An Authentication Mechanism to prevent SQL Injection Attacks

Indrani Balasundaram
Department of Computer Science
Madurai Kamaraj University
Tamil Nadu, India

E. Ramaraj
Department of Communication
Madurai Kamaraj University
Tamil Nadu, India

ABSTRACT
SQL Injection attacks target databases that are accessible through a web front-end, and take advantage of flaws in the input validation logic of Web components such as CGI scripts.In the last few months application-level vulnerabilities have been exploited with serious consequences by the hackers have tricked e-commerce sites into shipping goods for no charge, usernames and passwords have been harvested and confidential information such as addresses and credit-card numbers has been leaked. The reason for this occurrence is that web applications and detection systems do not know the attacks thoroughly and use limited sets of attack patterns during evaluation. SQL Injection attacks can be easily prevented by applying more secure authentication schemes in login phase itself. To address this problem, this paper presents an authentication scheme for preventing SQL Injection attack using Advance Encryption Standard (AES). 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 preventing SQL Injection attack using AES (PSQLIA-AES).

General Terms
Protocols, authentication scheme

Keywords
SQL Injection attack, Web security, authentication, AES, Secret Key, password security.

1. INTRODUCTION
SQL injection attacks have become the most widely exploited security attacks on the Internet as they can usually bypass layers of security such as firewalls and any other network detection sensors. They are used most often to attack databases and for extracting any confidential information such as Social Security Numbers, Credit Card information etc. SQL injection is an attack in which malicious code is inserted into strings that are later passed to an instance of SQL Server for parsing and execution. The primary form of SQL injection consists of direct insertion of code into user-input variables that are concatenated with SQL commands and executed. A less direct attack injects malicious code into strings that are destined for storage in a table or as metadata. When the stored strings are subsequently concatenated into a dynamic SQL command, the malicious code is executed. A study conducted in 2005 by the Gartner Group found that on over 300 tested web sites, 97% were vulnerable to SQL injection attacks (W. R. Cook and S. Rai (2005), C. Anley (2002)). Through SQL injection attacks, an attacker may extract undisclosed data, bypass authentication, escalate privileges, modify the content of the database, execute a denial-of-service attack (C. Anley (2002), D. Aucsmith, (2004) , F. Bouma, (2003)). These kinds of attacks are happened only if they can able to introduce forged characters on SQL queries. A sophisticated attacker can able to compromise the user name and password by lunching on-line and off-line guessing attack. Web based applications are normally has three tire model, Application (Front End*),* Middle tire (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 tire, 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 bypass the authentication process using some mechanism like SQL injection.

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 "Alice" and "Bob," the servlet dynamically builds the query:

Query_result="SELECT info FROM users_account WHERE username ='Alice' AND password ='Bob'"

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

Query_result="SELECT info FROM user_account WHERE username =" 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. (The characters "--" mark the beginning of a comment, so everything after them is ignored.)

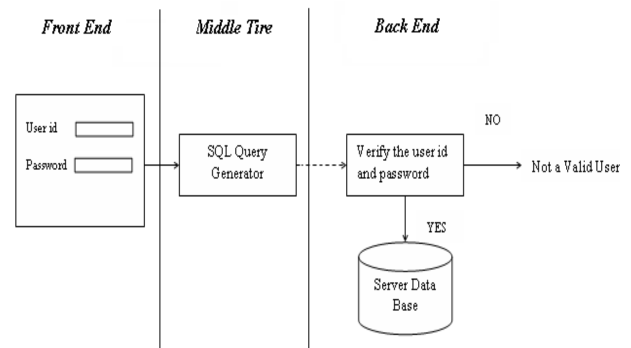

Figure 1: Basic Model for Web Applications

As a result, the database would return information about all users. An attacker could insert a wide range of SQL commands via this exploit.

2. RELATED WORK

Many techniques have been proposed to prevent SQL injection Attacks for example, dynamic monitoring tools (Pietraszek, T. and C. V. Berghe (2005), Halfond, W. G. J. and A. Orso (2008), Su, Z. and G. Wassermann (2006)). Each of these techniques has some advantages and disadvantages. Major problems with these techniques are either high code modifications or it takes large extra time overhead. E. G. Barrantes, D. H. Ackley (2003) is related to Randomized instruction set emulation to disrupt binary code injection attacks, which applies a randomization technique similar to our Instruction-Set Randomization (G. S. Kc, A. D. Keromytis, and V. Prevelakis, (2003)) for binary code only, and uses an emulator attached to specific processes. Kemalis et al (2008) proposed Specification based approach in this technique they build a model for SQL statements. It is based on set of rules lexical analysis and syntactical verification is used to valid the SQL statement of the query either declares it legitimate or malicious. It maintains the log file for the system process which will facilitate the administrator. More recent approaches (J. Foster, M. Fa¨hndrich, and A. Aiken (1999), D. Larochelle and D. Evans (2001), N. Dor, M. Rodeh, and M. Sagiv (2003)) have focused on detecting specific types of problems, rather than try to solve the general "bad code" issue, with considerable success, while such tools can greatly help programmers ensure the safety of their code. Dynamic analysis tools such as (E. Larson and T. Austin (2003)) offer incomplete protection, as they cannot prevent modern class of attacks and vulnerabilities. They intercept system calls inside the kernel, and use policy engines to decide whether to permit the call or not. The main problem with all these is that the attack is not prevented: rather, the system tries to limit the damage such code can do, such as obtain super-user privileges. In the context of a web server, this means that a web server may only be able to issue queries to particular databases or access a limited set of files. T. Garfinkel (2003) scheme identifies several common security-related problems with such systems, such as their susceptibility to various types of race conditions. A number of techniques are in use for securing the web applications. The most common way is the authentication process through the username and password. One of the major problems in the authentication process is the input validation checking (S. W. Boyd and A. D. Keromytis, (2004)). SQL rand is a practical defense mechanism against SQL injection attacks. Such attacks target databases that are accessible through a web front end, and take advantage of flaws in the input validation. These works apply the concept of instruction-set randomization to SQL. Queries injected by the attacker will be caught and terminated by the database parser. This mechanism imposes negligible performance overhead to query processing and can be easily retrofitted to existing systems. Injecting SQL code into a web application requires little effort by those who understand both the semantics of the SQL language and CGI scripts (S. W. Boyd and A. D. Keromytis, (2004)). 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.

3. PROPOSED MODEL

This section proposed an authentication scheme using AES for preventing SQL Injection attack. This method has three phases, 1) Registration Phase, 2) Login Phase and 3) Verification Phase.

1) Registration Phase

The following steps are executed, when ever a new user is enter into server for register as a new user,

a. Every user must select a unique user name U_{Name} and password $U_{Password}$ and send it to the server along with registration request.
b. Server receives the request from the user and register as a new user. Server maintains a user account table with three field's user name, user password, and user secrete key (unique key value) as shown in the table1. The user secrete key $K_{U_{Name}}$ is generated by the server and this key is unique for all the users.
c. Server sent a registration conformation to the user along with user secrete key $K_{U_{Name}}$

Table1: User Account Table

User Name	User Password	User Secret Key
Murugesh	Uthaandakalai	$SecretKey_{Murugesh}$
Amuthan	Sethubai	$SecretKey_{Amutha}$

2) Login Phase

Through Login phase user can access the data base from the server so, the following steps are executed.

a. The user name and password is encrypted by using Advance Encryption Standard algorithm by applying **user secrete key.**
b. SQL query generator generates the query by using the encrypted user name and password as shown in figure 2.
c. The query will be send to server.

Query_result=SELECT * FROM user_account WHERE username = 'abc' AND password = 'xyz' AND encrypted_username = '$E_{SecretKey}$ (abc)' AND encrypted_password = '$E_{SecretKey}$ (xyz)'

Figure 2: Sample query generation in the proposed model

System Model for Login and Verification Phase

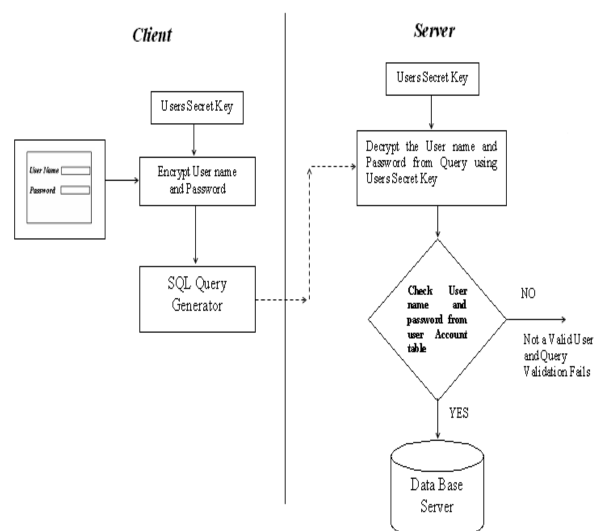

Figure 3: Login and Verification Phase

3) Verification Phase

In the verification phase, server receives the query result send by the user and performs the following steps,

a. The server receives the login query and verifies the corresponding users secrete key. If the username and password matches the **user name and password can be** decrypted from the query by using this key.

b. Check the decrypted user name and password from the user account table. If it match then accept the user, otherwise reject as malicious attacker.

The detailed system model of login and verification phase is given in figure3. In the proposed scheme, verification phase first verifies the user name from the query and corresponding secrete key of the user is taken from user account table. So, SQL injection attack is avoided if there is given a query like username= 'a' or '1'= '1';--.

4. IMPLEMENTATION RESULTS AND DISCUSSIONS

The proposed scheme is implemented and tests are tested in windows machine having configuration Intel® Pentium® Core™ 2 Duo CPU E4500 @ 2.20GHz, 2.19GHz, 0.99GB of RAM. The proposed scheme is compare with two existing schemes, first scheme SQLIPA (Shaukat Ali, Azhar Rauf, and Huma Javed (2009))is based on hash functions and the second scheme is based on Elgamal cryptosystem. We have developed the code by using Core Java jdk1.6. We have compared the proposed in three different ways,

1. The proposed scheme is evaluated with different key sizes (128, 256, and 512) as shown in table 2 and the corresponding chat is given in figure 4.
2. Compared the processing overhead (time needed for encryption of user name and password) of proposed scheme with existing related schemes PSQLIA and AQE-PSQLIA as shown in table 3 and figure 5.
3. Compared the processing overhead of the proposed scheme by using different number of users (10, 20, 30, 40, and 50) as shown in table 4 and figure 6.

The proposed scheme is very secure with minimum computation overhead. An experiment was designed to measure the additional processing time required by different sets of concurrent users. The worst-case scenario adds approximately 5.2 milliseconds to the processing time of each query. We used a stand alone computer with the SQL Server with approximately 1,000 tuples in relevant tables and created views. Next we run a transaction that included Select statement directly on the table and on the view with encrypted user name and password using AES. To average the processing time we repeated this for 10 and 100 serial transactions for a single user.

Table 2: Execution Time comparison with different Key size

Cryptosystem	Execution Time in Millisecond
AES-128	≈0.315
AES-256	≈0.572
AES-512	≈0.785

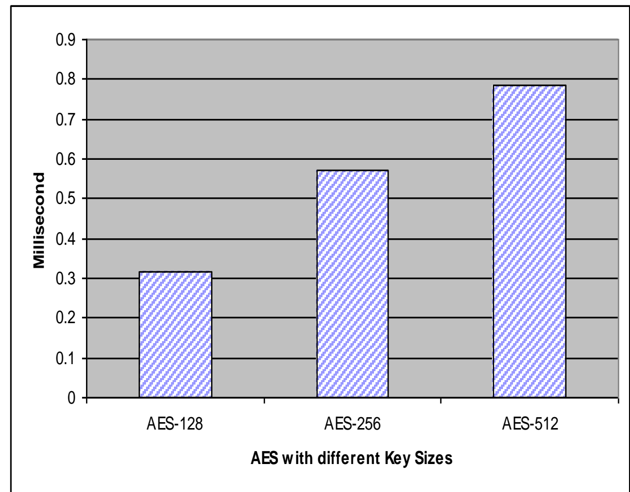

Figure 4: Execution Time comparison of user input

Table 3: Comparison of processing overhead with existing schemes in Millisecond

Techniques	Processing Overhead in Millisecond
Proposed Scheme	≈0.785
SQLIPA	≈1.472
AQE-PSQLIA	≈0.125

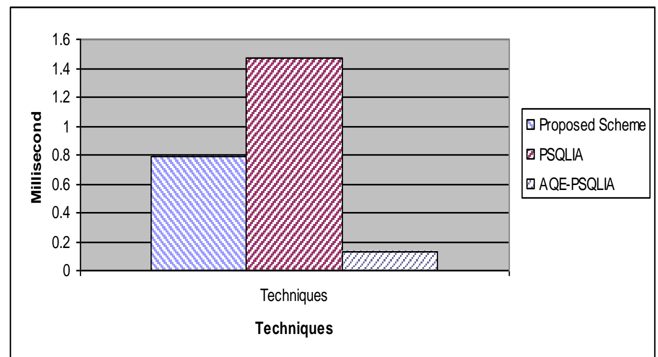

Figure 5: Comparison of Processing Overhead

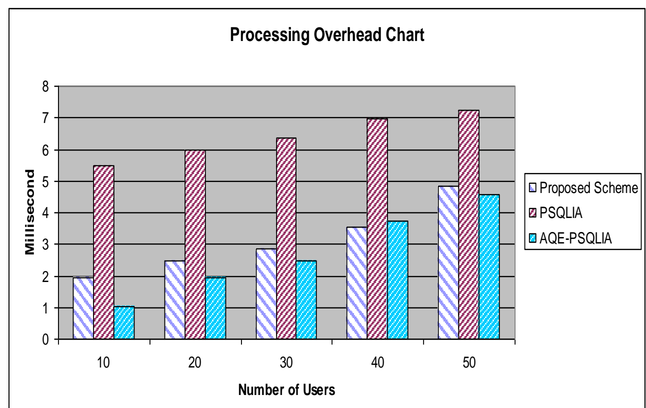

Figure 6: Processing Overhead Chart for different number of users

5. CONCLUSION

This paper presents an authentication scheme for preventing SQL Injection attack using Advance Encryption Standard (PSQLIA-AES). Encrypted user name and password are used to improve the authentication process with minimum overhead. We have implemented and tested the proposed scheme with three different ways, 1) evaluated with different key sizes (128, 256, and 512), 2) compared the processing overhead (time needed for encryption of user name and password) of proposed scheme with existing related schemes SQLIPA (Shaukat Ali, Azhar Rauf, and Huma Javed (2009)) and AQE-PSQLIA, and 3) compared the processing overhead of the proposed scheme by using different number of users (10, 20, 30, 40, and 50). The proposed scheme is more efficient, it needs 3.144ms for encryption or decryption and this can be negligible.

6. REFERENCES

[1] C. Anley, 2002, "Advanced SQL Injection In SQL Server Applications," White paper, Next Generation Security Software Ltd.

[2] D. Aucsmith, 2004 "Creating and Maintaining Software that Resists Malicious Attack," http://www.gtisc.gatech.edu/bioaucsmith.html, September 2004. Distinguished Lecture Series.

[3] F. Bouma, 2003. Stored Procedures are Bad, O'kay Technical report, Asp.Net Weblogs, November http://weblogs.asp.net/fbouma/archive/2003/11/18/38178.aspx.

[4] S. W. Boyd and A. D. Keromytis, 2004. "SQLrand: Preventing SQL Injection Attacks," In Proceedings of the 2nd Applied Cryptography and Network Security (ACNS) Conference, pages 292–302.

[5] Kemalis, K. and T. Tzouramanis 2008. "SQL-IDS: a specification-based approach for SQLinjection detection," SAC'08. Fortaleza, Ceará, Brazil, ACM: pp. 2153-2158.

[6] W. R. Cook and S. Rai, 2005, "Safe Query Objects: Statically Typed Objects as Remotely Executable Queries," In Proceedings of the 27th International Conference on Software Engineering (ICSE 2005).

[7] E. G. Barrantes, D. H. Ackley, S. Forrest, T. S. Palmer, D. Stefanovic, and D. D. Zovi, 2003. "Randomized Instruction Set Emulation to Disrupt Binary Code Injection Attacks," In Proceedings of the 10th ACM Conference on Computer and Communications Security (CCS), pp 281–289.

[8] Shaukat Ali, Azhar Rauf, and Huma Javed, 2009. "SQLIPA: An Authentication Mechanism Against SQL Injection," European Journal of Scientific Research, ISSN 1450-216X Vol.38 No.4, pp 604-611.

[9] E. Larson and T. Austin, 2003. "High Coverage Detection of Input-Related Security Faults," In Proceedings of the 12th USENIX Security Symposium, pages 121–136

[10] C. Cowan, S. Beattie, J. Johansen, and P. Wagle, 2003. "PointGuard: Protecting Pointers From Buffer Overflow Vulnerabilities," In Proceedings of the 12th USENIX Security Symposium, pages 91–104.

[11] G. S. Kc, A. D. Keromytis, and V. Prevelakis, 2003. "Countering Code-Injection Attacks With Instruction-Set Randomization," In Proceedings of the ACM Computer and Communications Security (CCS) Conference, pages 272–280.

[12] D. Larochelle and D. Evans, 2001. "Statically Detecting Likely Buffer Overflow Vulnerabilities," In Proceedings of the 10th USENIX Security Symposium, pages 177–190.

[13] T. Garfinkel, 2003. "Traps and Pitfalls: Practical Problems in System Call Interposition Based Security Tools," In Proceedings of the Symposium on Network and Distributed Systems Security (SNDSS), pages 163–176.

[14] N. Dor, M. Rodeh, and M. Sagiv, 2003. "CSSV: Towards a realistic tool for statically detecting all buffer overflows in C," In Proceedings of the ACM Conference on Programming Language Design and Implementation (PLDI).

[15] J. Foster, M. Fa¨hndrich, and A. Aiken, 1999. "A theory of type qualifiers," In Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)

[16] Pietraszek, T. and C. V. Berghe 2005. "Defending against Injection Attacks through Context- Sensitive String Evaluation," Recent Advances in Intrusion Detection (RAID2005).

[17] Halfond, W. G. J. and A. Orso 2008. "WASP: Protecting Web Applications Using Positive Tainting and Syntax-Aware Evaluation." IEEE 34(01): pp. 65-81.

[18] Su, Z. and G. Wassermann 2006. "The Essence of Command Injection Attacks in Web Applications," *POPL.* Charleston, South Carolina, USA, ACM**:** pp. 372 – 382