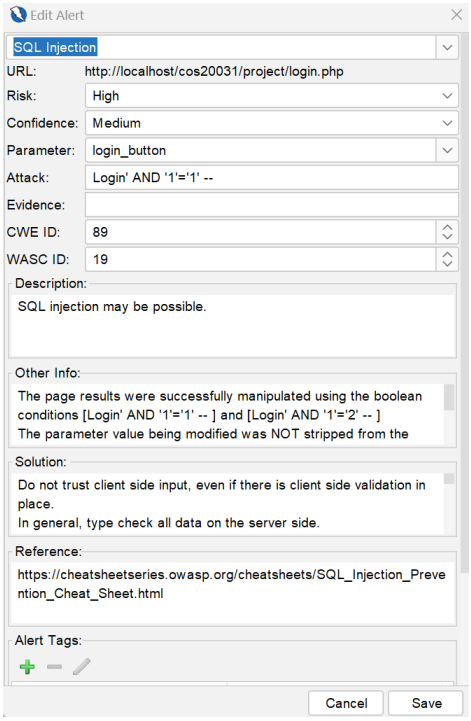


Figure 20: SQL Injection Session Properties

SQL injection vulnerability was found in the 'login.php' endpoint of the application that would allow a person to login without going through the login system. This vulnerability was confirmed with Boolean conditions in the query. In the login parameter, by using Login' AND '1'= '1' --, the system provided the access to the user although no proper authentication was done which proves that the authentication checks can be manipulated with a crafted input.

On the other hand, when an input of Login' AND '1='2' was entered, it led to granting limited access to data which means that SQL queries are being manipulated using straight inputs without using query parameters to sanitize inputs. To counter this, the server-side validation should be mandatory; the use of parameterized queries should be called for and the inputs from the users sanitized to eliminate the use of scripts.

The images below show the request and response of the first payload, **Login' AND '1'= '1' --**.

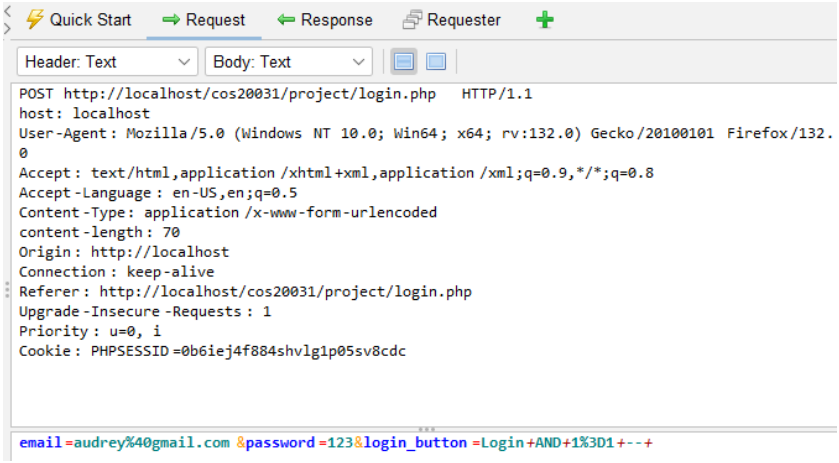


Figure 21: Request of SQL Injection Alert

In the request, the payload takes advantage of the SQL query parsing by adding an always-true statement ('1'='1') after an 'AND' operator. -- indicates that it is a comment which leaves the rest of the SQL query unprocessed which makes the whole basic query true regardless of the password entered. This malicious-crafted request's goal is to sneak past the authentication mechanism to directly access the targeted application.

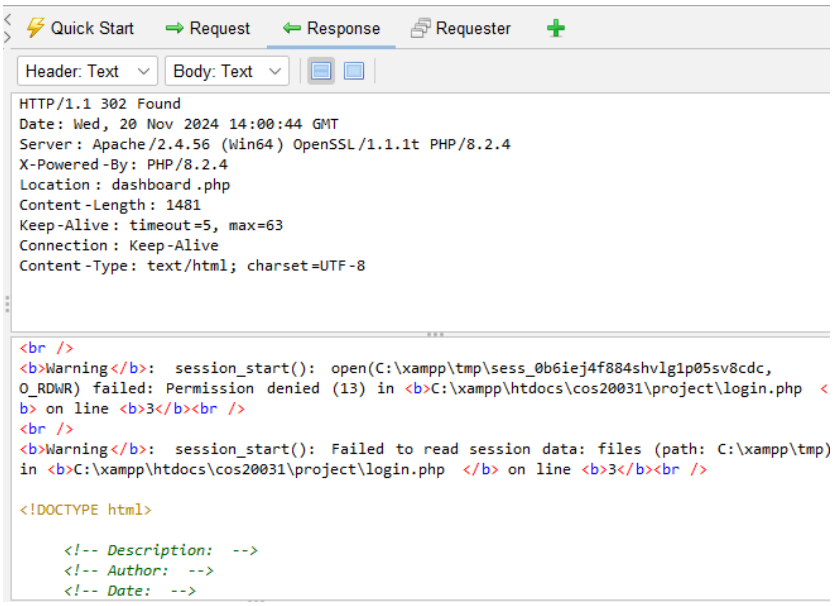
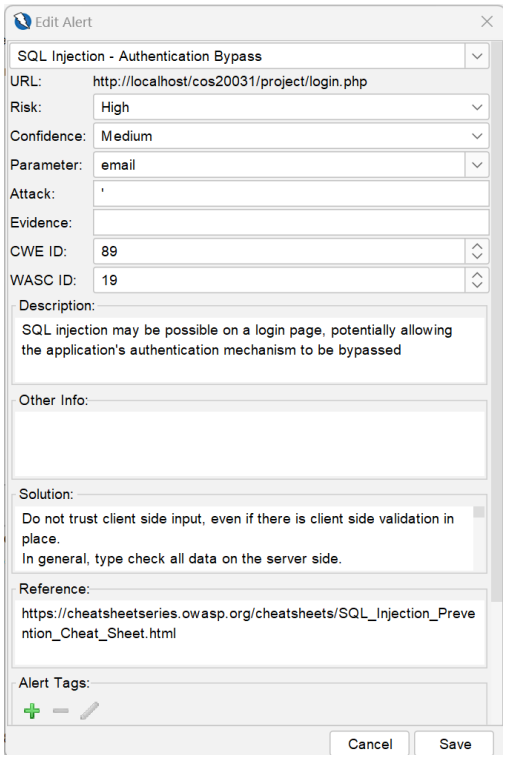


Figure 22: Response of SQL Injection Alert

On the other hand, the response shows that the payload was able to unlock the flawed authentication mechanism. The HTTP status code '302 Found' indicates that the server offers a redirection for rest of the response, after a login. The attacker is redirected to the application's console which means that the SQL injection validated the credentials without the use of the appropriate password.

However, there are some words about session management problems such as 'Permission denied' for session file access in the server's directory. These warnings indicate misconfigurations in the server, but they do not stop the attack from succeeding. This proves that there is a flaw in the web application's SQL query and sessions.

4. SQL Injection (Authentication Bypass) via 'email' Parameter



SQL injection was found in email input field of the login.php form in the database parameter list. When the attacker entered the single quote ('), the system permitted authentication bypass, showing that SQL inputs are concatenated with user inputs instead of being properly parameterized. This is a severe danger because an attacker who gets to the login credentials will have unauthorized access.

This problem can be solved by deploying Prepared Statements and including strict server-side input validation. Moreover, to protect the system from injection attacks input data could be sanitized so that special characters used in SQL are not allowed.

Figure 23: SQL Injection (Authentication Bypass) via 'email' Parameter Session Properties

Figure 24: Request of SQL Injection (Authentication Bypass) via 'email' Parameter Alert

In this POST request the 'email' parameter is given a single quote (') while the 'password' is 123. This payload is testing the application's input handling for one certain field that is the email field. The single quote (') is one of the most frequently used proxies for finding the possible SQL injection vulnerabilities, because it can alter the construction of the SQL query if the user content is not properly sanitised. The attacker inserts this character to disrupt the query syntax or log in with the application without authorisation when the inputs are not validated in a proper manner. Other headers and session cookies in request show that payload is tailored to mimic a normal login attack.

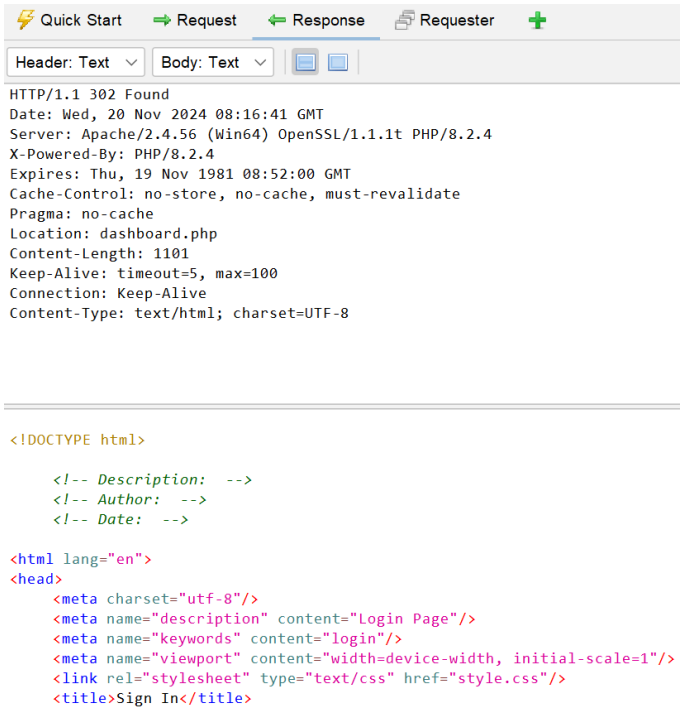


Figure 25: Response of SQL Injection (Authentication Bypass) via 'email' Parameter Alert

In the response section, we get an HTTP '302 Found' status code and redirects the user to the dashboard.php file. This confirms that the program recognizes a successful login during which the password entered was incorrect. The successful request means that the application's SQL query is prone to injection. It appears that the single quote (') in the email parameter somehow modified the SQL query and got around the authentication mechanism. This is because the input might have triggered the backend to run a query that is always true, or points to the first record in the database and grants the intruder access into the system.

5. SQL Injection (Authentication Bypass) via 'login_button' Parameter

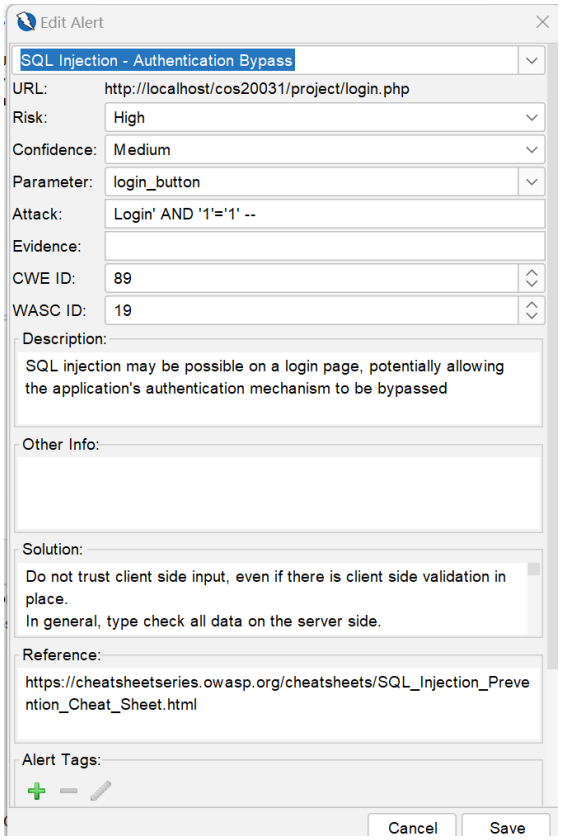


Figure 26: SQL Injection (Authentication Bypass) via 'login_button' Parameter Session Properties

A SQL injection in the login.php was identified in the login_button parameter when an attacker attempts to enter their credentials. The vulnerability was exploited when Login' AND '1'='1' -- was entered in place of the login_button parameter and it opened the application for unauthorized access. This means that the application passes data from regular users directly into the login_button parameter and then adds it into SQL queries without opinions. This may enable the attackers to forge ordinary users or take access to private information.

Preventions and protection procedures encompass parameterized queries and Prepared Statements, verifying inputs server side, and exploiting Web Application Firewalls (WAFs) to decipher and eliminate attempts at SQL injection. One way to achieve this would be to turn off all the unnecessary debug messages thrown by libraries, frameworks and platforms and similarly one would need to camouflage the structure of SQL queries that are to be executed in the backend.

Figure 27: Request of SQL Injection (Authentication Bypass) via 'login_button' Parameter Alert

The request focuses on the login.php using the POST method. In the email parameter, it is made to a normal value, audrey@gmail.com; the password field has a correct password, 123. However, in the login_button parameter, the vulnerability exploitation happens as the payload Login OR 1=1 -- is inserted. This payload changes the string, the SQL query used in the login, authentication process as string. The OR 1=1 part of the code makes the condition true all the time excluding any authentication check. The -- sequence excludes any of the subsequent SQL logic used to authenticate logins. Thus, this provides the attacker with the access they are attempting to gain regardless of entering the actual password.

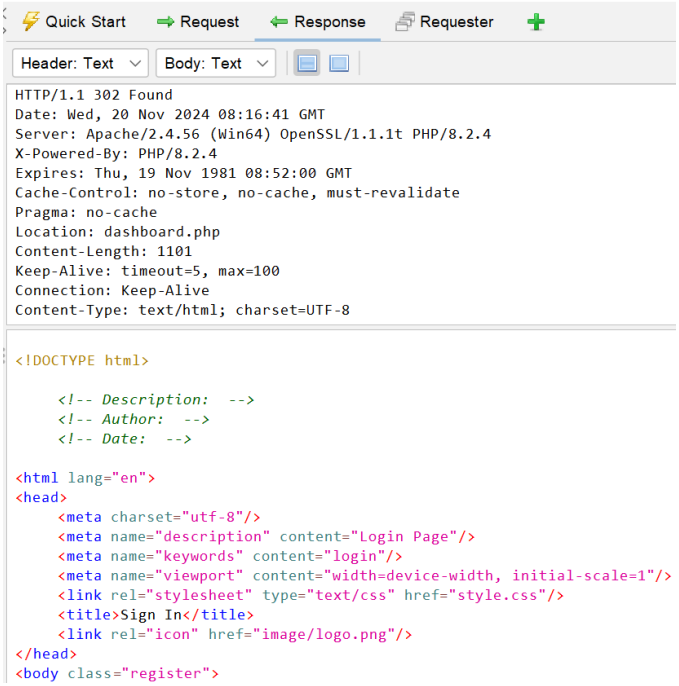


Figure 28: Response of SQL Injection (Authentication Bypass) via 'login_button' Parameter Alert

The reply from the server is an HTTP status '302 Found' shows that there is redirection to 'dashboard.php' which means the login is successful. The use of Cache-Control: no-store, no-cache in the headers, implies that the system is built for secure dynamic context challenging by the current SQL injection flaw.

The following logic of HTML code explains the structure of the login page to have email and password to be filled separately with reset and login buttons. Nevertheless, redirection to dashboard.php proves that SQL injection gets through authentication. The server regards the request above as a valid login because of the tainted SQL query hence unauthorized access to restricted resources.

6. SQL Injection – MySQL in the fname and lname parameter

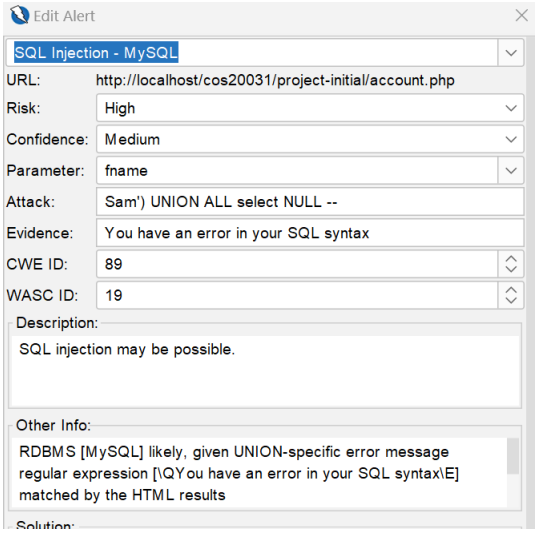


Figure 30: Payload in the fname parameter

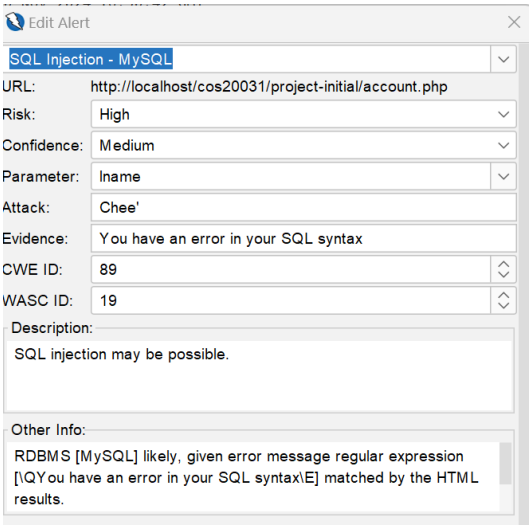


Figure 29: Payload in the lname parameter

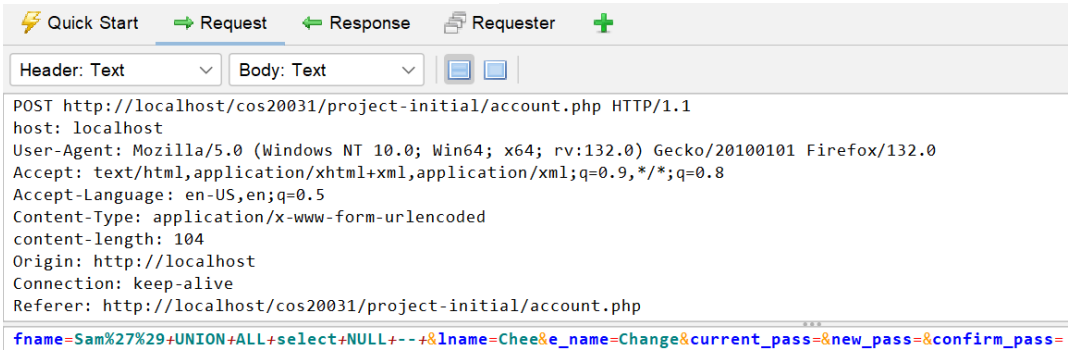


Figure 31: Request for fname payload

In the SQL injection payload of 'fname' which is (Sam' UNION ALL select NULL —), use of structure in query causes SQL manipulation. The payload `') UNION ALL select NULL ``` closes the original query context (`fname='Sam"), adds a `UNION" to try and append the results with another attempt at an SQL command (returning NULLs), and then comments out the rest of the command "```. This can expose the database to query for such information if the database has been configured wrongly to allow it to do so.

For `lname`, the apostrophe injection (`Chee'`) introduces an opening single quote without its corresponding closing single quote in the SQL query thus generating a syntax error. The resulting error provides a lot of information about the database schema and query kind of structure which helps attackers in creating more efficient payloads.

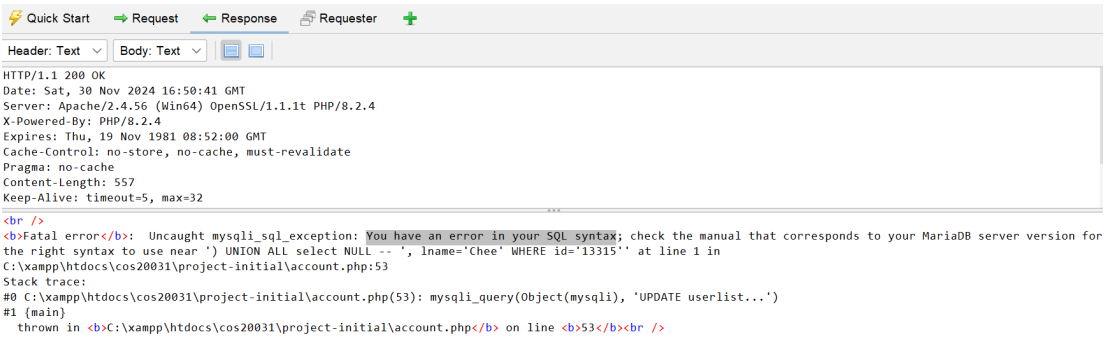


Figure 32: Response for both attacks

In the response for both attacks, it states that there is an error in the SQL syntax. The image above clearly shows that the server is not coping or even 'masking' errors. This is a weakness because of the reveal of sensitive details of the database such as table and column names, SQL commands, and file locations.

7. SQL Injection – MySQL in the email parameter

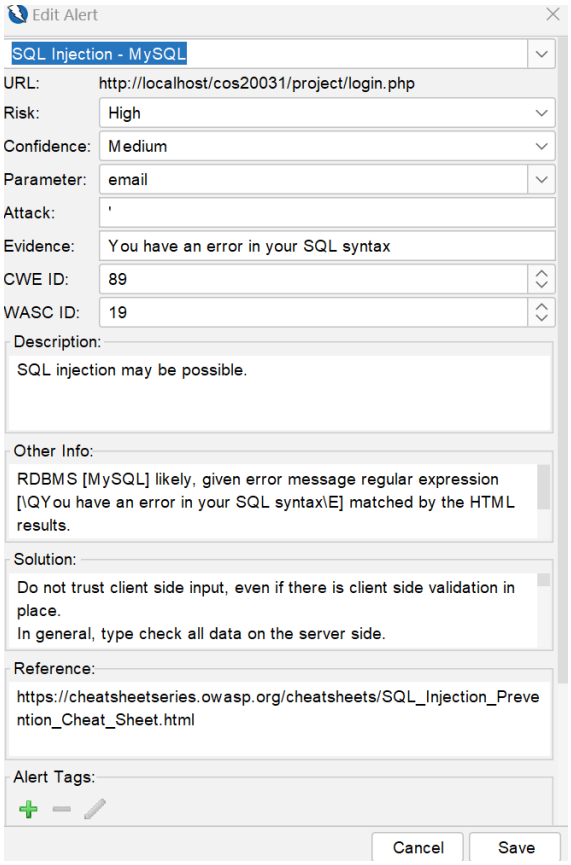


Figure 33: SQL Injection – MySQL Session Properties

The most evident example of SQL injection was also found in input parameter 'email' of the 'login.php' form; if ' was typed, the response indicated a SQL syntax error. This implies that the user input is directly appended to the SQL statements meaning the entire system can be manipulated. Unlike the previous attack, this targets vulnerabilities in error handling and construction of queries, as it is clearly seen from the detailed error in database message in the server's response. This problem might be used to grab important information and even modify the physical structure of the database.

As a solution, there is a need to enhance server-side validation, and input sanitization to lock down all special characters that may harm queries. The above issues will be offset using parameterized queries or prepared statements, and the use of stored procedures. Reducing the permissions of the database user and omitting the exact error messages improve the general security against SQL I/AA even more.

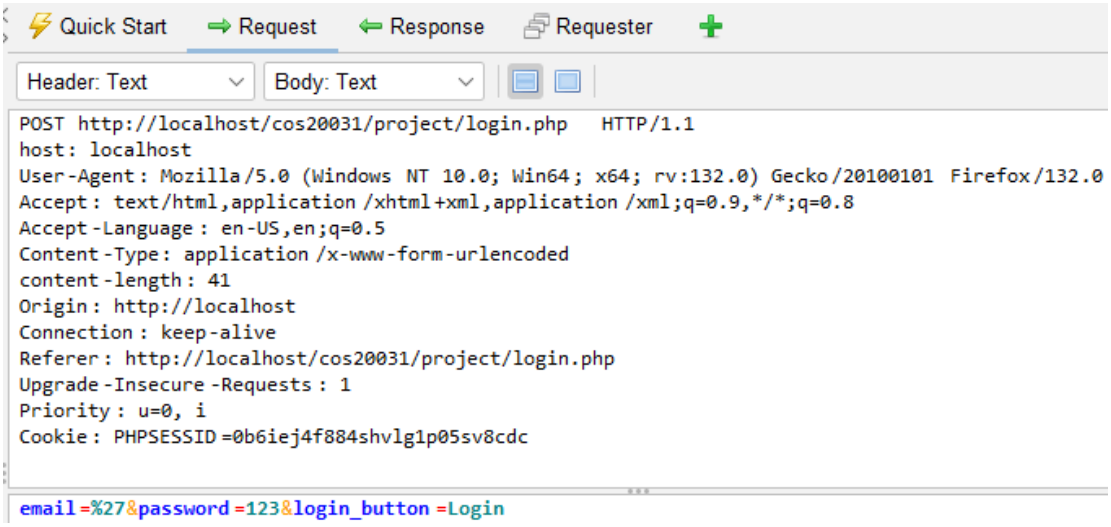


Figure 34: Request of SQL Injection – MySQL Alert

The request exploits the `email` parameter by injecting the payload `'` (a single quote) into the field. This payload is designed to test for SQL injection vulnerabilities by deliberately breaking the SQL syntax. The `password` and `login_button` parameters seem normal but the sequence of single quote `'` in the `email` parameter most probably destroys the structure of SQL query the server uses.

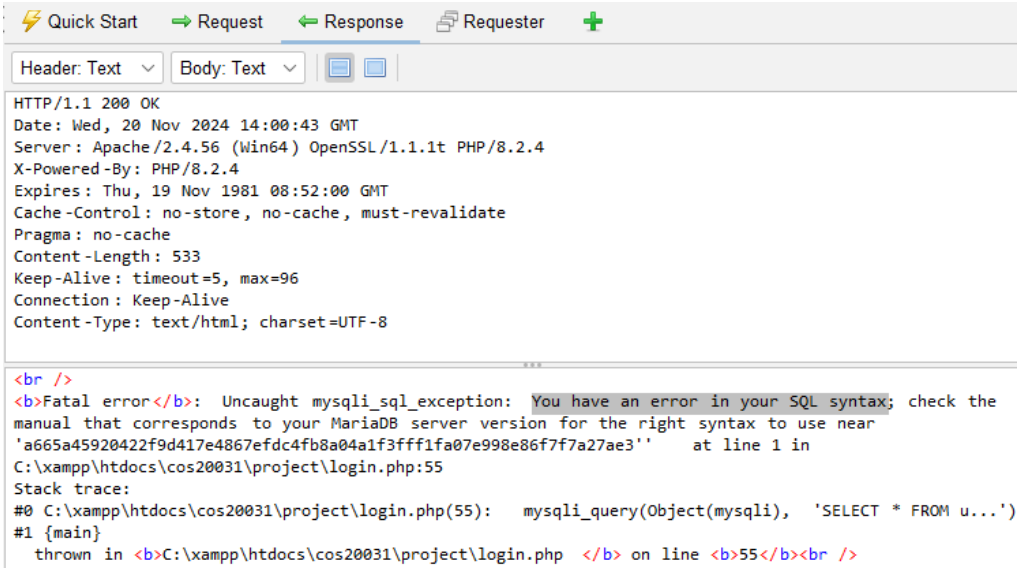


Figure 35: Response of SQL Injection – MySQL Alert

The response indicates an HTTP `200 OK` status which shows that the server received the request but encountered an error. The body of the response reveals an error message indicating a SQL syntax error caused by the unescaped single quote. This happened when trying to concatenate the SQL query, due to an undesired query structure which was produced that cannot be executed with database. This response proves that the application developed failed to sanitize inputs and validate the entries, leading the application to a SQL injection. Moreover, the exposure of sensitive details in the error message, such as the file path 'C:\xampp\htdocs\cos20031\project\login.php' and the exact query string, represent severe vulnerabilities because the attackers can use this information for further refining.

8. SQL Injection - MySQL in the email parameter

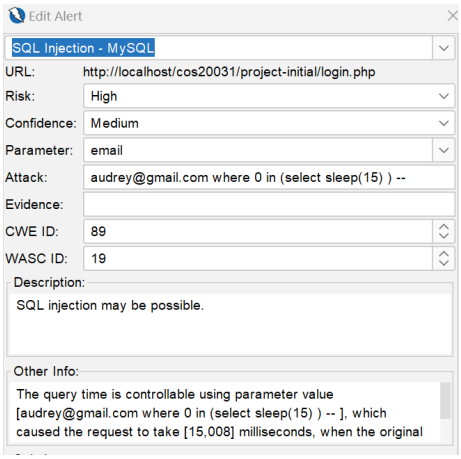


Figure 36: Payload at email parameter

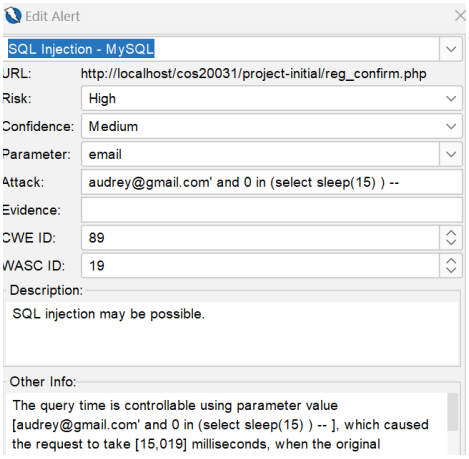


Figure 37: Payload at email parameter

Lastly, the sleep payload in the email field is a defined form of time-based SQL injection attack. The first payload, `'audrey@gmail.com where 0 in (select sleep(15)) —`, includes a `SELECT` statement that utilizes the `sleep()` function. The `sleep(15)` function causes the whole database to take a 15-second break before proceeding with the rest of the query. This delay is applied in blind SQL Injection by attempting to guess the presence of the weakness on the side of the server depending on the time taken to respond. A typical SQL injection attack does not allow the attacker to view the data, but the attacker can test this on a localized basis by timing the lag. When the delay happens it just means that the server is processing the malicious query, so the tester got an exploitable SQL injection point.

The second variation of the payload, 'audrey@gmail.com' and 0 in (select sleep(15)) -- ' is not very different in formulation from the first one. By getting the `and` condition the attacker can make sure the query has valid logical structure. The `0 in (select sleep(15)) ` part of the payload is executed in case of the conditional request, which makes the server wait for the set delay time. This can be applied where the query structure of the original query is complex, with more than one condition, and the attacker wishes to introduce the `sleep()` function for testing purposes while at the same time ensure that the condition does not affect the working of the query. This form of the attack also exploits the conditional logic most applications use to filter input based on the logic they provide, which can be manipulated to introduce a time delay.

Web Application's Database Table

	id	fname	lname	email	pass	session_id	time
Edit Copy Delete	13476	Sam	Chee	audrey@gmail.com	e3b0c44298fc1c149afbf4c8996fb92427ae41e4649b934ca4...	2q1hq9qsccq7sf48irfvm3qs8s	1732987065

Figure 38: Database Table before the automated attack

Before the attack with ZAP, the initially created database table of the application only contains a legitimate user record. The user data attributes are quite conventional and consist of a first name, the last name, an email address, and password. This is a normal state of the application where the users who have registered correctly are the only one stored in the database.

Edit Copy Delete	13529	Audrey	Chee	audrey@gmail.com AND 1=1 --	e3b0c44298fc1c149afbf4c8996fb92427ae41e4649b934ca4...	none	0
Edit Copy Delete	13530	Audrey	Chee	audrey@gmail.com\' AND \'1\'=\'1\' --	e3b0c44298fc1c149afbf4c8996fb92427ae41e4649b934ca4...	none	0
Edit Copy Delete	13531	Audrey	Chee	audrey@gmail.com\" AND \"1\"=\"1\" --	e3b0c44298fc1c149afbf4c8996fb92427ae41e4649b934ca4...	none	0
Edit Copy Delete	13532	Audrey	Chee	audrey@gmail.com AND 1=1	e3b0c44298fc1c149afbf4c8996fb92427ae41e4649b934ca4...	none	0
Edit Copy Delete	13533	Audrey	Chee	audrey@gmail.com\' AND \'1\'=\'1	e3b0c44298fc1c149afbf4c8996fb92427ae41e4649b934ca4...	none	0
Edit Copy Delete	13534	Audrey	Chee	audrey@gmail.com\" AND \"1\"=\"1	e3b0c44298fc1c149afbf4c8996fb92427ae41e4649b934ca4...	none	0
Edit Copy Delete	13535	Audrey	Chee	audrey@gmail.com UNION ALL select NULL --	e3b0c44298fc1c149afbf4c8996fb92427ae41e4649b934ca4...	none	0
Edit Copy Delete	13536	Audrey	Chee	audrey@gmail.com\' UNION ALL select NULL --	e3b0c44298fc1c149afbf4c8996fb92427ae41e4649b934ca4...	none	0
Edit Copy Delete	13537	Audrey	Chee	audrey@gmail.com\" UNION ALL select NULL --	e3b0c44298fc1c149afbf4c8996fb92427ae41e4649b934ca4...	none	0
Edit Copy Delete	13538	Audrey	Chee	audrey@gmail.com) UNION ALL select NULL --	e3b0c44298fc1c149afbf4c8996fb92427ae41e4649b934ca4...	none	0
Edit Copy Delete	13539	Audrey	Chee	audrey@gmail.com\') UNION ALL select NULL --	e3b0c44298fc1c149afbf4c8996fb92427ae41e4649b934ca4...	none	0

Figure 39: Database table after the automated attack

As a result of the SQL injection attack by using ZAP, there was some changes observed in the database table shown below. Initially, there was only one entry on the table – the first identified user, but after the attack, many more entries were added. These new entries are due to the SQL injection payloads that have been launched.

The new entries in the table highlighted above are examples of automated attack attempts made from OWASP ZAP to locate vulnerabilities in an application. These payloads have a logical sequence to verify the different types of injections that the application can respond to starting with Boolean ones, as well as the union injections.

The first set of payloads shows the basic way of using Boolean injections where conditions such as `AND 1=1`, and `AND 1=2` are added to the real SQL statement. This condition `AND 1=1` has an always true condition, while this `AND 1=2` condition has an always false condition. It means that an attacker can comprehend whether the application is vulnerable to SQL injection by the responses. Syntax comments such as `--` and `/*` are used to skip the rest of the SQL query, that are not part of the injected payload, to not throw out SQL syntax errors. These techniques assist the attackers to check whether their input affects the query logic.

The second set of payloads is related to union-based SQL injection where the attack tries to add more queries to the main one using the UNION keyword. For instance the payload ') UNION ALL select NULL–' could be used since it was created to check if the application will allow and execute other queries. Here, the attack increases the number of `NULL` in the payload in increments (e.g. `NULL,NULL`, `NULL,NULL,NULL`) to get the correct number of arguments expected by the database. If the number is discovered, the attacker can instigate the program to replace the `NULL` with certain column names or data which allows sensitive information to be passed over.

After Implementing Fixes

Confirmation Test Using Manual SQL Injection

1. Testing with **random' OR 1=1 -- '** and a dummy password

Figure 40: Testing of Manual Injection with random' OR 1=1 -- '

From the image on the left, when the attacker tries to enter the payload, they are unable to enter the account of the user. When they try to login, the output 'Invalid username or password' appears, proving that the SQL injection vulnerability has been fixed

2. Testing with **audrey@gmail.com' -- '** and a dummy password

Figure 41: Testing of Manual injection with audrey@gmail.com' -- '

From the image on the left, when the attacker tries to enter the payload, they are unable to enter the account of the user. When they try to login, the output 'Invalid username or password' appears, proving that the SQL injection vulnerability has been fixed.

Confirmation Test Using Automated SQL Injection with ZAP

Number of SQL alerts

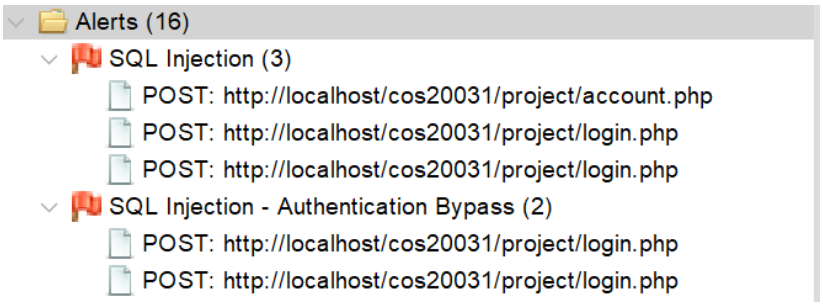


Figure 42: Alerts after applying fixes

The number of SQL Injection alerts have reduced from 16 to 5 alerts with most of them being false positives.

Analysis of the remaining alerts and their requests/responses

1. SQL Injection with the attack of Chee AND 1=1 – to the parameter lname in account.php

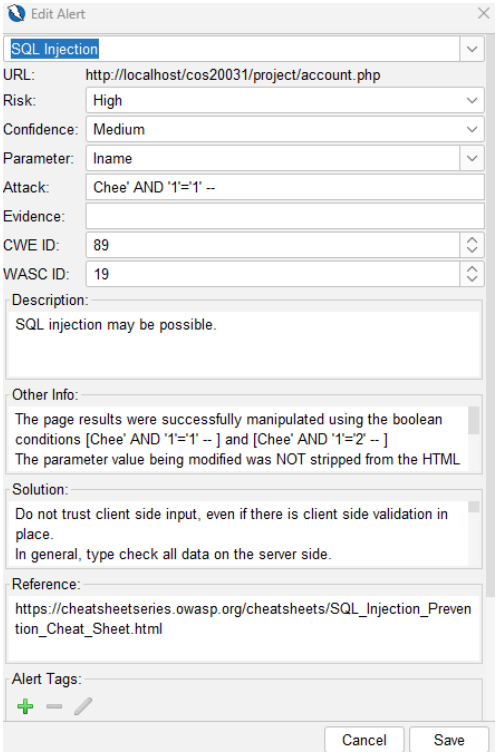


Figure 44: Alert for lname parameter

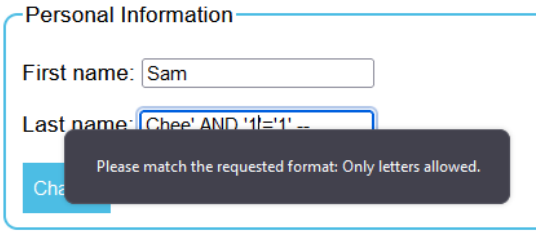


Figure 43: Message for invalid format

Based on the images above, it can be shown that the alert is a false positive as when the payload Chee AND 1=1 -- is entered into the 'Last name:' parameter, it will not allow the attacker from changing as there is an input validation that says, 'Please match the requested format: Only letters allowed'.

2. SQL Injection and Authentication Bypass Alert for login_button with the payload Login" AND "1"="1" -- and LOGOUT AND 1=1 -- in login.php.

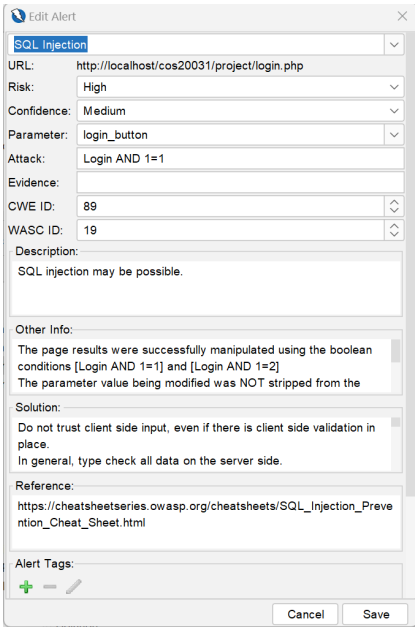


Figure 46: Payload in Login button

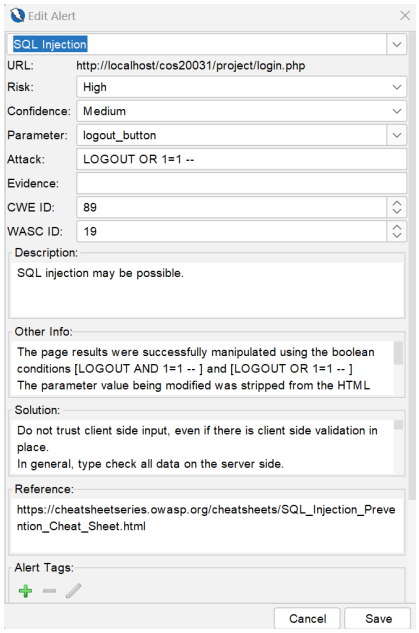


Figure 45: Payload in logout button

These two payloads, Login" AND "1"="1" -- and LOGOUT AND 1=1 -- are false positives as there are no SQL queries for these parameters in the code. In typical SQL injections, an attacker attempts to insert an unauthorized query into an SQL parameter that is expected by the application. However, the values like Login and LOGOUT do not match with any SQL query in the backend application which means that these payloads appearing as valid SQL injection attack, are harmless.

Web Application's Database Table

	id	fname	lname	email	pass	session_id	time
Edit Copy Delete	13576	Audrey	Chee	case randomblob(1000000000) when not null then 1 e...	e3b0c44298fc1c149afbf4c8996fb92427ae41e4649b934ca4...	none	0
Edit Copy Delete	13577	Audrey	Chee	sfp8tlja3rhy6pc0uh686ilvout6ge2w67p9p9ghuuiagp5ut1...	e3b0c44298fc1c149afbf4c8996fb92427ae41e4649b934ca4...	none	0
Edit Copy Delete	13578	Audrey	Chee	audrey@gmail.com WAITFOR DELAY \'0:0:15\' --	e3b0c44298fc1c149afbf4c8996fb92427ae41e4649b934ca4...	none	0
Edit Copy Delete	13579	Audrey	Chee	audrey@gmail.com\' WAITFOR DELAY \'0:0:15\' --	e3b0c44298fc1c149afbf4c8996fb92427ae41e4649b934ca4...	none	0
Edit Copy Delete	13580	Audrey	Chee	audrey@gmail.com\" WAITFOR DELAY \'0:0:15\' --	e3b0c44298fc1c149afbf4c8996fb92427ae41e4649b934ca4...	none	0
Edit Copy Delete	13581	Audrey	Chee	audrey@gmail.com) WAITFOR DELAY \'0:0:15\' --	e3b0c44298fc1c149afbf4c8996fb92427ae41e4649b934ca4...	none	0
Edit Copy Delete	13582	Audrey	Chee	audrey@gmail.com) \' WAITFOR DELAY \'0:0:15\' --	e3b0c44298fc1c149afbf4c8996fb92427ae41e4649b934ca4...	none	0
Edit Copy Delete	13583	Audrey	Chee	audrey@gmail.com) \" WAITFOR DELAY \'0:0:15\' --	e3b0c44298fc1c149afbf4c8996fb92427ae41e4649b934ca4...	none	0
Edit Copy Delete	13584	Audrey	Chee	audrey@gmail.com)) WAITFOR DELAY \'0:0:15\' --	e3b0c44298fc1c149afbf4c8996fb92427ae41e4649b934ca4...	none	0
Edit Copy Delete	13585	Audrey	Chee	audrey@gmail.com)) \' WAITFOR DELAY \'0:0:15\' --	e3b0c44298fc1c149afbf4c8996fb92427ae41e4649b934ca4...	none	0
Edit Copy Delete	13586	Audrey	Chee	audrey@gmail.com)) \" WAITFOR DELAY \'0:0:15\' --	e3b0c44298fc1c149afbf4c8996fb92427ae41e4649b934ca4...	none	0
Edit Copy Delete	13587	Audrey	Chee	audrey@gmail.com) WAITFOR DELAY \'0:0:15\' (	e3b0c44298fc1c149afbf4c8996fb92427ae41e4649b934ca4...	none	0
Edit Copy Delete	13588	Audrey	Chee	\";print(chr(122).chr(97).chr(112).chr(95).chr(116...	e3b0c44298fc1c149afbf4c8996fb92427ae41e4649b934ca4...	none	0
Edit Copy Delete	13589	Audrey	Chee	\';print(chr(122).chr(97).chr(112).chr(95).chr(116...	e3b0c44298fc1c149afbf4c8996fb92427ae41e4649b934ca4...	none	0
Edit Copy Delete	13590	Audrey	Chee	${@print(chr(122).chr(97).chr(112).chr(95).chr(116...	e3b0c44298fc1c149afbf4c8996fb92427ae41e4649b934ca4...	none	0
Edit Copy Delete	13591	Audrey	Chee	${@print(chr(122).chr(97).chr(112).chr(95).chr(116...	e3b0c44298fc1c149afbf4c8996fb92427ae41e4649b934ca4...	none	0
Edit Copy Delete	13592	Audrey	Chee	;print(chr(122).chr(97).chr(112).chr(95).chr(116)....	e3b0c44298fc1c149afbf4c8996fb92427ae41e4649b934ca4...	none	0
Edit Copy Delete	13593	Audrey	Chee	\"+response.write(537,669*631,609)+\"	e3b0c44298fc1c149afbf4c8996fb92427ae41e4649b934ca4...	none	0
Edit Copy Delete	13594	Audrey	Chee	+response.write({0}*{1})+	e3b0c44298fc1c149afbf4c8996fb92427ae41e4649b934ca4...	none	0
Edit Copy Delete	13595	Audrey	Chee	response.write(537,669*631,609)	e3b0c44298fc1c149afbf4c8996fb92427ae41e4649b934ca4...	none	0
Edit Copy Delete	13596	Audrey	Chee	cat /etc/passwd	e3b0c44298fc1c149afbf4c8996fb92427ae41e4649b934ca4...	none	0
Edit Copy Delete	13597	Audrey	Chee	audrey@gmail.com&cat /etc/passwd&	e3b0c44298fc1c149afbf4c8996fb92427ae41e4649b934ca4...	none	0
Edit Copy Delete	13598	Audrey	Chee	audrey@gmail.com;cat /etc/passwd;	e3b0c44298fc1c149afbf4c8996fb92427ae41e4649b934ca4...	none	0
Edit Copy Delete	13599	Audrey	Chee	audrey@gmail.com\"&cat /etc/passwd&\"	e3b0c44298fc1c149afbf4c8996fb92427ae41e4649b934ca4...	none	0
Edit Copy Delete	13600	Audrey	Chee	audrey@gmail.com\";cat /etc/passwd;\"	e3b0c44298fc1c149afbf4c8996fb92427ae41e4649b934ca4...	none	0
Edit Copy Delete	13601	Audrey	Chee	audrey@gmail.com\'&cat /etc/passwd&\'	e3b0c44298fc1c149afbf4c8996fb92427ae41e4649b934ca4...	none	0
Edit Copy Delete	13602	Audrey	Chee	audrey@gmail.com\';cat /etc/passwd;\'	e3b0c44298fc1c149afbf4c8996fb92427ae41e4649b934ca4...	none	0
Edit Copy Delete	13603	Audrey	Chee	audrey@gmail.com&sleep 15.0&	e3b0c44298fc1c149afbf4c8996fb92427ae41e4649b934ca4...	none	0
Edit Copy Delete	13604	Audrey	Chee	audrey@gmail.com;sleep 15.0;	e3b0c44298fc1c149afbf4c8996fb92427ae41e4649b934ca4...	none	0
Edit Copy Delete	13605	Audrey	Chee	audrey@gmail.com\"&sleep 15.0&\"	e3b0c44298fc1c149afbf4c8996fb92427ae41e4649b934ca4...	none	0
Edit Copy Delete	13606	Audrey	Chee	audrey@gmail.com\";sleep 15.0;\"	e3b0c44298fc1c149afbf4c8996fb92427ae41e4649b934ca4...	none	0
Edit Copy Delete	13607	Audrey	Chee	audrey@gmail.com\'&sleep 15.0&\'	e3b0c44298fc1c149afbf4c8996fb92427ae41e4649b934ca4...	none	0
Edit Copy Delete	13608	Audrey	Chee	audrey@gmail.com\';sleep 15.0;\'	e3b0c44298fc1c149afbf4c8996fb92427ae41e4649b934ca4...	none	0
Edit Copy Delete	13609	Audrey	Chee	type %SYSTEMROOT%\\win.ini	e3b0c44298fc1c149afbf4c8996fb92427ae41e4649b934ca4...	none	0
Edit Copy Delete	13610	Audrey	Chee	audrey@gmail.com&type %SYSTEMROOT%\\win.ini	e3b0c44298fc1c149afbf4c8996fb92427ae41e4649b934ca4...	none	0
Edit Copy Delete	13611	Audrey	Chee	audrey@gmail.com|type %SYSTEMROOT%\\win.ini	e3b0c44298fc1c149afbf4c8996fb92427ae41e4649b934ca4...	none	0
Edit Copy Delete	13612	Audrey	Chee	audrey@gmail.com\"&type %SYSTEMROOT%\\win.ini&\"	e3b0c44298fc1c149afbf4c8996fb92427ae41e4649b934ca4...	none	0
Edit Copy Delete	13613	Audrey	Chee	audrey@gmail.com\"|type %SYSTEMROOT%\\win.ini	e3b0c44298fc1c149afbf4c8996fb92427ae41e4649b934ca4...	none	0
Edit Copy Delete	13614	Audrey	Chee	audrey@gmail.com\'&type %SYSTEMROOT%\\win.ini&\'	e3b0c44298fc1c149afbf4c8996fb92427ae41e4649b934ca4...	none	0
Edit Copy Delete	13615	Audrey	Chee	audrey@gmail.com\'|type %SYSTEMROOT%\\win.ini	e3b0c44298fc1c149afbf4c8996fb92427ae41e4649b934ca4...	none	0
Edit Copy Delete	13616	Audrey	Chee	audrey@gmail.com&timeout /T 15.0	e3b0c44298fc1c149afbf4c8996fb92427ae41e4649b934ca4...	none	0
Edit Copy Delete	13617	Audrey	Chee	audrey@gmail.com|timeout /T 15.0	e3b0c44298fc1c149afbf4c8996fb92427ae41e4649b934ca4...	none	0
Edit Copy Delete	13618	Audrey	Chee	audrey@gmail.com\"&timeout /T 15.0&\"	e3b0c44298fc1c149afbf4c8996fb92427ae41e4649b934ca4...	none	0
Edit Copy Delete	13619	Audrey	Chee	audrey@gmail.com\"|timeout /T 15.0	e3b0c44298fc1c149afbf4c8996fb92427ae41e4649b934ca4...	none	0
Edit Copy Delete	13620	Audrey	Chee	audrey@gmail.com\'&timeout /T 15.0&\'	e3b0c44298fc1c149afbf4c8996fb92427ae41e4649b934ca4...	none	0
Edit Copy Delete	13621	Audrey	Chee	audrey@gmail.com\'|timeout /T 15.0	e3b0c44298fc1c149afbf4c8996fb92427ae41e4649b934ca4...	none	0
Edit Copy Delete	13622	Audrey	Chee	get-help	e3b0c44298fc1c149afbf4c8996fb92427ae41e4649b934ca4...	none	0
Edit Copy Delete	13623	Audrey	Chee	audrey@gmail.com;get-help	e3b0c44298fc1c149afbf4c8996fb92427ae41e4649b934ca4...	none	0
Edit Copy Delete	13624	Audrey	Chee	audrey@gmail.com\";get-help	e3b0c44298fc1c149afbf4c8996fb92427ae41e4649b934ca4...	none	0
Edit Copy Delete	13625	Audrey	Chee	audrey@gmail.com\';get-help	e3b0c44298fc1c149afbf4c8996fb92427ae41e4649b934ca4...	none	0
Edit Copy Delete	13626	Audrey	Chee	audrey@gmail.com;get-help #	e3b0c44298fc1c149afbf4c8996fb92427ae41e4649b934ca4...	none	0
Edit Copy Delete	13627	Audrey	Chee	audrey@gmail.com;start-sleep -s 15.0	e3b0c44298fc1c149afbf4c8996fb92427ae41e4649b934ca4...	none	0
Edit Copy Delete	13628	Audrey	Chee	audrey@gmail.com\";start-sleep -s 15.0	e3b0c44298fc1c149afbf4c8996fb92427ae41e4649b934ca4...	none	0
Edit Copy Delete	13629	Audrey	Chee	audrey@gmail.com\';start-sleep -s 15.0	e3b0c44298fc1c149afbf4c8996fb92427ae41e4649b934ca4...	none	0
Edit Copy Delete	13630	Audrey	Chee	audrey@gmail.com;start-sleep -s 15.0 #	e3b0c44298fc1c149afbf4c8996fb92427ae41e4649b934ca4...	none	0
Edit Copy Delete	13631	Audrey	Chee	\"\'	e3b0c44298fc1c149afbf4c8996fb92427ae41e4649b934ca4...	none	0
Edit Copy Delete	13632	Audrey	Chee	<!--	e3b0c44298fc1c149afbf4c8996fb92427ae41e4649b934ca4...	none	0
Edit Copy Delete	13633	Audrey	Chee	]]>	e3b0c44298fc1c149afbf4c8996fb92427ae41e4649b934ca4...	none	0
Edit Copy Delete	13634	Audrey	Chee	zj 8688*1488 zj	e3b0c44298fc1c149afbf4c8996fb92427ae41e4649b934ca4...	none	0
Edit Copy Delete	13635	Audrey	Chee	zj{2306*5882}zj	e3b0c44298fc1c149afbf4c8996fb92427ae41e4649b934ca4...	none	0
Edit Copy Delete	13636	Audrey	Chee	zj${2253*1936}zj	e3b0c44298fc1c149afbf4c8996fb92427ae41e4649b934ca4...	none	0
Edit Copy Delete	13637	Audrey	Chee	zj#{9336*8954}zj	e3b0c44298fc1c149afbf4c8996fb92427ae41e4649b934ca4...	none	0
Edit Copy Delete	13638	Audrey	Chee	zj{#1645*4834}zj	e3b0c44298fc1c149afbf4c8996fb92427ae41e4649b934ca4...	none	0
Edit Copy Delete	13639	Audrey	Chee	zj{@3123*5098}zj	e3b0c44298fc1c149afbf4c8996fb92427ae41e4649b934ca4...	none	0
Edit Copy Delete	13640	Audrey	Chee	zj{{8832*2246}}zj	e3b0c44298fc1c149afbf4c8996fb92427ae41e4649b934ca4...	none	0
Edit Copy Delete	13641	Audrey	Chee	zj{{=9848*2792}}zj	e3b0c44298fc1c149afbf4c8996fb92427ae41e4649b934ca4...	none	0
Edit Copy Delete	13642	Audrey	Chee	zj<%=8963*8965%>zj	e3b0c44298fc1c149afbf4c8996fb92427ae41e4649b934ca4...	none	0
Edit Copy Delete	13643	Audrey	Chee	zj#set($x=4339*9672)${x}zj	e3b0c44298fc1c149afbf4c8996fb92427ae41e4649b934ca4...	none	0
Edit Copy Delete	13644	Audrey	Chee	zj<p th:text=\"${3529*4625}\"></p>zj	e3b0c44298fc1c149afbf4c8996fb92427ae41e4649b934ca4...	none	0
Edit Copy Delete	13645	Audrey	Chee	zj{@math key=\"2962\" method=\"multiply\" operand=...	e3b0c44298fc1c149afbf4c8996fb92427ae41e4649b934ca4...	none	0
Edit Copy Delete	13646	Audrey	Chee	zj{{19170|add:16560}}zj	e3b0c44298fc1c149afbf4c8996fb92427ae41e4649b934ca4...	none	0
Edit Copy Delete	13647	Audrey	Chee	zj{{print \"4637\" \"6277\"}}zj	e3b0c44298fc1c149afbf4c8996fb92427ae41e4649b934ca4...	none	0
Edit Copy Delete	13648	Audrey	Chee	<#assign ex=\"freemarker.template.utility.Execute\...	e3b0c44298fc1c149afbf4c8996fb92427ae41e4649b934ca4...	none	0
Edit Copy Delete	13649	Audrey	Chee	#set($engine=\"\")#set($proc=$engine.getClass()....	e3b0c44298fc1c149afbf4c8996fb92427ae41e4649b934ca4...	none	0
Edit Copy Delete	13650	Audrey	Chee	{{= global.process.mainModule.require(\'child_proc...	e3b0c44298fc1c149afbf4c8996fb92427ae41e4649b934ca4...	none	0
Edit Copy Delete	13651	Audrey	Chee	<%= global.process.mainModule.require(\'child_proc...	e3b0c44298fc1c149afbf4c8996fb92427ae41e4649b934ca4...	none	0
Edit Copy Delete	13652	Audrey	Chee	#{global.process.mainModule.require(\'child_proces...	e3b0c44298fc1c149afbf4c8996fb92427ae41e4649b934ca4...	none	0
Edit Copy Delete	13653	Audrey	Chee	{{range.constructor(\"return eval(\\\"global.proce...	e3b0c44298fc1c149afbf4c8996fb92427ae41e4649b934ca4...	none	0
Edit Copy Delete	13654	Audrey	Chee	{{\"\".__class__.__mro__[1].__subclasses__()[157]....	e3b0c44298fc1c149afbf4c8996fb92427ae41e4649b934ca4...	none	0
Edit Copy Delete	13655	Audrey	Chee	${__import__(\"subprocess\").check_output(\"sleep ...	e3b0c44298fc1c149afbf4c8996fb92427ae41e4649b934ca4...	none	0
Edit Copy Delete	13656	Audrey	Chee	{{__import__(\"subprocess\").check_output(\"sleep ...	e3b0c44298fc1c149afbf4c8996fb92427ae41e4649b934ca4...	none	0
Edit Copy Delete	13657	Audrey	Chee	<%=%x(sleep 15)%>	e3b0c44298fc1c149afbf4c8996fb92427ae41e4649b934ca4...	none	0
Edit Copy Delete	13658	Audrey	Chee	#{%x(sleep 15)}	e3b0c44298fc1c149afbf4c8996fb92427ae41e4649b934ca4...	none	0
Edit Copy Delete	13659	Audrey	Chee	{system(\"sleep 15\")}	e3b0c44298fc1c149afbf4c8996fb92427ae41e4649b934ca4...	none	0
Edit Copy Delete	13660	Audrey	Chee	RxsTmGxmuZPmLTrZACYwDxckqGvGCpBdPFvtBMvCMSqIlagII...	e3b0c44298fc1c149afbf4c8996fb92427ae41e4649b934ca4...	none	0
Edit Copy Delete	13661	Audrey	Chee	ZAP	e3b0c44298fc1c149afbf4c8996fb92427ae41e4649b934ca4...	none	0
Edit Copy Delete	13662	Audrey	Chee	ZAP%n%s%n%s%n%s%n%s%n%s%n%s%n%s%n%s%n%s%n%s%n%...	e3b0c44298fc1c149afbf4c8996fb92427ae41e4649b934ca4...	none	0
Edit Copy Delete	13663	Audrey	Chee	ZAP %1$s%2$s%3$s%4$s%5$s%6$s%7$s%8$s%9$s%10$s%11$s...	e3b0c44298fc1c149afbf4c8996fb92427ae41e4649b934ca4...	none	0
Edit Copy Delete	13664	Audrey	Chee	Set-cookie: Tamper=3772df81-f062-4a05-8269-d70e118...	e3b0c44298fc1c149afbf4c8996fb92427ae41e4649b934ca4...	none	0
Edit Copy Delete	13665	Audrey	Chee	any\r\nSet-cookie: Tamper=3772df81-f062-4a05-8269-...	e3b0c44298fc1c149afbf4c8996fb92427ae41e4649b934ca4...	none	0
Edit Copy Delete	13666	Audrey	Chee	any?\r\nSet-cookie: Tamper=3772df81-f062-4a05-8269...	e3b0c44298fc1c149afbf4c8996fb92427ae41e4649b934ca4...	none	0
Edit Copy Delete	13667	Audrey	Chee	any\nSet-cookie: Tamper=3772df81-f062-4a05-8269-d7...	e3b0c44298fc1c149afbf4c8996fb92427ae41e4649b934ca4...	none	0
Edit Copy Delete	13668	Audrey	Chee	any?\nSet-cookie: Tamper=3772df81-f062-4a05-8269-d...	e3b0c44298fc1c149afbf4c8996fb92427ae41e4649b934ca4...	none	0
Edit Copy Delete	13669	Audrey	Chee	any\r\nSet-cookie: Tamper=3772df81-f062-4a05-8269-...	e3b0c44298fc1c149afbf4c8996fb92427ae41e4649b934ca4...	none	0
Edit Copy Delete	13670	Audrey	Chee	any?\r\nSet-cookie: Tamper=3772df81-f062-4a05-8269...	e3b0c44298fc1c149afbf4c8996fb92427ae41e4649b934ca4...	none	0

Figure 47: Database table after applying fixes

Most of the vulnerabilities have been fixed as improvements can be seen from the database table above. In the previous table, opening statement injections such as 'AND 1=1 —', 'AND 1=2 —', were entered into the table. After fixing the flaws in the codes, the system does not accept or execute such payloads with the help of input validation and parameterized queries.

With the parameterized queries, the application no longer considers the user inputs as a part of the SQL statement. This prevents manipulation of legitimate SQL code by inputs such as `AND 1 = 1 --`. In this case, if the attacker tries to input a Boolean injection the application will either return an error or the input will be sanitized before the execution of the SQL query. This shows that the employment of prepared statements checks when a user tries to place an expression such as `AND 1=1` in the input parameters. The payload will be interpreted as a string or data instead of getting interpreted as part of the query in the database.

However, other attacks such as time-based injections using different commands like`WAITFOR DELAY` or `SLEEP` are not focused on this project. The current improvements are mainly targeting the simple Boolean SQL injection as the improvement will provide a fundamental layer for protecting this system against SQL injection. For the remainder of the attacks, more work would be needed to deploy additional protections that should include limiting input length or implementing more comprehensive validation against known malicious payloads or general enhancement of error handling. These steps would safeguard the application against these attacks of higher levels, but for the purposes of the current assignment, the main concentration will be to solve the simple Boolean SQL injection problems to ensure that the above application is safe from these forms of attacks.

Results and Discussion

Outcomes of the Project

The amended PHP files for this project are fixed mostly with input validation, parameterized queries, escaping user-supplied input and other SQL injection preventions. Below are the screenshots of outputs and behaviours of the system:

1. register.php

- First name and last name input validation

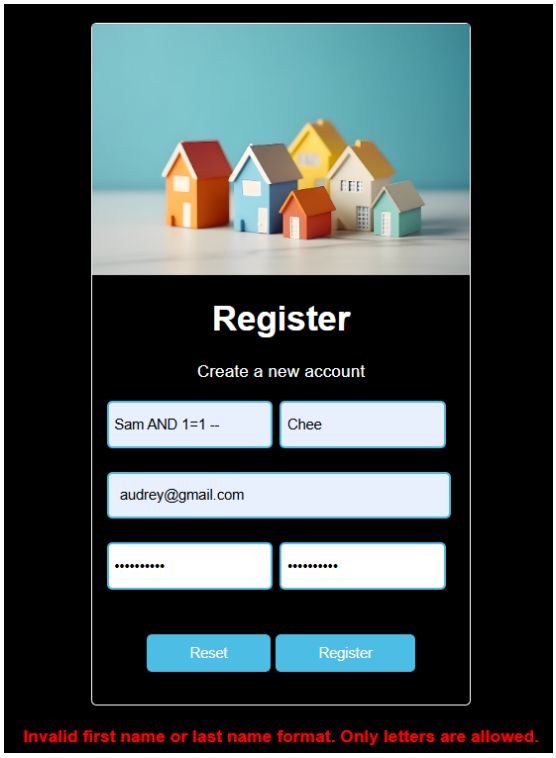


Figure 48: First name validation

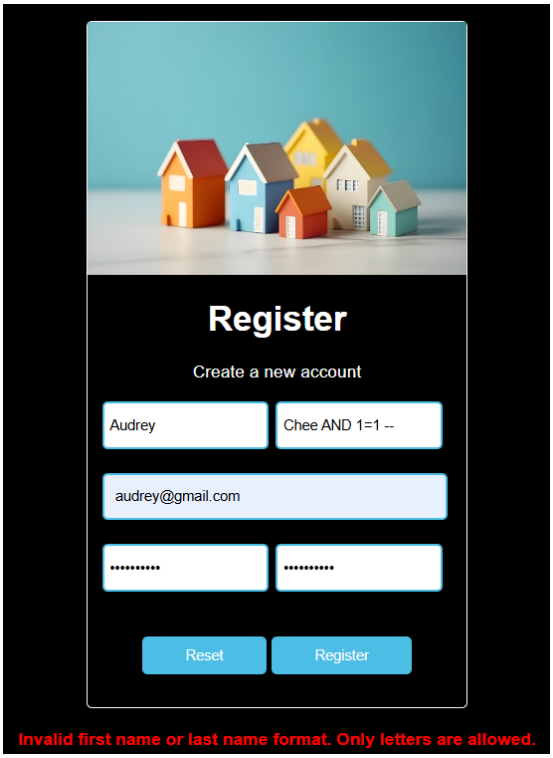


Figure 49: Last name validation

Based on the images above, it can be proved that the messages 'Invalid first name or last name format. Only letters are allowed.' prevents the attacker from using the payloads to register as a user.

- Email validation

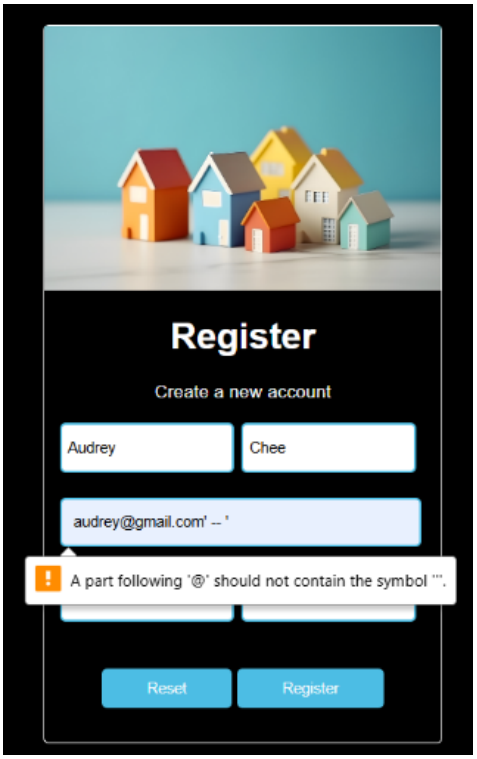


Figure 50: Email validation

A message is shown when the attacker tries to insert audrey@gmail.com' -- ' into the email parameter. This prevents the attacker from entering the payload as the email.

- Password validation

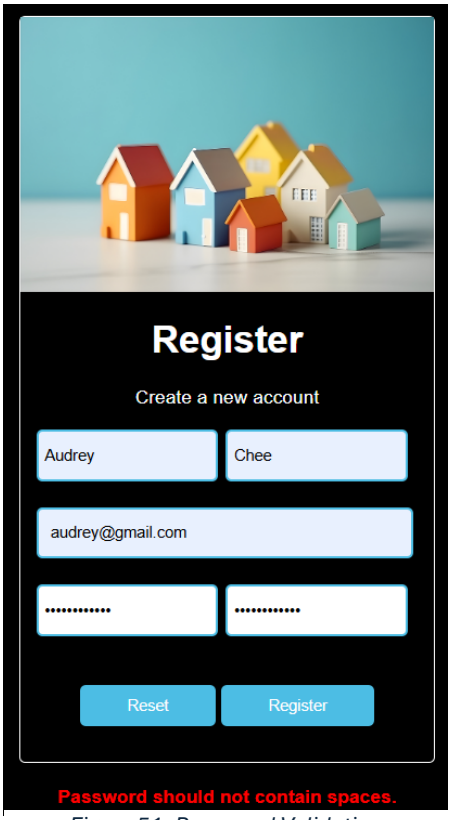


Figure 51: Password Validation

The message states that 'Password should not contain spaces.' when the user tries to enter AND 1=1 – into the password parameter.

2. login.php

Figure 53: Login email validation

Figure 52: Login password validation

Based on the images above, when the attacker tries to login with the email with a payload, the system will prevent the login and print out that the email format is invalid whereas when the payload AND 1=1 -- is entered into the password parameter, the attacker will face a message saying that the username or password is invalid.

3. account.php

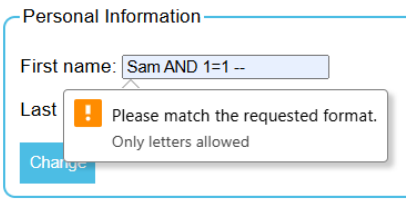


Figure 55: Changing fname validation

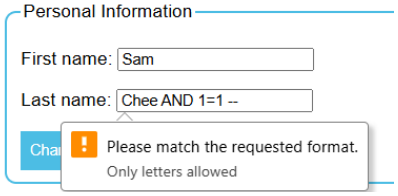
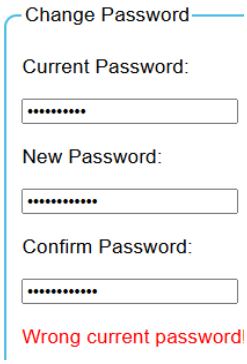


Figure 54: Changing lname validation

When the attacker tries to change the names by adding AND 1=1 --, there will be a client-side validation which states that only letters are allowed.



However, when the attacker tries to enter the same payload into the new password and confirm password parameter, the message of wrong current password is printed out instead. This will be further explained in the unresolved issues in the last section.

Figure 56: Changing password validation

Detailed Analysis of Results

1. Input Validation

The email validation is enforced to prevent characters or patterns such as (user@example.com' OR 1=1;--) to exploit the email parameter. The validation checks for correct email input without any payloads inserted into it. The snippet of code below supports these statements.

```php
// NEW CODE: Validate email input
if (!filter_var($_POST['email'], FILTER_VALIDATE_EMAIL)) {
    $Msg = "<p style='color:red'><strong>Invalid email format.</strong></p>";
} else {
    $id = mysqli_real_escape_string($conn, $_POST['email']);
    // Validate password to only allow alphanumeric characters and a reasonable length
    $pass = $_POST["password"];
```

Figure 57: Email input validation code

Code Explanation:

- filter_var($POST[‘email], FILTER_VALIDATE_EMAIL) checks if the email is in a valid format.

Next, the password validation in this system is strict to ensure that no payloads such as (AND 1=1 –) can be used to manipulate the password parameters. The code below demonstrates the password requirements which are a minimum length of 8 characters, no spaces and a mix of uppercase, lowercase, numeric and special characters which the users must have when registering, logging in and changing their password.

```php
function validatePassword($pass) {
    // Ensure password does not contain spaces and meets minimum strength requirements
    if (preg_match('/\s/', $pass)) {
        return "Password should not contain spaces.";
    }
    if (strlen($pass) < 8) {
        return "Password should be at least 8 characters long.";
    }
    if (!preg_match('/[A-Z]/', $pass)) {
        return "Password should contain at least one uppercase letter.";
    }
    if (!preg_match('/[a-z]/', $pass)) {
        return "Password should contain at least one lowercase letter.";
    }
    if (!preg_match('/[0-9]/', $pass)) {
        return "Password should contain at least one number.";
    }
    if (!preg_match('/[\W_]/', $pass)) {
        return "Password should contain at least one special character.";
    }
    return ""; // No error
}
```

Figure 58: Password validation code

Lastly, the names in the system also require input validation to prevent payloads such as (Sam OR 1=1 --). The snippet of code below shows the validation of the first name and last name of the user. This is required for all three PHP files which are login, registration and account pages.

```php
// NEW CODE - Validation and sanitization for names
if (preg_match("/^[A-Za-z]+$/", $_POST['fname']) && preg_match("/^[A-Za-z]+$/", $_POST['lname'])) {
    $fname = filter_input(INPUT_POST, 'fname', FILTER_SANITIZE_STRING);
    $lname = filter_input(INPUT_POST, 'lname', FILTER_SANITIZE_STRING);
    $ename = filter_input(INPUT_POST, 'ename', FILTER_SANITIZE_STRING);
```

Figure 59: fname and lname validation code

From the above code, the input validation of first name (`fname`) and last name (`lname`) values accept only letters using the regular expression check `preg_match()`. If both conditions are met, filters for type filter input are applied for values using the FILTER_SANITIZE_STRING filter to remove any dangerous character or HTML tags in the input data.

2. Parameterized Queries

Parameterised queries are used to ensure safe data is accepted to prevent SQL injections during database interactions. The code above shows the use of prepared statements in reg_confirm.php.

```php
// OLD CODE (Vulnerable to SQL Injection)
// $query = "INSERT INTO userlist(fname,lname,email,pass,session_id) VALUES('$fname','$lname','$email','$pass','$session_id');";
// @mysqli_query($conn, $query);

// NEW CODE (Prepared Statements for Security)
$stmt = $conn->prepare("INSERT INTO userlist (fname, lname, email, pass, session_id) VALUES (?, ?, ?, ?, ?)");
$stmt->bind_param("sssss", $fname, $lname, $email, $pass, $session_id);
$stmt->execute();
$stmt->close();
```

Figure 60: Parameterized queries

The new code includes an effective execution of prepared statements that help against SQL injection attacks. The code does not directly put the user inputs into the SQL query but constructs a query which includes question mark (?) for the placeholder of the values. The bind_param() method took arguments $fname, $lname, $email, $pass, $session_id which binds them with placeholders where ‘s’ stands for string data type. This makes sure that input values are properly managed and eliminates the possibility of an SQL injection attack. The execute() method executes the query and close() method releases the prepared statement after operation is over.

3. Client-side Validation (JavaScript)

```javascript
<script>
    function validateNameInput() {
        const nameRegex = /^[A-Za-z]+$/;
        const fname = document.forms["account"]["fname"].value;
        const lname = document.forms["account"]["lname"].value;

        if (!nameRegex.test(fname) || !nameRegex.test(lname)) {
            alert("Names can only contain letters.");
            return false;
        }
        return true;
```

Figure 61: Client-side Validation (JavaScript)

The use of JavaScript improves the user experience and enforces security of the web application by validating before form submission. In the account.php file, validateNameInput function that is used to check if the first name (fname) and last name (lname) field contain only alphabets. The function also creates a regular expression nameRegex to match only letters (A-Z and a-z). When the user submits the form, the function uses the document.forms[“account”][“fname”].value for fname and documents.forms[“account”][“lname”].value for lname then tests them against the regular expression using the test(). If either of the names includes invalid characters, including numbers and special symbols, an alert message pops up informing the user of the mistake and the submission of the form is aborted by returning false.

Discussion of the Solution's Effectiveness

These solutions implemented provide the much-needed security updates on the web application focusing on SQL injection, and the proper handling of user input. It can be said that input validation, parameterized queries, and client-side validation form a strong barrier wall, which reduces the vulnerability of common SQL injection attacks and at the same time enhancing experience for the user.

One of the most important enhancements in the solutions is the ability to insert parameterized queries with prepared statements, as this eliminates the way users previously input data into SQL statements. With the use of placeholders (?) in the place of the actual user input, the problem of SQL injection no longer exists. When a user enters data for instance an email, password or name, bind_param () function safely inserts these parameters to the prepared statement so that the interpreter treats them as data and not code. This approach does not allow the hacker to feed the query with malicious code as the parameters are sanitized before being passed to the database.

The input validation mechanisms made it more secure system. The email validation checks that only valid e-mail addresses are entered and excludes from input any string that contains a potential SQL injection or other noxious code. The system also has the possibility to filter out ineligible or malicious email addresses using a PHP filter_var function with the FILTER_VALIDATE_EMAIL parameter. This is a very efficient and simple approach to make certain that only authentic email addresses are being recorded in the database.

Likewise, the password validation to ensure that users set the password of a specific length to a certain standard and that passwords do not contain dangerous characters or SQL injections. This enhances security as passwords cannot be used to corrupt the process of user authentication and hence increase security. This capability provides an effective mechanism of enforcing a strict regime on the creation of passwords, hence, effectively counteracting the creation of the frivolous passwords that are a common permeable layer for hackers to infiltrate into a network.

Client-side validation with JavaScript improves the usability of the website considerably. The use of regular expressions to filter specific form fields such as fname and lname returns the input error to the user before the data reaches the server. This cuts down the amount of work to be done by the server and makes the use of the site easier by eliminating recurrent typographical mistakes. Even though client-side validation is mostly designed for boosting user interface, it serves as the first filter so that the server does not accept hacks. However, it is important to note that client-side validation can be easily worked around by an attacker by simply disabling JavaScript, or editing the source code, thus it should not be used alone as a form of security. Nonetheless, server-side validation is still a necessity to check the data provided by the form.

All in all, these solutions are efficient in incorporating different kinds of security measures that will help to prevent attacks of SQL injection, improper input handling as well as weak password security. The implemented server-side validation, parameterized queries, and client-side checks comprise a strong system which enhances the general security protection level of the application greatly. But like many security mechanisms, it is vital to carry on upgrading the system for emerging threats and risks.

Discussion of Unresolved Issues and Proposals for Future Work

As mentioned previously in the report, several important security problems remain unsolved, although the present solution successfully resolves many of them that are crucial for the secure running of the web application. Continuing with the stated objectives and research questions, there are still unsolved problems along with some ideas for future research.

1. Password Change Issues in account.php

While the system has provided validation checks in cases of user registration and login, the process of password change is relatively vulnerable. Presently, the system avoids SQL injection on password fields by indicating a 'wrong current password' sign in case of invalid input. It remains uncertain to users that are within the lawful use of systems whether the problem was due to presenting an incorrect password or flawed attempt at input verification. Moreover, this area can still be vulnerable to attacks where an attacker might try to guess the location of the connection or perform other manipulations if they have already partially contacted the system.

A better solution would be to have a stronger password change process, or a password directive coming through that would check whether the existing password is properly authenticated before making any change. Furthermore, complex passwords shall be applied and stored as password hashes such as bcrypt and Argon2. These modern hashing algorithms, for example, employ the use of salting and stretching just in case passwords are vulnerable due to the breach on database. Besides practicing passwords safeguarding, other measures such as password control measures and password standards concerning the time for which a password remains active should be put in place to minimize chances of the passwords getting hacked.

Another improvement that could be implemented is multi-factor authentication (MFA). This would require users to give another factor (for instance, a code sent once to the user's email or phone number) before they could reset their passwords. The use of MFA is slowly becoming more common in companies for it increases security as even if the attacker gets the password, they cannot access the account.

2. Enhanced Client-side Validation

Although the current client-side script work that includes requests like checking for alphabetic characters in the name field improves the presentation quality and eliminates certain types of input errors, it can be more extensive. Presently, name validation makes input into fname and lname field to contain only letters only but could be improved to allow common naming etiquette like; the first letter of each name in upper-case or allow some special characters like apostrophe or hyphen that may occur in some names. For instance, although such titles as "O'Connor" or "Anne-Marie" would be acceptable to users of most cultures, the system dwells on their entry.

To improve client-side validation further, the following enhancements could be made:

a. Enhanced Name Validation:
   Adding valid name formats by incorporating special characters in a name such as apostrophes, hyphens and spaces as are common in names of different culture groups.

b. Field Length Limits:
   A validation process should also be put in place to check on the maximum number of characters allowed to be entered in the fields to avoid input overload that comes with long names that may stress the server or database.

c. Cross-field Validation:
   For the fields such as password and confirmation password, make client-side validations where the two values must match before sending it. Although this is normally done on the server, it is useful to first do a simple check in the client side for better user experience feedback.

Nonetheless it is essential to stress that client-side validation is not the only protection line as valid JavaScript can be disabled or manipulated by a user. Client-side validation should always be backed up by server-side validation so that any malicious data to the system is detected and addressed before it reaches the database or the application's logical flow.

3. Two-Factor Authentication (2FA)

One of the suggestions for future work is the integration of Two Factor Authentication (2FA) to enhance the security feature on the user accounts. This is because 2FA combines the factors of a password, which the user knows and a code sent to the user's mobile device or email, which the user has. If a hacker gets the user's password, he will still need the second factor which is the phone's or email's code to gain access to the account.

2FA has now become a common security practice and the utilization of this practice is even more demanding for applications that work with sensitive information. There are various methods for implementing 2FA such as, SMS tokens, email tokens, or authenticator applications like google authenticator, authy, and so on.

Conclusion

The final deliverables of this project comprise of the modified registration and log-in module for the Online Property Marketplace with strong security measures against SQL injection cases. These changes were done on the project files containing the HTML, CSS, JavaScript, SQL and PHP which are crucial in the web application deployment.

Upon evaluation of the original code, several flaws were detection in the login, registration and account files that could allow penetration of SQL injection attacks on the database. The following fixes were implemented:

1. Prepared Statements: Changed from the use of dynamic SQL queries to utilizing prepared statements in order not to directly manipulate the database query strings.
2. Input Validation and Sanitization: Improved checking on the user input which prevents invalid data type or format to be passed to the database.
3. Escaping Special Characters: Added escaping of characters that could be potentially used in the construction of SQL commands to avoid them being used in the command.
4. Error Handling: Enhanced measures to ensure errors log does not expose database information to potential assailants into the system.

After applying these fixes to the codes, the system underwent multiple tests to ensure the robust security measures were applied effectively and mostly ensure the system is immune to SQL injection attacks. The complete details of the tools used, the vulnerabilities identified, and the fixes that were implemented were recorded as well. These updates guarantee that there is mostly no risk of SQL injection attack to the Online Property Marketplace's login and registration module making the web application stronger and users' information safer.

However, some challenges in the future work are still visible, even though the current deliverables prevent most of the SQL injections. The future work includes adding the security issues related to changing passwords, improving the client-side validation and incorporating MFA and 2FA. All these measures would further enhance security of the system and therefore make the Online Property Marketplace even more secure against potential attacks as well as enhance the security of the user authentication process.

References

1. Balasundaram, I. and Ramaraj, E., 2011. An authentication mechanism to prevent SQL injection attacks. *International Journal of Computer Applications*, *19*(1), pp.30-33.
2. Halfond, WG, Viegas, J & Orso, A 2006, 'A classification of SQL-injection attacks and countermeasures', Proceedings of the IEEE International Symposium on Secure Software Engineering, pp. 13-15.
3. Awad, MM & Dhabi, A 2024, Building an Impenetrable PHP & SQL Login and Registration System, viewed 13 November 2024, <https://wordpress.org/plugins/secure-login-registration/>.
4. W3Schools 2024, PHP mysqli_real_escape_string() Function, viewed 26 November 2024, https://www.w3schools.com/php/func_mysqli_real_escape_string.asp.
5. Tiernok, J 2024, The Role of Client-Side and Server-Side Validation in Web Security, viewed 28 November 2024, https://www.securityjourney.com/blog/client-server-validation.
6. SecureCoding, 2024, Client-Side Validation vs. Server-Side Validation: Which One to Use?, viewed 28 November 2024, https://www.securecoding.org/validation.
7. Brightsec 2024, 'SQL Injection Payloads: Understanding and Mitigating Attacks', viewed 1 December 2024, https://brightsec.com/blog/sql-injection-payloads/#union-based-payloads.
8. Thiyab, R.M., Ali, M. and Basil, F., 2017, April. The impact of SQL injection attacks on the security of databases. In *Proceedings of the 6th International Conference of Computing & Informatics* (pp. 323-331). School of Computing.
9. OWASP Foundation 2023, *SQL Injection*, viewed 13 November 2024, <https://owasp.org/www-community/attacks/SQL_Injection>.