CPSC 470 Final Project Report Dynamic Prepare Statements

Robert Booth

10th April 2022

1 Introduction

Every day the environment for website protection is evolving, constantly forcing developers to make changes to improve their security, and hackers to improve their attacks. SQL Injection Attacks are very common amidst websites that allows its users to enter input into a text box that is then compiled into a query and executed in the database. Without functions to check each input box to ensure that the user has not entered malicious data, the user can easily enter in MySQL keywords that can be executed by the database and perform very malicious injections, such as the deletion of entire tables, editing tables that should otherwise not be changed, among many other common attacks. This is possible because in a normal query, the user's input is compiled with the MySQL keywords. When it is compiled in this manner the user can inject virtually any MySQL keywords they want to change what the query will do. Prepare statements use a function called Prepare to construct a query before user input is inserted into the query. These Prepare statements are constructed and compiled before user input is inserted into the query, avoiding many of the common SQL Injection Attack techniques.

2 The Problem

The problem with Prepare statements is they are normally only usable for static inputs, for example, it would be easy to setup a prepare statement for a login page requiring two inputs for username and password. However, is it possible to use prepare statements for more dynamic inputs? The problem I am trying to solve with this research, is being able to use Prepare statements with dynamic, possibly limitless inputs, where the user has the possibility to enter in data into an unknown, possibly very large number of text boxes, and still be able to protect the database against SQL Injection Attacks. Additionally I will be attempting to execute single prepared statements for the entirety of their input, no matter how large, as having a potentially limitless number of queries executed to the database would be very stressful on the database, plus having one query for each row would simulate static inputs, meaning that nothing new would be being explored. I searched online but I have been unable to find anything that is asking this specific question, I only found articles that are using Prepare statements for their intended purpose, which is to defend against simple attacks.

3 Literature Survey

- Prepare statements are a MySQL keyword used for creating a different kind of query into the database. They are used normally to execute a query many times without the requirement of recompiling the query for each time it needs to be executed. Instead the Prepare statement is compiled before inputs are given to the query, and then the inputs are filled each time it needs to be executed. Because of this functionality, Prepare statements can be used many times without recompiling the query. Compiling a query is the most dangerous part of interacting with a database, because users, in certain situations, may be able to modify the query to perform other actions. Because prepare statements are able to be compiled before user input has been added, it eliminates this risk. Because of these unique properties, a variety of methods have been used to properly use Prepared Statements. One source used Prepared Statement Algorithms to generate a variety of Prepared Statements based on the inputs, allowing them to mitigate attacks through the strength of Prepared Statements. [3] These algorithms must keep prepared statement structure between the database and the user's input separate so that the user is unable to manipulate the query. [1] These algorithms can act as the mediator between the database and the user's input, and based on the inputs, it creates a proper Prepared Statement to send it to the database. [3]

- Types of SQL Injection Attacks can vary widely based on the keywords injected and what failures in the database occur when the attack is successful. There are 5 general categories of SQL Injection Attacks.

- Bypassing Web Application Authentication, where the attacker exploits an input field that is used the a query's "WHERE clause".
- Getting Knowledge of Database Fingerprinting which is when the user enters incorrect inputs to force error messages to appear and to learn about the internal structure of the database.
- Injection with Union Query, where the hacker is able to extract data from other tables that are not intended to be queried by using a UNION keyword to connect the correct table with additional tables.
- Additional Injected Query, where the hacker is able to enter inputs such that an additional query is injected and run in addition to, or instead of the original query.
- Remote Execution of Stored Procedures which is where the hacker executes procedures previously stored by the web developer.

These attacks are generally the most common methods of damaging or retrieving illegal data when attempting to hack a website. [1]

- The original inspiration for my project is when an article mentions PREPARE statements are useful for defending simpler inputs, but moves on to say that they cannot be used for dynamic inputs. [2]

4 Technical Material

All queries require the execution of a large string using SQL keywords. Normally the query is hard-coded to be executed on a certain number of inputs with a static string, only with variables being filled in. Dynamic Queries require a far different approach. The MySQL keywords are still the same, however the query requires each input to be ended in a comma, something not normally possible unless using a loop of some kind when the number of inputs is unknown. Additionally, Prepare statements require a Question Mark for each piece of data that is to be inputted. these question marks must be properly surrounded by Parentheses based on what row they are going to be inserted in. For a dynamic Query to work properly, as string of parentheses, question marks, and commas must be concatenated properly to allow the proper insertion of all the data. Additionally, prepare statements require an identifier in the Bind Parameters function call that tells the function the type of every single variable entered. In this case, it is a string of letters with each letter an identifier for a single piece of inserted data. All the data is stored in arrays, one for each separate text box in the input page, and these must be inserted in proper order, which means it must all be pushed in order into a single array, and using the unpacking operator, the array can be unpacked into the Bind Parameters function, such that all the data is in proper order. All of these factors together can generate a proper Prepared Statement.

On the user end, JavaScript was needed to allow HTML to be copied repeatedly whenever the user needed. This JavaScript appends a copy of a hidden display to the form that appears and then replaces its hidden type with a block display. The Name of the input box is the variable being passed through, holding the user input, and because we do not know the number of inputs the user desires, we must make the name an array type instead of a normal string.

The website features many tables and an input page for every table that uses standard, unprotected MySQL queries that are not Prepare statements and do not have any input validity checking before the input is sent to the database.

5 Methodology

I have implemented an algorithm that is able to generate a single prepared statement and insert it into the database no matter how many additional rows the user wants to add. The algorithm is able to do this because all of the additional forms the user creates in their input page are merely extensions on a single form. This form can handle an unlimited number of entries, and as it is one form, all of the data is sent to the querying page through the POST method contained in Arrays. Due to the nature of Prepared statements, this data now needs to be reorganized in a singular array, rather than a collection of arrays. This is required because the bind parameters function requires that data be inputted as single arguments in the order that they are to be inserted. Therefore, we must go piece by piece, in the order of the columns in the tables, moving the data from their individual arrays into a single large array. If we have two arrays by the name: lastName[], firstName[], then the data in the large array will be in the form: [firstName[0], lastName[0], firstName[1], lastName[1], firstName[2], lastName[2]... ...firstName[n], lastName[n]].

With that, completed, we need to generate a string of question marks to be used as the placeholder variables when compiling the Prepare statement. These question marks are wrapped in parentheses for each row, so using the above example the row would be 2 elements, therefore the question mark string, for each row, would be: (?, ?). Each set of these question marks would be separated by a space and a comma. Lastly, with the data prepared in a large array and the arguments string completed, a variable type identifier as the first argument of the bind parameters function must be constructed. Similar to the question mark string, it identifies the types of the that is about to be inserted into the query, and will require an identifier for every single piece of data in the array. This string, for each row, using the above example, would be 'ss' with no commas or separation between rows. The 's' tells the bind parameters function that the data is a string. similarly, 'i' for integer, 'd' for double. With that the only thing remaining to do is to use the unpacking operator, which in PHP is: "..." which, when an unpacked array is fed to a function, it does not see an array it instead sees each individual piece of data in that array, properly separated by spaces and commas between each.

6 Results

All attacks I have attempted do not succeed in properly performing the attack on the Prepared Statements. Any form or variation of the attack I tried the most, which was inserting a ;DROP TABLE 'test'; string into the query, would result in the table not being dropped, and either an error being thrown for invalid input, or would insert the data as normal, and that string would merely be stored in the database.

7 Future Related Questions

- How well will Dynamically Constructed Prepared Statements fare against Cross Site Scripting attacks versus Statically Constructed Prepared Statements. Would it be possible for Dynamically Constructed Prepared Statements to be more resilient against other types of attacks? Considering the fact that Cross Site Scripting attacks, if successful, are able to modify the query itself, I doubt they would make much difference.
- Can we quantify the time it takes to execute a dynamically constructed Prepared Statement versus executing a number of static Prepared statements that would perform the same query, to see which is more efficient?

References

[1] S. M. Sadegh Sajjadi and B. T. Pour. *Study of SQL Injection Attacks and Countermeasures*. Sept. 2013. URL: `http://ijcce.org/papers/244-E091.pdf` (visited on 01/27/2022).

[2] A. D. K. Stephen W. Boyd. *SQLrand: Preventing SQL Injection Attacks*. 2004. URL: `https://link.springer.com/content/pdf/10.1007%2F978-3-540-24852-1_21.pdf` (visited on 01/19/2022).

[3] S. Thomas, L. Williams, and T. Xie. *On automated prepared statement generation to remove SQL injection vulnerabilities*. 2009. URL: `https://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.143.6132&rep=rep1&type=pdf`.