2012 IEEE 11th International Conference on Trust, Security and Privacy in Computing and Communications

Random4: An Application Specific Randomized Encryption Algorithm to prevent SQL injection

Srinivas Avireddy[*],Varalakshmi Perumal[*], Narayan Gowraj[*],Ram Srivatsa Kannan[*],Prashanth Thinakaran[*],Sundaravadanam Ganapathi[*],Jashwant Raj Gunasekaran[#] and Sruthi Prabhu[#]
[*]*Department of Information Technology,*
Madras Institute of Technology, Anna University,
TamilNadu,Chennai-600044
[#]*Department of Computer Science,*
Madras Institute of Technology,Anna University,
TamilNadu,Chennai-600044

***Abstract* — Web Applications form an integral part of our day to day life. The number of attacks on websites and the compromise of many individuals' secure data are increasing at an alarming rate. With the advent of social networking and e-commerce, web security attacks such as phishing and spamming have become quite common. The consequences of these attacks are ruthless. Hence, providing increased amount of security for the users and their data becomes essential. Most important vulnerability as described in top 10 web security issues by Open Web Application Security Project is SQL Injection Attack(SQLIA) [3]. This paper focuses on how the advantages of randomization can be employed to prevent SQL injection attacks in web based applications. SQL injection can be used for unauthorized access to a database to penetrate the application illegally, modify the database or even remove it. For a hacker to modify a database, details such as field and table names are required. So we try to propose a solution to the above problem by preventing it using an encryption algorithm based on randomization. It has better performance and provides increased security in comparison to the existing solutions. Also the time to crack the database takes more time when techniques such as dictionary and brute force attack are deployed. Our main aim is to provide increased security by developing a tool which prevents illegal access to the database.**

Keywords- ***randomization; SQL injection; Vulnerability; web security.***

I. INTRODUCTION

According to the report by the White Hat on web security vulnerabilities 2011, it shows that nearly 14-15% of web application attacks account for SQL Injection [8]. With the increasing attacks on web applications, it is very important to have awareness about the existing attacks, because vulnerabilities such as phishing, social engineering attack, denial of service attacks have become very common. The most basic Social Engineering attacks are Phishing and Email spamming.

Another emerging phishing attack is Tab Nabbing, which can deceive even tech savvy online users [9]. A survey on web security was conducted by us and a total of about 100 students participated. It is really surprising to note that nearly 80% were unable to identify phishing attack and around 70% could not identify Email spam. Hence there is a need that everyone has basic awareness about web security, since most of the confidential transactions are carried out on the web.

In this paper we take up SQL Injection, a critical web security vulnerability. SQLIA is a type of code-injection attack [12].It is caused mainly due to improper validation of user input. Solutions addressed to prevent SQL Injection Attack include existing defensive coding practices alongside encryption algorithms based on randomization. Defensive coding mechanisms are sometimes prone to errors,hence not complete in eradicating the effect of vulnerability. Defensive programming is sometimes very labour intensive, thus not very effective in preventing SQLIA. SQL Injection Attack is an application level security vulnerability.

The main intent to use SQL injection attack include illegal access to a database, extracting information from the database, modifying the existing database, escalation of privileges of the user or to malfunction an application.Ultimately SQLIA involves unauthorized access to a database exploiting the vulnerable parameters of a web application.

A novel idea to detect and prevent SQLIA,an application specific encryption algorithm based on randomization is proposed and its effectiveness is measured. There are many methods to illegally access a database using SQLIA and most of the solutions proposed to detect and prevent it are able to solve only problems related to a subset of the attack methods.

The related work which works on similar concept named SQLrand [11] uses randomization to encrypt SQL keywords. But this needs an additional proxy and computational overhead and the need to remember those keywords. The overhead associated with this concept is removed in our proposed algorithm. It belongs to application specific class of coding methodology.

The major contributions by us in this paper include, the proposal of Random4 encryption algorithm to prevent SQLIA. A tool to generate cipher text written in C# programming language is presented. An empirical analysis based on brute force attack to show its effectiveness is

978-0-7695-4745-9/12 $26.00 © 2012 IEEE
DOI 10.1109/TrustCom.2012.232

emphasized. The proposed technique is applied in several applications to prove its correctness.

Section II gives an overview on SQL Injection and the attack types associated with it. Section III provides our proposed solution to detect and prevent SQLIA effectively. Section IV describes the prevention strategy while section V providing the results and analysis of applying the randomised algorithm .Section VI provides the concluding part of the paper.

II. SQL INJECTION

In order to locate the hotspots where SQLIA vulnerability occurs, we first discuss about the 3-tier logical view architecture of web applications [1].

A. 3-tier Architecture of web application

1) User interface tier: This layer forms the front end of the web application. It interacts with the other layers based on the inputs provided by the user.

2) Business logic tier: The user request and its processing are done here. It involves the server side programming logic. Forms the intermediate layer between the user interface tier and the database tier.

3) Database tier: It involves the database server. It is useful in storage and retrieval of data.

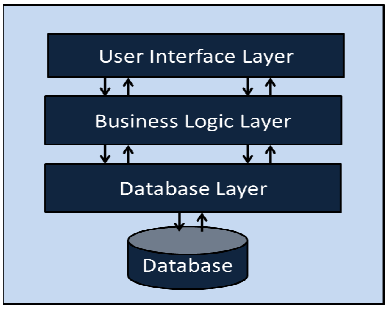

Fig 1. Web 3-tier architecture

B. Basic principle in SQL injection

SQL injection attack is a web security attack by using SQL statements exploiting the poorly designed input elements of a web form. This compromises the confidentiality and integrity of user's sensitive data. SQLIA takes place between the user interface layer and the business logic layer [1]. To understand the essence of SQL injection let us see the following example [2].

SELECT * from tablename WHERE user='' and password= ' ';

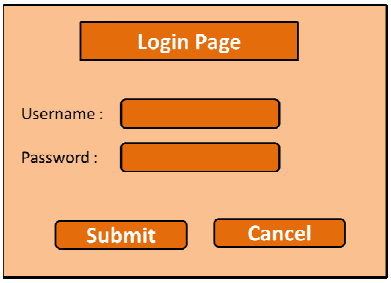

Figure 2. Sample login form

A sample SQL statement containing two input parameters is considered .Instead of typing the actual username and password, if a hacker tries to gain access to the application by inputting SQL statements, it is said to be a SQLi attempt. For example if the hacker inputs, ' OR '1'='1' --, the statement becomes,

SELECT * from tablename WHERE user='' OR '1'='1 '- - and password= ' ';

Here the user gets unauthorized access to the system because 1=1 is true always and -- indicates the statements following it are comments. Therefore, if the inputs by the user were not properly sanitized, it might lead to a critical web security attack. This describes the basic principle involved in SQL injection.

C. Attack types

This section briefly describes about the types of SQLi attacks [10] for which we propose solutions in the next section.

1) Using tautology: The example quoted in the section II.B to explain SQL injection describes the attack using tautology.

2) Using illegal/incorrect queries: By providing incorrect inputs, the database might return some important information regarding the table and fields used in the database. Using successive requests like these, the security can be compromised. In the following example instead of a valid username (xyz), an incorrect input (xyz') is provided and an error message is returned giving clues about the database making it vulnerable to intrusion.

Error: ***SELECT username, password from student WHERE username=xyz'***

Here field names such as username and password from table student are exposed.

3) Using Piggy-Backed Queries: Previously mentioned attack types try to gain unauthorized access to the application or fetch details about the database without any distinct queries

added to the input query. If additional queries are piggy-backed in the input area using special characters such as " ", " -- " or " ; " [5] hackers can modify or delete the database. An example to this is:

SELECT * from User WHERE id=123;drop table User;

Here ";" acts as a delimiter and additional drop statement may be executed and deletes the table.

4) Blind injection: Programmers try hiding database information in case of incorrect queries as a measure to protect the application from attacks using illegal/incorrect queries. Now, the hacker does not get any clues about the database. The option hence used to compromise security is, by querying the database with a lot of true/false queries and checking its result. Based on the response of the application to the queries, the hackers attempt illegal access. This type of attack is most prominently used. It includes timing attacks [10] as one of the technique to identify the behavior of the application.

III. PROPOSED SOLUTION

Section III.A describes the basic solution resembling the existing solution which is a client side validation and the Section III.B describes the proposed randomized algorithm. Section III.C deals with the tool generating the random keys for a given input.

A. Client side validation

Using client side script validation such as JavaScript, a lot of SQL injection attacks can be prevented in Web application. Though this approach does not solve all the attack types it is necessary to provide the basic security to prevent illegal attacks. The sequence of steps that increase the level of security in case of vulnerabilities is depicted in the activity diagram [Fig 3].

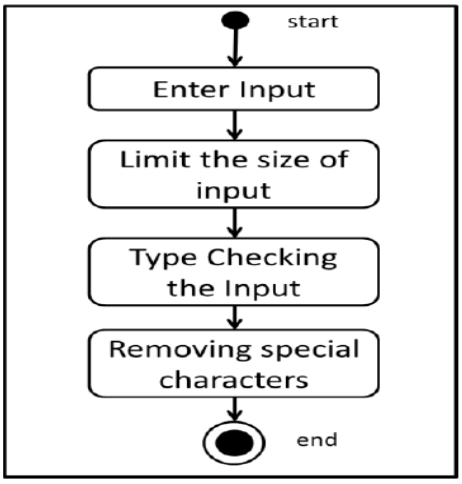

Fig 3. Activity Diagram for Client Side Validation

The advantage of client side validation is that it reduces CPU cycles since it avoids a number of round trips to the server. Some of the steps involved in client side validation include limiting the input size, restricting the use of special characters etc. But limiting the size of the input and restricting the use of special characters cannot be imposed on users in all applications. Also the protection provided by client side scripts can be easily bypassed. The use of this approach can solve attacks using tautology or incorrect queries. It cannot solve the threat posed by blind injection techniques. Hence we prefer server side validation techniques.

B. Random4 Algorithm

The random4 algorithm is based on randomization and is used to convert the input into a cipher text incorporating the concept of cryptographic salt. This algorithm forms the basis of the proposed approach. Any input in web forms will contain numbers, uppercase, lowercase or special characters. Keeping this in mind the input from user is encrypted based on randomization. In our algorithm the valid inputs are numbers, lowercase or uppercase characters and at most 10 special characters. The reason for choosing only 10 special characters is that they are rarely used and for additional security. Each character in the input can have 72 combinations (26 lowercase, 26 uppercase, 0-9 and 10 special characters). Hence for a 6 character input there can be 72^6 combinations possible. To encrypt the input, each input character is given four random values.

TABLE I
LOOKUP TABLE FOR RANDOM4

	R[1]	R[2]	R[3]	R[4]
a	;	1	x	W
…				
z	7	k	@	U
A	i	J	)	0
…				
Z	M	6	f	.
0	9	B	g	"
…				
9	c	R	j	1
@	(	a	5	0
…				
-	6	a	H	K

A sample lookup table is given in TABLE I. Based on the next input character, one of these four values is substituted for a given character. For example, let username be the input to be provided. If the username is "abc", the first character 'a' is given 4 random values in the lookup table. Based on the next input character which is lowercase 'b' here, the value R [1] is chosen. If the next character was uppercase R [2] would have been chosen and R[3] if it was a number and R [4] if it was a

special character or no character. These random values for each character are application specific. The idea behind making these random values different for different application is that, it decreases the probability of decrypting it by hackers. The algorithm to encrypt the input is presented as follows [Fig 4].The algorithm below shows that one of the four random values are chosen for each character based on the lookup table as in TABLE I.

Input: input string ip[]
Output: Encrypted string en[]
N: Length of ip[]
R[]: Random values of character

For i=1 to N
if (ip[i+1]=*null* || ip[i+1]=*lowercase*)
then en[i]=R[1]
End { if }
else if(ip[i+1]=*uppercase*)
then en[i]=R[2]
End {else if }
else if(ip[i+1]=*number*)
then en[i]=R[3]
End {else if }
else if(ip[i+1]=*spl.char*)
then en[i]= R[4]
End {else if }
End { For }
return en[]

Fig 4. Algorithm for Encryption

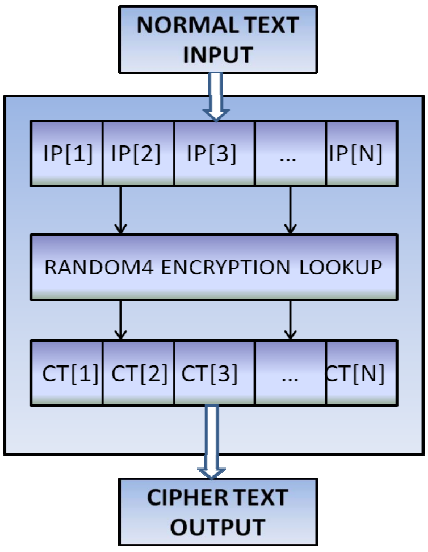

Fig 5.Framework for RANDOM4 Algorithm

C. Framework for RANDOM4

A framework to build the tool generating the encrypted text based on Random4 is shown in [Fig 5]. A normal text is given as input . Each input character is mapped to one of its four values as in the lookup table. The key values of each character are concatenated to form the cipher text.

A c# application using Microsoft Visual Studio 2007 was developed by us which is the tool to generate the encrypted keys for the given input values. This can then be stored in database and used by programmers in server side programming to prevent illegal access through SQL injection. A sample screen shot of the output generated by this tool is shown in [Fig 6].

Fig 6. A sample output for encrypted key

IV. PREVENTION STRATEGY

A. For piggy-backed query

For a piggy-backed query to be successful, we need the information about the table such as table name and the corresponding field names. By concatenating the original table and field names along with its encrypted string makes it difficult to crack the table and field names, hence preventing SQL injection attack through piggy-backed queries.

For example the table "user" becomes "userA2;h".The time to crack such cipher texts are in years. Hence for the hacker to successfully attempt an attack by piggy backing he needs the table and field names. Since it is encrypted using Random4 algorithm and the encryption logic is also application specific, it is highly difficult and time consuming for decryption and deploy an attack. Thus attack by piggy backing is prevented.

B. For Blind SQL injection

In case of piggy-backed query attacks we encrypted only the table and field names. In case of blind injection attacks, we encrypt the user input and validate it for authorized access. Hence even if attacks are attempted, they will be prevented because only the encrypted values are checked with the database and not the actual input. The following example explains this strategy of how blind injection works in a web application which is poorly designed.

SELECT * FROM user WHERE id='' and password='';

For the above the SQL statement if the id=567 and password=xyz, the SQL statement will be checked with an encrypted value such as id=g4" and password=j8G.This is based on the random values used for the particular application. Instead of valid data if illegal attempts to access database is tried, it becomes unsuccessful. For example, if an illegal SQL statement is given as input such as id=";drop table users;--",the SQL statement becomes,

SELECT * FROM user WHERE id='2dg9Jlo9&aBd3nK0SV4;' and password='';

Hence the above statement when checked with the database will not match any entry, thus returning false. An attempt to attack the database was prevented successfully using the idea of encryption based on randomization. The overall strategy of the proposed solution providing the sequence of stages involved in web application transactions is shown in [Fig 7].

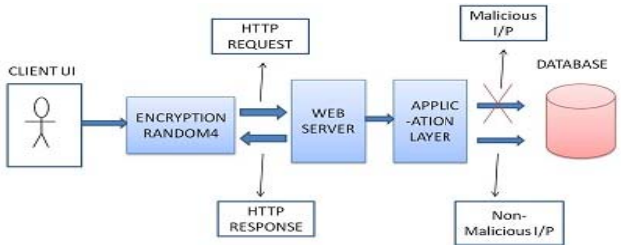

Fig 7. An overall view of the proposed system

V. RESULTS AND ANALYSIS

Section V.A deals with the experimental setup and its analysis. Section V.B compares the proposed encryption algorithm with the other tools. Section V.C provides a test case to prove Random4 algorithm and its performance.

A. Experimental Setup

A set of five PHP web applications with MySQL as the backend database were chosen for the experiment. This empirical study was to prove the effectiveness of Random4 algorithm. Table II gives information such as the web application considered, its size measured in Lines Of Code and the number of vulnerable parameters. Lines of code was measured using a open source software CLOC [17]. The web applications chosen were online college magazine, online library management system, social networking application, online payment system and online ticket booking system. The following were the results obtained on various web applications considered.

Table II
SCENARIO FOR EXPERIMENTAL SETUP

Name of the Web App.	Lines Of Code	Vulnerable parameters
Online college Magazine	3403	40
Online Library Management system	2798	35
Social networking application	5608	47
Online payment system	4214	23
Online Ticket Booking system	3309	34

The experiment results show the number of attacks made, the number of attempts that were successful and the number of attacks that were prevented using the Random4 algorithm. A graph was plotted and is shown in [Fig 8].

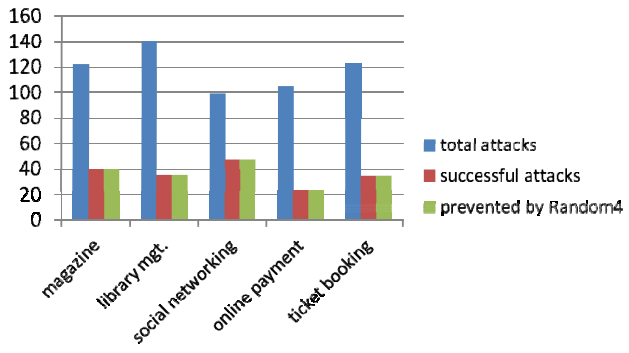

Fig 8. Graph comparing various web applications

B. Comparison with other tools

Random4 algorithm is compared with other tools used in detection and prevention of SQLIA. The considered AMNESIA [12],WAVES [13],SQL check [14], SQL rand [11] and SQL DOM [15]. The comparison was made on whether

any encoding was part of the technique and the automation in detecting and preventing SQLIA. The results are tabulated in [TABLE III].

TABLE III
COMPARISON OF PREVENTION TOOLS

Tool	Encoding	Detection	Prevention
AMNESIA	No	Automated	Automated
WAVES	No	Automated	Reports
SQL check	yes	Semi-Automated	Automated
SQL rand	yes	Automated	Automated
SQL DOM	yes	N/A	Automated
Random4	yes	Automated	Automated

C. *Test case for Random4*

A PHP based web application was built with MySQL as the backend database. To the test the idea proposed, we used a web application and the results before and after applying the encryption algorithm are shown in [Fig 9] and [Fig 10] respectively.

Fig 9. Before applying encryption

Before applying the encryption algorithm the attack using tautology technique which was discussed in section II.B was successful. An illegal access was allowed. A sample screenshot shows the effect in [Fig 9].

After applying the encryption algorithm it was found that the previously successful attack using tautology became unsuccessful. A sample screenshot to show this effect is in [Fig 10].

The performance comparison of cipher text over normal text shows that, cipher text is very difficult and time

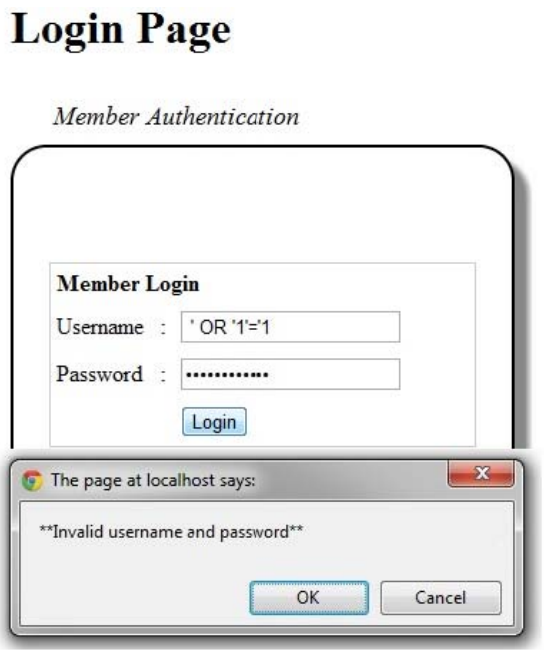

Fig 10: After applying encryption Algorithm

No of i/p characters	Normal Text	Cipher Text
5	30 mins	21 days
6	1 day	14 years
7	13 years	3659 years

Fig 11. Comparison of Normal and Cipher Text

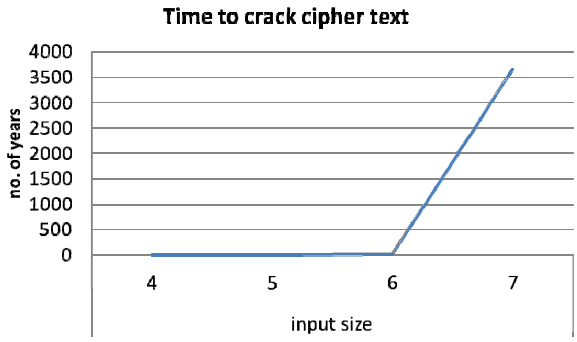

Fig 12: Time to crack Cipher text using Brute Force Technique

consuming to crack. Because the time to crack cipher text generated by Random4 using Brute force technique is in the

order of years in comparison to normal text which is normally in the order of days for smaller input size [4]. Time to crack normal and cipher text by a computer whose CPU is capable of cracking 500,000 passwords or key is tabulated in [Fig 11] and the graph is plotted for cipher text in [Fig 12].

We can infer from [Fig 12] that how powerful is a cipher text when compared to ordinary text without any encryption algorithm used. This reason why we chose is the difficulty in cracking the logic which makes use of randomization.

VI. CONCLUSION

SQL injection is one most important web security threat that needs attention so as to improve security for the users and their data. This paper deals with an application specific randomized encryption algorithm to detect and prevent it.

Futher its effectiveness was compared with other existing techniques and its performance was quantified. Hence we took up this web security vulnerability and analyzed its attack types. Security threat posed SQLIA is really high and it is very necessary to protect users' data in a web application, since it is very confidential and sensitive.

REFERENCES

[1] Jeom-Goo Kim "Injection Attack Detection using the Removal of SQL Query Attribute Values"

[2] R.Ezumalai, G.Aghila "Combinatorial Approach for preventing SQL Injection Attacks.

[3] The open Web Application Security Project, "OWASP TOP 10 project", http://www.owasp.org/

[4] http://lastbit.com/rm_bruteforce.asp

[5] NTAGWABIRA Lambert, KANG Song Lin "Use of Query Tokenisation to detect and prevent SQL injection Attacks"

[6] MeiJunjin "An Approach for SQL injection vulnerability detection"

[7] Diallo Abdoulaye Kindy, Al-sakib Khan pathan "A Survey On SQL Injection: Vulnerabilities, Attacks and Prevention techniques"

[8] https://www.whitehatsec.com/resource/stats.html

[9] Seckin Anil Unlu, Kemal Bicakci "NoTabNab: Protection against The Tabnabbing Attack"

[10] Tajpour, A., Massrum, M., Heydari, M.Z. "Comparison of SQL injection detection and prevention Techniques"

[11] Stephen W.Boyd , Angelos D.Keromytis "SQLrand: Preventing SQL injection Attacks

[12] W. G. Halfond and A. Orso. AMNESIA: Analysis and Monitoring for NEutralizing SQL-Injection Attacks. In Proceedings of the IEEE and ACM International Conference on Automated Software Engineering (ASE 2005), Long Beach, CA, USA, Nov 2005.

[13] Y. Huang, S. Huang, T. Lin, and C. Tsai. Web Application Security Assessment by Fault Injection and Behavior Monitoring. In Proceedings of the 11th International World Wide Web Conference (WWW 03), May 2003.

[14] Z. Su and G. Wassermann. The Essence of Command Injection Attacks in Web Applications. In The 33rd Annual Symposium on Principles of Programming Languages (POPL 2006), Jan. 2006.

[15] R. McClure and I. Kr¨uger. SQL DOM: Compile Time Checking of Dynamic SQL Statements. In Proceedings of the 27th International Conference on Software Engineering (ICSE 05), pages 88–96, 2005.

[16] William G.J. Halfond, Jeremy Viegas, and Alessandro Orso, "A Classification of SQL Injection Attacks and Countermeasures"

[17] http://cloc.sourceforge.net/