SQL Injection Prevention Using Query Parsing

Pooja Yadav
Department of Computer Science and Engineering
Gurgaon College of Engineering, Gurgaon, Haryana, India

Abstract- Web sites are dynamic, static, and most of the time a combination of both. Web sites need protection in their database to assure security. An SQL injection attacks interactive web applications that provide database services. These applications take user inputs and use them to create an SQL query at run time. In an SQL injection attack, an attacker might insert a malicious SQL query as input to perform an unauthorized database operation. Using SQL injection attacks, an attacker can retrieve or modify confidential and sensitive information from the database.

It may create problem with the confidentiality and security of Web sites which totally depends on databases. This thesis presents an implementation of SQL Injection Prevention Technique that implicitly protects the applications which are written in PHP from SQL injection attacks. It uses an original approach that combines static as well as dynamic analysis.

Keywords – SQL Injection, Malicious, Confidential, Dynamic analysis

I. INTRODUCTION

In recent years, widespread adoption of the internet has resulted in to rapid advancement in information technologies. The internet is used by the general population for the purposes such as financial transactions, educational endeavors, and countless other activities. The use of the internet for accomplishing important tasks, such as transferring a balance from a bank account, always comes with a security risk. Today's web sites strive to keep their user's data confidential and after years of doing secure business online, these companies have become experts in information security. The database systems behind these secure websites store non-critical data along with sensitive information, in a way that allows the information owners quick access while blocking break-in attempts from unauthorized users.

A common break-in strategy is to try to access sensitive information from a database by first generating a query that will cause the database parser to malfunction, followed by applying this query to the desired database. Such an approach to gaining access to private information is called SQL injection. Since databases are everywhere and are accessible from the internet, dealing with SQL injection has become more important than ever. Although current database systems have little vulnerability, the Computer Security Institute discovered that every year about 50% of databases experience at least one security breach. The loss of revenue associated with such breaches has been estimated to be over four million dollars. Additionally, recent research by the "Imperva ApplicationDefense Center" concluded that at least 92% of web applications are susceptible to "malicious attack" (Ke Wei, M. Muthuprasanna, Suraj Kothari, 2007).

A. EXAMPLE

To get a better understanding of SQL injection, we need to have a good understanding of the kinds of communications that take place during a typical session between a user and a web application. The following figure shows the typical communication exchange between all the components in a typical web application system.

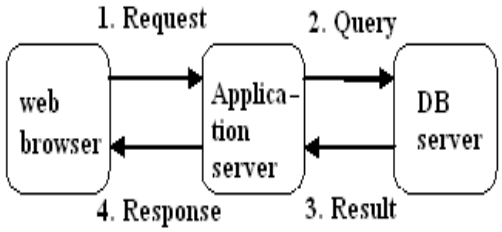


Figure 1: Web application Architecture

A web application, based on the above model, takes text as input from users to retrieve information from a database. Some web applications assume that the input is legitimate and use it to build SQL queries to access a database. Since these web applications do not validate user queries before submitting them to retrieve data, they become more susceptible to SQL injection attacks. For example, attackers, posing as normal users, use maliciously crafted input text containing SQL instructions to produce SQL queries on the web application end. Once processed by the web application, the accepted malicious query may break the security policies of the underlying database architecture because the result of the query might cause the database parser to malfunction and release sensitive information.

B. Impact of SQL Injection

Once an attacker realizes that a system is vulnerable to SQL Injection, he is able to inject SQL Query / Commands through an input form field. This is equivalent to handing the attacker your database and allowing him to execute any SQL command including DROP TABLE to the database!

An attacker may execute arbitrary SQL statements on the vulnerable system. This may compromise the integrity of your database and/or expose sensitive information. Depending on the back-end database in use, SQL injection vulnerabilities lead to varying levels of data/system access for the attacker. It may be possible to manipulate existing queries, to UNION (used to select related information from two tables) arbitrary data, use sub selects, or append additional queries.

In some cases, it may be possible to read in or write out to files, or to execute shell commands on the underlying operating system. Certain SQL Servers such as Microsoft SQL Server contain stored and extended procedures (database server functions). If an attacker can obtain access to these procedures, it could spell disaster.

Unfortunately the impact of SQL Injection is only uncovered when the theft is discovered. Data is being unwittingly stolen through various hack attacks all the time. The more expert of hackers rarely get caught.

II. SQL INJECTION DISCOVERY TECHNIQUE

It is not compulsory for an attacker to visit the web pages using a browser to find if SQL injection is possible on the site. Generally attackers build a web crawler to collect all URLs available on each and every web page of the site. Web crawler is also used to insert illegal characters into the query string of a URL and check for any error result sent by the server. If the server sends any error message as a result, it is a strong positive indication that the illegal special meta character will pass as a part of the SQL query, and hence the site is open to SQL Injection attack. For example Microsoft Internet Information Server by default shows an ODBC error message if an any meta character or an unescaped single quote is passed to SQL Server. The Web crawler only searches the response text for the ODBC messages.

III. SQL INJECTION ATTACKS

A SQL Injection attack is a form of attack that comes from user input that has not been checked to see that it is valid. The objective is to fool the database system into running malicious code that will reveal sensitive information or otherwise compromise the server.

There are two main types of attacks. First-order attacks are when the attacker receives the desired result immediately, either by direct response from the application they are interacting with or some other response

mechanism, such as email. Second-order attacks are when the attacker injects some data that will reside in the database, but the payload will not be immediately activated.

IV. SQL PARSE TREE VALIDATION

A Parse tree is nothing but the data structure built by the developer for the parsed representation of a statement. To parse the statement, the grammar of that parse statement's language is needed. In this method, by parsing two statements and comparing their parse trees, we can check if the two queries are equal. When attacker successfully injects SQL into a database query, the parse tree of the intended SQL query and the resulting SQL query generated after attacker input do not match. The following figure shows the representation of a parse tree.

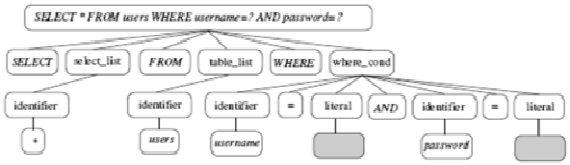


Figure 2: A SELECT query with two user inputs

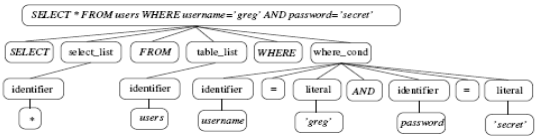


Figure 3: The same SELECT query as in Figure 2, with the user input inserted

In the above parse tree the programmer-supplied portion is hard-coded, and the user- supplied portion is represented as a vacant leaf node in the above parse tree. A leaf node must be the value of a literal, and it must be in the position where vacant space is located. The SQL query for the above parse tree is as below.

```
SELECT * FROM users WHERE
username=? AND password=?
```

V. METHOD TO PREVENT SQL INJECTION ATTACKS

A. *MODEL-BASEDGUARD CONSTRUCTOR PREVENTION APPROACH*

Model-based guard constructor prevention is an efficient method in preventing an SQL injection attack. This method is established on breaking the suitable conjunction of input, code, data, and database access situation that would employ an SQL injection attack. Spontaneously inserting appropriate guards before allowing the access to the database, we can avoid an SQL injection attack. As shown in the Figure-5, initially instrument the PHP string to collect the samples of query which authentically used at database application program interface call point. These queries are called as a set of trusted test cases. From the flow of the diagram, we can easily understand the prevention of an SQL injection attack. Instrumentation is nothing but to add an output instruction before database application interface calls, as below.

```
Sql_query(… Expression…);
```

After passing this expression through automated approach it becomes:

```
$string = Expression;

fRead($file handle; $string));

$result = sql_query($ string);
```

After running the trusted test cases to gather the plain text strings that are produced dynamically at various call sites matching to trusted queries. It is a straightforward to create model guards from sets of ASTs leading to legitimate queries. Justifiable queries are parsed by automated approach and corresponding "ASTs" is stored for every call site. To avoid generalization between queries "ASTs" are stored independently.

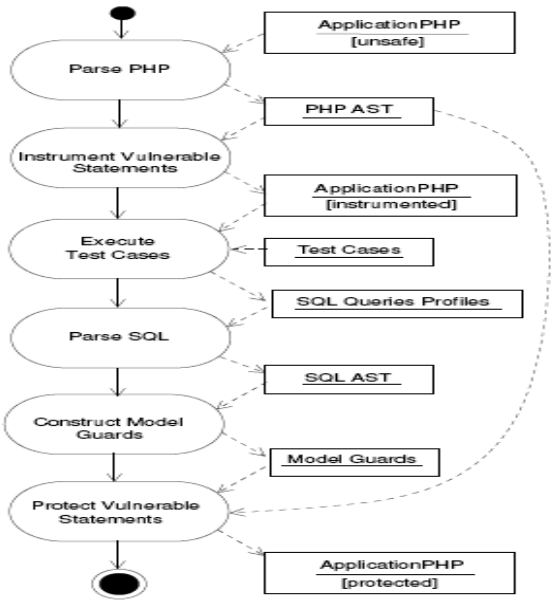


Figure 4: Automated Protection

ASTs are generalized by type rather than image, because constants, strings and additional types of data are also stored in the ASTs. On the other hand, application dependent identifiers, such as the names of the tables, number of columns, and rows, are counted as a part of syntactic structure of the SQL query which plays crucial role to prevent malicious substitution of table or column names in the valid queries. Therefore this method permits number of queries with same syntactic structure, but with different values of data. Using special call site, model guard invokes the SQL parser on the database, where we are working currently to and obtains the matching SQL AST. The formed "AST" is compare with the stored valid "ASTs" for the same call site. If the match points towards positive result then the current query has a compatible syntactic structure with the valid query from the trusted set. Only positively equivalent queries are allowed to be processed in the database application programming interface, and all other queries are rejected. In this way we prevent the access to the database from crafted malicious queries.

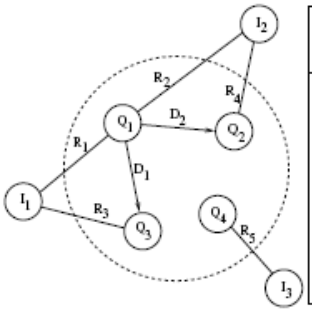
Generally "ASTs" are stored as images, but it is stored as token strings containing token types where an application table names and field names have become keywords. Tables' names have been stored as identifier token types in local configuration. In the subsequent section, these token strings called as reference patterns.

Giving proper call to the model-based guard, we can protect the application from an SQL injection attack. Model guard construction is equivalent to the average length of the SQL queries executed in the test, and number of test cases. Ettore Merlo, DominicLetarte, and Giuliano Antoniol used automated construction process for the model-guards which is very simple and have an enough scope to make this model more complicated to increase power of the parsing queries. Model-base guard is much better than the fixed form per call site method. Model-guard build automatically depends on the dynamic approximation for security specification. Small

number of legitimate queries at a call site also affects the efficiency of automation. This approach provides a feasible amount of protection from an SQL injection attack.

B. Static analyses of stored procedure

In static analysis authors' provides the parser called stored procedure parser which is used to extracts the "control flow graph" from the saved procedures, we can see in detail about the control graph in following section. At the start, we label every execution statement in the control flow graph and then use the backtracking method to verify all statements participated in the formation of the SQL statement in the control flow graph. In the SQL graph, statements which are depended on the user's input are screened and flags are set on it to monitor their behavior at run time. In this method, using Finite State Automaton, we compare the statement with dynamically created SQL statement of user inputs with the original SQL statement. The statement created by user's input which tries to change the original pattern of the parser will indicated by flag as dangerous statement and provides the related information. More than one execution statement may be possible for single "stored procedure" statement. There are different kinds of procedure statements available, and only the statements which accept input from user are vulnerable to an SQL injection attack. Now using SQL control graph we try to optimize the query that need to process dynamically in order to provide validation. Following figure gives a clear understanding of static analysis. Four different SQL queries Q1, Q2, Q3, and Q4 are in the stored procedure shown as nodes within a boundary displayed in dotted circle. I1, I2, and, I3 are the three different inputs received from users which are from outside of the logical boundaries. Suppose a user enter the input I in the SQL query Q and the relationship between input I and query Q is represented by R. D represents the dependencies in SQL diagram that links the one SQL query to another. The user input 'I' accepted by previous query is transfers to another query through the dependency link. In SQL queries one of these nodes is selected as a representative query and it is considered a start point to point other queries. Dependency in the figure is shown by directed arrows.



Q_1, Q_2, Q_3, Q_4	SQL Queries
I_1, I_2, I_3	User Inputs
R_1	$I_1 \in Q_1$
R_2	$I_2 \in Q_1$
R_3	$I_1 \in Q_3$
R_4	$I_2 \in Q_2$
R_5	$I_3 \in Q_4$
D_1	$Q_3 \subset Q_1$
D_2	$Q_2 \subset Q_1$

Figure 5: SQL- control graph

A.

1) Advantages of static analysis

- SQL graph representation used to reduce the runtime scanning overhead of program by preventing the number of queries that are not required to execute in stored procedure.
- SQL control graph does not include the query which does not take an input from user.
- The queries which includes input from user to access the database information are counted towards SQL control graph representation.

C. Dynamic analysis

In dynamic analysis, SQL injection attack checker function is used to categorize the user input. In this method, author used "current session" identifier to identify the input taken from user, and using same session id, builds a finite state automaton. Figure 5 shows the finite state automaton that accepts inputs from user. To check legitimacy of SQL statement received from user, the SQL statement along with user inputs is compared with corresponding SQL statement of finite state automaton. If the SQL queries generated at run time uses the user input is not satisfy the semantics of the intended SQL queries in the FSA (Finite State Automata), then these SQL queries are set as SQL injection attack otherwise these queries should passed to the database access.

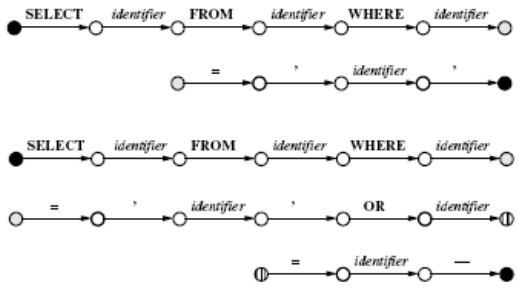


Figure 6: SQL Injection Attack Recognition and SQL-FSM Contravention

Hence, we can easily obviate the crafted malicious queries and only permits the legitimate queries to access databases. Due to use of finite state automata this method achieves perfect secrecy to screen the legitimate queries.

VI. GOAL

The goal of this project is to build and use efficient method to prevent SQL injection vulnerability from plain text SQL statements. In an automated method approach, a server will gather information about previously known vulnerabilities, specifically SQL statements, generate a patch, and apply patch. The process can be completed by someone with no security expertise and secure legacy code, which will allow developers to fix the SQL injection vulnerability.

VII. CONCLUSION

Most of the web applications uses intermediate layer to accept a request from the user and retrieve sensitive information from the database. Most of the time they use scripting language to build intermediate layer. To breach security of database hacker often uses SQL injection techniques. Generally attacker tries to confuse the intermediate layer technology by reshaping the SQL queries. Perhaps, attacker will change the activities of the programmer for their benefits. A number of methods are used to avoid SQL injection attack at application level, but no feasible solution is available yet. This paper covered most powerful techniques used for SQL injection prevention. From my research it concludes that automated technique for preventing, detecting and logging the SQL injection attack in 'stored procedure' is commonly used and they are concrete method. Graph control method is also good for small databases systems.

REFERENCES

[1] Wei, K., Muthuprasanna, M., & Suraj Kothari. (2006, April 18). "Preventing SQL injection attacks in stored procedures". Software Engineering IEEE Conference.
[2] Thomas, Stephen, Williams, & Laurie. (2007, May 20). "Using Automated Fix Generation to Secure SQL Statements". Software Engineering for Secure Systems IEEE CNF.
[3] Merlo, Ettore, Letarte, Dominic, Antoniol & Giuliano. (2007 March 21). "Automated Protection of PHP Applications Against SQL-injection Attacks Software Maintenance and Reengineering, 11th European *Conference IEEE CNF.*
[4] Wassermann Gary, Zhendong Su. (2007, June). "Sound and precise analysis of web applications for injection vulnerabilities". ACM SIGPLAN conference on Programming language design and implementation PLDI.
[5] Gregory T. Buehrer, Bruce W.Weide,and Paolo A.G.Sivilotti. The Ohio State University Columbus, OH 43210 "Using Parse Tree Validation to Prevent SQL Injection Attacks"
[6] William G.J. Halfond and Alessandro Orso. College of Computing Georgia Institute of Technology. "Preventing SQL Injection Attacks Using AMNESIA".
[7] Frank S. Rietta 10630 Greenock Way Duluth, Georgia 30097." Application Layer Intrusion. Detection for SQL Injection".
[8] Martin Bravenboer, Eelco Dolstra, Eelco Visser, Delft University of Technology The Netherlands. "Preventing Injection Attacks with Syntax Embeddings A Host and Guest Language Independent Approach"
[9] Yuji Kosuga, Kenji Kono, Miyuki Hanaoka Department of Information and Computer Science Keio University. "Syntactic and Semantic Analysis for Automated Testing against SQL Injection".
[10] Hal Berghel, "Hijacking the Web".