User Authentication Method against SQL Injection Attack

Manju Khari, Nikunj Kumar

Abstract- The Internet and web applications are playing very important role in our today's modern day life. Most of the web applications use the database as a back end to store critical information. SQL injection attacks represent a serious threat to any database-driven site. The methods behind an attack are easy to learn and the damage caused can range from considerable to complete system compromise. Despite these risks an incredible number of systems on the internet are susceptible to this form of attack. Organizations are using a reactive approach towards these threats, instead of a proactive approach that would help avoiding them. In this paper, we propose a technique for user authentication to prevent SQLIAs by hashing and salting the user name and password.

Keywords- Web applications vulnerabilities, SQL injection, Authentication, Hashing, Salting, Database security, Internet

1 INTRODUCTION

Web applications are extremely popular today because they are ubiquitous and can be easily maintained and updated. Users access the interface via a web browser and send requests to the web server, which in turn translates these requests to database SQL commands and, using the results of those commands, generates the response that is sent back to the browser for final presentation to the user. A major problem is that web applications are often insecure. In fact, web application developers are normally not specialized in security and the usual time to market constraints direct the effort on satisfying the user's requirements, causing security aspects to be easily neglected.

SQL injection is a particularly dangerous threat that exploits application layer vulnerabilities inherent in web applications. Instead of attacking instances such as web servers or operating systems, the purpose of SQL injection is to attack RDBMSs, running as back-end systems to web servers, through web applications [1].

Every web application, using a relational database, can theoretically be a subject for SQL injection attacks. If successful, SQL injection attacks may therefore result in exposure of and serious impact on the corporations most valuable information assets. These attacks may in the worst case result in a completely destroyed database schema, which in turn may affect a corporation's ability to perform business [2].

Login page is the most complicated web application which allows users to enter into the database after authenticating them. In this page, the users provide their identity like username and password. A sophisticated attacker can able to compromise the user name and password by launching on-line and off-line guessing attack.

Web based applications are normally has three tier model, Application (Front End), Middle tier (Protocol), and backend (Data base), given in figure 1. If a user wants to access the data base form remote place then he has to logon to the system through web site using the user name and password. In the middle tier, SQL query is generated with the given input data. The server verifies the user name and password, if it matches then the user will be allowed to access the data base. Login page is the most complicated in the web application which allows users to access database after the completion of authentication process. In this page, the user provides his identity like username and password. There might be some invalid input validations which can

• *Manju Khari is working as Asst. Professor in Department of Computer Science and Engineering of AIACT&R, Delhi, India.*
E-mail: manjukhari@yahoo.co.in

• *Nikunj Kumar is currently pursuing M.Tech in Information Security in AIACT&R, Delhi, India*
E-mail: nikunjkumar14@gmail.com

bypass the authentication process using some mechanism like SQL injection [3].

If username and password are defined by the user, the method embeds the submitted credentials in the query. For instance, if a user submits username and Password as "XYZ" and "111," then the query will be:

Query_result = "SELECT * from User_Credentials WHERE username = 'XYZ' and password = '111'"

A web site that uses this servlet would be vulnerable to SQLIAs: For example, if a user enters "'OR 1=1 --" and "", instead of "XYZ" and "111", the resulting query is:

Query_result = "SELECT *from User_Credentials WHERE username = ''OR 1 = 1 --'AND password = ''"

Analyzing the above query, the result is always true for variable Query_result. It is because malicious code has been used in the query. Here, in this query the mark (') tells the SQL parser that the user name string is finished and " OR 1=1 " statement is appended to the statement which always results in true. The (– –) is comment mark in the SQL which tells the parser that the statement is finished and the password will not be checked. So, the result of the whole query will return true for Query_result variable which authenticates the user without checking password.

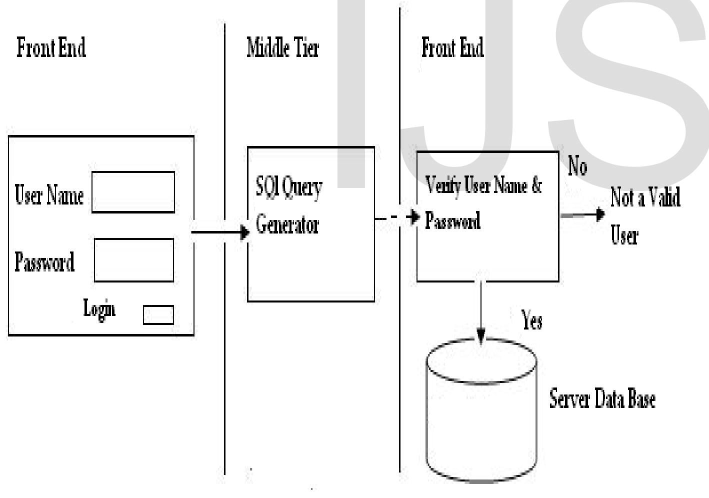

Fig. 1. Basic Model for Web Applications

The paper is structured as follows: Section 2 discuses background. Section 3 describes related work. Section 4 is focused on proposed user authentication method and section 5 discuses the implementation and testing. Finally section 6 concludes the work done.

2 BACKGROUND

SQL injection is an attack technique that mainly occurs due to the insecure coding practices [4]. This attack modifies the SQL statement in such a way so that the legitimate inputs or the authentication of a legitimate user is bypassed and the database executes the malicious code supplied by the attacker. The basic cause of the vulnerability is the un-sanitized user input.

Any web application can be formalized with respect to SQL injection attack as follows:

- It accepts the input from a user or system.
- It concatenates input with hardcoded SQL statement and builds complete query structure.
- The generated query gets executed and concatenates result with HTML code.

In the context of above formalization SQL injection attack is targeted on a program at the database layer which is connected to a web application. This SQL injection attack exploits weakness or vulnerability in the target program to properly verify the input supplied to it through a web form [5]. In a typical SQL injection attack the attacker posts specially crafted structured query language (SQL) statements which are executed in the database server and produce malicious outcomes.

3 RELATED WORK

There are many ways to prevent SQL Injection attacks. Prevention concerns with correctness of input value which is supplied by client or user at coding level. These techniques force the client to enter correct data and can be barred to enter illegal value which is harmful to database server. Such type of prevention can be done at both sides whether it may be client side or server side but SQL injection cannot be prevented with this technique. The tools and techniques for detecting and preventing SQL injection are given below:

CANDID [5] is proposed by Bandhakavi et al. This approach automatically prevents SQLIA using dynamic candidate evaluations. It dynamically mines the programmer intended query structure on any input and detects attacks by comparing it against the structure of actual query.

SQLGuard [6] is proposed by Buehrer et al. SQLGuard checks at runtime whether SQL queries conform to a model of the expected queries. The model is deduced at runtime by examining the structure of the query before and after a client's requests. SQLGuard requires the application developer to rewrite code to use a special intermediate library.

SQLIPA [7] is proposed by Shaukat Ali et al. In this approach they used hash value to improve performance of authentication for web application. This hash value for username and password is created at runtime when user account is created. SQLIPA (SQL Injection Protector for Authentication) prototype was developed in order to test the framework. Though the proposed framework was tested on few sample data and had an overhead of 1.3 ms, it requires further improvement to reduce the overhead time. It also requires to be tested with larger amount of data.

Tautology Checker [8] is proposed by Wassermann et al. This approach uses static analysis combined with automated reasoning to verify that the SQL queries generated in the application layer cannot contain a tautology. The primary drawback of this technique is that its scope is limited to detecting and preventing tautologies and cannot detect other types of attacks.

AMNESIA [9] is proposed by Halfond et al. AMNESIA is model based technique that combines the static and dynamic analysis. In the static phase, it uses a static analysis to build the models of the SQL queries that an application legally generates at each point of the access to the database. In dynamic phase, it intercepts all the SQL queries before they are sent to the database and checks each query against the statically built models queries that violate the model are identified as SQL injection attacks. The accuracy of AMENSIA depends on that of static analysis. Unfortunately, certain types of obfuscation codes and/or query generation technique make this step less precise and the result is both false positive and negative.

SQLrand [3] is proposed by Boyd et al. SQLrand provides a framework that allows developers to create SQL queries using randomized keywords instead of the normal SQL keywords. A proxy between the web application and database intercepts SQL query and de-randomizes the keywords. The SQL keywords injected by an attacker would not have been constructed by the randomized keywords, and thus the injected command would result in a syntactically incorrect query. Since SQLrand uses a secret key to modify keywords, its security relies on attackers not being able to discover this key. SQLrand requires the application developers to re-write the code.

SANIA (Syntactic and Semantic Analysis for Automated Testing against SQL Injection) [10] is propose by Kosuga et

al. Sania is designed to used by a web applications developer during the development and debugging phase, and thus is able to intercept SQL queries between an application and the database as well as HTTP requests between a client and application. After capturing HTTP requests and SQL queries, Sania checks for any SQL injection vulnerabilities.

WebSSARI [11] is proposed by Huang et al. WebSSARI detects input-validation related errors using information flow analysis. In this approach, static analysis is used to check taint flows against preconditions for sensitive functions. The analysis detects the points in which preconditions have not been met and can suggest filters and sanitization functions that can be automatically added to the application to satisfy these preconditions. The WebSSARI system works by considering as sanitized input that has passed through a predefined set of filters. In their evaluation, the authors were able to detect security vulnerabilities in a range of existing applications. The primary drawbacks of this technique are that it assumes that adequate preconditions for sensitive functions can be accurately expressed using their typing system and that having input passing through certain types of filters is sufficient to consider it not tainted. For many types of functions and applications, this assumption is too strong.

Unless developers properly design their application code to protect against unexpected data input by users, alteration to the database structure, corruption of data or evaluation of private and confidential information may be granted inadvertently.

4 PROPOSED METHOD

This section proposes an authentication scheme using hash function and salt (random string concatenated to the password string when it created) for preventing SQL injection attacks. In this method there is need for three extra columns in User_Credentials table. The first one is for hash value of user name, second one is for salt and the last one is for hash value of password||salt. The hash values are calculated for user name and password||salt when a user account is created for the first time and stores it in the User_Credentials table. When users want to login to database hash values of user name and password||salt are calculated at runtime and checked with stored hash values in the User_Credentials table. Whenever user enters his/her user name and password in the user name and password fields, the query at the back end server will be creating as:

Query_result = "SELECT * FROM User_Credentials WHERE Username_Hash_value = "Username_Hash_value" AND Password_Hash_value = "Password_Hash_value" AND Username = ' 'OR 1=1 – – AND password = 'Password'"

In the query shown above, if hackers enter the SQL injection query still they cannot bypass the authentication process. The benefit of the proposed method is that the hackers do not know about the salt and hash values of user name and password. So, it is not possible for the hacker to bypass the authentication process through the general SQL injection techniques. The SQL injection attacks can only be done on codes which are entered through user entry form but the hash values are calculated at run time at backend before creating SELECT query to the underlying database therefore the hacker cannot calculate the hash values as it dynamic at runtime.

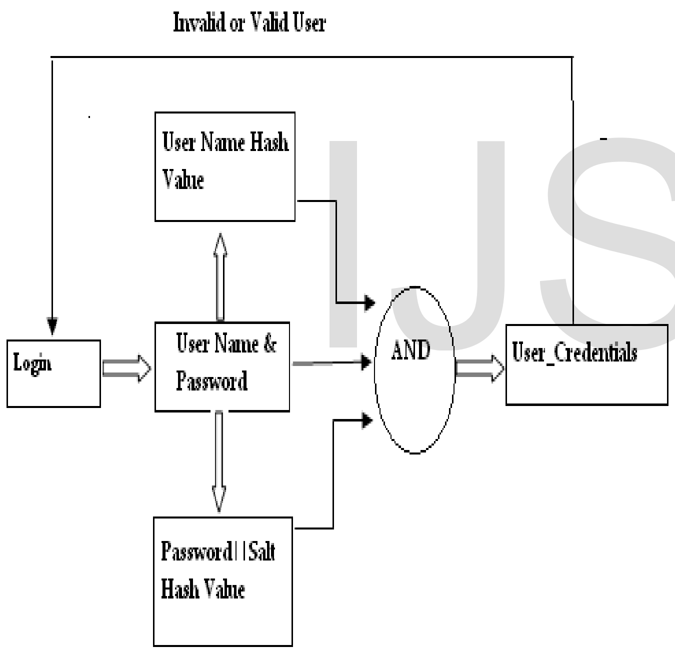

Fig. 2. Model for Proposed Method

5 IMPLEMENTATION AND TESTING

The system has been implemented using Microsoft SQL server as a DBMS. Three stored procedures with name Create_User_Credentials, Generate_Random_String and User_Authentication have been used. The proposed method has been test by compare the processing overhead of the proposed method with existing related techniques such as PSQLIA-AES and SQLIPA whose result is shown in fig. 3.

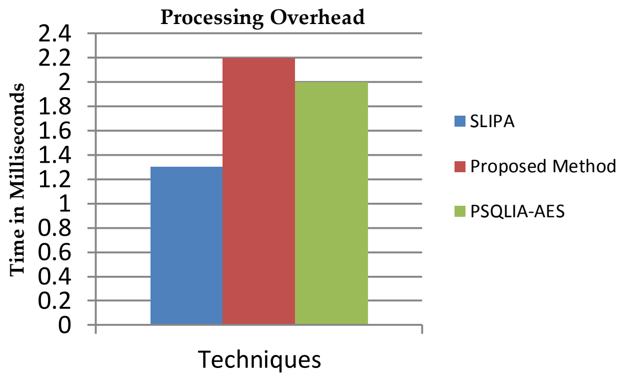

Fig. 3. Comparison of Processing Overhead of Proposed Method and related techniques

The graph in fig. 3 shows the result of comparison of processing overhead of proposed method, SQLIPA, PSQLIA-AES. The time overhead of the proposed method is that it takes 0.9 ms of extra time overhead to protect database against SQL injection which is almost negligible fraction of time and has no significance compared to securing the authentication process. Salting makes the dictionary attack more difficult. If the original password is 6 digits and the salt is 4 digits, then hashing is done over a 10 digit value. This means that hacker now needs to make a list of 10 million items and create a hash for each of them. The list of hashes has 10 million entries, and the comparison takes much longer.

6 CONCLUSION

This paper presents a new method to secure the authentication process of the database. Proposed method uses user name, password, salt and their hash values for authentication process. The method is tested on sample data of different records in User_Credentials table. It takes very little time overhead of 0.9 ms for authentication process.

ACKNOWLEDGEMENTS

Authors wish to express sincere gratitude to the administration of Department of Computer Science and Engineering, AIACT&R, Delhi for providing the academic environment to pursue research activities.

REFERENCES

[1] Martin Eizner, “Direct SQL command injection,” Technical report, The Open Web Application Security Project, 2001.

[2] Mitchell Harper, “SQL injection attacks - are you safe? Technical report, DevArticles,May 2002.

[3] S. W. Boyd and A. D. Keromytis, “SQLrand: Preventing SQL Injection Attacks,” *Proc. of the 2nd Applied Cryptography and Network Security (ACNS) Conference*, pp. 292–302, June 2004.

[4] Common Weakness Enumeration, “CWE-89: improper neutralization of special elements used in SQLcommand,” http://cwe.mitre.org/data/definitions/89.html. 2012

[5] P. Bisht, P. Madhusudan, and V. N. Venkatakrishnan, “CANDID: Dynamic Candidate Evaluations for Automatic Prevention of SQL Injection Attacks,” *ACM Trans. Information System Security*, 13(2), pp. 1- 39, 2010.

[6] G. T. Buehrer, B. W. Weide, and P. A. G. Sivilotti, “Using parse tree validation to prevent SQL injection attacks,” *Proc. of the 5th International Workshop on Software Engineering and Middleware SEM*, pp. 106–113, 2005.

[7] S. Ali, S.K. Shahzad, and H. Javed, “SQLIPA: An Authentication Mechanism against SQL Injection,” *European Journal of Scientific Research*, vol. 38, no. 4, pp. 604-611, 2009.

[8] G.Wassermann and Z. Su, “An Analysis Framework for Security in Web Applications,” *Proceedings of the FSE Workshop on Specification and Verification of Component-Based Systems (SAVCBS ‘04)*, pp. 70–78, 2004.

[9] W. G. Halfond and A. Orso, “AMNESIA: Analysis and Monitoring for NEutralizing SQL-Injection Attacks,” *Proc. of the IEEE and ACM International Conference on Automated Software Engineering (ASE ‘05)*, Long Beach, CA, USA, Nov 2005.

[10]Y. Kosuga, K. Kono, M. Hanaoka, M. Hishiyama, and Y. Takahama, “Sania: Syntactic and Semantic Analysis for Automated Testing against SQL Injection,” *Proc. of 23rd Annual Computer Security Applications Conference (ACSAC 2007)*, pp. 107-117, Dec. 2007.

[11]Y. Huang, F. Yu, C. Hang, C. H. Tsai, D. T. Lee, and S. Y. Kuo, “Securing Web Application Code by Static Analysis and Runtime Protection,” *Proc. of the 12th International World Wide Web Conference (WWW ‘04)*, May 2004.