International Journal of Scientific Research in Computer Science, Engineering and Information Technology
© 2019 IJSRCSEIT | Volume 5 | Issue 3 | ISSN : 2456-3307
DOI : https://doi.org/10.32628/CSEIT1952339

Neutralizing SQL Injection Attack on Web Application Using Server Side Code Modification

Sarjiyus, O. and El-Yakub, M. B.
Department of Computer Science, Adamawa State University, Mubi, Nigeria

ABSTRACT

SQL Injection attacks pose a very serious security threat to Web applications and web servers. They allow attackers to obtain unrestricted access to the databases underlying the applications and to the potentially sensitive and important information these databases contain. This research, "Neutralizing SQL Injection attack on web application using server side code modification" proposes a method for boosting web security by detecting SQL Injection attacks on web applications by modification on the server code so as to minimize vulnerability and mitigate fraudulent and malicious activities. This method has been implemented on a simple website with a database to register users with an admin that has control privileges. The server used is a local server and the server code was written with PHP as the back end. The front end was designed using MySQL. PHP server side scripting language was used to modify codes. 'PDO prepare' a tool to prepare parameters to be executed. The proposed method proved to be efficient in the context of its ability to prevent all types of SQL injection attacks. Acunetix was used to test the vulnerability of the code, and the code was implemented on a simple website with a simple database. Some popular SQL injection attack tools and web application security datasets have been used to validate the model. Unlike most approaches, the proposed method is quite simple to implement yet highly effective. The results obtained are promising with a high accuracy rate for detection of SQL injection attack.

Keywords : Injection, Vulnerability, Modification, Attacks, Database.

I. INTRODUCTION

The alarming rise in fraudulent and malicious activities carried out over the Internet has brought about the classification of nine types of frauds, developed from the data reported by Internet Crime Complaint Centre (IC3) with SQL injection ranked as the second most common and destructive attack on the web. A number of recent surveys, the Open Web Application Project [1] indicate that SQLI is among top three worst vulnerabilities discovered in today's web-based applications after their deployment. Moreover, a large portion of data security breaches have been caused by SQLI attacks in real-world resulting in huge financial losses to business organizations [2]. So, detecting SQLI attacks early can reduce the potential losses.

The IC3 website has seen a number of increases in frauds involving the exploitation of valid online banking credentials belonging to small and medium sized businesses (IC3, IBM Internet Security Systems 2008). The greater percentage of Internet threats are from software application vulnerabilities and flaw in the design of software system [3].

Vulnerabilities in software may allow a third party or program to gain unauthorized access to some

resource with malicious intent. Software vulnerability control is one of the essential parts of computer and network security. Intruders use vulnerabilities in operating system and application software to gain unauthorized access, to attack, tamper with and damage other systems. This means, avoiding software vulnerability is a major counteracting action to protect software applications from threats posed by intruders through the Internet.

It is difficult to design and build a secure web application until the designer knows the possible threats in application. Hence, threat modeling is recommended as part of the design stages in web application. The purpose of threat modeling is to analyze the application's architecture and identify all potentially vulnerable areas. Developers must follow secure coding techniques to develop secure, robust, and hack-resilient solutions. The design and development of application layer software must be supported by a secured network and hosting systems.

A weak input validation is an example of an application layer vulnerability, which can result to SQL injection attack. SQL injection is a technique for exploiting web applications that uses client-supplied data in SQL queries without stripping potentially harmful characters [4]. The primary target of malicious attackers may be to obtain data from the databases. However, SQL injection offers more than the data. SQL injection enables the attacker to run arbitrary commands in the database. SQL injection bugs lead to disclosing sensitive information, tampering with the data, running SQL commands with a highly undeserved opportunity. SQL Injection (SQLI) vulnerability is a well-known growing security concern for web applications that alters the implemented query structures with supplied malicious inputs. The execution of altered queries may lead to security breaches such as unauthorized access to application resources, escalation of privileges, and modification of sensitive data [5].

The connection from the web application to the database management system is made through Application Programming Interfaces (APIs) like Open Database Connectivity (ODBC) and Java Database Connectivity (JDBC). By using the built-in objects and methods, we make the connection to the database server and execute the Structured Query Language (SQL) queries. The queries are passed into the SQL query processor and are executed. The results of the queries are returned to the application server. The application server checks the returned data and takes the decision and then renders the data in the dynamic web page. Most of the time, the query that is passed to the database server for execution contains user-supplied parameters. The input parameters provided by the user may or may not be trustworthy [6]. It is obvious that the query processor will execute the query and return the result to the user without considering about its type. But the query can still contain some malicious codes and/or may be logically incorrect.

As a result of intrusion, the data losses its confidentiality, integrity, and authenticity. This research sets out to mitigate such vulnerabilities and defend against the attacks through a different methodology that can detect and protect against SQL Injection in web applications through independent web services in a layered approach and through focusing on the identification of invalid SQL statements, analyzing the query for invalid non-SQL keywords which are not present in database, capturing errors, generalizing the error of illegal and logically incorrect queries, printing outputs, detecting SQL injections and maintaining file system for further references.

II. CONCEPTUAL FRAMEWORK

There is no doubt that Internet security is of great importance and should be highly prioritized. Web servers are the basis to respond to users' requirements using Hypertext Transfer Protocols(HTTPs), but can cause irreversible problems in case of insecurity.

Therefore, security needs should be met in web servers so as to achieve key elements in security, namely authentication, permission, confidentiality, integrity, auditing, and availability. Using vulnerabilities, hackers threaten and attack websites and web servers. A threat can potentially lead to a catastrophe that can damage an entire organization's resources and use the system's vulnerabilities after a successful attack. These attacks damage web servers using weak points and insufficient validation in web applications and web servers' infrastructures. On the other hand, weak points can help identify vulnerabilities and improve web servers' security.

SQL injection attacks are a form of injection attack, where the attacker deliberately inserts SQL commands in the input parameters with a view to altering the execution of the SQL query at the server [1].

Attackers take benefit of such situations where the developers often combine the SQL statements with user-submitted parameters and thus insert SQL commands within those parameters to modify the predefined SQL query. The result is that the attacker can run arbitrary SQL commands and queries on the database server through the application processing layer [7].

A successful SQL injection attack can read confidential data from the database, change the data (insert/alter/update/delete), run administrative processes, and retrieve the content of a given file present on the database server and can also execute operating system level commands [8].

A typical instance of SQL injection attack is given below.

Suppose a web page is generated dynamically by taking the parameter from the user in the URL itself, like;

http://www.domainname.com/Admission/
Studnets.asp?Sid=165

The corresponding SQL query associated in the application code is executed as

SELECT Name,Branch,Department FROM
Student WHERE StudentId = 165

An attacker may misuse the point that the parameter "Sid" is accepted by the application and passed to the database server without necessary validation or escaping. Therefore, the parameters can be manipulated to create malicious SQL queries. For example, giving the value "165 or 2=2" to the variable "Sid" results in the following URL:

http://www.domainname.com/Admission/
Studnets.asp?Sid=165 or 2=2

The SQL statement will now become:

SELECT Name, Department, Location FROM
Student WHERE Student Id = 165 or 2=2

This condition is always true and all the Name, Department, and Location triplets will be returned to the user. The attacker can further exploit this vulnerability by inserting arbitrary SQL commands. For example, an attacker may give request for the following URL:

http://www.domainname.com/Admission/
Studnets.asp?Sid=165; DROP TABLE Student

The semicolon in the above URL terminates the server side SQL query and appends another query for execution. The second query is "DROP TABLE Student" which causes the database server to delete the table. In a similar way, an attacker can use "UNION SELECT" statement to extract data from other tables as well. The UNION SELECT statement allows combining the result of two separate SELECT queries. For example, consider the following SQL query:

http://www.domainname.com/Admission/
Studnets.asp?Sid=165 UNION SELECT
UserId, Username, Password FROM Login;

The default security model for many web applications considers the SQL query as a trusted command. This allows the attackers to exploit this vulnerability to bypass access controls, authorization, and authentication checks. In some cases, SQL queries allow access to server operating system commands using stored procedures. Stored procedures are usually bundled with the database management server. For example, in Microsoft SQL Server the extended stored procedure xp cmd shell executes operating system commands.

Therefore, in the previous example the attacker can set the value of "Sid" to be "165; EXEC master..xp cmdshell dir – –"; this if executed will return the list of files in the

current directory of the SQL Server process. The use of LOAD FILE('xyz.txt') in MySQL allows the attacker to load and read arbitrary files from the server.

The default security model for many web applications considers the SQL query as a trusted command. This allows the attackers to exploit this vulnerability to evade access controls, authorization, and authentication checks. In some cases, SQL queries allow access to server operating system commands using stored procedures. Stored procedures are usually bundled with the database management server [6].

Khaimar [9] pointed out that Code Injection Attack is broadly classified into SQL Injection Attack (SQLIA) and Cross Site Scripting Attack (XSS).

These vulnerable applications are targeted by attackers or malicious users to perform their illegal activities.

III. METHODS AND MATERIAL

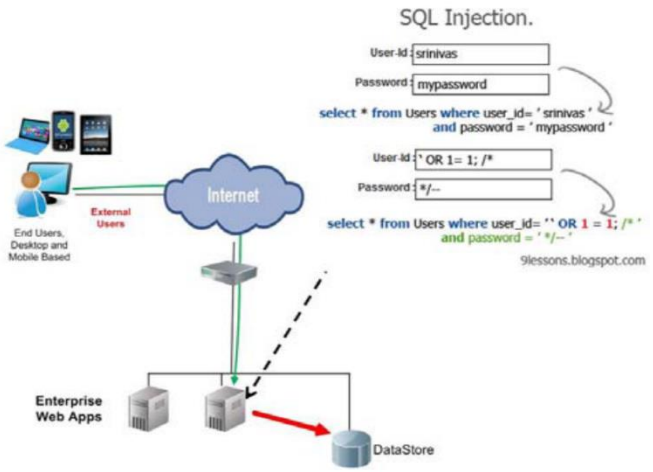

Fig. 1: An illustration of how SQL Injection is carried out [10].

2.1 Types of SQL Injection Attacks.

According to [6] there are different types of SQL injection attack as presented in many studies. These attack types have been named based on the technique implemented to exploit the injection vulnerability as listed below:

(1) Tautology.

Tautology is such a logical statement which is TRUE in every possible interpretation. In SQL queries, the same concept may be used in the conditional statement of the query, that is, in the WHERE clause, to make it always TRUE returning all data. The simple use of tautology is as follows

select ∗from admin where user id= ' '
and password = ' ' or 'a' = 'a'

This is often inserted in the vulnerable parameter to perform the injection attack. This tautology is mainly applied to bypass the login authentication. Tautology is also used to confirm the blind SQL injection vulnerability.

(2) Commenting the Code.

Like other programing languages, SQL also can specify comment line in the code. By adding a double

hyphen in MS-SQL or a # in the case of MySQL, one can comment the code. The comment line prevents the code from execution. The attackers take advantage of this and insert a comment in the vulnerable parameter to disable the rest of the code following the vulnerable parameter. A simple example of using a comment line is

SELECT ∗from admin where userid= 'xxx';
-- and password ='yyy';

The above code can bypass the login authentication by giving only valid user id.

(3) Type Mismatch.

In case of type mismatch in the query, SQL provides a verbose error message, for instance,
http://www.domainname.com/Admission/
Studnets.asp?Sid=system user

The error output is like:

[Microsoft][ODBC SQL Server Driver][SQL Server] error: xxx, Conversion failed
when converting the varchar value 'sa'
to data type integer.

From the above error message, we can clearly know that the current user is 'sa'; hence, the attacker takes advantage of this and provides type mismatch queries like giving characters to a numeric type and vice versa and can easily extract a lot of information.

(4) Stacked Query.

When a sequence of multiple SQL queries executed in a single connection to the database server this is called stacked or piggybacked query. Being able to terminate the existing query and attach a completely new one, taking advantage of the fact that the database server will execute both of them, provides more freedom and possibilities to the attacker compared to simply injecting code in the original query. Most of the DBMS supports the stacked query.

An example of stacked query for DROP and UPDATE is given below:

http://www.domainname.com/Admission/
Studnets.asp?Sid=165; DROP TABLE Student
http://www.domainname.com/Admission/
Studnets.asp?Sid=165; UPDATE login set
password = 'xxx' where userid = 'yyy'

Similarly, stacked query can be written and executed for ALTER, DELETE, and so forth. This can severely impact the back-end database.

(5) Union Query.

The union operator combines the results of two SELECT queries and returns the result as one. Hence, once we enumerate the table names and column names, we can inject the UNION SELECT statement in the vulnerable parameter to combine the results with the original query and retrieve the data. The example of using UNION SELECT is:

http://www.domainname.com/Admission/
Studnets.asp?Sid=165 UNION SELECT
userid, password FROM login;

The above request will combine the user-id and password pair with the original query and will be displayed to the client. We can further modify the query to iterate through all the rows of the login table.

(6) Stored Procedure and System Functions.

In DBMS, a stored procedure is a group of SQL statements combined to create a procedure that is stored in the data dictionary. Stored procedures are present in compiled form so that many programs can share them. The practice of using stored procedures can be useful in improving productivity, preserving data integrity, and controlling data access. The attacker can take help of these stored procedures to impact the SQL injection attack severely.

An example of using the stored procedure is:
exec master..xp cmdshell 'ipconfig'

xp cmdshell is an extended stored procedure available in MSSQL which allows the administrator to run operating system level commands and get the desired output.

The use of system defined functions also helps in performing SQL injection. In SQL Server 2005 hashes are stored in the sql logins view. The system hash can be retrieved using the query:

SELECT password hash FROM sys.sql logins
http://www.domainname.com/Admission/
Studnets.asp?Sid=165+union+select+master.
dbo.fn varbintohexstr(password hash)+
from+sys.sql logins+where +name+=+'sa'

The function fn varbintohexstr() converts the password hash stored in the varbinary form into hex so that it can be displayed in the browser and then tools like "Cain and Abel" are used to decrypt the hash into plain text.

(7) Inference.

Inference is the act or process of deriving logical conclusions. Sometimes we test through inference to extract some information; that is, "if we get this output, then this might be happening at the back-end." Inference techniques can extract at least one bit of data by noticing the response to a specific query. Observation is the key, as the response of the query will have a separate signature when the query is true and when it is false. An example of using inference in SQL injection is:

http://www.domainname.com/
Admission/Studnets.asp?Sid=165 and
SUBSTRING(user name(),1,1)='c' –

If the first character of the USER is indeed 'c' then the second condition (SUBSTRING(user name(),1,1)='c') is true and we would see the same result and if not then we may get the output as "no records exist" or something other than the usual output.

The False and True conditions states are inferred from the response on the page after each request is submitted; that is, if the response contains "no records exist" the state was False; otherwise, the state was True. Similarly, by repeating the process, starting with the letter 'a' and moving through the entire alphabet, we can infer all successive character of the USER name, for instance,

Sid=165 AND SUBSTRING(user name(),2,1)=
'c' (False)
Sid=165 AND SUBSTRING(user name(),2,1)=
'd' (True)
Sid=165 AND SUBSTRING(user name(),3,1)=
'e' (False)
Sid=165 AND SUBSTRING(user name(),3,1)=
'b' (True)

(8) Alternative Methods.

Web applications often use input filters that are designed to protect against basic attacks, including SQL injection. To evade such filters, attackers may use some encoding technique. The technique is achieved using case variation, URL encoding, CHAR function, dynamic query execution, null bytes, nesting striped expressions, exploiting truncation, and so forth [6]. By using the above methods, the attacker bypasses the defending mechanisms. Examples of using alternative methods are as follows:

CHAR Function
UNION = CHAR(85) + CHAR(78) + CHAR(73) +
CHAR(79) + CHAR(78)
HEX Encoding
SELECT = 0x53454c454354
URL Encoding
SELECT%20%2a%20FROM%20LOGIN%20WHERE%
20USERID%20%3E%2010
Case Variation
uNiOn SeLeCt usErID, password FrOm

tblAdmins WhErE uname='admin'—

2.2 Existing Methods for Mitigating SQL Injection Attack

Categorizing the different approaches to tackle the issue of SQL injection attack involve the following:

(i) Static Analysis. Some approaches rely purely on static analysis of the source code [11]. These methods scan the application and use heuristics or information flow analysis to detect the code that could be vulnerable to SQL injection attack. Each and every user input is inspected before being integrated into the query. Because of the inaccurate nature of the static analysis that is being used, these methods can produce false positives. Moreover, since the method relies on declassification rules to convert untrusted input into safer one, it may generate false negatives too. Wassermann and Su [12] proposed a method that combines static analysis and automated reasoning techniques to detect whether an application can generate queries that contain tautologies. This technique is limited to the types of SQL injection attack that it can detect.

(ii) Static Analysis and RuntimeMonitoring. Some approaches like Analysis and Monitoring for Neutralizing SQL Injection Attack (AMNESIA) [12] have combined both static analysis and runtime monitoring. In the static part, they build legitimate queries automatically that the application could generate. In the dynamic part, the dynamically created runtime queries are monitored and are checked for the amenability with that of the queries generated in the static part. This approach depends on the following:

(i) First is scanning the whole application code to define the critical spots.
(ii) Within each critical spot, the authors of that paper "AMNESIA" generate SQL query models by figuring the possible values of query string that may be passed to the database server.
(iii) For each critical spot, this approach makes a call to the monitoring procedure with two different parameters (the string that contains the actual query to be submitted and a unique identifier).

(iv) During execution when the application reaches that spot, the runtime monitor is being invoked, and the string that is about to be submitted as a query is passed as a parameter with unique id.

(v) Then the method AMNESIA retrieves the SQL query model for that spot and checks the query against the previously generated static model.
This tool limits the SQL injection attack during static analysis phase for query building and also it has certain limitations particularly in thwarting attacks related to stored procedures.

(iii) Context-Oriented Approach. Context-oriented approach provides a novel method for protection against different types of attack in web applications [7].

This work presents a single generic solution for various types of injection attack associated with web applications. The authors have taken an alternative view of the core root of the vulnerabilities. In this work the common attack traits are analyzed and on this basis a context-oriented model for web applications protection is developed. But the presence of a backdoor in the code may not get detected by the model. In the case of code obfuscation, code hiding, and so forth the method may not be able to function as intended. Another approach by [7] provides a generic and extensible PHP-oriented protection framework. The proposed framework is mainly based on intention understanding of the application developer. It makes a real-time supervision of the execution and detects deviations from the intended behavior, which helps it in preventing potentially

malicious activity. This method purely focuses on attack detection in six Security and Communication Networks PHP environment. This method fails to defend the attacks if the application is developed using technologies other than PHP.

(iv) Input Validation. The cause of many injection vulnerabilities is the improper separation of code and input data. Hence various techniques have been proposed on the basis of input validation. Security Policy Descriptor Language (SPDL) [13] is used for controlling the flow of user input through the secure gateway. The specified policy analyses and transforms each request/response by enforcing user input constraints. Tools like PowerForms [14], AppShield [15], and InterDo [16] use similar methodology. As these approaches are signature-based, they can have insufficient input validation routines and may introduce false positives. As the approaches are also human based, much effort is required to determine the data that needs to be filtered and the right policy to be applied.

(v) Instruction Set Randomization. The SQLrand [17] is such a method which adds a random token to each keyword and operator to all SQL statements in the program code. Before the query is being sent to the database, it is checked that all the operators and keywords must contain the token. The attacks would be easily detected as the operators and keywords injected by the attacker would not have that token. This method involves randomizing both the underlying SQL parser in the database and the SQL statements in the program code which makes it cumbersome. Adding the random tag to whole SQL statement and each keyword makes the query arbitrarily long. Also using this method makes it open to the possibility of brute-force attack.

(vi) Learning-Based or Anomaly Detection Methods. A set of learning-based approaches has been proposed to learn all the intended query structure statically or dynamically [18]; [19]. The effectiveness of detection largely depends on the accuracy of the learning algorithms. The approach in [20] focuses on securing the web application from external and internal attacks. SQL Injection and Insider Misuse Detection System (SIIMDS) is a technique that takes advantage of both misuse detection methods and anomaly detection methods to reduce the risk resulting from SQL injection attack. It consists of three modules such as misuse detection, anomaly detection, and a response module. The SQL statement is compared with a list of stored SQL injection signature patterns. If there is a match, there is an attack and the SQL statement is now passed to the response module for necessary action.

Furthermore, if there is no-match found with the stored attack pattern, the SQL statement is forwarded to anomaly detection module for behavioral analysis. If some abnormality is found, then the SQL statement is passed to the response module for appropriate action. Otherwise, the SQL statement is considered to be perfectly attack-free and ready for execution.

In summary, here are the list of the existing approaches:

i. AMNESIA (Analysis and Monitoring for Neutralizing SQL Injection Attacks)
ii. SQLrand (SQL randomization)
iii. SPDL (Security Policy Descriptor Language)
iv. SIIMDS (SQL Injection Insider Misuse Detection Systems)
v. SQLIPA (SQL Injector Protector for Authentication)

A detailed study of literatures shows that considerable efforts have been made to devise many techniques for preventing SQL injection attacks. However, security in web applications cannot be disregarded as it has a wide existence.

In accordance with this, existing literatures for preventing SQL injection attacks in web applications have been studied and below are some of the methods used to mitigate the menace of SQL Injection attack.

Asagba and Ogheneovo [21] stated that Detection or prevention of SQLIAs is a topic of active research in computer security. State-of-the-practice SQL countermeasures are far from effective.
OWASP [1] notes that almost all Web applications deployed today are still vulnerable to SQL injection attacks. Signature-based Web application firewalls which act as proxy servers filtering inputs before they reach Web applications and other network-level instruction detection methods may not be able to detect SQLIAs that employ evasion techniques [22].
Boyd and Keromytis [17] proposed a method named as Instruction Set Randomization. In this approach a random integer is appended with the valid SQL keyword before sending to database. Recognition of a random integer is difficult for both attacker and database too. So author proposed independent module which decodes SQL keywords with their original name before sending to database. This method has negligible performance overhead.

Halfond and Orso [19] proposed an approach AMNESIA (Analysis and Monitoring for Neutralizing SQL Injection Attacks). This approach is based on static and dynamic analysis of queries. In Static phase, query model is generated at each point of access to the database. In dynamic phase, queries are intercepted before sending to the database and are checked against statically to build model. Performance of this approach is totally based on the static analysis for building query models.

McClure and Krüger [23] pointed that their approach is based on the object oriented programming. Their solution consists of an executable Sqldomgen which is executed against a database [24]. This is referred as a SQL domain object model (SQL). These classes are useful to construct a dynamic SQL statement with manipulating any string. In this approach every valid SQL statement is constructing using an object data model. Next, they obtain the schema of the database, and then iterate through the tables and columns contained in the schema and output number of files containing a strongly typed instances of the abstract object model [24].

Buehrer, Weide and Siviliotti [25], proposed an approach SQL Guard. In this approach queries are checked at runtime. Here, the runtime evaluation of a query based on a model which is expressed as a grammar that only accepts legal queries. SQL Guard approach use secret key to delimit user input during parsing by runtime checker. SQL Guard approach is stopping all type of SQLIA except stored procedures.
Su and Wassermann [26] proposed an approach named as SQL Checker here they implement their algorithm on a real time environment and are checked at a runtime. A secret key is used to delimit the user input. Overhead of this method is low.

Bisht and Madhusudan [27], proposed an approach CANDID. This method is based on a dynamic candidate evaluations method, which automatically prevent SQLIA. This framework dynamically extracts the query structures from every SQL query location which are projected by the developer (programmer). This approach solves the issue of a manually modifying the application to create the prepared statements.

Putthacharoen and Bunyatnoparat [28], proposed a method based on a concept named as a dynamic cookies rewriting. This approach is work with cookies. A proxy agent is placed between client and server. Cookies rewriting method can change the value of name attribute in the cookies field. Those cookies are stored with their original names at server. As browser's database do not store original

information of cookies, so even if attackers steal cookies from the database, they cannot be used later to impersonate the users [24].

The tool detected both categories of XSS attack without having any changes made at the client and server site. But the proxy failed to intercept https requests coming from the client [24].

Galan, Alcaide, Orfila, and Blasco [29] proposed a method name as a multi-agent scanner. This method is work well with both type of XSS Attack namely stored and reflected attack. This proposed method is tested in different scenarios; secured and unsecured, but only basic attack vectors were tested, more vectors can be added to test the accuracy of their approach [24].

Choudhary, and Dhore [24], proposed an approach CIDT (code injection detection tool). Here query detector is used to detect SQLIA and script detector is used to detect XSS. This approach is work with both type of code injection attack. But this method fails to detect stored procedure attack.

IV. MATERIALS AND METHOD USED

The methodology adopted for the application development is the waterfall model. This is because it explicitly outlines each step and processes associated with developing an application. Also, the use of unified modeling language (UML) as a visual language allows the modeling of processes, software and system in order to clearly express the design of the system architecture. Interview was administered to 55 system administrators in various organizations including institutions of higher learning, all in Nigeria. In addition, thirty (30) journal articles relating to server and database security were reviewed. The framework was actualized using Hypertext Markup Language, Cascading Style Sheet (CSS3) and JavaScript, PHP and structured Query Language MySQL. In designing the front- end interface, the Hypertext Markup Language (HTML5) was used. The server used was the local server and the server code was written with PHP as the back-end. PHP server side scripting language was used to modify codes while Acunetix was used in testing the vulnerability of the codes.

3.1 Designing the System

The query written by the developer is static until it gets the input parameters from the user. As the input provided by the user may not be trusted, the aim here is to take care of the query which contains any user input. The attacker may input malicious code along with the input parameter. The malicious input can make a terrible impact on the database server, starting from extracting the sensitive data from the database to taking complete control over the database server. Hence, this new system methodology 'Neutralizing SQL Injection Attack on web applications using server side code modification', monitors the query to check whether the user has added any such additional character other than the intended parameter. The method involves the following steps for dealing with the SELECT query which contains a WHERE clause.

Step 1. From the SELECT query, all characters after the WHERE clause are extracted and stored in a string S1.

Step 2. Input parameters are accepted from the user(it could be in the form of login information or injection malware). The parameters are checked for their appropriate type. If the input type matches the required type, the input parameters are added to the query. Otherwise, the parameters are rejected, and the page is reloaded with a warning message of "Invalid Parameters."

Step 3. The query string is normalized to convert it into a simple statement by replacing the encoding if any.

Step 4. Using the string extraction method all characters after the WHERE clause is extracted.

Step 5. The input parameters from the extracted string are removed sequentially as they were added. For numeric parameters, the numbers are removed, and for alphanumeric parameters, the characters enclosed in single quotes are removed. The new string is named as S2.

Step 6. Strings S1 and S2 are compared if they match and then it is considered that there is no injection attack, and the query is sent to the database server for execution. Otherwise, the query is dropped and the page is reloaded with a warning message of "Error processing bad or malformed request OR YOU ARE AN UNAUTHORISED PERSONNEL". Here in this step the neutralization is done.

The SQL query may have NONWHERE clauses such as HAVING, LIKE, and ORDER BY, which may contain the user-supplied parameter. In such cases at Steps 1 and 4 the developer has to replace the WHERE with these NONWHERE clauses.

The Fig. 2 below explains the conceptual model design for preventing SQL Injection attack.

The proposed model is incorporated in the test web application for implementation purposes. The web application contains queries to display pages containing data from several tables. The similar set of codes with necessary changes are tested against all types of SQL queries, using a combination of all parameter types, and queries for INSERT, UPDATE, and DELETE operations.

The input parameters from the user are checked for its appropriate type. Type checking reduces the chance of attack to some extent. Then, the query string is normalized to replace the encoding. The string extraction function is called again to extract the string. Then, by specifying the number of parameters and their types, the parameters are removed. For numeric parameters, numbers are removed and, for character type, characters enclosed in single quotes are removed. Finally, strings are compared for their equality. If the strings are equal, then the query is sent to the database for execution. Otherwise, a warning is generated suspecting SQL injection attack.

Thus using PHP; PDO::prepare- prepares an SQL statement to be executed. The SQL statement can contain zero or more named (:name) or question mark (?) or slashes (/) parameter markers. The prepare method helps to remove such parameters for which real values will be substituted when the statement is executed.

Parameterized queries are simple to write and understand. They force you to define the SQL query beforehand, and use placeholders for the user-provided variables within the query. You can then pass in each parameter to the query after the SQL statement is defined, allowing the database to be able to distinguish between the SQL command and data inputted by a user. If SQL commands are inputted by an attacker, the parameterized query would treat these as untrusted input, and the injected SQL commands will never get to execute.

- Using PDO (PHP Data Objects):

Many web developers likely learned to access databases by using PHP's mysql or mysqli extensions. While it is possible to write parameterized queries with PHP's mysqli extension, PHP 5.1 introduced a better way to work with databases — PHP Data Objects (PDO). PDO not only provides methods that make parameterized queries easy to use, but also makes code more portable (PDO works with several databases, not just MySQL) and is easier to read.

By properly parameterizing SQL queries, any user input that is passed to the database is treated as data and can never be confused as being part of a command.

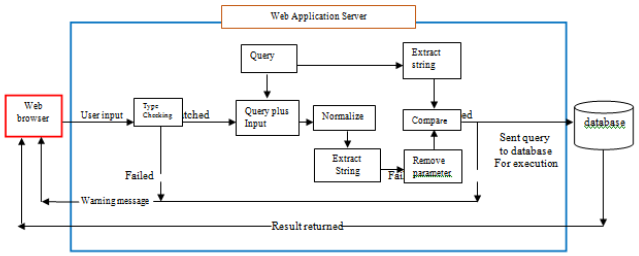

Figure 2. Conceptual Model design for preventing of SQL injection attack

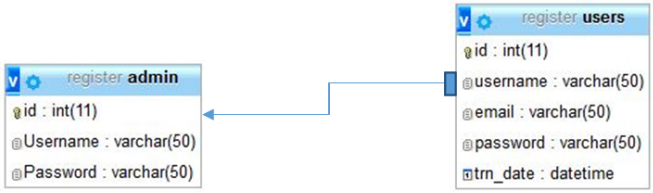

Figure 3. Entity Relationship (E-R) Diagram

3.1.1 Input Design/Specification

The expected input to the system are email, username, password and date. The password is encrypted with MD5 an encryption algorithm.

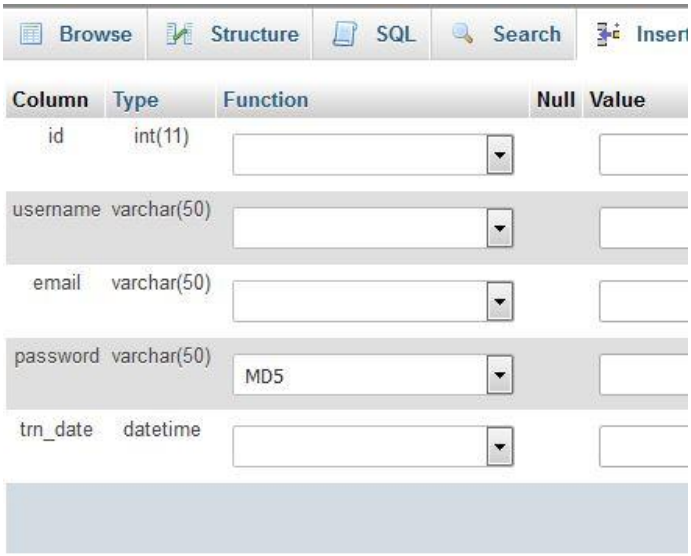

Fig. 4: Input Design

3.2 SYSTEM REQUIREMENTS

The system requirements are divided into software and hardware requirements.

3.2.1 Software Requirements:

The software requirements include:

- Windows 7 Operating System (at least).
- HTML5
- Php5.5
- CSS3
- JavaScript
- MySQL.
- Brackets (text editor)
- Apache Server (WAMP or XAMPP)

3.2.2 Hardware Requirements.

The hardware requirements include:

- Computer System (Laptop or Desktop)
- 1Gb Ram (at least)
- 50Gb Hard disk (at least).

V. IMPLEMENTATION

Administrator Login Page

The administrator has privileges to view the database and delete users using the administrator username and password.

Fig. 5: Administrator Login page

After login in, the administrator gains access to the panel below.

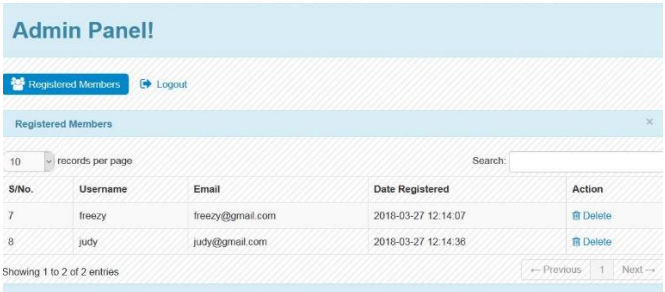

Fig.6: Administrator Panel

Database of the system

The simple web application has a database with the following table called 'users'.

id	username	email	password	trn_date
4	freezy	freezy@gmail.com	827ccb0eea8a706c4c34a16891f84e7b	2016-06-14 14:37:46
5	judy	judy@yahoo.com	c6a1ca47b645f4c4b786ce951f8d26a7	2016-06-14 14:49:09
6	freezy	freezy@gmail.com	ee23cd19091ba88bc3cf974d9a5c66ca	2016-08-26 19:15:53
7	phknklsi	sample@email.tst	32cc5886dc1fa8c106a02056292c4654	2018-03-15 10:09:08
8	ivjyhdnb	sample@email.tst	32cc5886dc1fa8c106a02056292c4654	2018-03-15 10:12:24
9	ijstfxsn	sample@email.tst	32cc5886dc1fa8c106a02056292c4654	2018-03-15 10:39:33

Fig.7: Database Table 'users'

Users Login page

Registered users login using this page.

Fig.8: User Login page

Users Registration Page

Intending users login using this page.

Fig.9: User Registration Page

The following is a display of a legitimate HTTP request that could be made to the vulnerable code above:

http://localhost/?id=2

OUTPUT > judy

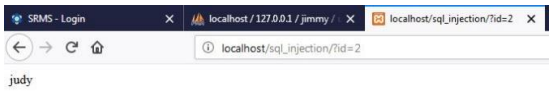

Fig.10: Legitimate HTTP request.

Injecting the code below displays the password for the user with the inputted id

http://localhost/?id=-2 UNION SELECT password FROM users where id=2

OUTPUT>3d8ea529e337658e5b84749970f41796

The encrypted password can be decrypted.

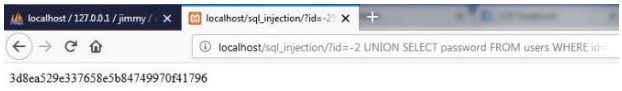

Fig.11: Result of SQL Injection on vulnerable code.

SQL Injection on secure code

Injecting this code:

http://localhost/?id=-2 UNION SELECT password FROM users where id=2

OUTPUT>Error processing bad or malformed request OR YOU ARE AN UNAUTHORIZED PERSONNEL

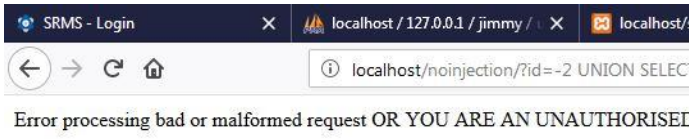

Fig.12: Result of SQL Injection on Secure Code

Acunetix test on vulnerable code

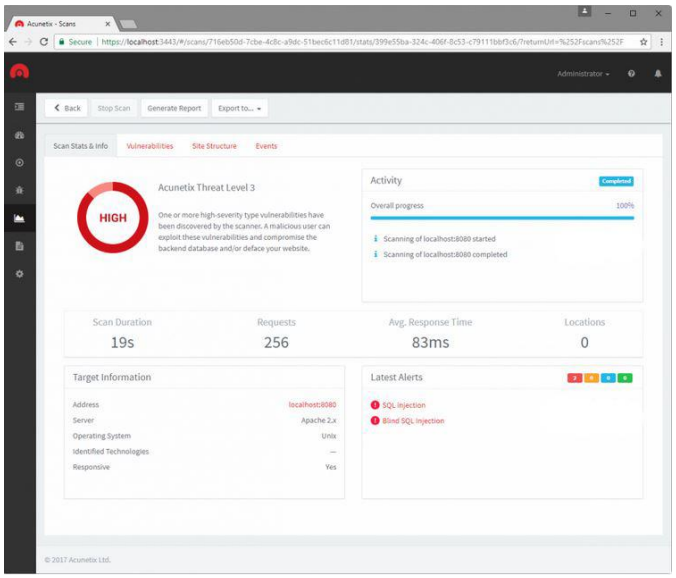

Fig. 13: Acunetix SQL Injection Vulnerability Scan with threat detected.

Acunetix test on secure code

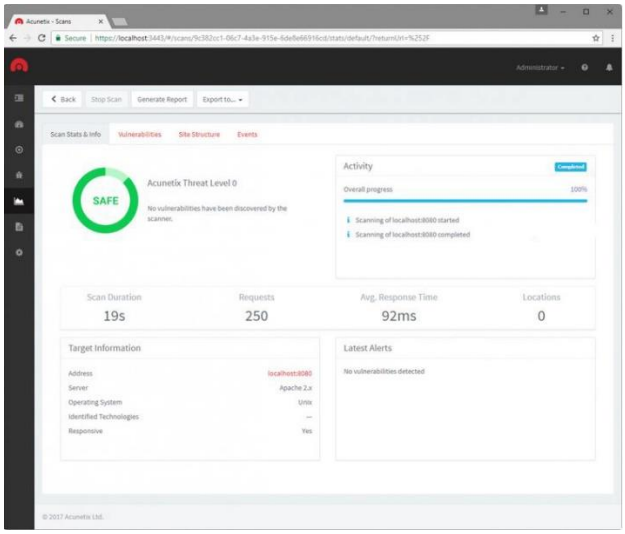

Fig. 14: Acunetix SQL Injection Vulnerability Scan with no threat detected.

A quick scan with the 'SQL Injection' Scan Type in Acunetix confirms the vulnerability.

VI. CONCLUSION

The new system framework for the application termed 'Neutralizing SQL Injection attack on web application using server-side code modification' is a novel online detection method against SQL injection attack. Its strength depends on sequentially extracting the intended user input from the dynamic query string to check for any malicious input. Unlike other existing approaches, this method is quite simple to implement yet highly efficient and effective in detecting attacks. The method has been implemented in the test web application to demonstrate its effectiveness.

VII. REFERENCES

[1]. OWASP (2010). Open Web Application-Top-Ten-Projects.

[2]. Curtis S. (2012). "Barclays: 97 percent of data breaches still due to SQL injection", http://news.techworld.com/security/3331283/b

arclays-97-percent-of-data-breaches-still-due-sqlinjection

[3]. Hossain Shahriar, Sarah North, and Wei-Chuen Chen, (2017).“Early Detection of SQL Injection Attacks”.

[4]. Shanmughaneethi V.and Swamynathan S. (2012). “Detection of SQL Injection Attack in Web Applications using Web Services”.

[5]. Johns M., BeyerleinC., GieseckeR., PoseggaJ., “Secure Code Generation for Web Applications,”Proc. of the 2nd International Symposium on Engineering Secure Software and Systems (ESSoS '10),Pisa, Italy, LNCS 5965, pp. 96-113, Springer.

[6]. Dalai A. K. and Jena S. K. (2017). “Neutralizing SQL Injection Attack in Web Applications Using Server Side Code Modification”.

[7]. Prokhorenko V., Chook. R., and Ashman H., (2016). “Context-oriented Web application protection model,” Applied Mathematics and Computation, vol. 285, pp. 59–78.

[8]. Guimaraes B. D. A., (2009). Advanced SQL injection to operating˜ system full control, Black Hat Europe, white paper.

[9]. Khaimar, C. (2015). “Detection and automatic prevention against SQL Injection Attack and XSS attack performed on web applications”.

[10]. Parveen, S. and Chandrakant, S. (2017). “SQL Injection Impact on Web Server and Their Risk Mitigation Policy Implementation Techniques: An Ultimate solution to Prevent Computer Network from Illegal Intrusion”.

[11]. Livshits V. and Lam M., (2008). “Finding security vulnerabilities in Java applications with static analysis,” in Proceedings of the 14th Conference on USENIX Security Symposium, pp. 18–25, Baltimore, Md, USA.

[12]. Wassermann G. and Su Z., (2004). “An analysis framework for security in Web applications,” in Proceedings of the FSE Workshop on Specification and Verification of Component-Based Systems (SAVCBS ’04), pp. 70–78, Citeseer.

[13]. Scott D. and Sharp R., (2003). Abstracting Application-level Web Security. In Proceedings of the 11th International Conference on the World Wide Web, pages 396–407.

[14]. Brabrand C., Møller A., Christensen R. M., and Schwartzbach M. I., (2000) “Power Forms: declarative client-side form field validation,” World Wide Web Journal, vol. 7, no. 43, pp. 205–314.

[15]. Sanctum Inc, (2002). App Shield 4.0 Whitepaper, http://www .sanctuminc.com.

[16]. Kavado I, (2003). InterDo Version 3.0, http://www.protegrity.com/data-security-platform.

[17]. Boyd S.W. and Keromytis A.D. (2004). SQLRand: Preventing SQL Injection Attacks. In Proceedings of the 2nd International Conference of Applied Cryptography and Network Security (ACNS ‘04), Yellow Mountain, China, pp. 292 -302.

[18]. Lee S., Low W, and Wong P., (2002). “Learning fingerprints for a database intrusion detection system,” in Computer Security— ESORICS 2002, pp. 264–279, Springer.

[19]. Halfond W. G. J. and Orso A., (2005). “AMNESIA: analysis and monitoring for Neutralizing SQL-injection attacks,” in Proceedings of the 20th IEEE/ACM international Conference on Automated Software Engineering (ASE ’05), pp. 174–183, ACM, Long Beach, Calif, USA.

[20]. Asmawi A., Sidek Z. M., and Razak S. A., (2008). “System architecture for SQL injection and insider misuse detection system for DBMS,” in Proceedings of the International Symposium on Information Technology (ITSim ’08.

[21]. Asagba P.O. and Ogheneovo (2011). “A Proposed Architecture for Defending Against Command Injection Attacks in A Distributed Network Environment”

[22]. Maor O. and Shulman A. (2005). SQL Injection Signatures Evasion. White Paperof Imperva International.

[23]. McClure R.A., and Kruger I.H., (2005). "SQL DOM: compile time checking of dynamic SQL statements," Software Engineering, 2005. ICS 2005. Proceedings. 27th International Conference on, pp. 88- 96, 15-21.

[24]. Choudhary, A. S. and Dhore, M.L. (2012) CIDT: Detection of Malicious Code Injection Attacks on Web Application" International Journal of Computer Application, Volume 52-No.2.

[25]. Buehrer G., Weide B. W., and Sivilotti P. A. G., (2005). "Using parse tree validation to prevent SQL injection attacks," in Proceedings of the 5th International Workshop on Software Engineering and Middleware (SEM '05), pp. 106–113, ACM, Lisbon, Portugal.

[26]. Su Z. and Wassermann G. (2006). The Essence of Command Injection Attacks in Web Applications. In Conference Record of the 33rd ACM SIGPLAN—SIGACT Symposium on Principles of Programming Language POPL'06, New York, NY, pp. 372 – 382.

[27]. Bisht P., Madhusudan P. and Venkatarish-nan V.N. (2010). CANDID: Dynamic Candidate Evaluations for Automatic Prevention of SQL Injection Attacks, ACM Transactions on Information and System Security, 13(2),1-39

[28]. Putthacharoen, Rattipong, Pratheep Bunyantoparat, (2011). "Protecting Cookies from Cross Site Script Attacks Using Dynamic Cookies Rewriting Technique", ICACT 2011, pp 1090-1094.

[29]. Galan E, Alcaide A, Orfila A, Blasco J., (2010). "A Multi-Agent Scanner to Detect Stored-XSS Vulnerabilities", Internet Technology ansd Secured Transactions (ICITST) pp 1-6.

Cite this article as :

Sarjiyus O., El-Yakub M. B. , "Neutralizing SQL Injection Attack on Web Application Using Server Side Code Modification", International Journal of Scientific Research in Computer Science, Engineering and Information Technology (IJSRCSEIT), ISSN : 2456-3307, Volume 5 Issue 3, pp. 158-173, May-June 2019. Available at doi : https://doi.org/10.32628/CSEIT1952339
Journal URL : http://ijsrcseit.com/CSEIT1952339