ISSN:0975-9646

Mayank Namdev et al, / (IJCSIT) International Journal of Computer Science and Information Technologies, Vol. 3 (5) , 2012,4981 - 4987

A Novel Approach for SQL Injection Prevention Using Hashing & Encryption (SQL-ENCP)

Mayank Namdev , Fehreen Hasan, Gaurav Shrivastav

Department of Computer Science and Engineering
RKDF Institute of Science & Technology (RGPV)
Bhopal (M.P.), 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 decisive 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. It also contains strengths and weaknesses of various SQL injection attacks.**

It is known to all that SQL injection attacks easily prevented by applying more secure schemes in login phase and after login phase. Therefore, we implement our proposed scheme called SQLENCP, the SQL injection prevention by encryption & hashing techniques, to handle the SQLIA and prevent them. Although, the proposed implemented system is unable to handle all the SQL injection attacks, but it can prevent tautology attacks, union based query attacks & illegal structured query attacks.

Keywords: **SQL injections, SQL injection attacks, SQL attacks, database attacks, hashing, encryption, decryption.**

I. INTRODUCTION

SQL injection is one of the most devastating vulnerabilities to affect a business, as it can lead to exposure of all of the sensitive information stored in an application's database, including handy information such as usernames, passwords, names, addresses, phone numbers, and credit card details. SQL injection has probably existed since SQL databases were first connected to Web applications. Most of us either use Web applications on a daily basis, as part of our vocation or in order to access our e-mail, book a holiday, purchase a product from an online store, view a news item of interest, and so forth. One thing that Web applications have in common, regardless of the language in which they were written, is that they are interactive and, more often than not, are database-driven. Database-driven Web applications are very common in today's Web-enabled society. They normally consist of a back-end database with Web pages that contain server-side script written in a programming language that is capable of extracting specific information from a database depending on various dynamic interactions with the user. One of the most common applications for a database-driven Web application is an e-commerce application. While, on the other side, the organization increases the use of office automation software & services, that helps them to maintain the confidential information with less efforts. Therefore, in this scenario it is not wrong to say that Information will be the single most important business asset today and achieving a high level of information security can be viewed as imperative in order to maintain a competitive edge. SQL Injection Attacks (SQLIA's) are one of the most severe threats to web application security. They are frequently employed by malicious users for a variety of reasons like financial fraud, theft of confidential data, website defacement, sabotage, etc. The number of SQLIA's reported in the past few years has been showing a steadily increasing trend and so is the scale of the attacks. It is, therefore, of paramount importance to prevent such types of attacks, and SQLIA prevention has become one of the most active topics of research in the industry and academia. There has been significant progress in the field and a number of models have been proposed and developed to counter SQLIA's, but none have been able to guarantee an absolute level of security in web applications, mainly due to the diversity and scope of SQLIA's. One common programming practice in today's times to avoid SQLIA's is to use database stored procedures instead of direct SQL statements to interact with underlying databases in a web application, since these are known to use parameterized queries and hence are not prone to the basic types of SLQIA's. However, there are vulnerabilities in this scheme too, most notably when dynamic SQL statements are used in the stored procedures, to fetch the database objects during runtime. Our work is centered on this particular type of vulnerability in stored procedures and we develop a scheme for detection of SQLIA in scenarios where dynamic SQL statements are used. This paper is divided into following section In Section I introduction, in section II basic of SQL injection, in section III various methodologies of SQL injection attack, in section IV effects of SQL injection, in section V problem statement, in section VI various approaches used to prevent SQLIA, in VII our proposed approach, in section VIII implementation details and in IX conclusion and future work is present.

II. SQL INJECTION

SQL Injection is a type of vulnerability to web application security in which an attacker is able to submit a database SQL command, which is executed by a web application, exposing the back-end database. SQL Injection attacks can occur when a web application utilizes user-supplied data without proper validation or encoding as part of a command or query. The specially crafted user data tricks the application into executing unintended commands or changing data. SQL Injection allows an attacker to create, read, update, alter, or delete data stored in the back-end database. In its most common form, SQL Injection allows attackers to access sensitive information such as social

security numbers, credit card number or other financial data. According to Vera code's State of Software Security Report SQL Injection is one of the most prevalent types of vulnerability to web application security.

Key Concepts of SQL Injection

- SQL injection is a software vulnerability that occurs when data entered by users is sent to the SQL interpreter as a part of an SQL query
- Attackers provide specially crafted input data to the SQL interpreter and trick the interpreter to execute unintended commands
- Attackers utilize this vulnerability by providing specially crafted input data to the SQL interpreter in such a manner that the interpreter is not able to distinguish between the intended commands and the attacker's specially crafted data. The interpreter is tricked into executing unintended commands

SQL injection exploits security vulnerabilities at the database layer. By exploiting the SQL injection flaw, attackers can create, read, modify, or delete sensitive data.

III. SQL INJECTIONS METHODOLOGIES

Database applications have become a core component in control systems and their associated record keeping utilities. Traditional security models attempt to secure systems by isolating core software components and concentrating security efforts against threats specific to those computers or software components.

Tautology: The general goal of a tautology-based attack is to inject code in one or more conditional statements so that they always evaluate to true. The consequences of this attack depend on how the results of the query are used within the application. The most common usages are to bypass authentication pages and extract data. In this type of injection, an attacker exploits an inject-able field that is used in queries WHERE conditional. Transforming the conditional into a tautology causes all of the rows in the database table targeted by the query to be returned.

Example: In this example attack, an attacker submits [" or 1=1 - -] for the user name input field (the input submitted for the other fields is irrelevant). The resulting query is:

```
SELECT status FROM tbl_user WHERE name = '' or 1=1 –
AND pwd= ' '
```

The code injected in the condition [OR 1=1] transforms the entire WHERE clause into a tautology. The database uses the conditional as the basis for evaluating each row and deciding which ones to return to the application. Because the conditional is a tautology, the query evaluates to true for each row in the table and returns all of them.

Illegal/Logically Incorrect Query: This attack lets attacker gather important information about the type and structure of the back-end database of a Web application. The attack is considered a preliminary, information gathering step for other attacks. The vulnerability leveraged by this attack is that the default error page returned by application servers is often overly descriptive. In fact, the simple fact that an error messages is generated can often reveal vulnerable/inject-able parameters to an attacker. Additional error information, originally intended to help programmers debug their applications, further helps attackers gain information about the schema of the back-end database.

Example: This example attacks goal is to cause a type conversion error that can reveal relevant data. To do this, the attacker injects the following text

```
UNION SELECT TOP 1 COLUMN_NAME FROM
INFORMATION_SCHEMA.COLUMNS WHERE
TABLE_NAME='tbl_user'--
```

into the following URL:

```
http://www.domain.com/users/userinfo.asp?userid =123
```

The resulting query is:

```
http://www.domain.com/users/userinfo.asp?userid =123
UNION SELECT TOP 1 COLUMN_NAME FROM
INFORMATION_SCHEMA.COLUMNS WHERE
TABLE_NAME='tbl_user'—
```

The injected query extracts the 1st column name of "admin_login" table from the information _schema database.

Union Query: By this technique, attackers join injected query to the safe query by the word UNION and then can get data about other tables from the application.

Example: Following executed from the server

```
SELECT name, phone FROM tbl_user WHERE
userid=$id
```

By injecting the following Id value:

```
$id= 1 UNION ALL SELECT credit Card Number, 1 FROM
Credit CardTable
```

We will have the following query:

```
SELECT name, phone FROM tbl_user WHERE userid =
1 UNION ALL SELECT creditCardNumber, 1 FROM
Credit CardTable
```

This will join the result of the original query with all the credit card users.

The proposed implemented system contains the mechanisms, which will protect the web application from the above discussed SQL injection attacks.

IV. EFFECTS OF SQL INJECTION

As the SQL injections are related with the database and in today's scenario where the database is one of the primary assets of any organization. Therefore, with these SQL injections, cyber-criminals can take complete remote control of the database, and become able to manipulate the database to do anything they wish, including:

- Insert a command to get access to all account details in a system, including user names and retrieve VNC passwords from registry

- Upload files
- Through reverse lookup, gather IP addresses and attack those computers with an injection attack
- Corrupting, deleting or changing files and interact with the OS, reading and writing files
- Online shoplifting e.g. changing the price of a product or service, so that the cost is negligible or free
- Insert a bogus name and credit card in to a system to scam it at a later date
- Delete the database and all its contents
- Shut down a database

91% of database attacks lead to financial loss [1], but the financial impact can be dwarfed be the long-term damage to an organizations reputation. In fact, research by Ipsos MORI[1] at march 2007 revealed that 58% of consumers would stop using an organizations service following a security breach involving their personal data.

V. PROBLEM STATEMENT

As shown in [2] the number of web servers are growing and so the number of installed web applications on these servers is rising as well. Many web sites use open source web applications to provide certain services that are part of the web site, such as a bulletin board (e.g., phpBB [3]) or a blog (e.g., WordPress [4]), or they use a content management system (e.g., Mambo [5], Typo3 [6], drupal [7]) that can be used to operate the complete web site. Web applications are not only used by private web site providers but also by companies and governmental institutions. Most often a database is used as the primary resource to retrieve information that is requested by the user. The information contained in the database has been stored by somebody responsible of tending the web site, or the information is created by an internal business process of the company (e.g., the currently available articles in an on-line shop). Another possible source of content in a web page may be a remote web service of a news agency that provides current news. The number of security problems found in software has increased within the last years [8]. Some of the security problems affect web applications that provide dynamic web pages to their customers. Attacks that exploit these security problems either prying on data contained in the web application (e.g., credit card numbers of customers) or they use the web application as an attack vector on the visiting customer. Both types of attack rely on user input that is not validated by the web application. To extract personal information from the web application, "SQL injection" can be used [9, 10]. In this kind of attack, information that is entered by the user is included in database queries that are used to extract content for the web page. Because the user input is not checked for malicious content, arbitrary SQL queries can be executed. These queries can then be used to circumvent safety procedures incorporated in the web application (e.g., bypass logins), retrieve personal data of customers (e.g., credit card numbers, social security numbers) or execute system commands on the targeted web server (e.g., to install malicious software on the server).

VI. RELATED WORK

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 [11, 12, 13, 14, 15 & 16]. There are many measures that can be taken to prevent SQL. Out of which, the following mechanisms & methodologies are to be discussed.

In 2005 by Halfond and Orso et al proposed Analysis and Monitoring for Neutralizing SQL-Injection attacks [17]. In the figure 4 AMNESIA, the authors are using runtime checking of the query and declare it valid or malicious. AMNESIA checks query in different steps. In the first step it identifies the "hotspot". Hotspots are application code which issues SQL query to database. Second, it forms a model for legitimate query in the form of NDFA (Non-Deterministic Finite Automata). Finally, as a request comes it checks the query with NDFA and declares it legitimate or malicious. It actually works by combining static analysis and runtime monitoring of database queries. In its static part, technique uses program analysis to automatically build a model of the legitimate queries that will be generated by the application. While in the dynamic part, the technique monitors the dynamically runtime generated queries and checks them for acceptability with the statically-generated model. Query that doesn't match with the model represent potential SQLIAs and are hence prevented from executing on the database and reported.

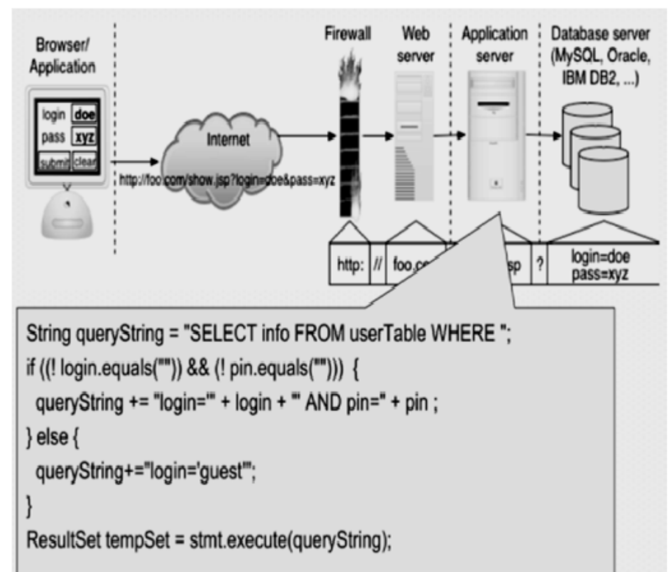

Fig 1: Analysis and Monitoring for Neutralizing SQL-Injection attacks (AMNESIA)

In 2005 Buehrer et al proposed a mechanism "Using parse tree validation to prevent SQL injection attacks"[18] which uses a parse tree, data structure, for the validation of query. In this technique the authors have used a parse tree as a model and every query entering to database is checked against that tree. After checking with parse tree the query is either declared valid or malicious.

In 2008 Kemalis et al proposed a specification based technique "SQL-IDS: a specification-based approach for

SQLinjection detection" [19]. In this technique they specify a model for SQL statements. The model is based on a set of rules. The SQL statements are intercepted to the model. After lexical analysis and syntactical verification of the query either declares it valid or invalid. It keeps the log file of the whole process which will facilitate the administrator.

In 2009 MeiJunjin is using an approach called "An approach for SQL injection vulnerability detection" [20] for the detection of SQL injection vulnerabilities. The above mentioned author has used static, dynamic and automatic testing method for the detection of SQL injection vulnerabilities. The approach traces user queries to vulnerable location.

In 2009 by Ezumalai et al. used a signature based approach "Combinatorial Approach for Preventing SQL Injection Attacks"[21] for the protection of SQL injection. In this approach they used three modules to detect security issues. A monitoring module which takes input from web application and sent to analysis module. An analysis module which finds out the hotspots from application, it uses Hirschberg algorithm [22]. This is a string comparison algorithm which works on divide and conquer rule. It stores all the keywords in the specifications module.

SQLIA Prevention Using Stored Procedures - Stored procedures are subroutines in the database which the applications can make call to [23]. The prevention in these stored procedures is implemented by a combination of static analysis and runtime analysis. The static analysis used for commands identification is achieved through stored procedure parser and the runtime analysis by using a SQL Checker for input identification.

VII .PROPOSED APPROACH

After studying the various SQL injection prevention techniques, we proposed a technique in which we implement a mechanism, that detect & prevent the SQL injections by incorporating the techniques of "HASHING" & "ENCRYPTION".

The concept behind our implemented system is simple: instead of relying on user's permissions, we implement cumbersome defensive coding techniques with which, we can detect & prevent the SQL injections and provide security to the web application.

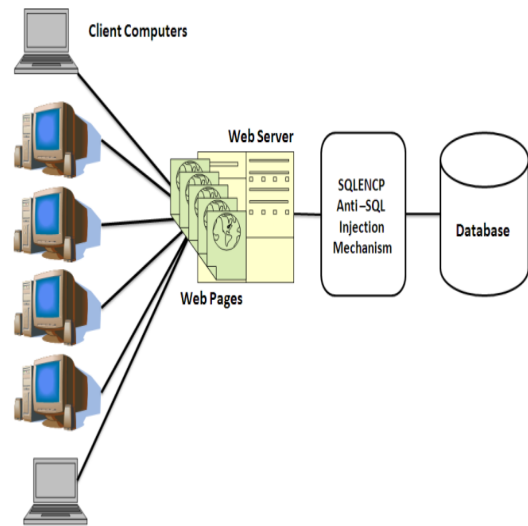

Fig 2: Proposed SQL-ENCP Architecture

Our proposed method simply works on the "HASHING" methods for the secure login technique. In which, we compute the hash value of the username and passwords for any user and store it in database table along with the simple username & passwords.

Table I: User table without security guidelines contains only username & passwords

ID	Username	Password
1	krishnpal	krsh123
2	mayank	mayank123

Table II: User table with security guidelines also contains the hash values
(* only for representation, not in actual table)

ID	user-name	password	Hash_username*	Hash_ password*	Hash_EX-OR
1	krishn pal	krsh123	F59295350096DBA03 3BEA802EC1A573FB E937EB0	19AA6EE730DFD50936A5 45A683A0716380D9E8E5	EC73D2D20E29051BEDA 413A265C349655
2	mayank	mayank12 3	7FE94AC4AC6B9336 9B5E8C290ECB15E4 43771657	947F02F46B4CA59418870 C21CC4A9391DDC52F1A	2964830C7278B53108C28 1899EB2394D

Table III: Query Testing

Standard Query	SELECT username, password FROM tbl_user WHERE username = @usrname AND password = @pwd
Malicious Code	SELECT username, password FROM tbl_user WHERE username ' ' OR 1=1; '/* AND password = '*/'
SQL Injection	SELECT username, password FROM tbl_user WHERE hash_exor = Exor(hashval('$user_nm'), hashval('$pass'))
Output	Not Possible, As the direct values not passing to SQL Query.

HASH FUNCTION ALGORITHM

In the proposed approach there is a need for one extra column in database, which contains the EX-OR of the Hash values of username and password at the time, when a user account is created for the first time and stores it in the User table. Whenever user wants to login to database his/her identity is checked using user name and password and its hash values. These hash values are calculated at runtime using store procedure when user wants to login into the database.

During the authentication of user, the SQL query with hash parameters is used. Hence, if a user tries the injection to the query, and our proposed methodology is working with SQL query, it will automatically detect the injections as the potentially harmful content and rejects the values.

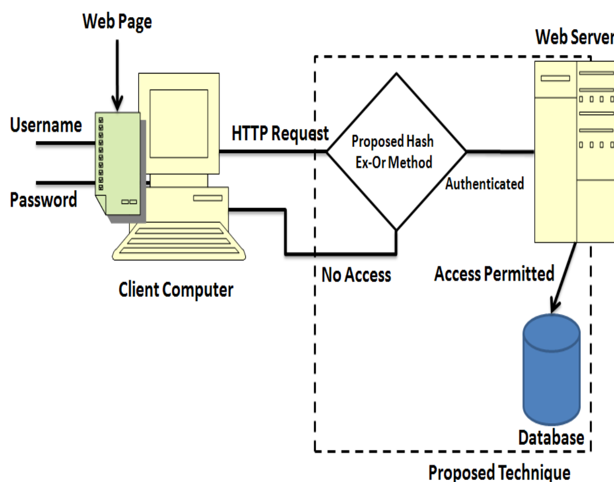

Fig 3: System design for SQLIA Detection & Prevention (SQL-ENCP)

Therefore, it cannot bypass the authentication process. The advantage of the proposed technique is that the hackers do not know about the 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.

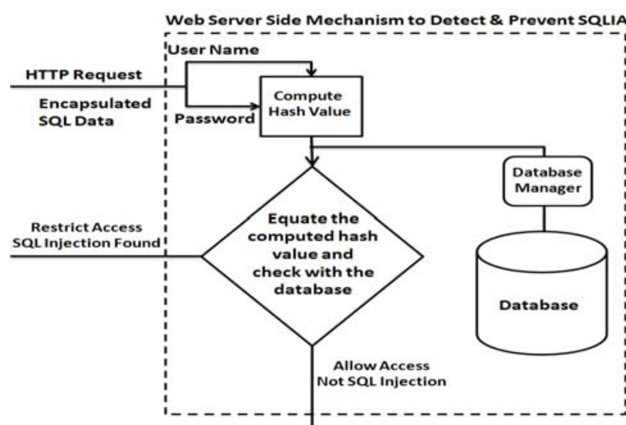

Fig 4: Proposed Hash Scheme for Detecting SQLIA & Prevent Them (SQLENCP)

Consider a scenario, where a user is authenticated by the secure login mechanism and login the system. Now, if this authenticated user make any intrusion into the system. How can we defend it?

Hence to prevent after-login attacks we have taken the help of data encryption. As we saw in the previous section that it is possible to collect the highly confidential information by using union operator we find out an alternative way to store all these confidential information's.

In our database, instead of directly storing all confidential information's, we store them in encrypted format with a secure and confidential encryption-key. Now even if the dispatcher user can able to see the atm_pin by using union operation, he cannot able to decrypt it without knowing the exact encryption method and encryption- key. So he cannot able to do any damage with that encrypted atm_pin.

VIII. IMPLEMENTATION DETAILS & RESULT ANALYSIS

The proposed scheme implemented by developing a website, using Apache 2.4.2, MYSQL 5.5.25a, PHP 5.4.4 and tested on windows environment with XAMPP 1.8.0. The proposed implemented system for SQL Injections Prevention Using Hashing & Encryption Techniques works on following two modes:

1- SIMPLE MODE
2- PROTECTED MODE

SIMPLE MODE:

In this mode the implemented system works in the non-protected environment. In which the SQL injections and vulnerabilities are not prevented & neither detected, because user inputted data is directly delivers to the database manager for the execution. It means there is no mechanism running in between the application & database to detect malicious code or query & stop the execution. Hence, there is no protection of data from the SQL injections or any illegal activity. Consider an example:

Table IV: SQL Query Become An Injection (Malicious Code)

Standard Query	SELECT * FROM tbl_user WHERE username = '@Username' AND password='@password'
Malicious Code	@username = ' OR 1=1 – – ' @password = password
SQL Injection	SELECT * FROM tbl_user WHERE name = ' ' OR 1=1 – – ' ' AND password= ' Password'
Output	Yes * As condition become TRUE

In table IV, it is clear that for any login form, if the user can inputs the malicious code in corresponding username field & password field. The Genuine SQL query will become an injection (Malicious Code). This malicious code can bypassed the authentication process easily through SQL injection.

Similarly, In Simple mode of proposed implemented system no protection from the unauthorized data access is provided. Hence, even after authorized login, one can perform the intrusion by passing malicious code to the textboxes (web control).Consider an example:

Table V: Union Query SQL Injections

Standard Query	SELECT name, phone FROM tbl_user WHERE userid=@id
Malicious Code	@username = 1 UNION ALL SELECT credit Card Number, 1 FROM Credit CardTable
SQL Injection	SELECT name, phone FROM tbl_user WHERE userid = 1 UNION ALL SELECT creditCardNumber, 1 FROM Credit CardTable
Output	Yes * Join the result of the original query with all the credit card users

We have shown the UNION queries SQL Injection in table V. Where, a user can modify the original SQL query by passing UNION operator with the malicious code in textbox. This will allow the user to access the data, which has restriction for the general user access.

PROTECTED MODE:

In this mode, we took action against the SQL injections by enforcing certain policies and mechanism, which analyses the user inputted data and make decision whether it is malicious or not. If found malicious code, discard the database access request.

Our proposed implemented system protects the database from the following three SQL injections attacks:

- Tautologies Attacks
- Illegal/Logically Incorrect Query
- Union Query SQL Injections

For the TAUTOLOGIES type's attacks, our proposed system has the key feature of using ex-or of hash values of username & password for the safe authentication as we have shown in table 3.

And to rectify the problems associated with ILLEGAL/LOGICALLY INCORRECT QUERY SQL injection, we make a module guard that checks the structure of query as well as the data passing through as argument, and returns Boolean value. TRUE means LEGITIMATE request and FALSE means ILLEGAL STRUCTURE, therefore on the basis of this decision. In this module, we map all the error messages to the programmer defined errors, which protect the application database from the leak of schema definition.

Example: consider the above standard query in table 3. Inputted value is "'abc", query formed will be

```
Select * from tbl_user where username=''abc' and password='';
```

Automated Error message:

```
System.Data.OleDb.OleDbException:Syntax error (missing operator) in query expression 'username=''abc' and password='';'
```

From the error message intruder can deduce that the type of database connectivity and the fields name in the "dept" table. This information can be used to perform further attacks on the application. Therefore, in our proposed system, we have the user defined errors that doesn't appear automated generated errors.

Hence, in above discussed case: the error message will be.

Programmer Handled Error message:

```
Incorrect Username / Password.
```

Similarly, to handle the UNION based SQL injections and after login attacks, we have taken the help of data encryption. As we saw in the previous section (table 5) that, it is possible to collect the highly confidential information by using union operator we find out an alternative way to store all these confidential information's. In our database, instead of directly storing all confidential information's, we store them in encrypted format with a secure and confidential encryption-key. Hence, with this approach the intruder cannot able to decrypt it without knowing the exact encryption method and encryption- key. Therefore, No one other than legitimate user can make changes in the data.

Though we are hiding information from spurious users, according to our business rule admin user should able see the actual confidential data of each user (Please don't compare it with real life situation. It's just an assumption). Now to allow this, we have to first decrypt the encrypted the confidential data. To decrypt a value we use the MYSQL function AES_DECRYPT (). We also need the encryption-key. This key must be store safely because if anyone can able to know both encryption method and the key then he can able to decrypt the information.

Therefore, In comparison with existing systems for SQL injections prevention, our proposed implemented system is much secure, efficient and prevents the above discussed SQL attacks (i.e. Tautology Attacks, Illegal Structured Query Attacks & Union Based Query Attacks) to assure the security of database.

The table 6 shown below is the comparison of proposed implemented system. Sign convention for the following table is (Y) for Yes, (D) for Detection and (P) for Prevention. In which, it is clear that existing system like AMNESIA[17], SQL GUARD[18], SQL-IDS[19] can only detects the SQLIAs and HASH + SALT Technique can prevents the SQLIAs of tautology & piggybacking category. While, the proposed implemented system is able to detect as well as prevents the SQLIAs of various types (i.e. Tautology, Illegal Structure, Piggy backing & Union type Injections attacks

Table VI: Comparison with existing system for SQL injections detection & prevention

Technique	Tautology	Illegal Structured	Piggybacking	Union	Stored Procedure	Inference	Alternate Encoding
PROPOSED	Y-D-P	Y-D-P	Y-D-P	Y-D-P	-	-	-
AMNESIA[17]	Y-D	Y-D	Y-D	Y-D	-	Y-D	Y-D
SQL GUARD[18]	Y-D	Y-D	Y-D	Y-D	-	Y-D	Y-D
SQL-IDS[19]	Y-D	Y-D	Y-D	Y-D	Y-D	Y-D	Y-D
Hash + Salt[24]	Y-P	-	Y-P	-	-	-	-

IX.CONLUSION & FUTURE WORK

During the study of several researches based on the SQL injection Prevention & Attacks, we found that certain cases are there, when these approaches are not found to be effective. Hence, these approaches become un-useful cannot able to detect the injections to prevent them. In addition, the attackers can access the database directly in an illegal way. Therefore, we are at the proposed a new approach that is completely based on the hash method of using the SQL queries in the web-based environment, which is much secure and provide the prevention from the attackers SQL. But, our proposed strategy requires the alterations in the design of existing schema database and a new guideline for the database user before writing any new database. Through these guidelines, we found the effective outcomes in SQL injections Preventions. After that we compared these techniques in terms of their ability to stop SQLIA. Still, we need to improve our approach so that, it can prevent the web application & database from all kind of SQL Injections. We also plan to apply SQL Prevent to dynamic discovery of SQLIA vulnerabilities.

In this work, we have concentrated on the specific area of SQL injection. We believe that this area is in need of further investigation, mainly because of many reasons: SQL injection attacks are most likely to evolve and new vulnerabilities will be found, together with new countermeasures to deal with them. Since many hacking sites are available on the web, and since attack methods are well described and distributed between hackers, we believe that information about new attack methods should continuously be surveyed and new counter measures should be developed.

REFERNCES

[1]. secerno.com," SQL Injection Attack: A Security Threat", http://www.secerno.com/?pg=SQL-Injection#2
[2]. Netcraft. Netcraft: Web Server Survey Archives. http://news.netcraft.com/archives/web_server_survey.html, February 2006.
[3]. phpBB Group. phpBB.com :: Creating Communities. http://www.phpbb.com, 2006.
[4]. WordPress. WordPress Free Blog Tool and Weblog Platform. http://wordpress.org/, 2006.
[5]. Miro International Pty Ltd. Mamboserver.com - Home. http://www.mamboserver.com/, 2006.
[6]. TYPO3 Association. TYPO3 CMS: typo3.com. http://www.typo3.com/, 2006.
[7]. Dries Buytaert. drupal.org Community plumbing. http://drupal.org/, 2006.
[8]. CERT CoordinationCenter. CERT/CC Statistics 1988-2005.http://www.cert.org/stats/,January 2006.
[9]. Chris Anley. Advanced SQL Injection In SQL Server Applications. In An NGSSoftware Insight Security Research (NISR) Publication, 2002.
[10]. Cesar Cerrudo. Manipulating Microsoft SQL Server Using SQL Injection. Technical report,Application Security, Inc., 2002.
[11]. Martin Eizner. Direct sql command injection. Technical report, The Open Web Application Security Project, 2001. http://qb0x.net/papers/MalformedSQL/sqlinjection.html.
[12]. Mitchell Harper. Sql injection attacks - are you safe? Technical report, DevArticles, may 2002. http://www.devarticles.com/content.php?articleId=138&page=2.
[13]. Stuart McDonald. Sql injection: Modes of attack, defence, and why it matters. Technical report, The SANS Institute, jul 2002. http://www.sans.org/rr/appsec/SQL_injection.php.
[14]. Aaron C. Newman. Protecting oracle databases. Technical re-port, Application Security, Inc., 2001. http://www.appsecinc.com/presentations/Protecting_Oracle_Databases_White_Paper.pdf.49
[15]. William A. Qualls. Exploit in action: A beginners view of inci-dent handling for sql injection techniques. Technical report, SANS Institute, 2003. http://www.giac.org/practical/GCIH/William_Qualls_GCIH.pdf.62
[16]. Kevin Spett. Security at the next level - are your web applica-tions vulnerable? Technical report, SPI Dynamics, 2002. http://www.spidynamics.com/whitepapers/webappwhitepaper.pdf.
[17]. Halfond, W. G. J. and A. Orso (2005). AMNESIA: analysis and monitoring for Neutralizing SQL-injection attacks. . ASE'05. Long Beach, California, USA.
[18]. G.T. Buehrer, B. W. Weide. and P. A. G. Sivilotti (2005). Using parse tree validation to prevent SQL injection attacks. Proceedings of the 5th international workshop on Software engineering and middleware. Lisbon, Portugal, ACM: pp. 106-113
[19]. Kemalis, K. and T. Tzouramanis (2008). SQL-IDS: a specification-based approach for SQLinjection detection. SAC'08. Fortaleza, Ceará, Brazil, ACM: pp. 2153 2158.
[20]. MeiJunjin (2009). An approach for SQL injection vulnerability detection. Sixth International Conference on Information Technology: New Generations: pp. 1411-1414.
[21]. R. Ezumalai, G. A. (2009). Combinatorial Approach for Preventing SQL Injection Attacks. 2009 IEEE International Advance Computing Conference (IACC 2009). Patiala, India: pp. 1212-1217.
[22]. Hirschberg, D. S. (1975). "A linear space algorithm for computing maximal common subsequences." A.C.M 18(06): pp. 341-343.
[23]. K. Amirtahmasebi, S. R. Jalalinia, S. Khadem, "A survey of SQLinjection defense mechanisms," Proc. Of ICITST 2009, vol., no., pp.1-8, 9-12 Nov. 2009.
[24]. Shubham Shrivastava, Rajeev Ranjan Kumar Tripathi, Attacks Due to SQL injection & their Prevention Method for Web-Application, International Journal of Computer Sciecne and information technologies, Vol 3 (2), pp.3615-3618, 2012.