Automated Fix Generator for SQL Injection Attacks

Fred Dysart and Mark Sherriff
Department of Computer Science, University of Virginia, Charlottesville, VA 22903
{ftd4q, sherriff}@virginia.edu

Abstract

A critical problem facing today's internet community is the increasing number of attacks exploiting flaws found in Web applications. This paper specifically targets input validation vulnerabilities found in SQL queries that may lead to SQL Injection Attacks (SQLIAs). We introduce a tool that automatically detects and suggests fixes to SQL queries that are found to contain SQL Injection Vulnerabilities (SQLIVs). Testing was performed against phpBB v2.0, an open source forum package, to determine the accuracy and efficacy of our software.

1. Introduction

According to the National Institute of Standards and Technology, SQL Injection Vulnerabilities (SQLIVs) amounted 14% of the total Web application vulnerabilities in 2006 [3]. This statistic is surprising due to the simple approach of using prepared statements to write secure applications without injection vulnerabilities. *Our objective is to design a tool that would allow developers to easily ascertain how secure their applications were against SQLIVs and how to immediately fix any vulnerabilities found.*

The tool presented in this paper identifies potentially vulnerable SQL queries and provides a new secure solution using prepared statements. Prepared statements separate the structure of an SQL statement from the data that is provided by an outside entity. When used properly, prepared statements can turn a previously vulnerable query into one that is secure while maintaining the same functionality. For this particular version of our tool, we only targeted applications written in PHP that interact with a MySQL database though subsequent versions may expand upon the particular programming language and database used.

2. Related Work

Thomas et. al. introduced a solution that used prepared statements to automatically remove vulnerabilities from SQL queries [4]. He developed an algorithm in Java that parsed Java files and created functionally equivalent prepared statements for every query that was found to be vulnerable to injection attacks. Our technique improves upon Thomas's work by improving the algorithm for generating prepared statements and by expanding the work to PHP. In Thomas's solution, any variable that defines a prepared statement's structure must be within the same scope as the execution of that statement. Our solution does not incur this limitation because the generation of the prepared statement fix is not dependant on the actual statement declaration in the code.

3. SecurePHP

Our solution is called SecurePHP. It was written in C# and utilized the .NET framework. There were two main design decisions that guided us through development. First, we wanted the tool to be easy to use. While all applications should be simple for a user to interact with, the reasoning behind our decision was that the increasing amount of SQLIVs present in software might be from legacy software that is not maintained by a developer with experience fixing security issues [1]. We wanted to provide a solution that could be run against a source code repository and quickly provide visual feedback to the developer on the current state of their application.

Second, we wanted the results presented in such a manner that it is easy for the user to find and fix any vulnerabilities found. The goal of this tool is to aid developers in their maintenance of software, so the speed at which the vulnerabilities could be found and fixed was of great importance.

4. Algorithm Details

SecurePHP has three main steps: vulnerability detection, prepared statements creation, and report generation. After the user has specified the root directory of their application via SecurePHP's GUI, as seen in Figure 1, SecurePHP recursively scans every file ending in the .php extension in that directory, including those found in subsequent directories.

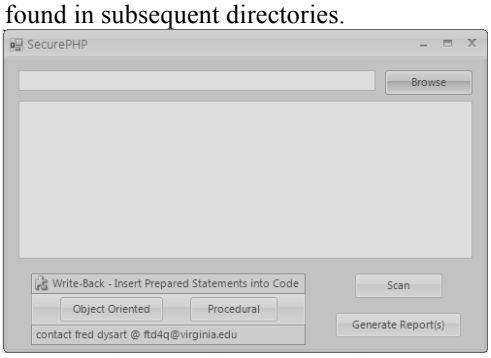

Figure 1: SecurePHP GUI

Vulnerability detection is accomplished by parsing an SQL statement and identifying any variables used. If these variables are used as part of the SQL structure without having been previously validated, then the query is marked as vulnerable. Once a vulnerable query is detected, a `SQLStmt` object is created and the following information is stored therein: the full SQL query, position found in file, and the name of the file where it was found.

Once the application completes the scan of every file, it then creates a `PreparedStmt` object for each `SQLStmt`. This manipulation incurs a bit of overhead as there are now two objects stored in memory, a `SQLStmt` and a `PreparedStmt`, for each vulnerability found, but this is required to generate adequate reports. A new query is then generated for the `PreparedStmt` with an array of replaced parameters and an array of integers corresponding to the parameter locations.

After the creation of the prepared statements is completed, the user has the option to generate reports for the files containing SQLIVs. Report generation consists of creating a new directory in the application's root directory called [`Application name`]_Reports. The reports folder will contain a text file corresponding to each PHP file found to contain SQLIV(s). An example of a section of code found in a report file can be found in Figure 2.

```
************************************
********** Vulnerable SQL **********
************************************

"SELECT user, pass
        FROM users
        WHERE pass = $pass"

************************************
*********** Secure SQL ************
************************************

$sql = "SELECT user, pass
                FROM users
                WHERE pass = ?";

if( $stmt = $db->prepare($sql) )
{
   $stmt->bind_param('s', $pass);
   $stmt->execute();
   $stmt->bind_result($user, $pass);

   while( $stmt->fetch() )
   {
      //Information stored in $user
      //and $pass available here
   }
   $stmt->close();
}
```

Figure 2: Example Report

The user also has the option of writing-back the prepared statements to the .php source files. We chose to create entirely new files for this action rather than writing over the user's original code. This allows the user to revert back to their original code if for some reason the change has some undesired effects on their application.

5. Results

We used SecurePHP to examine phpBB, an open source-source forum system, (www.phpbb.com) to determine the efficacy of the developed solution. PhpBB's root directory contains seventy-seven .php files and over 3 MB worth of code. SecurePHP was able to parse and build the prepared statements in well under one second (.421 and .0015 seconds for parsing and prepared statement creation respectively). In total, over 37,000 lines of code were parsed and 663 SQL statements were found. Out of these 663 SQL statements, 328 were found to contain possible SQLIVs. All of which had a prepared statement counterpart generated.

While all of the 328 possible SQLIVs may not lead to injection attacks, the purpose of our solution is to notify the user of any possible weaknesses and suggest fixes. PhpBB used older techniques to secure their queries from injection attack, such as escaping potentially dangerous characters. These techniques work against protecting against injection attacks, but they incur more processing on the application server. Also, when a query is used more than once, prepared statements can actually lead to an increase in performance because the procedure is temporarily stored on the MySQL server [2].

6. Conclusion and Future Work

Having the ability to quickly obtain visual feedback on the current state of security of one's application would help developers maintain their large code bases easily. The reports can also be used as a metric to display how secure one's system is to managers or lead developers.

Future work could be done to build in more features to SecurePHP. Currently it has difficulty tracking SQL statements that are built conditionally through multiple code paths. Also, more PHP projects could be tested to further increase the SQL detection of SecurePHP due to the varying nature of which developers can write SQL.

It is also important to note that while this solution specifically targets PHP and MySQL, the idea of automated fix generation for SQLIAs can be extended to any programming language that supports prepared statements.

7. References

[1] S. Barnum, McGraw, G., "Knowledge for Software Security," *Security and Privacy Magazine, IEEE*, vol. 3, no. 2, 2005, pp. 74 – 78.

[2] Z. Greant & G. Richter, *ext/mysqli: Part I – Overview and Prepared Statements*, 2004, http://devzone.zend.com/node/view/id/686, accessed October 15, 2007.

[3] NIST, National Vulnerability Database, 2007, http://nvd.nist.gov/, accessed January 16, 2007.

[4] S. Thomas and L. Williams, "Using Automated Fix Generation to Secure SQL Statements," 3rd International Workshop on Software Engineering for Secure Systems, Minneapolis, Minnesota, USA, 2007, pp. 1-7.