A Simple and Efficient Framework for Detection of SQL Injection Attack

Witt Yi Win
University of Technology (Yatanarpon Cyber City)
Pyinoolwin, Myanmar
wittyiwin09@gmail.com

Hnin Hnin Htun
University of Technology (Yatanarpon Cyber City)
Pyinoolwin, Myanmar
hninhninhhtun@gmail.com

Abstract—Web applications have become an integral part of the daily life. One of the most serious types of attack against web applications is SQL injection. SQL injection is a type of attack which the attacker adds Structured Query Language code to a web form input box to gain access or make changes to data. This paper proposes a simple and efficient framework to detect SQL injection attacks. The method converts the runtime query into sequence of tokens and then compares it with the predetermined queries. In order to reduce runtime validation, the possible queries at the query execution points are separately stored during static analysis. This method uses combined static and dynamic analysis.

Keywords: Static Query, SQLIA, Dynamic Query, Database Security, Static Analysis, Runtime Validation

I. INTRODUCTION

As networks and the internet have advanced, many offline services have moved online. Nowadays, most online services use web services. The ability to access the web anywhere and anytime is a great advantage; however, as the web becomes more popular, web attacks are increasing. Most web attacks target the vulnerabilities of web applications, which have been researched and analyzed at OWASP [1].

Database-driven web applications have become widely deployed on the Internet, and organizations use them to provide a broad range of services to their customers. These applications, and their underlying databases, often contain confidential, or even sensitive, information, such as customer and financial records. This information can be highly valuable and makes web application an ideal target for attacks. In fact, in recent years there has been an increase in attacks against these online databases.

Currently, one of the most dangerous and common threats to databases and Web applications is the SQL injection attack. It typically involves malicious modifications of the user SQL input either by adding additional clauses or by changing the structure of an existing clause. SQL injection enables attackers to access, modify, or delete critical information in a database without proper authorization. In spite of being a well-known type of attack, the SQL injection remains at the top of the published list of security threats. The solutions proposed so far seem insufficient to prevent and block this type of attack because these solutions lack the learning and adaptation capabilities for dealing with 0-day (previously unseen) attacks as well as new or future variations of attacks. Furthermore, the vast majority of these solutions are based on centralized mechanisms, with little capacity to work in distributed and dynamic environments [2].

For preventing the SQLIAs, defensive coding has been offered as a solution but it is very difficult. Not only developers try to put some controls in their source code but also attackers continue to bring some new ways to bypass these controls. Hence it is difficult to keep developers up to date, according the last and the best defensive coding practices. On the other hand, implementing of defensive coding is very difficult and need to special skills and also erring. These problems motivate the need for a solution to the SQL injection problem [3].

II. TYPES OF SQLIA

In this section, we present and discuss the different kinds of SQLIAs known to date. The different types of attacks are generally not performed in isolation; many of them are used together or sequentially, depending on the specific goals of the attacker.

A. Tautologies

A SQL tautology is a statement that is always true. The most common usages are to bypass authentication pages and extract data. In this type of injection, an attacker exploits an injectable field that is used in a query’s WHERE conditional.

B. Logically Incorrect Queries

This attack takes advantage of the error messages that are returned by the database for an incorrect query. These database error messages allow attacker to find out the vulnerable parameter in an application and the database schema. When performing this attack, an attacker tries to inject statements that cause a syntax, type conversion, or logical error into the database. Using this information about the schema of the database, an attacker can then create further attacks.

C. Union Query

Union query injection is called as statement injection attack. In this attack, attackers insert additional statement into the original SQL statement. The result of this attack is that the database returns a dataset that is the union of the results of the original first query and the results of the injected second query.

D. PiggyBacked Queries

In this attack type, an attacker tries to inject additional queries into the original query. Attackers are not trying to modify the original intended query; instead, they are trying to include new and distinct queries that “piggy-back” on the original query. This type of attack can be extremely harmful because attacker can use it to inject virtually any type of SQL command.

E. Stored Procedures

Stored procedure is a part of database that programmer could set an extra abstraction layer on the database. SQLIAs of this type try to execute stored procedures present in the database. Stored procedures are often written in special scripting languages, they can contain other types of vulnerabilities, such as buffer overflows, that allow attackers...
to run arbitrary code on the server or escalate their privileges. This type of attack works as piggy-back attack.

III. RELATED WORK

Over the past decade, a lot of work has been accomplished by the research community in providing new techniques to detect and prevent SQLIAs. In this section, we discuss state-of-the-art in SQLIA detection and prevention techniques.

Yadav et al. [5] introduced an authentication scheme for preventing SQL Injection attack using Advance Encryption Standard (AES) using hash function for stored password and user id. Encrypted user name and password are used to improve the authentication process with minimum overhead. The server has to maintain three parameters of every user: user name, password, and user’s secret key. This paper proposed a protocol model for avoiding SQL Injection attack using AES (PSQLIA-AES) with hash function. This modeling explicitly captures the particular subtle incidents triggered by SQLIA adversaries and corresponding state transitions. This technique has a limitation also. This technique can be implemented in the beginning of website development. Reengineering of website will have to be done to implement this technique on the existing website.

Kadirvelu et al. [6] proposed an intelligent dynamic query intent evaluation technique to learn and predict the intent of the SQL queries provided by users and to compare the identified query structure with the query structure which has been generated with user input in order to detect possible attacks by unethical users automatically. This type of evaluation is helpful in reducing the need for the user to have more consciousness when SQL queries are written. The main advantage of this system is that it applies the decision tree classification algorithm which is enhanced with temporal rules to find the unethical users intelligently at the query execution points where the database manager of the system can be informed of the new possible query execution points with an intent for attacks, and thereby preventing the SQL injection attacks.

Salama et al. [7] proposed a framework based on misuse and anomaly detection techniques to detect SQL injection attack. The main idea of this framework is to create a profile for legitimate database behavior extracted from applying association rules on XML file containing queries submitted from application to the database. It then used data mining technique for fingerprinting SQL statements and uses them to identify SQLIA. This set of fingerprints is then used to match incoming database transactions. If the set of fingerprints in the legitimate set is complete, any incoming transaction whose fingerprint does not match any of those in the legitimate set is very likely to be an intrusion. As a second step in the detection process, the structure of the query under observation will be compared against the legitimate queries stored in the XML file thus minimizing false positive alarms.

Hidhaya et al. [8] developed a method to detect the SQL injection. It used a Reverse proxy and MD5 algorithm to check out SQL injection in user input. Using grammar expressions rules to check for SQL injection in URL’s. This system does not do any changes in the source code of the application. The detection and mitigation of the attack is fully automated. By increasing the number of proxy servers the web application can handle any number of requests without obvious delay in time and still can protect the application from SQL injection attack. In future work, the focus will be on optimization of the system and removing the vulnerable points in the application itself.

Dharam et al. in [9] proposed a Runtime Monitoring framework that is used in the development of runtime monitors. The framework uses two pre-deployment testing techniques, such as basis-path and data-flow to identify a minimal set of all legal/valid execution paths of the application. Runtime monitors are then developed and integrated to perform runtime monitoring of the application, during its post-deployment for the identified valid/legal execution paths. The results of their study show that runtime monitor developed for the application was successfully able to detect all the tautology based attacks without generating any false positives. The important limitation of the proposed technique is that it can detect only tautology based SQLIAs.

Sharma et al. in [10] introduced an effective detection method RDUD for SQL injection attack which is an enhanced version of DUD [11]. The method comprises a supervised machine learning approach using a Support Vector Machine (SVM) to learn and to classify a query at runtime. Legitimate web profile and attack web profile are generated for each of the web-application software which consists of a set of production rules extracted from the dynamic SQL queries. Both the web profiles are generated during training phase. At runtime a dynamic SQL query is matched with each of the web profile and accordingly it classify based on the matching distance. RDUD is independent of the developer’s initialization of syntax rules, valid trusted string database, static or pre-generated program code checking, etc.

IV. PROPOSED SYSTEM

Various SQLIA detection techniques have been proposed in literature using offline static analysis and runtime attack detection. Our approach addresses SQLIAs by combining static analysis and runtime monitoring. In its static part, our approach uses program analysis technique to automatically build the abstract legitimate queries that could be generated by the application. Although our technique uses the same basic framework, we store the obtained abstract legitimate queries separately according to the query statement for reducing the runtime validation time. In its dynamic part, our approach monitors the dynamically generated queries at runtime and checks them for compliance with the statically-generated queries. Fig.1 shows the overview of the proposed system.

![Figure 1: Overview of the Proposed System](http://ijcser.org)

© http://ijcser.org e-ISSN: 2321-4198 p-ISSN: 2321-418X Page 27
public class Show extends HttpServlet {
    ...
    1. public ResultSet getEmpInfo(String login, String password) {
        ...
    2.   Connection conn = DriverManager.getConnection("Empldir");
    3.   Statement stmt = conn.createStatement();
    4.   String queryString = ""
    5.   queryString = "SELECT info FROM emps WHERE ";
    6.   if ((! login.equals("")) && (! password.equals("")))
    7.       queryString += "login=" + login + " AND pass=" + password + ""
    8.   ResultSet rs = stmt.executeQuery(queryString);
    9.   }
    10. else
    11.    queryString += "login='guest'
    12.   ResultSet rs = stmt.executeQuery(queryString);
    13.   return rs;
    ...
    ...
    ...
}

Figure 2: Example Servlet.

A. Static Analysis

We perform a simple scanning of the application code to identify hotspots. A hotspot is defined as a point in the application code that issues SQL queries to the underlying database. Within each hotspot, we are interested in computing the possible values of the query string passed to the database. To do this, we use Java String Analyzer (JSA) that is a tool for performing static analysis of the application source code. JSA constructs a flow graph that abstracts away the control flow of the program and represents string-manipulation operations performed on string variables. For each string of interest, the technique analyzes the flow graph and simulates the string manipulation operations that are performed on the string.

Fig. 2 represents the sample web application where we identify lines 8 and 11 as hotspots. The result of the string analysis is a Non-Deterministic Finite Automaton (NDFA) that expresses all the possible values a particular string can assume using single character transitions in the automaton. The string analysis is conservative, so the NDFA for a string is an overestimate of all the possible values of the string. We perform a depth first traversal of the NDFA for the hotspot and group characters as SQL keywords, operators, or literal values and create a transition. For example, a sequence of transitions labeled ‘S’, ‘E’, ‘L’, ‘E’, ‘C’, and ‘T’ would be recognized as the SQL SELECT keyword and suitably grouped into a token labeled “SELECT”. Any token that is not a SQL keyword, SQL operator, or delimiter is recognized as a string token. A string token can either be a constant string, hard-coded in the application or a variable string, a string that corresponds to a variable related to some user input. We use the generic label ‘Var’ for the user input string variables that can be changed at runtime.

Each token is defined as Token ID using Table 1 as shown in Table 1. SQL keywords, Logical Operators and Relational Operators are initialized in the Token Table. The obtained literal values which are not initialized in the table are marked as new tokens and stored in the token table. Then, the NDFA of each hotspot is transformed as a static query pattern. Fig. 3 shows Non-Deterministic Finite Automaton, query model and query pattern for the hotspot at line 8 in sample web application. We can construct a query pattern for each of the hotspots in the program. These data structures capture the semantics of the different SQL queries that are to be sent to the database at runtime. Any user input would be compared against this template and any change in the query pattern would indicate a possible SQLIA. We note that running each and every query under the scanner at runtime could be an expensive process.

In order to optimize number of queries that need to be put under the scanner during runtime to ensure the validity of dynamically generated queries, we first categorize the collected query patterns based on different query operations. For example, SELECT statements and UPDATE statements will be categorized into two different clusters. Secondly, we find the number of tokens in the query pattern. We initially create four tables for SELECT, INSERT, UPDATE and DELETE operations in Master database. These operations are four common SQL operations commonly used by most applications. Then, the query pattern for each hotspot is stored in the corresponding table, according to the query statement and using the number of tokens as a field.

<table>
<thead>
<tr>
<th>Token ID</th>
<th>Token Type</th>
<th>Token</th>
</tr>
</thead>
<tbody>
<tr>
<td>S</td>
<td>Keyword</td>
<td>Select</td>
</tr>
<tr>
<td>D</td>
<td>Keyword</td>
<td>Delete</td>
</tr>
<tr>
<td>I</td>
<td>Keyword</td>
<td>Insert</td>
</tr>
<tr>
<td>U</td>
<td>Keyword</td>
<td>Update</td>
</tr>
<tr>
<td>F</td>
<td>Keyword</td>
<td>From</td>
</tr>
<tr>
<td>W</td>
<td>Keyword</td>
<td>Where</td>
</tr>
<tr>
<td>K₃</td>
<td>Keyword</td>
<td>'</td>
</tr>
<tr>
<td>K₄</td>
<td>Keyword</td>
<td>--</td>
</tr>
<tr>
<td>R₁</td>
<td>Relational Operator</td>
<td>=</td>
</tr>
<tr>
<td>L₁</td>
<td>Logical Operator</td>
<td>And</td>
</tr>
<tr>
<td>L₂</td>
<td>Logical Operator</td>
<td>Or</td>
</tr>
<tr>
<td>V</td>
<td>Variable</td>
<td>Var</td>
</tr>
<tr>
<td>A₁</td>
<td>Attribute</td>
<td>info</td>
</tr>
<tr>
<td>A₂</td>
<td>Attribute</td>
<td>user</td>
</tr>
<tr>
<td>A₃</td>
<td>Attribute</td>
<td>login</td>
</tr>
<tr>
<td>A₄</td>
<td>Attribute</td>
<td>pin</td>
</tr>
</tbody>
</table>
FROM USER WHERE
LOGIN = ' VAR '
AND PIN = VAR

Figure 3: NDFA, query model and query pattern for hotspot B. Runtime Validation

During runtime, the dynamically generated queries (with the user inputs embedded) are also converted into query pattern using token table. The user inputs that are not included in the token table are defined as variable. To compare the runtime query, the corresponding query patterns are retrieved from Master database. If the runtime query pattern is compared with every query pattern in Master database, the process is expensive. Firstly, we find the number of tokens in the dynamic query pattern.

Then, we retrieve the static query patterns which have the same number of tokens and the same query statement type of the dynamic query from Master database. We can ignore the static query patterns which have the different number of tokens although they have the same query statements with the dynamic query pattern. By restricting the number of queries that need to be scanned along any execution path that is taken in the program, we can reduce the runtime scanning overhead.

The converted runtime query pattern is compared with each of the corresponding static query. It then compares each token of static query pattern with each of dynamic query pattern. If each token of dynamic query pattern is not matched with each token of static query pattern, then it is flagged as SQLIA, else it is passed through. Fig. 4 shows the case where an SQLIA has been caused and rejected as a potentially malicious query.

Figure 4: Query model and its query pattern

VI. CONCLUSION

The SQL Injection Attacks are extremely dangerous in comparison to other types of Web-based attacks, because the end result is data manipulation. In this paper we presented a new approach for detecting SQL Injection Attacks. This approach is based on positive tainting, which explicitly identifies trusted (rather than untrusted) data in a program. This way, we eliminate the problem of false negatives that may result from the incomplete identification of all untrusted data sources. Our proposed system can detect the attacks very efficiently by restricting the number of queries to be scanned during runtime.

REFERENCES


Table 2.: SQLIA’S Prevention Accuracy

<table>
<thead>
<tr>
<th>SQL Injection Types</th>
<th>Unprotected</th>
<th>Protected</th>
</tr>
</thead>
<tbody>
<tr>
<td>Tautologies</td>
<td>Not Prevented</td>
<td>Prevented</td>
</tr>
<tr>
<td>Piggy Backed Queries</td>
<td>Not Prevented</td>
<td>Prevented</td>
</tr>
<tr>
<td>Stored Procedure</td>
<td>Not Prevented</td>
<td>Prevented</td>
</tr>
<tr>
<td>Alternate Encoding</td>
<td>Not Prevented</td>
<td>Prevented</td>
</tr>
</tbody>
</table>

V. EVALUATION

Both the protected and unprotected web applications are tested using different types of SQLIA’s; namely use of Tautologies, Union, Piggy-Backed Queries, Stored procedure and various other SQLIA’s. Table 2 shows that the proposed technique prevented all types of SQLIA’s in all cases. The proposed technique is thus a secure and robust solution to defend against SQLIA’s.