**CPSC 4660 – Final Report**

**SQL Injection Attack Prevention**

**By: Amanda Munden**

**#001148507**

**Due: November 25, 2019**

**Background:**

SQL injection attacks are some of the most common attacks on databases. There are multiple SQL injection strategies that are used to accomplish different goals when attacking a database. Some of the most common types of attacks are using tautologies which can allow you to bypass authentication and retrieve data you would not normally have access to, using a Union query which allows you to return additional data from a query, and using Piggy-Backed queries which allows you to alter the database by removing tables or tuples, changing values or adding values. To test my different preventative techniques that I will be implementing in this project, I will be using attacks from all three strategies to see how accurately each of the prevention methods stops the different kinds of attacks.

**Summary of Component Being Implemented:**

I implemented two different preventative techniques that are intended to reduce or eliminate SQL injection attacks. The first technique I implemented was “type or length detection” which I put into the HTML file to keep the user from inputting the wrong type of information and to limit how much could be put into the different input fields. The second technique that I implemented was “sensitive key word/special character filtering” which I implemented in the PHP code to check for sensitive key words or special characters that could allow other SQL code to be run. Instead of returning an error message like I originally proposed, I decided to add a slash to the input on the backend as to have all the input treated as just standard input and not a piece of code that could be run. The goal of implementing two different techniques was to determine if one better prevents SQL injection attacks than the other.

**Implementation Details:**

For the implementation of this project, I started by creating a database that consisted of three tables which were student, faculty, and course. In the student table, the columns consist of Student ID, usernames, passwords, first name, last name, street address, city, province, postal code, phone number, date of birth, and gender. The primary key for this table is the Student ID number. In the faculty table, the columns consist of Staff ID, usernames, passwords, first name, last name, street address, city, province, postal code, phone number, date of birth, and gender. The primary key in this table is the Staff ID number. In the course table, the columns consist of Course ID, course number, course name, section, the days the course is offered, the time the course is offered, and what classroom the course is taught in. The primary key for this table is the Course ID number. The structure of the database was kept simple as it was just used to run tests on, and the focus of the project was SQL injection prevention.

After creating the database, I then implemented a very basic and unprotected front end. I created a menu webpage called “menu.html” that had links to adding information to each of the tables and being able to search for information in each of the tables. The files to add information are: “AddStudent.html”, “addStudent.php”, “AddStaff.html”, “addStaff.php”, “AddCourse.html”, and “addCourse.php”. The html files do specify the input type but have no restraints on how much input can be added in the fields that take in strings. The php files take all input from the user without checking for key words or special characters and insert it into the database. The files to search the database are: “SearchStudent.html”, “searchStudent.php”, “SearchStaff.html”, “searchStaff.php”, “SearchCourse.html”, and “searchCourse.php”. The html files also do not have restraints on input and the php files do not check the incoming input in any way.

After the initial implementation was completed, I conducted a test of all the SQL injection methods that I intended to use to make sure that the base case was susceptible to all of them. I was able to use the tautology “OR 1=1” to get the search function to output everything in the table. I was also able to search in the course table and use UNION to figure out what was in the student table and UNION on all the stored usernames, passwords, and first and last names of everyone stored in the student table. I did try multiple ways to piggy-back in and drop a table, but I was unsuccessful at getting that attack to work. Once I was satisfied that the original front-end coding was very susceptible to SQL injection attacks, I then recoded the HTML and PHP documents to implement the two prevention strategies.

The first strategy that I implemented was type and length detection. I took the HTML files and introduced a stricter type for the information that could be input. I also added in a max length of characters that could be put in the input fields on the webpage to line up with the lengths that were set in the type definition of the database. Although it was only the HTML files that were changing for this strategy, I decided the best way to keep the project organized and user friendly was to create all new files. I used the file names listed above and add TLD to the name to distinguish which strategy those files were to be used for. I also created a new menu called “menuTypeLengthDetection.html” to run this strategy. Once the implementation of this prevention strategy was completed, I then tested this strategy against all the same SQL injection tests that were run on the base case.

Lastly, I implemented the second strategy which was sensitive key word/special character filtering. For this strategy, I recoded the php files to filter the incoming input before sending it into the SQL query. I also created a file called “SKWFunction.php” which contains the filtering function that I used for filtering the user input. Initially, I proposed that it would be more of a key word filtering function; however, after testing out the SQL injection attacks on the unprotected code, I realized that all the attacks required some sort of special character to work. Therefore, in the implementation of this function, I focused only on the special characters. The code could; however, be expanded to include key words as well. Although it was only the PHP files that were changing for this strategy, I again created all new files to keep the project organized and user friendly. I used the file names listed above and add SKW to the name to distinguish that those files were used for the sensitive key word/special character filtering. I also created a new menu called “menuSensitiveKeyWord.html” to run this strategy. Once the implementation of this prevention strategy was completed, I then tested this strategy against all the same SQL injection tests that were run on the base case.

**Evaluation Strategy:**

My evaluation strategy was to apply one of the prevention strategies to my test database and then try multiple kinds of attacks against the database and see how many of the attacks the strategy can prevent and how accurately it prevents the attacks. I will repeat this for the two prevention strategies to see which one can prevent the most attacks and if there is a difference in how well one strategy can prevents attacks compared to the other strategy.

**Results of the Evaluation:**

The results for the type length detection prevention strategy was that in some cases it was able to prevent some of the longer UNION attacks, but it was not always able to prevent the shorter tautology type attacks. Because this prevention strategy only checked for the type and maximum length of the input, fields that required a longer string were still open to attacks if they fit within the maximum length that could be entered. I chose to make the search item for course the Course Name because it is the longest string that the database required, and this helped us to see that a lot of injections were still possible for a column that required a long string input. I chose the last name as the search item for student and faculty because its length was only set to 20 characters and this would allow us to see that the length detection was able to stop some of the SQL injection attacks.

The results for the sensitive key word/special character filtering strategy was that it appeared to be able to stop all attacks. I say appeared because it was able to stop all the attacks I had been testing; however, it is possible that there could be other attacks generated that this strategy alone might not be able to stop. I had initially thought that maybe this method should out put an error message when a statement was entered that contained any of the filtered special characters; however, in the end it worked best to just input a slash in front of the character and let the database take the input how it was entered. Because the program was adding slashes to the input with special characters, the search function was unable to match that input with what was in the database. Therefore, searches just return a response of 0 Results found if an unwanted special character was input in the search. I was concerned at first that this was not the way that this should work, but I compared it to how mysqli\_real\_escape\_string worked, and both my function and that function reacted in the same way.

**Conclusions:**

After the testing of the two different prevention strategies, it became clear that the sensitive key word/special character filtering is a far superior strategy than using only type length detection. Although the goal of this project was to compare the different prevention strategies and evaluate their performance, one other thing I would take away from this project is how important it is to write good code that protects your database. It was quite scary to see just how much damage could be done to a database that was not protected by some form of SQL injection prevention strategy.

**References**

Yeole, A. S., Meshram, B. B. (2011). Analysis of different technique for detection of SQL injection. *ICWET '11: Proceedings of the International Conference & Workshop on Emerging Trends in Technology*. Publisher ACM.

Zhang, Haiyan, Zhang, Xiao. (2018). SQL Injection Attack Principles and Preventive Techniques for PHP Site. *CSAE '18: Proceedings of the 2nd International Conference on Computer Science and Application Engineering*. Publisher ACM.