An Effective Approach for Protecting Web from SQL Injection Attacks

Veera Venkateswaramma P

Abstract- The databases that underlie web applications were facing issues like, unauthorized access, so many security threats in recent years. Many software systems have evolved to include a Web-based component that makes them available to the public via the Internet and can expose them to a variety of Web-based attacks. One of these attacks is SQL injection, which can give attackers unrestricted access to the databases and has become frequent and serious threat to them. Successful injection attack can give attackers access to and even control of the databases that underlay Web applications, which may contain sensitive or confidential information. This paper presents a new highly automated approach for protecting Web applications against SQL injection that has both conceptual and practical advantages over most existing techniques. From a practical standpoint, our technique is precise and efficient, has minimal deployment requirements, and incurs a very low performance overhead in most cases. We have implemented this technique (Injection preventer), which we used to perform an empirical evaluation on a wide range of Web applications that we subjected to a large and varied set of attacks and legitimate accesses.

Keywords- SQL Injection, Security, Syntax- aware, Positive tainting, Character level tainting.

1. INTRODUCTION

The security is an important aspect to the organizations which are developing web applications. It is a great challenge to the organizations to protect their valuable data against intruders, corruptions and malicious accesses [2].Actually the developers in the organizations are concentrating mostly on applications usability and functionality, rather than enforcing security standards. In general, SQL injection vulnerabilities are caused by inadequate input validation within an application [2].Input validation is a major security aspect if an attacker finds that an application makes attempt has been made to increase the efficiency of the above unfounded assumptions about the type, length, format, or techniques by a combinatorial approach for protecting web range of input data. The attacker can then supply a malicious application against SQL Injection attacks input that compromises an application. The external interfaces exposed by an application become the only source of attack besides the other interfaces network and host level entry points are secure. Since this injection attack is a major threat in web applications development that underlies databases which may breach the database security mechanism, Such as availability and issues like authorization, integration, and authentication. A SQL injection attack consists of
insertion or "injection" of a SQL query via the input data from the client to the application. A

successful SQL injection exploit can read sensitive data from the database, modify database data
(Insert/ Update/Delete), execute administration operations on the database (such as shutdown the DBMS), recover the content of a given file present on the DBMS file system and in some cases issue commands to the operating system. SQL injection attacks are a type of injection attack, in which SQL commands are injected into data-plane input in order to effect the execution of predefined SQL commands. These attacks may bypass the security mechanisms like intrusion detection systems, firewall and cryptography. Attackers take advantage of these vulnerabilities by submitting input strings that contain specially-encoded database commands to the application. When the application builds a query using these strings and submits commands are executed by the database and the attack succeeds.

The most worsening part of these injection attacks is they are very easy to perform, even though developers of the applications have an idea about these attacks. The main concept is based on the idea is a malicious user counterfeits the data that a web application sends to the database focusing at modification of sql query which gets executed by DBMS software. The input validation issues can allow the hackers to gain access to the database systems. All most all technologies that use database system were facing these vulnerabilities due to these attacks [3]. So many techniques have been developed to counter these attacks, but they are lack of practicality and efficacy. Initially a technique was proposed as a solution to counter these injection attacks based on defense coding. This practice was not efficient due to these problems. They are 1) solutions based on defensive coding will address only a subset of possible attacks. 2) Legacy systems address another problem because of expense and complexity of making the existing code so that is compliant with defensive coding. 3) It is a great difficult to develop code based on defense code practices.

2. PROPOSED WORK

In this paper an attempt has been made to overcome the above difficulties and to improve the efficiency of above

techniques by a conceptual approach for protecting web applications against sql injection attacks. It contains four sections namely section A. conceptual approach, section B. contains types of injection vulnerabilities, section C contains Mitigation. The rest of the paper section D. present's signature based approach and auditing of the database.

A. CONCEPTUAL APPROACH

Intuitively, our approach works by identifying "trusted" strings in an application and allowing only these trusted strings to be used to create certain parts of an SQL query, such as keywords or operators. The general mechanism that we use to implement this approach is based on dynamic tainting [4] which marks and tracks certain data in a program at runtime. The kind of dynamic tainting we use gives our approach several important advantages over techniques based on different mechanisms [5]. It involves positive Tainting, character level tainting, syntax-aware evaluation.

i. Positive Tainting: Positive tainting [5] differs from traditional tainting (hereafter, negative tainting) because it is based on the identification, marking, and tracking of trusted, rather than un-trusted data. This conceptual difference has significant implications for the effectiveness of our approach, in that it helps address problems caused by incompleteness in the identification of relevant data to be marked. In the case of negative tainting, incompleteness leads to trusting data that should not be trusted and, ultimately, to false negatives. Incompleteness may thus leave the application vulnerable to attacks and can be very difficult to detect, even after attacks actually occur, because they may go completely unnoticed. With positive tainting, incompleteness may lead to false positives, but it would never result in an SQLIA escaping detection. Moreover, as explained in the following, the false positives generated by our approach, if any, are likely to be detected and easily eliminated early during prerelease testing. Positive tainting uses a white-list, rather than a black-list, policy and follows the general principle of fail-safe defaults, as outlined by Saltzer and Schroeder [6]. In case of incompleteness, positive tainting fails in a way that maintains the security of the system. In the context of preventing SQLIAs, the conceptual advantages of positive tainting are especially significant. The way in which Web applications create SQL commands makes the identification of all un-trusted data especially problematic and, most importantly, the identification of most trusted data relatively straightforward. In general, it is difficult to guarantee that all potentially harmful data sources have been considered and even a single unidentified source could leave the application vulnerable to attacks. The situation is different for positive tainting because identifying trusted data in a Web application is often straightforward and always less error prone. In fact, in most cases, strings hard-coded in the application by developers represent the complete set of trusted data for a Web application. To account for these cases, our technique provides developers with a mechanism for specifying sources of external data that should be trusted. The data sources can be of various types such as files, network connections, and server variables. Our approach uses this information to mark data that comes from these additional sources as trusted .In a typical scenario, we expect developers to specify most of the trusted sources before testing and deployment. In other words, false positives are likely to occur only for totally untested parts of applications. Therefore, even when developers fail to completely identify additional sources of trusted data beforehand, we expect these sources to be identified during normal testing and the set of trusted data to quickly converge to the complete set.

ii. Character Level Tainting We track taint information at the character level rather than at the string level. We do this because, for building SQL queries, strings are constantly broken into substrings, manipulated, and combined. By associating taint information to single characters. Our approach can precisely model the effect of these string operations. Another alternative would be to trace taint data at the bit level, which would allow us to account for situations where string data are manipulated as character values using bitwise operators. However, operating at the bit level would make the approach considerably more expensive and complex to implement and deploy. To accurately maintain character-level taint information, we must identify all relevant string operations and account for their effect on the taint markings. Our approach achieves this goal by taking advantage of the encapsulation offered by object oriented languages, in particular by Java, in which all string manipulations are performed using a small set of classes and methods. Our approach extends all such classes and methods by adding functionality to update taint markings based on the methods' semantics.

iii. Syntax-Aware Evaluation Our technique performs syntax-aware evaluation of a query string immediately before the string is sent to the database to be executed. To evaluate the query string, the technique first uses a SQL parser [7] to break the string into a sequence of tokens that correspond to SQL keywords, operators, and literals. The technique then iterates through the tokens and checks whether tokens (that is, substrings) other than literals contain only trusted data. If all such tokens pass this check, the query is considered safe and is allowed to execute. If an attack is detected, a developer specified action can be invoked. This approach can also handle cases where developers use external query fragments to build SQL commands. In these cases, developers would specify which external Consider the malicious query, where the attacker submits "admin' – –" as the login and "0" as the pin.

Shows the sequence of tokens for the resulting query, together with the trust markings. Recall that "– –" is the SQL comment operator, so everything after this is identified by the parser as a literal. In this case, the Meta Checker would find that the last two tokens,' and __, contain untrusted characters. It would therefore identify the query as an SQLIA.

B. INJECTION VULNERABILITIES

i. Incorrectly filtered escape characters

This form of SQL injection occurs when user input is not filtered for escape characters and is then passed into an SQL statement. These results in the potential manipulation of the statements performed on the database by the end-user of the application. The following line of code illustrates this vulnerability Statement = "SELECT * FROM users WHERE name = '" + username + "';" This SQL code is designed to pull up the records of the specified username from its table of users. However, if the "username" variable is crafted in a specific way by a malicious user, the SQL statement may do more than the code author intended. For example, setting the "username" variable as 'or '1'='1 Or using comments to even block the rest of the query (there are three types of SQL comments): ' or '1'='1' --' 'or '1'='1' ({' ' or '1'='1' /*' Renders one of the following SQL statements by the parent language: SELECT * FROM users WHERE name = '' OR '1'='1'; SELECT * FROM users WHERE name = '' OR '1'='1' -- '; If this code were to be used in an authentication procedure then this example could be used to force the selection of a valid username because the evaluation of '1'='1' is always true.

The following value of "username" in the statement below would cause the deletion of the "users" table as well as the selection of all data from the "userinfo" table (in essence revealing the information of every user), using an API that allows multiple statements: a'; DROP TABLE users; SELECT * FROM userinfo WHERE 't' ='t' This input renders the final SQL statement as follows: SELECT * FROM users WHERE name = 'a'; DROP TABLE users; SELECT * FROM userinfo WHERE 't' ='t'; While most SQL server implementations allow multiple statements to be executed with one call in this way, some SQL APIs such as PHP's mysql_query(); function do not allow this for security reasons. This prevents attackers from injecting entirely separate queries, but doesn't stop them from modifying queries.

ii. Incorrect type handling This form of SQL injection occurs when a user supplied field is not strongly typed or is not checked for type constraints. This could take place when a numeric field is to be used in a SQL statement, but the programmer makes no checks to validate that the user supplied input is numeric. For example: Statement: = "SELECT * FROM userinfo WHERE id = " + a_variable + ";" It is clear from this statement that the author intended a_variable to be a number correlating to the "id" field. However, if it is in fact a string then the end-user may manipulate the statement as they choose, thereby bypassing the need for escape characters. For example, setting a_variable to 1; DROP TABLE users Will drop (delete) the "users" table from the database, since the SQL would be rendered as follows: SELECT * FROM userinfo WHERE id=1; DROP TABLE users;

iii. Blind SQL Injection Blind SQL Injection [8] is used when a web application is vulnerable to an SQL injection but the results of the injection are not visible to the attacker. The page with the vulnerability may not be one that displays data but will display differently depending on the results of a logical statement injected into the legitimate SQL statement called for that page. This type of attack can become time-intensive because a new statement must be crafted for each bit recovered. There are several tools that can automate these attacks once the location of the vulnerability and the target information has been established.

iv. Conditional Responses One type of blind SQL injection forces the database to evaluate a logical statement on an ordinary application screen. SELECT booktitle FROM booklist WHERE bookId = 'OOk14cd' AND '1'='1'; Will result in a normal page while SELECT booktitle FROM booklist WHERE bookId = 'OOk14cd' AND '1'='2'; Will likely give a different result if the page is vulnerable to a SQL injection. An injection like this may suggest to the attacker that a blind SQL injection is possible, leaving the attacker to devise statements that evaluate to true or false depending on the contents of another column or table outside of the SELECT statement's column list. SELECT 1/0 FROM users WHERE username='000';

C. MITIGATION

i. Parameterized Statements With most development platforms, these parameterized statements can be used that work with parameters (sometimes called placeholders or bind variables) instead of embedding user input in the statement [9]. In many cases, the SQL statement is fixed, and each parameter is a scalar, not a table. The user input is then assigned (bound) to a parameter. This is an example using Java and the JDBC API: Java.sql.PreparedStatement prep = connection.prepareStatement ("SELECT * FROM users WHERE USERNAME =? AND PASSWORD =?"); prep.setString (1, username); prep.setString (2, password); prep.executeQuery ();

ii. Enforcement at the code level Using object-relational mapping libraries avoids the need to write SQL code. The

ORM library in effect will generate parameterized SQL statements from object-oriented code [9].

iii. Escaping A straightforward, though error-prone, way to prevent injections is to escape characters that have a special meaning in SQL [10]. The manual for an SQL DBMS explains which characters have a special meaning, which allows creating a comprehensive blacklist of characters that need translation. For instance, every occurrence of a single quote (') in a parameter must be replaced by two single quotes (") to form a valid SQL string literal. For example, in PHP it is usual to escape parameters using the function mysql_real_escape_string (); before sending the SQL query:query = Sprintf ("SELECT * FROM `Users` WHERE Username='%s'AND password= '%s'" ,mysql_real_escape_string ($Username), mysql_real_escape_string ($Password)); mysql_query ($query); This function, i.e. mysql_real_escape_string (), calls MySQL's library function mysql_real_escape_string, which prepends backslashes to the following characters: \x00, \n, \r, \, ', "and \x1a. This function must always (with few exceptions) be used to make data safe before sending a query to MySQL. [10].

D. SIGNATURE APPROACH It is based on the signature based approach to address the injection attacks which occurred due to input validations. It contains three modules and used Hirschberg algorithm to compare the statements from specifications and a hotspot .Hot spot is that line where it gets the input from the user and vulnerable in execution. This step performs a simple scanning of the application code to identify hotspots. For example servlet in prg below, the set of hotspots contain a single element: the statement at line 6, 7, 8. (In Java stability database. based applications, interactions with the database occur through calls to specific methods in the JDBC library such as Public class Display extends HttpServlet { 1. Public ResultSet getUserInfo (String login, String pwd) { 2. String queryString = ""; 3. Connection con = DriverManager.getConnection ("connected"); 4. Statement st = con.createStatement (); 5. queryString = "select info from usertable where"; 6. If not ((login. equals ("")) && (pwd.equals (""))) { 7. queryString += "login=`" + login + "' AND pass="' +pwd + "'";} 8. ResultSet tempSet = st.execute (queryString); }…….. This step identifies the hotspot (6, 7, and 8) and it divides the hotspot in to tokens and it sends it to query validation phase.

i. Monitoring Module. In this module input has been taken from web application and sends it to analysis module for validation. If it fails validation (any culprit) it delivers error message to monitoring module to block further transactions.

ii. Specifications. It contains contains predefined keywords and sends it to analysis module for comparisons, these modules contains all predefined keyword which are stored in database.

iii. Analysis Module. In this module it takes input from monitoring module and finds hotspot from the application, applies Hirschberg algorithm for string comparison.

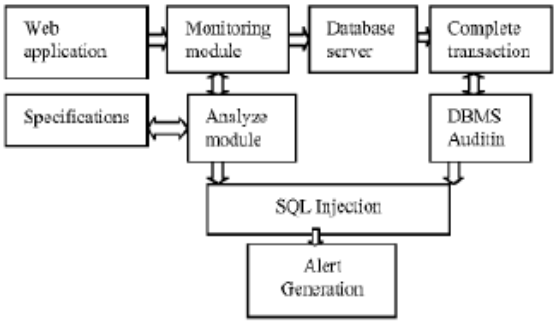

Fig.1. Signature based architecture

In this approach it provides a table which contains keywords present in the horizontal and vertical line and compares the incoming tokens with predefined values by applying this algorithm for identification. In the below sql statement there is a hotspot identified by analysis module which sends it to table to find and prevent attacks. Sql stmt = select * from clients where user = '"&ausername&"' and pwd = '"&apwd&"; By applying the above algorithm the statement is divided into tokens and validates each token with predefined tokens using divide and conquer methodology [1]. The injection code is: .Select * from clients where user = 'username' and pwd = 'anything or '1' = '1'; The analysis module finds an injection takes place after (anything) this token to prevent injection attack.

System Auditing Auditing is a method of tracking the use of database system availability, resources and authority. When it is active it provides the information about database operations like which database object was affected, who and when performed the operation. By enforcing strict security policies in DBMS by DBAs' they can easily identify information regarding that who is an authenticated user and up to which level he is authorized to access data. By enforcing these standards it can prevent some injection attacks and moreover it provides support to signature based method to prevent injection attacks more effectively.

3. CONCLUSION

This paper presents a systematic approach to prevent injection attacks and protect the web application against those attacks. Moreover from a conceptual standpoint; the approach is based on the idea of positive tainting and on the concept of

syntax-aware evaluation. From a practical standpoint, our technique is precise and efficient, has minimal deployment requirements, and incurs a very low performance overhead in most cases. By using auditing to analyze the transactions to prevent malicious access and on the other hand Signature based approach is used to reduce the time taken to detect and prevent the attacks. Moreover empirical evaluation is performed on wide range of web applications and WASP which automates the task very easily.

REFERENCES

[1]. Christina Yip Chung, "DEMIDS: A Misuse Detection System for Database Systems", Integrity and internal control information systems, Pages: 159 - 178, ACM, 2008.
[2]. David Geer (2008), "Malicious Bots Threaten Network Security".
[3]. G.T. Buehrer, B.W. Weide, and P.A.G. Sivilotti, (2005) "Using Parse Tree Validation to Prevent SQL Injection Attacks," Proc. Fifth Int'l Workshop Software Eng. and Middleware, pp. 106-113.
[4]. J. Clause, W. Li, and A. Orso (2007) "Dytan: A Generic Dynamic Taint Analysis Framework," Proc. Int'l Symp.Software Testing and Analysis, pp. 196-206.
[5]. Marco Cova, Davide Balzarotti, Viktoria Felmetsger, and Giovanni vigna (2007), " Swaddler: An approach for the anamoly based character distribution models in the detection of SQL Injection attacks", Recent Advances in Intrusion Detection System, Pages 63-86.

[6]. Nguyen-tuong, S. Guarnieri, D. Greene, J.Shirley, and D. Evans (2005)," Automatically hardening web applications using Precise Tainting", In Twentieth IFIP Intl, Information security conference.

[7]. R.Ezumalai, G.Aghila (2009) "A Combinatorial Approach for Preventing SQL Injection Attacks" IEEE International Advance Computing Conference 2009.
[8]. S.W. Boyd and A.D. Keromytis (June 2004), "SQLrand: Preventing SQL Injection Attacks," Proc. Second Int'l Conf. Applied Cryptography and Network Security, pp. 292-302.

[9]. SruthiBandhakavi (ACM, 2007), "CANDID: Preventing SQL Injection Attacks using Dynamic Candidate Evaluations".

[10]. W.G. J. Halfond and A. Orso (2005), "Combining Static Analysis and Runtime monitoring to counter SQL Injection attacks", 3rd International workshop on Dynamic Analysis, St. Louis, Missouri.