IJISET - International Journal of Innovative Science, Engineering & Technology, Vol. 2 Issue 4, April 2015.

www.ijiset.com

ISSN 2348 – 7968

Security against SQL injection attacks using AMNESIA

Shashwat Gupta

shashwat.591@hotmail.com

Saket S. Ektate

saketektate@gmail.com

Prof.Hema.K

hema53001@gmail.com

Deepak Yadav

deepakyadav_64@ymail.com

Sachin Pitrubhakt

sachin.pitrubhakt@gmail.com

(Department of Computer Engineering, DYPCOE Savitribai Phule University of Pune, Maharashtra, India)

Abstract

In today's world there is enormous and widespread use of the internet and web applications. If the web application is not secured its vulnerable to many of the command injection attacks like SQLIA. The current approach uses AMNESIA algorithm to detect and prevent the injection attacks. The confidential information of the user might get leaked from the database leading to severe losses of life and property. This security prevents the unauthorized access to your database and it also prevents your data from being altered or deleted by users without the appropriate permissions. Web applications typically interact with a back-end database to retrieve persistent data and then present the data to the user as dynamically generated output, such as HTML pages on web. However, this interaction is commonly done through a low-level API by dynamically constructing query strings within a general-purpose programming language like Java.

Keywords

AMNESIA, SQLIA, Hotspot, Pattern Matching, Query-based Model, NDFA, Monitor.

I. INTRODUCTION

SQLIA (Structured Query Language Injection Attacks) is type of code injection technique which targets the databases to steal data from the organizations. In this kind of attack, the attacker enters the SQL commands of meta characters or keywords into a SQL statement through unrestricted user input parameters to modify or change the SQL query's logic. It poses the threat to all those web applications that access their databases for its working, through SQL commands constructed with external input data. Those web applications which comprise of online transactions/banking and emails, social networking sites have brought with them the scope of computer security vulnerabilities like SQL injection attacks. SQL injection by-passes authentication logic and provides confidential information to the attacker. An authorized access to confidential information by a crafted user has threatened their authority, confidentiality and integrity. The consequences could be such as the system could not deliver proper services to its customers. In this paper, we present this technique, which will act against all those malicious content and will actively work at those hotspots where injection might occur. This technique is a combination of both static and dynamic type checking.

II. STUDY OF SQL INJECTION ATTACKS

There are certain hotspots in the Structured Query Language through which attacker can penetrate through the database. These hotspots are the target of attackers to inject code into the database without proper authentication. Through the SQL injection, the attacker in worst case might execute arbitrary commands with high system privilege.

1.By-passing authentication through Tautology

Tautology is an always true condition. A website uses this source (figure 2), which would be vulnerable to SQLIA. For example, if a user enters“‘‘ OR 1=1--” and“”, instead of userid =“Raja” and password = “rani”, the resulting query is:

SELECT * from FROM User_info WHERE userid=‘‘‘‘ OR 1=1 --‘‘ AND password =‘‘‘‘

The database interprets everything after the WHERE token as a conditional statement, and the inclusion of the “OR 1=1” clause turns this conditional into a tautology. As a result, the database returns the records for all users in the database. An attacker could insert a wide range of SQL commands via this exploit, including commands to modify or destroy database tables.

2.Union Exploitation Technique

The UNION operator is used in SQL injections to join a query, purposely forged by the attacker to the original query. The technique allows the attacker to obtain the values of columns of other tables. For example,

SELECT Name, Phone, Address FROM Users WHERE Id=$id

The $id values is set as,

$id=1 UNION ALL SELECT Creditcardno,1,1 FROM Creditcardtable

We will have the following query,

SELECT Name, Phone, Address FROM Users WHERE Id=1 UNION ALL SELECT Credircardno,1,1 FROM Creditcardtable

The result of the forged query is joined to the original query. The keyword ALL is necessary for the queries that use DISTINCT keyword. Also, the other two values with Creditcardno ",1,1" are necessary because the two queries must have an equal number of parameters(columns), in order to avoid a syntactical error.

3.Piggybacked Queries

Query delimiters are used by exploiters to append an extra query to the original query such as ';'. The following query will be obviously illegitimate. So attacker could inject any SQL statement or connect to the database. For example,

SELECT accounts From users WHERE username='abc' AND password='';drop table Users – ' AND pin=123

After completing the first query the database would recognize the query delimiter (";") and execute the injected second query. The result of executing the second query would be to drop table Users, which is likely to destroy information that may be valuable.

4.Injection through Stored Procedures

The stored procedure is actually an abstraction layer set by programmer. This type of attack is carried out by executing the procedures, stored previously by the application developer. Additionally, because stored procedures are often written in 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.

CREATE PROCEDURE DBO.isAuthenticated

@userName varchar2, @pass varchar2, @pin int

AS

EXEC("SELECT accounts FROM Users

WHERE username='"+@username+"' and password='"+@password+"'and pin="+@pin+);

GO

5.Blind Injection

To protect the error page to display to the attacker, the developer hides error details and writes code to generate a generic page to be shown to the attacker. This would make an injection attack difficult but not impossible. Still attacker can access the database by asking a series of TRUE/FALSE questions through SQL commands. This type of attacks are known as blind injection attacks. For example,

SELECT account_no FROM user_info WHERE userid='JAY' AND 1==0—AND password=AND pin=0

If the application is not secured then attacker could try the luck and will be successful to penetrate the database.

6.Hex String attack in absence of data type checks

The developer might make an error while constructing SQL statements by failing to check the data types of input from the user. Database has its own type conversion mechanism which performs the automatic conversion of the data types. So the attacker could make use of it by encoding HEX string

0 x 270 x 780 x 270 x 200 x 4f0 x 50- x 200 x 310 x 3d0 x 30 to parameter name. The parser at database will convert it into varchar value which could become a tautology like 'x' OR 1=1.

The escaping functions at the server's program mi9ght not escape such a string because it is not in the escaping characters list, which would result in an injection attack. The parser at the server's end should be designed in such a way that not only it escapes the special characters but it would even perform the type conversion which can prevent SQLIA.

7.Attacks through Special Characters

The developer usually escapes such special characters which might lead to a tautology or blind injection while designing the parser. But, it may not be possible always for the developer to find the full list of special characters to avoid SQL injection attacks. Identified escaping characters like ' or ; are not interpreted as SQL commands. Other unidentified special characters will be a part of SQL commands.

III. EXISTING SYSTEMS

If we talk about the defense techniques used, then the we have two defensive techniques namely 'Defensive Coding' and 'Runtime Monitoring'.

Defensive coding has subdivisions like Manual Defensive Coding practices, SQL DOM, Parameterized Query Insertion. This defensive coding techniques ensured secure code but are labour intensive and time consuming. Manual defensive Coding practices are performed manually and could be done with the help of OWASP. SQL DOM is useful in terms of greater flexibility when developer wishes to use the dynamic queries instead of parameterized one.

Runtime checking is a technique used for prevention of illegitimate SQL statements for all types of SQLIA's by checking them at the runtime. But its drawback is that it requires a strong dynamic monitoring system.

IV. ATTACK DETECTION STRATEGY

4.1 The Technique

Our technique uses a combination of static analysis and runtime monitoring to detect and prevent SQLIAs. It consists of four main steps.

1. Identify hotspots.
2. Build SQL-query models
3. Instrument Application
4. Runtime monitoring

4.1.1 Identify Hotspots

This involves performing a simple scanning of the entire application code to identify hotspots. For example, the servlet in Figure 1, the set of hotspots would contain a single element, the statement at line 10.

```
public class Show extends HttpServlet {
...
1. public ResultSet getUserInfo(String login, String password) {
2. Connection conn = DriverManager.getConnection("MyDB");
3. Statement stmt = conn.createStatement();
4. String queryString = "";
5. queryString = "SELECT info FROM userTable WHERE ";
6. if ((! login.equals("")) && (! password.equals(""))) {
7. queryString += "login=’" + login +
"’ AND pass=’" + password + "’";
8. } else {
9. queryString+="login=’guest’";
}
10. ResultSet tempSet = stmt.execute(queryString);
11. return tempSet;
}
...
}
```

Figure 1: Servlet

4.1.2 Build SQL-Query Models

For building the SQL-query model for each hotspot that is identifies, first all of the possible values for the hotspot's query string are computed. To do this, we leverage the Java String Analysis (JSA) library developed by Christensen, Møller, and Schwartzbach [3]. The JSA library produces a non-deterministic finite automaton (NDFA) that expresses all the possible values the considered string can assume at the character level. The NDFA for a string is an overestimate of all the possible values of the string.

To produce the final SQL-query model, an analysis of the NDFA is performed and then transformed into a model in which all of the transitions represent semantically meaningful tokens in the SQL language. The operation creates an NDFA in which all of the transitions are annotated with SQL keywords, operators, or literal values. In the model, transitions are marked corresponding to the externally defined strings with the symbol β. To illustrate, Figure 2 shows the SQL-query model for the hotspot. The model shows the two different query strings that can be generated by the code depending on the branch followed after the if statement at line 6 (Figure 1). In our model, β marks the position of the user-supplied inputs in the query string.

4.1.3 Instrument Application

Firstly, we instrument the application code with calls to a monitor that checks the queries during runtime. Then, we insert a call to the monitor before the call to the database for each hotspot. The query string, that is about to be submitted and a unique identifier, for the hotspot are the two parameters that are used to invoke the monitor. The monitor uses the

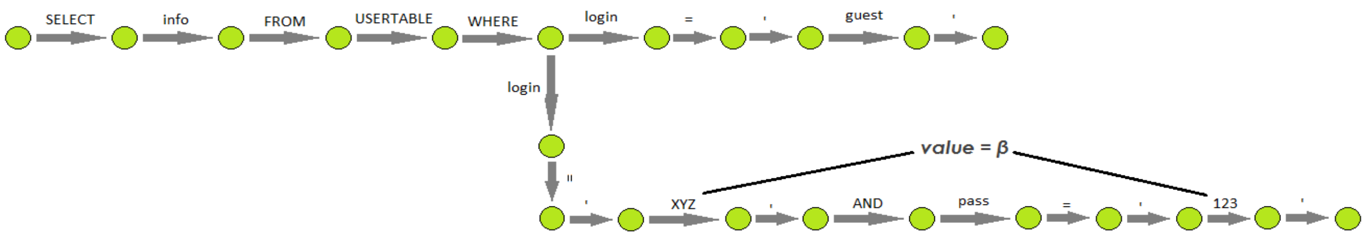


Figure 2: SQL Query model for Servlet in Figure 1

identifier to retrieve the SQL-query model for that hotspot.

Figure 3 shows how the example application would be instrumented by our technique. The hotspot, originally at line 10 in Figure 1, is now guarded by a call to the monitor at line 10a.

```
   ...
   10a.   if   (monitor.accepts   (<hotspot   ID>,
queryString))
   {
   10b.        ResultSet         tempSet         =
stmt.execute(queryString);
   11. return tempSet;
   }
   ...
```

Figure 3: Example hotspot after instrumentation.

4.1.4 Runtime Monitoring

The application executes normally until it reaches a hotspot at the runtime. The query string is now sent to the monitor. The runtime monitor parses the query string into a sequence of tokens according to the specific SQL dialect considered. Figure 4 shows how the last two queries would be parsed during runtime monitoring.

After the query is parsed, the runtime monitor checks if the hotspot's SQL-query model is violated by the query or not. The monitor checks whether the sequence of tokens in the query string is accepted by the model. On matching the query string against the SQL-query model, a token that corresponds to a numeric or string constant (including the empty string, ε) can match either an identical literal value or a β label. If the model does not accept the sequence of tokens the query is identifies an SQLIA, by the monitor.

To show the runtime monitoring, consider the queries shown in Figure 4. The tokens in query 'a' specify a set of transitions that terminate in an accepting state. Therefore, query 'a' is executed on the database. Query 'b' contains extra tokens which prevent it from reaching an accepting state and is identified as an SQLIA.

4.2 Implementation

In our demonstration, we show an implementation of our technique, AMNESIA that works for Java-based Web applications. The technique is fully automated, requiring only the Web application as input, and requires no extra runtime environment support beyond deploying the application with the AMNESIA library. We developed the tool in Java and its implementation consists of three modules:

Analysis module: This module implements Steps 1 and 2 of our technique. It inputs a Java Web application an outputs a list of hotspots and a SQL-query model for each hotspot. For the implementation of this module, we leverage the Java String Analysis library [3]. The analysis module is able to analyze

Java Servlets and JSP pages.

Instrumentation module: This module implements Step 3 of our technique. It inputs a Java Web application and a list of hotspots and instruments each hotspot with a call to the runtime monitor. We implemented this module using InsECTJ, a generic instrumentation and monitoring framework for Java [4].

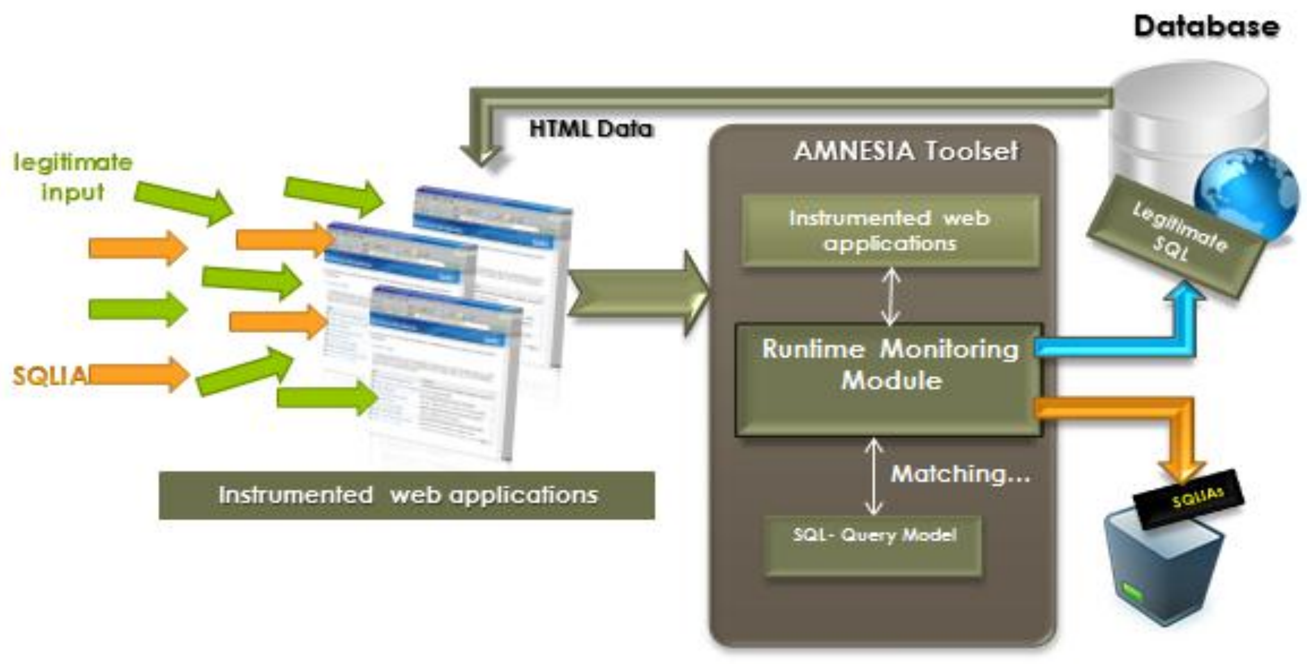
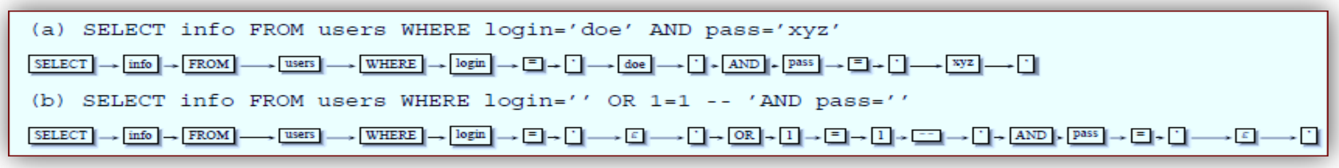


Figure 4: High-Level View of AMNESIA


(a) SELECT info FROM users WHERE login='doe' AND pass='xyz'
SELECT info FROM users WHERE login = ' doe ' AND pass = ' xyz '
(b) SELECT info FROM users WHERE login='' OR 1=1 -- 'AND pass=''
SELECT info FROM users WHERE login = ' ε ' OR 1 = 1 -- ' AND pass = ' ε '

Runtime-monitoring module: This module implements Step 4 of our technique. The module takes as input a query string and the ID of the hotspot that generated the query, retrieves the SQL-query model for that hotspot, and checks the query against the model.

Figure 5 shows a high-level overview of AMNESIA. In the static phase, the Instrumentation Module and the Analysis Module take as input a Web application and produce (1) an instrumented version of the application and (2) an SQL-query model for each hotspot in the application. In the dynamic phase, the Runtime-Monitoring Module checks the dynamic queries while users interact with the Web application. If a query is identified as an attack, it is blocked and reported.

To report an attack, AMNESIA throws an exception and encodes information about the attack in the exception. If developers want to access the information at runtime, they can leverage the exception-handling mechanism of the language and integrate their handling code into the application. Having this attack information available at runtime allows developers to react to an attack right after it is detected and develop an appropriate customized response. Currently, the information reported by AMNESIA includes the time of the attack, the location of the hotspot that was exploited, the attempted-attack query, and the part of the query that was not matched against the model.

Our tool makes one primary assumption regarding the Applications it targets—that queries are created by manipulating strings in the application. In other words, AMNESIA assumes that the developer creates queries by combining hard-coded strings and variables using operations such as concatenation, appending, and insertion. Although this assumption precludes the use of AMNESIA on some applications (e.g., applications that externalize all query-related strings in files), it is not an overly restrictive assumption. Moreover, it is an implementation-related assumption that can be eliminated with suitable engineering.

V. CONCLUSION:

Here we have developed prevention mechanism to protect the vulnerable website and its database from being exploited by the SQL Injection attacks. We have used the java string library for manipulating the injection attack and thereby preventing the injection inside of the database.These techniques will secure the web application and also strengthen the customer relationship with web to safely access the confidential information.

VI. FUTURE SCOPE:

In the future a special dedicated parser can be developed for scanning the illegal SQL strings, running at backend of the application. The parser at the backend will be running and discarding the invalid characters of SQL so that only trIn our future work we will investigate alternate techniques for building SQL models for cases in which the static analysis cannot be used.

VII. REFERENCES

[1] S. W. Boyd and A. D. Keromytis. SQLrand: Preventing SQL injection attacks. In
Proc. of the 2nd Applied
Cryptography and Network Security Conf. (ACNS 2004), pages 292–302, Jun. 2004.

[2] G. T. Buehrer, B. W. Weide, and P. A. G. Sivilotti. Using Parse Tree Validation to Prevent SQL Injection Attacks. In Proc. of the 5th Intern. Workshop on Software Engineering and Middleware (SEM 2005)
, pages 106–113, Sep. 2005.

[3] A. S. Christensen, A. Møller, and M. I. Schwartzbach. Precise analysis of string expressions. In Proc. 10th Intern. Static Analysis Symposium (SAS 2003) , pages 1–18, Jun. 2003.

[4] A. Seesing and A. Orso. InsECTJ: A Generic Instrumentation Framework for Collecting Dynamic Information within Eclipse. In Proc. of the eclipse Technology eXchange (eTX) Workshop at OOPSLA 2005, pages 49–53, Oct. 2005.

[5] SQL Server 2000 Extended Stored Procedure Vulnerability http://www.atstake.com/research/advisories/2000/a120100-2.txt

[6]Indrani Balsundaram , An efficient technique for detection and prevention of SQL injection attack using ASCII based string matching, International conference on communication

technology and system design 2012,madhurai university, May 2012.

[7] William G.J. Halfond, Alessandro Orso, and Panagiotis Manolios (2008): WASP: Protecting Positive Tainting and Syntax-Aware Evaluation .IEEE Transactions on Software Engineering, Vol. 34, No. 1

[8] Zhendong Su and Gary Wassermann (2006): The Essence of Command Injection Attacks in Web Applications. In ACM Symposium on Principles of Programming Languages (POPL)“top ten most critical web application vulnerabilities”, OWASP Foundation.

[9] S.V. Shanmughaneethi, S.C. E. Shyni, and S. Swamynathan (2009): SBSQLID: Securing Web Applications with Service Based SQL Injection Detection. IEEE Conference, Computer Society, pp. 702-704.

[10] “AMNESIA: Analysis and Monitoring for NEutralizing SQLInjection Attacks” William G.J. Halfond and Alessandro Orso College of Computing Georgia Institute of Technology whalfondorso@cc.gatech.edu