Research Article

Neutralizing SQL Injection Attack Using Server Side Code Modification in Web Applications

Asish Kumar Dalai and Sanjay Kumar Jena

Department of Computer Science and Engineering, National Institute of Technology Rourkela, Odisha 769 008, India

Correspondence should be addressed to Asish Kumar Dalai; dalai.asish@gmail.com

Received 6 July 2016; Revised 23 September 2016; Accepted 17 October 2016; Published 16 February 2017

Academic Editor: Kim-Kwang R. Choo

Copyright © 2017 Asish Kumar Dalai and Sanjay Kumar Jena. This is an open access article distributed under the Creative Commons Attribution License, which permits unrestricted use, distribution, and reproduction in any medium, provided the original work is properly cited.

Reports on web application security risks show that SQL injection is the top most vulnerability. The journey of static to dynamic web pages leads to the use of database in web applications. Due to the lack of secure coding techniques, SQL injection vulnerability prevails in a large set of web applications. A successful SQL injection attack imposes a serious threat to the database, web application, and the entire web server. In this article, the authors have proposed a novel method for prevention of SQL injection attack. The classification of SQL injection attacks has been done based on the methods used to exploit this vulnerability. The proposed method proves to be efficient in the context of its ability to prevent all types of SQL injection attacks. Some popular SQL injection attack tools and web application security datasets have been used to validate the model. The results obtained are promising with a high accuracy rate for detection of SQL injection attack.

1. Introduction

Today's web applications are built on n -tier architecture, in which, the data management, application processing, and presentation tier are logically separated. Instead of rewriting the entire application, now the developers have to add or modify a specific tier as needed, which helps in ease of design and maintenance. The data management tier consists of a database server, where confidential information relating to the application and the users is stored and retrieved. The data from the database is commonly used for authenticating the user, for storing the record and their relationship, and for displaying the data in a dynamically created web page.

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 to the SQL query processor and get 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. 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 or may be logically incorrect.

The attackers take advantage of such architecture and can provide malicious code in the input parameter. If the proper separation between program instructions and user data has not been done in the code, the malicious input by user/attacker may get executed. By modifying the SQL query, the attacker may extract confidential information from the database and may get full control over the database and the database server. This technique of exploiting the web application is popular among the hackers by the name of "SQL injection attack." The biggest plus point of the attack is that it uses port 80 (default port for HTTP) to communicate, and this port always remains open and neither blocked nor filtered by the firewall. In this paper SQL injection attack and the steps to exploit this attack have been described, and their classification has been done based on the technique that is

used to exploit the attack. The related work in preventing SQL injection attack has been studied, and a novel method has been presented to prevent such attacks. Some popular SQL injection attack tools and web application security datasets have been used to evaluate the performance of the proposed model.

The rest of the paper is structured as follows: Section 2 describes the SQL injection, various attack scenarios, and the classification of SQL injection attacks. In Section 3, the related works for mitigating the SQL injection attack and their pros and cons are discussed. The proposed method to prevent SQL injection attack has been given in Section 4. Evaluation of the model and results has been examined in Section 5 and Section 6, respectively. Finally, concluding remarks are given in Section 7.

2. SQL Injection

SQL injection attacks are a form of injection attack, where the attacker inserts SQL commands in the input parameters, to alter 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 [2]. 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 [3].

An example 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 such 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 StudentId = 165 or 2=2
```

This condition is always true and all the {Name, Department, 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 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. For example, in Microsoft SQL Server the extended stored procedure xp_cmdshell 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.

2.1. Vulnerability Exploitation Steps. To exploit SQL injection vulnerability, the steps that may be followed are reconnaissance, enumeration, data extraction, and command execution. The steps are explained below in detail with appropriate examples. In this discussion, Microsoft SQL Server has been considered as the back-end database.

2.1.1. Reconnaissance. It is the first and the foremost step in exploiting any application. It is a process of fingerprinting the technologies used, which helps the attacker to launch the SQL injection attack successfully. Sometimes, if the database server error messages are returned to the client, it reveals fairly precise information on the technology being used by the web application in the database server. However, to get the complete information about the back-end database server such as the particular version and the patch level, one can use the query “SELECT @@version”; for instance, if the web application displays the verbose error message returned by the database, then the URL like

```
http://www.domainname.com/Admission/
Studnets.asp?Sid=@@version
```

would display

```
Microsoft OLE DB Provider for SQL Server
error '80040e0x' [Microsoft] [ODBC SQL
ServerDriver] [SQL Server] Conversion
failed when converting the varchar
value 'Microsoft SQL Server 2008 -9.
0x.13xx.0x (Intel X86) Nov 15 2008
00:33:37 Copyright (c) 198X-2008
Microsoft-Corporation Express Edition
on Windows NT 5.5 (Build 379X: Service
Pack 2X)' to data type int. /Studnetsx.
aspx, line 213
```

This clearly shows that the victim is using Microsoft SQL Server 2008 as the back-end. It also includes the exact build level and information about the host operating system. Therefore, such techniques can be repeated for other pieces of information, to obtain more accurate fingerprints such as the following:

- (i) @@version: DBMS Version
- (ii) db_name(): Name of the database
- (iii) @@servername: The server name where MS-SQL has been installed
- (iv) @@language: the language name
- (v) @@spid: current user's Process ID

2.1.2. Enumeration. To perform a successful attack and to completely exploit the SQL injection vulnerability, one has to enumerate the tables and their corresponding column names that are present in the database. Some specific pre-defined tables in the database management system contain information about all the system and user defined tables, commonly referred to as metadata. Hence to enumerate the tables/columns of the database server attacker has to access those tables. The queries to extract database name, table, and column names are given below:

- (i) Databases: select name from master..sysdatabases
- (ii) Tables: SELECT name FROM Databasename..sysobjects WHERE xtype='U'
- (iii) Columns: SELECT name FROM Databasename..syscolumns WHERE id = (SELECT id FROM Databasename..sysobjects WHERE name = 'Tablename')

2.1.3. Data Extraction. Once the column names, table names, and the database names are known, the next step is to extract the data that resides in the tables. For extracting the data, we use the "UNION SELECT" statement. In UNION SELECT statement, the number of columns in the injected query must match that of the preexistent SELECT query. To know the exact number of columns present in the existing query, we can use ORDER BY statement such as

```
http://www.domainname.com/Admission/
Studnets.asp?Sid=165+order+by+1
```

```
http://www.domainname.com/Admission/
Studnets.asp?Sid=165+order+by+2
```

```
http://www.domainname.com/Admission/
Studnets.asp?Sid=165+order+by+3
```

We have to repeat these steps until the query executes without any error and the last successfully executed query reveals the number of columns. In another way by gradually increasing the number of columns in the "UNION SELECT" statement until the query executes correctly, we can also determine the number of columns, for example,

```
http://www.domainname.com/Admission/
Studnets.asp?Sid=165+union+select+1--
```

```
http://www.domainname.com/Admission/
Studnets.asp?Sid=165+union+select+1,2--
```

```
http://www.domainname.com/Admission/
Studnets.asp?Sid=165+union+select+
1,2,3--
```

As we know, UNION operator combines two separate SELECT statements and displays the result. Hence, the UNION SELECT statement can be used to retrieve the desired data from the database server.

2.1.4. Command Execution. This step involves executing system commands through the injection vulnerability. To execute system commands, the current user must have high-level privileges. In case of MS-SQL, by using xp_cmdshell we can execute system commands such as

```
exec master..xp_cmdshell 'ipconfig'
```

2.2. Types of SQL Injection Attacks. There are different types of SQL injection attack as presented in many studies [4–9]. These attack types have been named based on the technique implemented to exploit the injection vulnerability as listed.

(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

```
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 programming 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 userid 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 MS-SQL 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_varbintoohexstr(password_hash)+
from+sys.sql_logins+where +name+=+'sa'
```

The function fn_varbintoohexstr() 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 example,


```

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. 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'--

```

3. Related Work

A detailed study of the literature shows that considerable efforts have been made to devise many techniques for preventing SQL injection attacks. One of the current security trends is focused mainly on the security of smart devices primarily working on the Android operating system. Some of the recent works [10–15] show the techniques for preserving security in Android environment. However, security in web applications can not be disregarded as it has a wide existence. In accordance with this, we have studied the existing literature for preventing SQL injection attacks in web applications.

(i) *Static Analysis*. Some approaches rely purely on static analysis of the source code [16–19]. 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 propose a method [20] 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 Runtime Monitoring*. Some approaches like Analysis and Monitoring for Neutralizing SQL Injection Attack (AMNESIA) [21–23] 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 by Prokhorenko et al. [24] provides a novel method for protection against different types of attack in web applications. 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 Prokhorenko et al. [25] 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

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) [26, 27] 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 [28], AppShield [29], and InterDo [30] use the similar methodology. As these approaches are signature-based, they can have insufficient input validation routines and may introduce false positives. As these approaches are human based, much effort is required to determine the data that needs to be filtered and the policy to be applied.

(v) *Instruction Set Randomization.* The SQLrand [31] 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 [22] or dynamically [32, 33]. The effectiveness of detection largely depends on the accuracy of the learning algorithms. The approach in [34] 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.

4. Proposed Method

The query written by the developer is static until it gets input parameters from the user. As the input provided by the

user may not be trusted, our aim 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 severe impact on the database server, starting from extracting the sensitive data from the database to taking complete control over the database server. Hence, the proposed method 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. 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 are extracted.

Step 5. The input parameters from the extracted string are removed sequentially as they were added. For numeric parameters, we remove the numbers and, for alphanumeric parameters, we remove the characters enclosed in single quotes. 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 "The user is trying for SQL Injection!!!"

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 NON-WHERE clauses.

Figure 1 explains the architecture of the proposed model. 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 has been tested with all types of SQL queries, a combination of all parameter types, and queries for INSERT, UPDATE, and DELETE operations. A sample code developed using C#.Net, for making a connection to the database and executing the SELECT query with WHERE construct, is given below.

```
string source = @"Provider=Microsoft.
Jet.OLEDB.4.0;DataSource=
|DataDirectory|sricce.mdb;Jet
OLEDB:Database Password= * * * * *";
```

FIGURE 1: Model for prevention of SQL injection attack.

```

ocn = new OleDbConnection(source);
ocn.Open();
string staticquery = "select password
from login where code = # #";
string substaticquery =
RightOf(staticquery, "where");
string dynamicquery = "select password
from login where code = " + param1 +
"";
if(param1.IsAlphanumeric)
{
dynamicquery = Normalize(dynamicquery);
string subdynamicquery =
RightOf(dynamicquery, "where");
string subdynamicquerypless =
RemoveParameter(subdynamicquery, "str");
substaticquery = substaticquery.Trim();
subdynamicquerypless =
subdynamicquerypless.Trim();
if(substaticquery ==
subdynamicquerypless)
{
ocmd = new OleDbCommand(dynamicquery,
ocn);
odr = ocmd.ExecuteReader();
odr.Read();
}
else
{
Response.Write("The user is trying for
SQL Injection!!! The web page will be
reloaded.");
}
}
ocn.Close();

```

The above-given code only considers the SELECT statement of the SQL query containing a WHERE clause. The SELECT statement can have NONWHERE clauses such as HAVING, LIKE, and ORDER BY. In such cases, the developer has to replace the WHERE with these NONWHERE clauses in the string extraction function. In a similar way, the code is also applicable for queries like UPDATE, INSERT, and DELETE with little modification.

5. Evaluation of the Proposed Model

The proposed method is easy to implement by the web application developers. The method involves few clearly illustrated steps which can be easily implemented irrespective of the platform. All SELECT and DELETE queries which contain a conditional statement can have WHERE/HAVING, LIKE, and ORDER BY clause in it. All UPDATE queries can contain SET clause in it. All INSERT queries may contain VALUES clause in it. Hence, we have used a string extraction function to extract all characters from the query string just after the WHERE, HAVING, LIKE, ORDER BY, SET, and VALUES clause, respectively, till the end of the string. 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.

Further, the proposed method has few more advantages in comparison with existing techniques. Unlike input validation approach [26, 27], no input filtering is done, and the user is free to supply any input character he wants. Further, the method does not involve any white listing or black listing approach, as it is difficult to maintain such list. Unlike SQLrand [31], no brute-force or guessing attack is possible. However, if the query contains both user input parameters

and developers specified parameters, developer specified parameters have to be present at the end. For example,

```
Select * from table name where
id = ? name = ? and city not
in('delhi','chennai','mumbai')
```

The “city not in(‘delhi’,‘chennai’,‘mumbai’)” has to be present at the end of the query.

It has been explained how our proposed model withstands the various types of SQL injection attacks.

(1) *Tautology*. In the case of tautology the attackers insert commands like “or 1 = 1” to make the query true for all conditions. As per the proposed model when the two strings S1 and S2 are compared at Step 6, they fail to match, and the attack will be detected.

```
Actual Query: select * from admin where
user_id= ' ' and password= ' '
```

```
S1: user_id= ' ' and password= ' '
```

```
Attack Query: select * from admin where
user_id= 'abc' and password= 'xyz' or
'a'='a'
```

```
S2: user_id= ' ' and password= ' ' or
'a'='a'
```

(2) *Commenting the Code*. In the case of Commenting the Code the attacker inserts some character sequence which converts the rest of the statement into comment line. As per the proposed model when the two strings S1 and S2 are compared at Step 6, they fail to match, and the attack will be detected.

```
Actual Query: select * from admin where
user_id= ' ' and password= ' '
```

```
S1: user_id= ' ' and password= ' '
```

```
Attack Query: select * from admin where
user_id= 'abc'; -- and password= 'xyz'
```

```
S2: user_id= ' '; -- and password= ' '
```

(3) *Type Mismatch*. In the case of type mismatch the attacker inserts the input with the different data type, which creates a type mismatch error and responds with a verbose error message containing sensitive data. As per the proposed model at Step 2 the input parameters are checked for their appropriate type. If the input type mismatch occurs, it blocks the code from further execution.

```
Actual Query: select * from
student_details where sid=
```

```
S1: sid=
```

```
Attack Query: select * from
student_details where sid=system_user
```

Here the intended parameter is of type integer but the attacker has given a string. The input type checking phase at Step 2 will detect the attack

(4) *Stacked Query*. By using stacked query, the attacker appends malicious SQL command at the end of the actual query. As per the proposed model when the two strings S1 and S2 are compared at Step 6, they fail to match, and the attack will be detected.

```
Actual Query: select * from
product_details where product_id= ' '
```

```
S1: product_id= ' '
```

```
Attack Query: select * from
product_details where product_id= 'P1';
drop table acnts--
```

```
S2: product_id= ' '; drop table accounts
--
```

(5) *Union Query*. By using a union, the attacker combines the results of two SQL statements and displays the output. As per the proposed model when the two strings S1 and S2 are compared at Step 6, they fail to match, and the attack will be detected.

```
Actual Query: select * from
product_details where product_id= ' '
```

```
S1: product_id= ' '
```

```
Attack Query: select * from
product_details where product_id= 'P1'
union select uid, pwd form login--
```

```
S2: product_id= ' ' union select uid, pwd
form login --
```

(6) *Stored Procedure and System Functions*. The attacker uses the stored procedures to impact the SQL injection attack severely. As per the proposed model when the two strings S1 and S2 are compared at Step 6, they fail to match, and the attack will be detected.

```
Actual Query: select * from
student_details where roll_no= ' '
```

```
S1: roll_no= ' '
```

```
Attack Query: select * from
student_details where roll_no=
'S123' +union +select+ master.dbo.
fn_varbintohexstr(password_hash)
+from+sys.sql_logins+where +name+=+'sa'
```

```
S2: roll_no=' ' +union +select+master.dbo.
fn_varbintohexstr(password_hash)+from+
sys.sql_logins+where +name+=+' sa '
```

(7) *Inference*. In the case of inference the attackers add some commands to infer the information from the resultant output. As per the proposed model when the two strings S1 and S2 are compared at Step 6, they fail to match, and the attack will be detected.

Actual Query: select * from
product_details where product_id= ' '
S1: product_id= '
Attack Query: select * from
product_details where product_id= 'P1'
and SUBSTRING(user name(),1,1)='c'--
S2: product_id= ' and SUBSTRING(user
name(),1,1)= ' c' --

(8) *Alternative Methods.* To evade input filters, attackers may use some encoding technique. As per the proposed model at Step 3, the query is normalized and converted into plain characters which can detect the SQL injection attack if any.

Actual Query: select * from admin where
user_id= ' ' and password= ' '
S1: user_id= ' ' and password= '
Attack Query: select * from
admin where user_id= 'abc'
UNION%20SELECT%20%2a%20FROM%20LOGIN
%20WHERE%20USERID%20%3E%2010

Here the attacker has used URL encoding technique to evade the web application firewall. But the use of string decoding at Step 3 will detect the attack.

6. Results and Discussion

The method has been implemented in the web application named *srice*, a web application developed for managing the ongoing projects in the Institute. The website (<http://Server-IP/srice/login.aspx>) is hosted in the Internet Information Server (IIS) server. The web application is developed in ASP.NET with MS-SQL as the back-end database. The site contains pages such as an authentication page, a change password page, page containing retrieval of the various project related data, search pages, and pages for inserting new records and updating the existing record. To test the trustworthiness of the model, SQL injection attacking tools such as SQLInjectMe, NTO SQL Invader, Scrawl, SQLPowerInjector, and SQLSentinel have been used. It has been found that the web application can withstand these attacking tools. The snapshots showing the output of these tools are given in Figures 2–6.

The performance of the proposed model has been tested with attack vectors collected through web resources and HTTP dataset CSIC 2010 [35]. TECAPI Attack Vectors [36] contain the resources for SQL injection attacks of different variants including Direct: SQL injection attack, Persistent: second-order SQL injection attack, and Session variants: SQL injection via session puzzling. We have collected all these types of attack vectors from the corresponding links to generate a set of attack vectors for validating the proposed model.

The HTTP dataset CSIC 2010 [35] contains thousands of automatically generated web requests. The author of the dataset has provided it for testing of web attack protection

FIGURE 2: Test result of SQLInjectMe.

FIGURE 3: Test result of NTO SQL Invader.

FIGURE 4: Test result of Scrawl.

FIGURE 5: Test result of SQLPowerInjector.

FIGURE 6: Test result of SQLSentinel.

FIGURE 7: Test result of manual attack vectors.

systems. It was developed at the “Information Security Institute” of CSIC (Spanish Research National Council). The dataset is generated automatically and contains 36,000 normal requests and more than 25,000 anomalous requests. The HTTP requests are labeled as normal or anomalous, and the dataset includes attacks such as SQL injection, buffer overflow, information gathering, files disclosure, CRLF injection, XSS, and parameter tampering. From the dataset, the SQL injection attack vectors have been collected for evaluating the proposed model.

The model is also verified by manually injecting the attack vectors. The snapshot displaying the output of manual attack vector is shown in Figure 7. It is clear from the figure that the proposed model also withstands the manual attack vectors.

6.1. Performance Measures Used. False Acceptance Rate (FAR), Genuine Acceptance Rate (GAR), False Rejection Rate (FRR), Receiver Operating Characteristics (ROC) curve, and Area Under ROC curve (AUC) have been used as the performance measures to evaluate the efficiency of the proposed model.

- (i) *False Acceptance Rate (FAR)*. FAR is the frequency of attack vectors able to bypass the attack detection mechanism. This statistic is used to measure the performance of the proposed approach when operating in the attack detection mode. A false acceptance occurs when the protection mechanism at the application server is unable to stop the malicious web request and the query having SQL injection code is sent to the database server for execution.
- (ii) *Genuine Acceptance Rate (GAR)*. GAR is the frequency of acceptance relative to the authentic web requests which are sent for execution. These statistics are used to measure the performance of the proposed

TABLE 1: Comparative analysis of the proposed method on different databases.

Databases	AUC (%)	FAR (%)	GAR (%)
TECAPI Attack Vectors	91.84	13.51	86.54
HTTP dataset CSIC 2010	98.05	11.44	89.61

FIGURE 8: Performance analysis of the proposed method for SQL injection attack detection in TECAPI Attack Vectors and HTTP Dataset CSIC 2010.

approach when operating in the attack verification mode. A genuine acceptance occurs when an authentic web request is classified as a normal (nonattack) pattern.

- (iii) *False Rejection Rate (FRR)*. FRR is the frequency of rejections relative to the genuine web requests which should be sent for execution. These statistics are used to measure the performance of the proposed approach when operating in the verification mode. A false rejection occurs when an authentic web request is classified as a malicious one.
- (iv) *Receiver Operating Characteristic (ROC)*. ROC curve depicts the dependence of GAR (Genuine Acceptance Rate) with FAR for change in the value of the threshold. The curve is plotted using linear, logarithmic, or semilogarithmic scales.
- (v) *Area Under ROC Curve (AUC)*. AUC is the percentage of coverage under the ROC curve. The more the coverage, the more the accuracy of the system. In ideal case, for a system with 100% accuracy, GAR = 1 at FRR = 0, causing AUC = 100%.
- (vi) *Equal Error Rate (EER)*. The EER refers to the point in a ROC curve, where the FAR equals the FRR. Thus a lower EER value indicates better performance.

The proposed method provides satisfactory results as presented in Table 1 and Figure 8. The proposed method is further compared with the existing techniques regarding

TABLE 2: Comparison of the proposed method with existing approaches.

Methods	Tautology	Comment line	Type mismatch	Stacked query	Union query	Stored procedure	Inference	Alternative methods
AMNESIA	Y	Y	Y	Y	Y	N	Y	Y
SQLrand	Y	P	P	Y	Y	N	Y	N
SPDL	P	Y	P	P	P	P	P	N
SIIMDS	N	P	P	P	N	P	P	N
SQLIPA	Y	Y	Y	Y	Y	Y	Y	N
Proposed Method	Y	Y	Y	Y	Y	Y	Y	Y

Y: successfully detect/prevent the attack types.

N: not able to detect/prevent the attack types.

P: partially handles the attack type.

their defending ability against the various SQL injection attack types. The result shows that the proposed model is more efficient than its counterparts. Table 2 summarizes the result of comparisons with existing methods. It clearly shows that the proposed method can withstand all types of SQL injection attacks.

7. Conclusion

The proposed model is a novel online detection method against SQL injection attack. It depends on sequentially extracting the intended user input from the dynamic query string to check for any malicious input. Unlike other approaches, the proposed method is quite simple to implement yet highly effective. The method has been implemented in the test web application to demonstrate its effectiveness. The model can also be incorporated into existing application of different environment. To measure the performance of the model we have manually tested the application with all attack vectors. Some of the popular SQL injection attacking tools have also been used to check for the effectiveness of the model. The experimental result shows that the proposed model can defend all kinds of SQL injection attack. The model is ideal for web application developers to prevent the threat of SQL injection attack in the web application. The proposed approach can be further extended to deal with another type of injection attack such as Command Injection, Code Injection, and File Injection.

Competing Interests

The authors declare that there is no conflict of interests regarding the publication of this paper.

References

- [1] Foundation TO: SQL Injection, [https://www.owasp.org/index.php/SQL Injection](https://www.owasp.org/index.php/SQL%20Injection).
- [2] V. Prokhorenko, K.-K. R. Choo, and H. Ashman, "Web application protection techniques: a taxonomy," *Journal of Network and Computer Applications*, vol. 60, pp. 95–112, 2016.
- [3] B. D. A. Guimarães, Advanced SQL injection to operating system full control, Black Hat Europe, white paper, 2009.
- [4] A. S. Yeole and B. B. Meshram, "Analysis of different technique for detection of SQL injection," in *Proceedings of the International Conference and Workshop on Emerging Trends in Technology (ICWET '11)*, pp. 963–966, ACM, Mumbai, India, February 2011.
- [5] G. Buehrer, B. W. Weide, and P. A. G. Sivilotti, "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, September 2005.
- [6] E. Al-Khashab, F. S. Al-Anzi, and A. A. Salman, "PSIAQOP: preventing SQL injection attacks based on query optimization process," in *Proceedings of the 2nd Kuwait Conference on e-Services and e-Systems (KCESS '11)*, pp. 10–18, ACM, Kuwait City, Kuwait, 2011.
- [7] A. Liu, Y. Yuan, D. Wijesekera, and A. Stavrou, "SQLProb: a proxy-based architecture towards preventing SQL injection attacks," in *Proceedings of the Annual ACM Symposium on Applied Computing (SAC '09)*, pp. 2054–2061, New York, NY, USA, March 2009.
- [8] S. W. Boyd, G. S. Kc, M. E. Locasto, A. D. Keromytis, and V. Prevelakis, "On the general applicability of instruction-set randomization," *IEEE Transactions on Dependable and Secure Computing*, vol. 7, no. 3, pp. 255–270, 2010.
- [9] K. Elshazly, Y. Fouad, M. Saleh, and A. Sewisy, "A survey of SQL injection attack detection and prevention," *Journal of Computer and Communications*, vol. 2, no. 8, pp. 1–9, 2014.
- [10] A. Azfar, K.-K. R. Choo, and L. Liu, "A study of ten popular Android mobile VoIP applications: are the communications encrypted?" in *Proceedings of the 47th Hawaii International Conference on System Sciences (HICSS '14)*, pp. 4858–4867, IEEE, Waikoloa, Hawaii, USA, January 2014.
- [11] A. Azfar, K. K. R. Choo, and L. Liu, "Forensic taxonomy of popular Android mHealth apps," in *Proceedings of the 21st Americas Conference on Information Systems (AMCIS '15)*, San Juan, Puerto Rico, August 2015.
- [12] A. Azfar, K. K. R. Choo, and L. Liu, "An android communication app forensic taxonomy," *Journal of Forensic Sciences*, vol. 61, no. 5, pp. 1337–1350, 2016.
- [13] A. Azfar, K. R. Choo, and L. Liu, "Forensic taxonomy of android productivity apps," *Multimedia Tools and Applications*, pp. 1–29, 2019.
- [14] A. Azfar, K.-K. R. Choo, and L. Liu, "Android mobile VoIP apps: a survey and examination of their security and privacy," *Electronic Commerce Research*, vol. 16, no. 1, pp. 73–111, 2016.

- [15] A. Azfar, K. K. R. Choo, and L. Liu, "An android social app forensics adversary model," in *Proceedings of the 49th Hawaii International Conference on System Sciences (HICSS '16)*, pp. 5597–5606, IEEE, Koloa, Hawaii, USA, January 2016.
- [16] Y. Xie and A. Aiken, "Static detection of security vulnerabilities in scripting languages," in *Proceedings of the 15th Conference on USENIX Security Symposium*, pp. 179–192, Vancouver, Canada, 2006.
- [17] V. Livshits and M. Lam, "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, 2005.
- [18] M. S. Lam, J. Whaley, V. Benjamin Livshits et al., "Context sensitive program analysis as database queries," in *Proceedings of the 24th ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems (PODS '05)*, pp. 1–12, New York, NY, USA, June 2005.
- [19] C. Gould, Z. Su, and P. Devanbu, "JDBC checker: a static analysis tool for SQL/JDBC applications," in *Proceedings of the 26th International Conference on Software Engineering (ICSE '04)*, pp. 697–698, IEEE Computer Society, Edinburgh, UK, May 2004.
- [20] G. Wassermann and Z. Su, "An analysis framework for security in Web applications," in *Proceedings of the FSE Workshop on Specification and Verification of Component-Based Systems (SAVCS '04)*, pp. 70–78, Citeseer, 2004.
- [21] W. G. J. Halfond and A. Orso, "Preventing SQL injection attacks using AMNESIA," in *Proceedings of the 28th International Conference on Software Engineering (ICSE '06)*, pp. 795–798, Shanghai, China, May 2006.
- [22] W. G. J. Halfond and A. Orso, "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, 2005.
- [23] W. G. J. Halfond and A. Orso, "Combining static analysis and runtime monitoring to counter SQL-injection attacks," *ACM SIGSOFT Software Engineering Notes*, vol. 30, no. 4, pp. 1–7, 2005.
- [24] V. Prokhorenko, K. R. Choo, and H. Ashman, "Context-oriented web application protection model," *Applied Mathematics and Computation*, vol. 285, pp. 59–78, 2016.
- [25] V. Prokhorenko, K. R. Choo, and H. Ashman, "Intent-based extensible real-time PHP supervision framework," *IEEE Transactions on Information Forensics and Security*, vol. 11, no. 10, pp. 2215–2226, 2016.
- [26] D. Scott and R. Sharp, "Abstracting application-level web security," in *Proceedings of the 11th International Conference on World Wide Web (WWW '02)*, pp. 396–407, ACM, May 2002.
- [27] D. Scott and R. Sharp, "Specifying and enforcing application-level web security policies," *IEEE Transactions on Knowledge and Data Engineering*, vol. 15, no. 4, pp. 771–783, 2003.
- [28] C. Brabrand, A. Møller, R. M. Christensen, and M. I. Schwartzbach, "PowerForms: declarative client-side form field validation," *World Wide Web Journal*, vol. 7, no. 43, pp. 205–314, 2000.
- [29] Sanctum Inc, AppShield 4.0 Whitepaper 2002, <http://www.sanctuminc.com>.
- [30] I. Kavado, *InterDo Version 3.0*, 2003, <http://www.protegrity.com/data-security-platform>.
- [31] S. Boyd and A. Keromytis, "SQLrand: preventing SQL injection attacks," in *Applied Cryptography and Network Security*, pp. 292–302, Springer, Berlin, Germany, 2004.
- [32] S. Lee, W. Low, and P. Wong, "Learning fingerprints for a database intrusion detection system," in *Computer Security—ESORICS 2002*, pp. 264–279, Springer, 2002.
- [33] F. Valeur, D. Mutz, and G. Vigna, "A learning-based approach to the detection of SQL attacks," in *Proceedings of the 2nd International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment (DIMVA '05)*, pp. 123–140, Springer, Vienna, Austria, July 2005.
- [34] A. Asmawi, Z. M. Sidek, and S. A. Razak, "System architecture for SQL injection and insider misuse detection system for DBMS," in *Proceedings of the International Symposium on Information Technology (ITSim '08)*, Kuala Lumpur, Malaysia, August 2008.
- [35] C. T. Giménez, A. P. Villegas, and G. Á. Marañón, HTTP DATASET CSIC, 2010.
- [36] "TECAPI I: List of Attack Vectors 2015," <http://www.tecapi.com/public/relative-vulnerability-rating-gui.jsp>.