A Proposed Framework Against Code Injection Vulnerabilities in Online Applications

Teresa K. George, K. Poulose Jacob, Rekha K. James

Department of Computer Science, Cochin University of Science and Technology, India
susanteresa12@gmail.com, kpj0101@gmail.com, rekhajames@cusat.ac.in

Abstract

Security vulnerabilities are frequently detected and exploited in modern web applications. Intruders obtain unrestricted access to the information stored at the back-end database server of a web application by exploiting security vulnerabilities. Code injection attacks top the list due to lack of effective strategies for detecting and blocking injection attacks. The proposed Token based Detection and Neural Network based Reconstruction (TbD-NNbR) framework is a unique approach to detect and block code injections with negligible processing overheads. This framework makes use of an efficient token mapping and validation technique to match the statically generated legal query tokens against the parsed dynamic query tokens at run time. The proposed approach also has the provision to reconstruct queries from authenticated users. The prototype implementation of TbD-NNbR shows that it does not demand any source code modifications and incurs only a negligible computational overhead without any incidents of false positives or false negatives.

Keywords: Code injection attack, Neural network, Query validation, Reconstruction, Security vulnerability

1 Introduction

Security vulnerabilities are becoming a severe issue in web applications as successful attacks lead to loss of integrity, confidentiality and make it a very sensitive subject in software security. Code Injection through a dynamic web page is one of the most dangerous threats that exploit the application layer vulnerabilities [1]. The existing techniques or strategies may not be enough to handle many of the vulnerabilities due to the unknown and often obscure nature of vulnerability issues. Existing input validation techniques still require more sophistication. The attack on a given database violates the Confidentiality, Integrity, Availability (CIA) triangle of security. Most of the SQL injection attack prevention approaches result in false positives, which will decrease the system availability for the authenticated users [2].

In SQL injection attacks, an attacker attempts to change the syntax and semantics of legitimate SQL statements by inserting unintended keywords, symbols or malicious codes on the SQL statements accepted through dynamic web pages. By exploiting this vulnerability, an attacker can directly interact with the database server and gain access to the critical data and thus compromise security. This paper proposes a Token based Detection and Neural Network based Reconstruction (TbD-NNbR) framework against code injection vulnerabilities. The proposed framework blocks all malicious entries and only the benign query can access the data from the back-end database server. The TbD-NNbR framework also has the provision to reconstruct the queries from authenticated users at run time, using the neural network, which increases the system availability and mitigates the denial of service attack [3].

The rest of the paper is organized as follows: Section 2 deals with SQL injection attack categories. Section 3 handles the related works. The proposed Token based Detection and Neural Network based Reconstruction (TbD-NNbR) framework are explained in Section 4. The prototype implementation of TbD-NNbR is described in Section 5. Section 6 discusses the evaluation of the proposed model and a conclusion is given in Section 7.

2 SQL Injection Attack Categories

SQL injection attack is one of the most dangerous types of vulnerability attacks adopted by web hackers to compromise the security features of a critical application. In most of the vulnerability analysis, Tautology, Union queries, Piggybacked queries, Logically-incorrect queries, Stored procedures, Inferences and Alternate encoding are the classifications of SQL injection attacks. A detailed description of SQL injection attacks along with examples are as follows [2, 4-5].

*Corresponding Author: Teresa K. George; E-mail: susanteresa12@gmail.com

DOI: 10.3966/160792642019012001008

2.1 Tautologies

In this attack, the hacker injects code into a conditional statement to evaluate it as true there by allowing the malicious user to bypass the user authentication or extract data from a database. For example, suppose that a malicious user inputs the SQL statement as SELECT * FROM books WHERE ID= ‘1’ or ‘1’= ‘1’--AND password= ‘pass’;

2.2 Logically Incorrect Query/Illegal Queries

This type of attack is used to gather information about the back-end database of a web application through error messages of type mismatch or logical error, while the query gets rejected. For example, the injected query on the given URL can be in the format: http://www.elearning/mct/? id_user= ‘123’.

2.3 Union Query

These types of queries trick the database into returning the results from database tables which are different from what was intended. For example, an injected query can be of the format: SELECT * FROM users WHERE userid=22 UNION SELECT item, results FROM reports.

2.4 Piggy Backed Queries

The attacker tries to inject additional queries along with the original queries, which are said to ‘piggyback’ onto the original query. Hence the database gets multiple queries for execution. For example, SELECT Login ID FROM users ID WHERE login ID= ‘john’ and password=’’; DROP TABLE users- AND ID=2345.

2.5 Alternate Encoding

These types of attacks use the char () function and ASCII /Hexadecimal encoding. For example: SELECT accounts FROM users WHERE login=”” AND pin=0; exec (char (0x73687574646j776e)).

2.6 Stored Procedure

In these attacks, hackers aim to perform privilege escalation, denial of service and remote command execution using stored procedures through the user interface to back-end servers. For example, UPDATE users SET password=‘Nicky’ WHERE id= ‘2’ UNION SHUTDOWN;--.

2.7 Inference Attack

Here, the hackers aim to identify the injectable parameters and extract data from databases. Blind SQL injection attack and Timing SQL injection attack are the two categories of Inference attacks. For example the injected queries in the Timing SQL injection category can be in the following format: SELECT name, password FROM user WHERE id = 12; IF (LEN (SELECT TOP 1 column_name from test DB.information_schema. columns where table_name= ‘user’),4) WAITFOR DELAY ‘00:00:10’.

3 Related Works

SQL Injection has been an issue for many years, and several tools and strategies are developed to tackle this situation [6-7]. Researchers Calvi and Vigan, in 2016 proposed some automated approaches of testing the security of the web application against the constant attack [8]. Researchers Johari and Pankaj, in 2012 conducted a detailed study on Injection vulnerabilities, and strategies for countermeasures [1]. As analyzed and documented by Skrupsky and Bisht et al. in 2013, the website security is improved by implementing appropriate web vulnerability scanners, and penetration testers [9]. During security testing phase, documented by Lebeau and Franck et al. in 2013, the vulnerability assessment should be carried out using appropriate strategies, mitigation process and with an in-depth understanding of the organization’s infrastructure and critical processes [10]. Martin Burkhart and Dominik deal with asymmetry in the attack space injected traffic pattern and the role of network data anonymization [11]. One of the recent research works carried out by Deepa and Santhi has proposed black box detection strategies for code injection attack and parameter tampering vulnerabilities on XML database.

Most of the model based countermeasures suggested by previous researchers have indicated some very useful strategies in the field of security and protection. Some of the very efficient model-based approaches are as follows: SQLrand suggests randomized SQL query language to identify and abort queries which contain injected code [12]. In SQLCHECK, a runtime checking algorithm evaluates a real-world web application with a real dynamic attack data as input. It prevents SQL injection attacks without any false positives and false negatives [13]. AMNESIA uses a combination of static and dynamic analysis to analyze web application codes and to monitor dynamically generated queries. In the next step, the building of a query model with all possible queries are identified by the hotspot. The runtime monitoring methodology will reject or report the queries that violate the model [13-14].

4 Proposed TbD-NNbR Frame Work

Token based Detection and Neural Network based Reconstruction (TbD-NNbR) is a hybrid model which consists of a Token based Detection (TbD) module and Neural Network based Reconstruction (NNbR) module. Figure 1 shows the proposed TbD-NNbR framework.

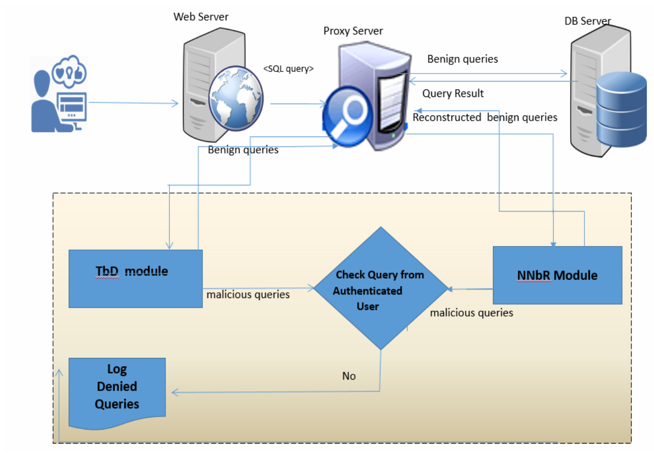


Figure 1. Proposed TbD-NNbR framework

Figure 1, shown above is a combined representation of the TbD module and NNbR module. The Web Server, accepts user requests through dynamic web pages and directs the queries to the proxy server that lies in between the web server and the database server. The detailed evaluation procedure is shown under corresponding module description. Reconstruction of queries are performed only within the NNbR module and only for the authenticated user queries. Reconstruction procedure is not considered in the TbD module but, has a strong token based detection and malicious query blocking strategy. The malicious queries from unauthenticated users are usually logged for future work of Identification of new injection pattern.

4.1 Token Based Detection (TbD)

One of the primary purposes of this module is to detect and block SQL-Injection attack. Only the benign query can access the data from the back-end database server. The proxy server executes the user query and validates it before redirecting it to the database server. The server blocks malicious queries and generates an alert message if the injection is detected. This method uses an efficient query validation technique to match the statically generated legal query tokens against the parsed dynamic query tokens at runtime [15]. A template repository stores the legal query tokens with a unique identity and a format. Figure 2 presents the main elements of the Token based Detection (TbD) module.

4.1.1 Standard Query Template Creator (SQTC)

The standard query template creator module statically analyzes the web application, identifies and collects the standard legal queries in the web pages. The template creator module makes use of the Standard Query Template Creator (SQTC) algorithm to parse the queries. Table 1 presents SQTC algorithm. In most of the web applications, there are multiple SQL requests in each web page as the hotspot for malicious

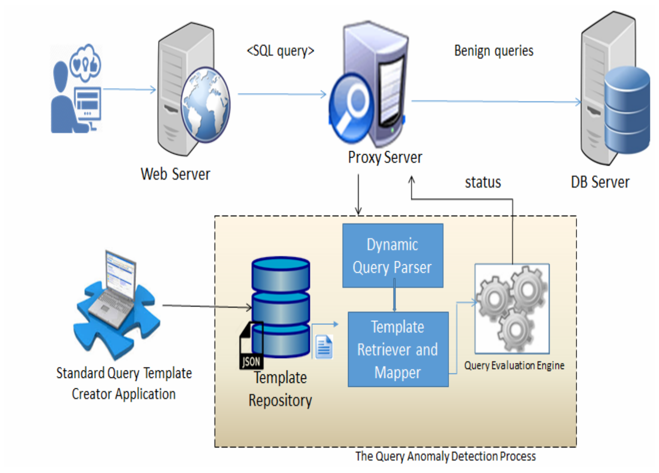


Figure 2. Token based Detection (TbD)

Table 1. Standard Query Template Creator Algorithm

```
Standard Query Template Creator Algorithm (SQTC)
Input: web application URL, user credentials
Begin
Procedure SQTC (Sq, Tk)
Begin
Sq←Standard query
Tk ←Tokens generated from Sq
Tk [i] ← {query-type, used-tables, columns, system-
variables, global-variables, functions,
              joins, special-symbols, operators,
              comment-symbols and keywords}
 Begin
   Sq-Id← get (Sq-Id) from JSON parser for each query
   Do
     get new Sq-Id
                //till all the pages are checked and queries
tokenized with a unique Sq-id
                  for each Query
              Tl (Sq-Id)← Template for each Sq-Id   // created
by the parser
          Return
            While   All ' Sq-Id' is  generated    // End of web
pages
     End Do
  End.
 End
```

user entries, which are directly accessing the database server. The module splits the complex queries into several independent queries by using a depth-first tree traversal procedure, and tokenizes each independent query. There can be multiple queries in each web page. During the static analysis, all the queries in the web application are identified for malicious entries and undergo the similar procedure to tokenize the query and further validation with a dynamic query at run time. For each input query accessing the backend database, it generates a unique ID [16].

In the algorithm given above, the web crawler identifies the input entry (the form entry field) by checking the user credentials entered and the URL or the specified path to the given web application. For each assigned form field, there is a corresponding SQL query. While splitting/parsing the query, the algorithm generates tokens as per the predefined template specification for each standard SQL statement. The tokens parsed are grouped under any one of the following attribute specification namely query-type, used-tables, columns, system-variables, global variables, functions, joins, special-symbols, operators, comment symbols and keywords. The attributes of the identified queries are grouped and assigned a Tl (sq-Id), which is unique for each query and stores the corresponding template details in the JSON format. During parsing, if there is any extra field in the queries other than the token-specification mentioned above, then each additional field identified in the query is expanded as an added column in the template specification to accommodate the fields [16]. There can be "n" added columns created based on the input query type.

4.1.2 Template Repository

The template repository holds all the possible legal query identity (ID), and template (TI) of the underlying online application stored in JSON format. It arranges the corresponding ID for each page and stores in the repository by using a particular jar/package file facility available in the application.

4.1.3 Dynamic Query Parser

During a dynamic interaction, the user queries accepted through dynamic web pages must undergo the tokenizing/parsing procedure by the parser to have the token mapping against the dynamic query.

4.1.4 Template Retriever and Mapper

The template retriever retrieves the corresponding standard tokens from the template repository. To check the validity, the template mapper maps the appropriate standard query from the template repository against the dynamic query with the support of a proposed SQL-Injection detection algorithm, which is shown in Table 2.

The algorithm mentioned above validates the tokens of user input query with the token of the legal query placed in the template repository. While validating tokens of both queries, each character, pattern and length of the user input query is compared with the tokens of the legal query stored in JSON format from the repository. The algorithm generates tokens as per the predefined template specification for each standard SQL statement while performing splitting/parsing the query. If each token of the input query and the legal query exactly matches (TL(sq) ⊕ TL(Dq) == 0), then there is no injection in the query. Otherwise (TL(Sq) ⊕ TL(Dq) ≠ 0) there is injection detected, where Sq is the standard/Legal query and Dq is the Dynamic /user-Input query. Figure 3 illustrates the key steps of the detection procedure with flow directions.

Table 2. SQL Token mapper algorithm

```
SQL Token mapper Algorithm
Input: Legal query model, user input query, user credentials
Output: Detection result
procedure for SQL-Tokenmapper (Sq-Id, Dq-Id)
Begin
DetectionResult [sq-Id, Dq-Id] ← Boolean getMatch (List1, List2 )
boolean getMatch (List1, List2) //To mapp Sq-Id with Dq-Id//
  Do while  all the tokens are mapped
    Retrieve  corresponding Sq-id & Dq-Id from Template Repository for maping
    List1 ←   All tokens from DynamicQuery:Dq
    List2←  All tokens from, StandardQuery:Sq
    If (LengthOfList1== LengthOfList2) Then
        For i= 1 to n
          Check
           If (List1 [i]!=List2 [i])
              Boolean getmatch ( )← False;//not matching//
           Else
              Boolen  getmatch ( )← True//Exact match//
          End If
        Next i
     End If
    If  (TL (Sq) ⊕ TL (Dq) == 0 ),
        there is an exact match
            set message as 'No injection detected  in the token'
    else
      exact match  not found
        set message as 'Injection detected in the token'
   endif
 End Do
End
```

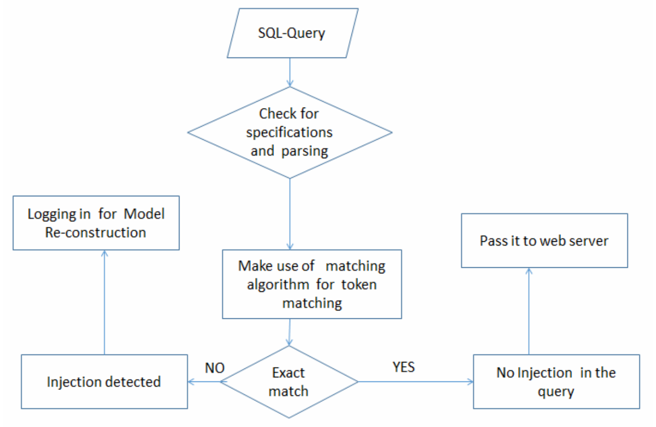

SQL-Query
Check for specifications and parsing
Logging in for Model Re-construction
Make use of matching algorithm for token matching
Pass it to web server
Injection detected
NO
Exact match
YES
No Injection in the query

Figure 3. Detection procedure

4.1.5 Query Evaluation Engine

The Query evaluation engine keeps track of all the detection results by the mapper and redirects the corresponding alert message (detection result) to the server. If there is an exact match found between the standard and the dynamic query, then the alert message is displayed as 'Injection not detected' otherwise 'Injection detected' message is displayed. The benign queries can move further to access the database server.

4.2 Neural Network Based Reconstruction (NNbR)

The Neural Network based Reconstruction (NNbR) provides better availability of a web application and reduces the denial of service attack by facilitating the reconstruction option for the authenticated user query. In this module, SQL queries are trained using an Artificial Neural Network (ANN) and a trained model is stored in the template repository. The main components of Neural Network based Reconstruction (NNbR) module is shown in Figure 4.

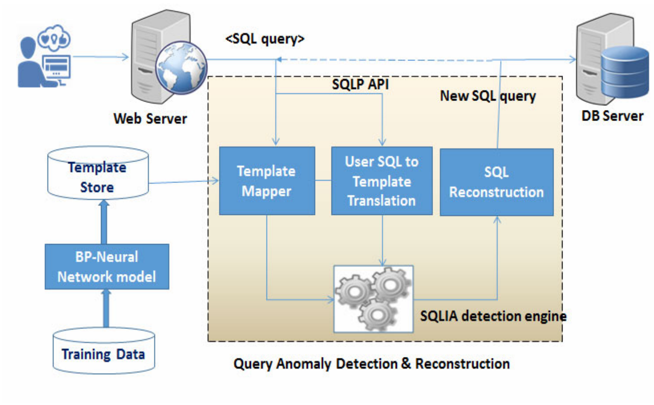


Figure 4. Neural Network based Reconstruction (NNbR)

The proposed model learns the ideal query model for the seven identified attack categories, using the machine learning technique, for further process and reconstruction. This framework facilitates reconstruction of queries from authenticated users, irrespective of the underlying database. As a pre-requisite of reconstruction procedure, it validates each query with authentication credentials of the user. If the query derives from an authenticated user, then the query is labelled as **reconstruction required**. The model redirects such queries with **reconstruction required** labels to the reconstruction procedure. The remaining queries with invalid authentication details can be logged in for model implementation and pattern matching procedure in the training stage.

4.2.1 Multilayer Artificial Neural Network (ANN) for Machine Learning

The multilayer artificial neural networks has three layers such as input layer, hidden layer, and output layer [3]. It has a set of synaptic weights, propagation function (Σ) and an activation function (φ) which takes the output of the propagation function. During the processing stage, each input is multiplied by their respective weighing factor (w (n)) and then the modified inputs are fed into the propagation function [17]. This function can produce some different values which are forwarded further and sent into a transfer function which will turn it into a real output value using the selected procedure. The transfer function also can scale up the output or control its value. The propagation function (Σ) includes sum, max, min, OR, AND, etc. The activation function (φ) is a Hyperbolic tangent, Linear, Sigmoid, etc. Figure 5 illustrates the multilayer representation of the neural network.

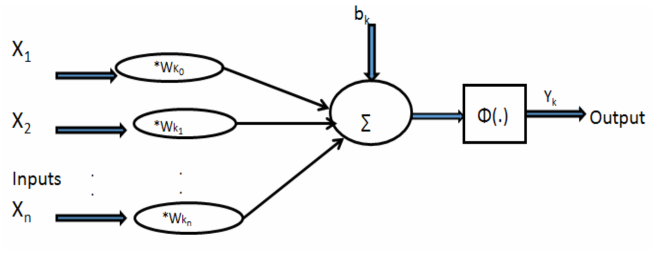


Figure 5. Multilayer representation of neural network

The multilayer neural network differentiates a feed-forward network from the feed-backward based on the architecture [4].

4.2.2 Back Propagated-Neural Network model

The Back Propagated-Neural Network (BP-NN) model trains the queries efficiently and gets the ideal model for further procedure specified by the authenticated user. The trained model can be used to perform various tasks such as pattern recognition and pattern association with the support of 'Back Propagation' algorithm [18]. Figure 6 shows the representation of BP-NN learning for SQL trained model.

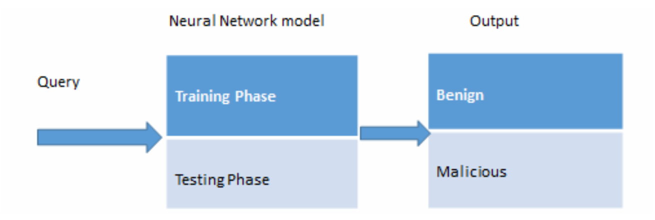


Figure 6. BP-NN learning for SQL trained model

The input layer accepts legal and injected SQL queries. From the collected queries, the training phase uses 80%, and the testing phase uses 20% of the queries at the Neural Network. The back-propagation algorithm classifies the output from the NN model as either 'benign' or 'malicious' query [4, 18].

4.2.3 Training Data

The NNbR module collects the SQL queries from online applications and trains these queries by Back Propagated-Neural Network (BP-NN). One of the best learning algorithms is Back Propagation Algorithm (BPA) [3-4].

4.2.4 Template Store

The template store has BP-NN trained ideal model, which are available for the API for further mapping and reconstruction of queries after being identified as 'queries from authenticated user'.

4.2.5 Template Mapper

The Template mapper component retrieves the legal query from template store and maps against the dynamic input query. In the proposed approach, we create a model file from the training set and validate every user query against this model. The SQLIAShield, a jar file, facilitates easy accessing of trained queries from template store to perform the pattern matching using the regular expressions with the dynamic query. The model template for each query from the repository is identified accurately to perform matching procedure [4, 19].

4.2.6 SQLIA Detection Engine

SQL Injection Attack (SQLIA) detection engine invokes the matching process against the trained data model from the template repository [20]. The detection engine generates the status report and passes it to SQL Reconstruction component for further process.

4.2.7 SQL Reconstruction

SQL Reconstructor component reconstructs dynamic queries from the authenticated user. Table 3 shows the reconstruction algorithm (R-Iq). Reconstruction process takes care of the task of re-establishing injected portions of the actual query by either eliminating it or substituting it with a null value. High accuracy of detection is obtained using the proposed method, without much loss of efficiency. [17-18]. We use regular expressions to search for complex patterns. REGEXP handles meta-characters and literals separately during the search function [21]. The meta-characters such as: +, ?, *, {n}, \, ^, \n, etc. identified are used for searching the pattern. Some of the basic string matching functions with SQL Regular expression is: REGEXP_LIKE, REWGEXP_ REPLACE, REGEXP_ INSTR and REGEXP_SUBSTR.

In the reconstruction algorithm in Table 3, if it detects injection (extra character/string) in the input query and if an authenticated user raises the query, then it is diverted for reconstruction. If the algorithm detects malicious user queries at first level (Case I): then it assigns each valid token to a regular expression. Tokens of injected query are matched with valid tokens of the model for detection of injected string and assigned a null value or remove the content of the injected (additional strings/characters) portion. Then the token is considered as the reconstructed one and is compared with the model token to recheck and prove that the token of injected query and token from the model query are equal and there are no more injected or new fields with the input query. The algorithm considers such queries as queries without injected fields. If the input query (complex queries) is required to undergo a multilevel detection procedure (Case II): then invoke tree traversal algorithm to split the complex query into independent queries. Invoke Reconstruction (R-Iq) algorithm for each separate query. Repeat the procedure for each independent query separately and perform reconstruction procedure.

Table 3. Reconstruction (R-Iq) algorithm

```
Reconstruction (R-Iq) Algorithm
Procedure Reconstruction (Iq, Sq)
Begin
Sq← Standard query
Iq ← Input query
Regexp [ ] ← Regexp (Sq)
Sq- List [ ]← Get query-spliter (Sq) // parser split Sq based
on the input field
Iq- List [ ]← Get input-extractor (Iq, sq-list)
 For i = 1 to length (lq-List)
     If Sq- List [i] == Iq- List [i] // validate Iq with regular
expression
     Valid-input [i] ← Iq- List [i]
            Else
             Valid-input [i] ← Null;
 Endif
 Next
   For i = 1to length (Sq-List-1)
      Rejuvenate-Iq = Sq-List[i] + Valid-Input[i];
   Next
End
```

5 Proposed TbD-NNbR Frame Work

The proposed prototype Token based Detection and Neural Network based Reconstruction (TbD-NNbR), is developed and implemented using Java based application software and MySQL as back-end database server. Web crawler functionality is implemented in the web application, to identify the hot spot or form field identification of user input queries and to make a template model for the detection framework. The captured queries are parsed or split into different tokens and stored in a template repository like the data structure server. Malicious queries are logged in and documented for developing the anomaly pattern to have a stronger detection model in the later stage for handling the zero-day vulnerability. The TbD-NNbR intercepts the SQL queries before placing them to the web server, with the intervention of the proxy server and allows only benign queries through the web server.

5.1 TbD Training Phase

This phase identifies the underlying web application for template creation of the SQL query. It also generates the pre-defined token formats. The primary components under this stage is

- Crawler to identify the hotspot or input field
- Standard query template creator (SQTC)

The TbD-NNbR, model makes use of Uniform Resource Locator (URL) of the web application and user authentication credentials as inputs and generates a number of tokens using the standard query template creator (SQTC) for dynamic user query validation at the run time or testing phase. Authentication details, URL for each web page are intercepted or captured and analyzed by a proxy server during the training phase. The SQTC creates the query model, which will be stored in the repository and later used for dynamic query template mapping. The model creation phase or the training phase is shown in Figure 7.

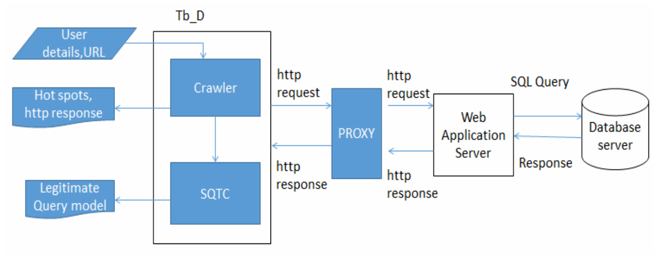
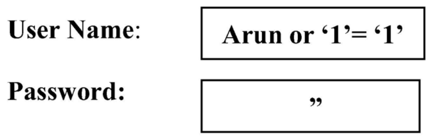


Figure 7. Tb-D training phase

5.2 Identification of Hot Spot/Form Field Entry in the Web Pages

A custom-made crawler is deployed/ implemented to browse through the underlying web application to identify the user entry/ input field on each web page. Each input field, which can be the vulnerable point, can be filled with appropriate value and submitted. The crawler also keeps track of the URL and authentication details of the http request. The entire application is crawled to record all the input entry fields in each web page without any missing entry in the forms, especially the entry fields which are critically vulnerable based on the type of data requested by that entry field. The deployed web crawler can identify with much precision, all the hot spots or the form entry fields, which the user fills on each web page.

For example, the http request on the form field of login page (an injected query) is shown in Figure 8.



User Name: **Arun or '1'= '1'**

Password: ”

Figure 8. http request on the form field of a login page (An injected query)

The corresponding SQL query: SELECT * FROM administrators WHERE username=Arun OR '1'= '1' AND password =”;

5.3 Standard Query Template Creator (SQTC)

The proposed framework creates the Standard Query Template Creator (SQTC) model by parsing each query into various predefined tokens, and a unique query ID and template is created and stored in the template repository (a database structure server) in JSON format. The use of JSON format substantially decreases the storage overhead. The crawler identifies the query, and the SQTC on the proxy server tokenizes it as per the pre-defined tokens designed and developed by analyzing the schema and grammar of the SQL statements. A similar token specification is applicable for the queries submitted to an Oracle server, MySQL and SQL Server. In this approach, while sending the standard query, it is possible to select any one of the databases as there is no difference in SQL query format in the databases mentioned above. It generates the unique Template-ID for each query with the corresponding template format and stores it in the template repository.

5.4 Learning Phase of Back Propagated Neural Network Learned Model

The reconstruction module has two options for generating the standard model. It can either make use of an SQTC application or a neural network based trained query model. This research work focused on neural network trained model for reconstruction. The learning phase of neural network based model also takes all the form field in response to the actual user inputs extracted from the crawler and set of SQL queries are collected from the URL of each web page. It also considers all possible input query injections for each form field during the learning of appropriate model template. Approximately, 32,000 URLs are tested during the training/learning phase to achieve a strong model template. The ideal Standard query model template corresponding to each page of the web application is learned to use a back-propagated artificial neural network and stored in the template store.

5.5 Testing Phase of TbD-NNbR

As shown in Figure 9, the testing phase of **TbD-NNbR** consists of the following major components for detecting the SQL injection attack:

- Template generator/parser
- The Model mapper
- SQL Injection Attack Detection Engine
- Reconstruction component

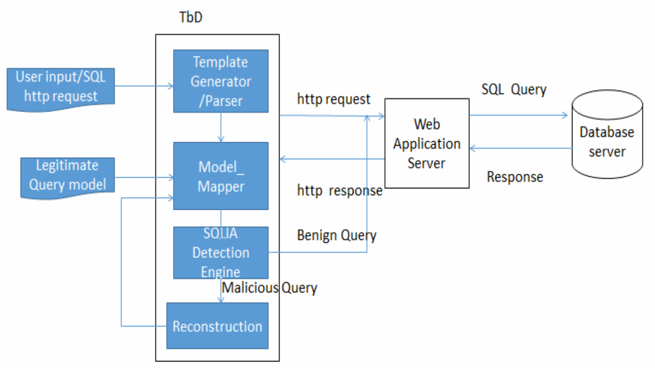


Figure 9. TbD testing phase

5.6 Template Generator/Parser for User Input Query

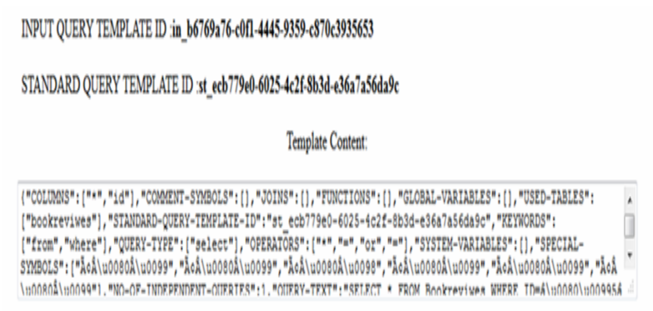
In the testing phase of the TbD framework, the template generator or parser splits the user queries accepted through the web pages. It performs the splitting of the queries by invoking the template creator procedure of Standard Query Template Creator (SQTC).

5.7 The Model Mapper

The major task of the model mapper is to locate and retrieve the accurate unique ID and template of the SQL query from the template repository, corresponding to the form field entry of the dynamic user queries. To have faster retrieval of the appropriate query, a SQLIAShield (JAR file) is deployed with this frame work. For example, the SQLIAShield for accepting user input through the log-in form can be specified by the path specification as:

SQLIAShield ("D:\\SQLIAConfig\\Template\\st_fdb52977-8288-4a7f-82e0-1b9c23e9a3d4.txt", "D:\\SQLIAConfig\\Output");

There is a SQLIAShield for each legal query in each page to specify the location/ path. This module invokes a template mapper algorithm at this stage for mapping the parsed dynamic user queries with the legal query model against Code Injection attack. Figure 10 shows the identified standard and Input Query ID for mapping in JSON format.


INPUT QUERY TEMPLATE ID :in_b6769a76-c0f1-4445-9359-c870c3935653

STANDARD QUERY TEMPLATE ID :st_ecb779e0-6025-4c2f-8b3d-e36a7a56da9c

Template Content:

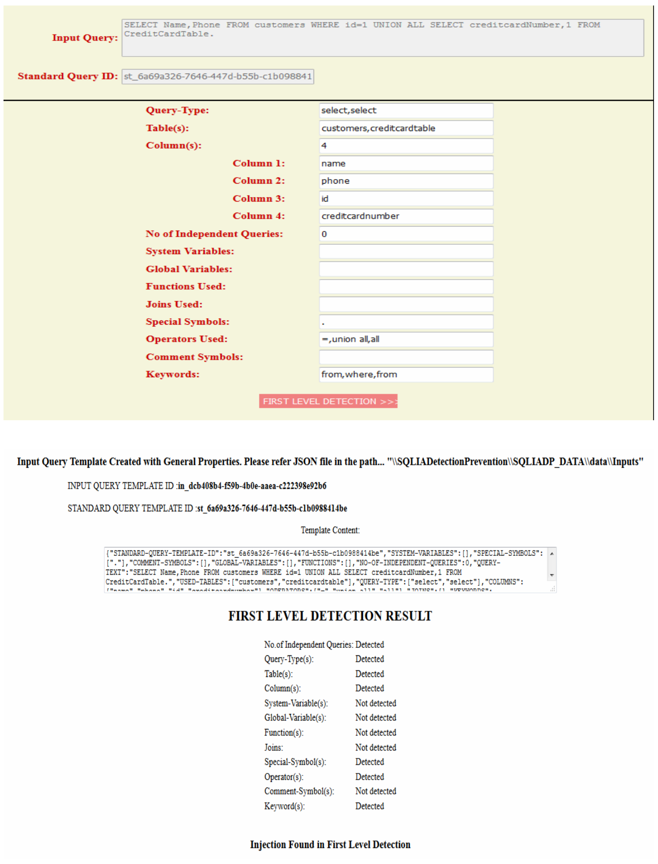
{"COLUMNS":["*","id"],"COMMENT-SYMBOLS":[],"JOINS":[],"FUNCTIONS":[],"GLOBAL-VARIABLES":[],"USED-TABLES":["bookrevives"],"STANDARD-QUERY-TEMPLATE-ID":"st_ecb779e0-6025-4c2f-8b3d-e36a7a56da9c","KEYWORDS":["from","where"],"QUERY-TYPE":["select"],"OPERATORS":["*","=","or","="],"SYSTEM-VARIABLES":[],"SPECIAL-SYMBOLS":["ÃcÂ\u0080Â\u0099","ÃcÂ\u0080Â\u0099","ÃcÂ\u0080Â\u0098","ÃcÂ\u0080Â\u0099","ÃcÂ\u0080Â\u0099","ÃcÂ\u0080Â\u0099"],"NO-OF-INDEPENDENT-QUERIES":1,"QUERY-TEXT":"SELECT * FROM Bookrevives WHERE ID=Â\u0080\u0099SÂ

Figure 10. Identified Query ID for mapping in JSON format

5.8 SQL Injection Attack Detection Engine

The detection engine detects SQL injection attacks by matching the parsed user input query against the legal query model developed during the training phase.

Figure 11 shows a sample evaluation result of the prototype tool TbD. SELECT Name, Phone from customers where id = 1 UNION ALL select creditcardNumber, 1 from CreditCardTable is the dynamic query. The SQL Token mapper algorithm checks the match between the dynamic query and the standard query. But there is no match on the tokens such as Tables, Special symbols, and Operators. So, the injection is detected and is displayed as shown in Figure 11. Since the above dynamic query had no sub-queries, the procedure is not repeated, and so it is a first level detection. The detection process of complex queries with multiple sub-queries is carried out by repeating the detection procedure for the second time, and then the detection engine displays the result as 'second level detection' based on the validation requirement of the queries in the given application.


Input Query: SELECT Name,Phone FROM customers WHERE id=1 UNION ALL SELECT creditcardNumber,1 FROM CreditCardTable.

Standard Query ID: st_6a69a326-7646-447d-b55b-c1b098841

Query-Type:	select,select
Table(s):	customers,creditcardtable
Column(s):	4
Column 1:	name
Column 2:	phone
Column 3:	id
Column 4:	creditcardnumber
No of Independent Queries:	0
System Variables:	
Global Variables:	
Functions Used:	
Joins Used:	
Special Symbols:	.
Operators Used:	=,union all,all
Comment Symbols:	
Keywords:	from,where,from

Input Query Template Created with General Properties. Please refer JSON file in the path... "\\SQLIADetectionPrevention\\SQLIADP_DATA\\data\\Inputs"

INPUT QUERY TEMPLATE ID :in_dcb408b4-f59b-4b0e-aaea-c222398e92b6

STANDARD QUERY TEMPLATE ID :st_6a69a326-7646-447d-b55b-c1b0988414be

Template Content:

{"STANDARD-QUERY-TEMPLATE-ID":"st_6a69a326-7646-447d-b55b-c1b0988414be","SYSTEM-VARIABLES":[],"SPECIAL-SYMBOLS":["."],"COMMENT-SYMBOLS":[],"GLOBAL-VARIABLES":[],"FUNCTIONS":[],"NO-OF-INDEPENDENT-QUERIES":0,"QUERY-TEXT":"SELECT Name,Phone FROM customers WHERE id=1 UNION ALL SELECT creditcardNumber,1 FROM CreditCardTable.","USED-TABLES":["customers","creditcardtable"],"QUERY-TYPE":["select","select"],"COLUMNS":

FIRST LEVEL DETECTION RESULT

No.of Independent Queries:	Detected
Query-Type(s):	Detected
Table(s):	Detected
Column(s):	Detected
System-Variable(s):	Not detected
Global-Variable(s):	Not detected
Function(s):	Not detected
Joins:	Not detected
Special-Symbol(s):	Detected
Operator(s):	Detected
Comment-Symbol(s):	Not detected
Keyword(s):	Detected

Injection Found in First Level Detection

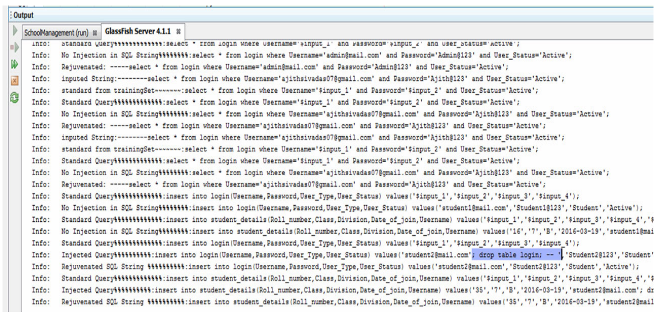
Figure 11. A sample evaluation result of the prototype tool TbD

Figure 11 shows the matched result by the detection engine using SQTC for a simple query of first level detection and mapping of user input query and the Standard query template ID of the SQTC model.

5.9 Reconstruction Component

In the proposed TbD-NNbR framework, reconstruction of queries is carried out after comparing the user input queries with BPNN learnt legal model and later reconstructing the queries with the support of REGEX function. The module invokes the reconstruction procedure only when the http request re-confirms that the malicious query is from an authenticated/ registered user.

Figure 12 shows the status report of the reconstruction process with several injected (malicious) queries and the corresponding reconstructed portion/ string (Rejuvenated query). For example, consider the dynamic query: **insert into login (Username, Password, User Type, User** Status) values (‘student2@mail.com’; drop table login--, ‘Student2@123’, ‘Student’, ‘Active’). Here “drop table login - -”; is the injected portion of the query which must be removed or replaced with a null value. So, the reconstruction algorithm rejuvenates the malicious query and converts it into a benign query. Reconstruction functionality included with this prototype mitigates the denial of service attack at a certain level. The deployed SQLIAShield with appropriate path specification for the corresponding SQL statement given in each web page will support the system with faster processing capability.



```
Info:   Standard Query%%%%%%%%%%%%%%:select * from login where Username='$input_1' and Password='$input_2' and User_Status='Active';
Info:   No Injection in SQL String%%%%%%%%%:select * from login where Username='admin@mail.com' and Password='Admin@123' and User_Status='Active';
Info:   Rejuvenated: -----select * from login where Username='admin@mail.com' and Password='Admin@123' and User_Status='Active';
Info:   inputed String:--------select * from login where Username='ajithsivadas07@gmail.com' and Password='Ajith@123' and User_Status='Active';
Info:   standard from trainingSet~~~~~~~:select * from login where Username='$input_1' and Password='$input_2' and User_Status='Active';
Info:   Standard Query%%%%%%%%%%%%%%:select * from login where Username='$input_1' and Password='$input_2' and User_Status='Active';
Info:   No Injection in SQL String%%%%%%%%%:select * from login where Username='ajithsivadas07@gmail.com' and Password='Ajith@123' and User_Status='Active';
Info:   Rejuvenated: -----select * from login where Username='ajithsivadas07@gmail.com' and Password='Ajith@123' and User_Status='Active';
Info:   inputed String:--------select * from login where Username='ajithsivadas07@gmail.com' and Password='Ajith@123' and User_Status='Active';
Info:   standard from trainingSet~~~~~~~:select * from login where Username='$input_1' and Password='$input_2' and User_Status='Active';
Info:   Standard Query%%%%%%%%%%%%%%:select * from login where Username='$input_1' and Password='$input_2' and User_Status='Active';
Info:   No Injection in SQL String%%%%%%%%%:select * from login where Username='ajithsivadas07@gmail.com' and Password='Ajith@123' and User_Status='Active';
Info:   Rejuvenated: -----select * from login where Username='ajithsivadas07@gmail.com' and Password='Ajith@123' and User_Status='Active';
Info:   Standard Query%%%%%%%%%%%%%%:insert into login(Username,Password,User_Type,User_Status) values('$input_1','$input_2','$input_3','$input_4');
Info:   No Injection in SQL String%%%%%%%%%:insert into login(Username,Password,User_Type,User_Status) values('student1@mail.com','Student1@123','Student','Active');
Info:   Standard Query%%%%%%%%%%%%%%:insert into student_details(Roll_number,Class,Division,Date_of_join,Username) values('$input_1','$input_2','$input_3','$input_4','$
Info:   No Injection in SQL String%%%%%%%%%:insert into student_details(Roll_number,Class,Division,Date_of_join,Username) values('16','7','B','2016-03-19','student1@mai
Info:   Standard Query%%%%%%%%%%%%%%:insert into login(Username,Password,User_Type,User_Status) values('$input_1','$input_2','$input_3','$input_4');
Info:   Injected Query%%%%%%%%%%%:insert into login(Username,Password,User_Type,User_Status) values('student2@mail.com'; drop table login; -- ','Student2@123','Student'
Info:   Rejuvenated SQL String %%%%%%%%%%%:insert into login(Username,Password,User_Type,User_Status) values('student2@mail.com','Student2@123','Student','Active');
Info:   Standard Query%%%%%%%%%%%%%%:insert into student_details(Roll_number,Class,Division,Date_of_join,Username) values('$input_1','$input_2','$input_3','$input_4','$
Info:   Injected Query%%%%%%%%%%%:insert into student_details(Roll_number,Class,Division,Date_of_join,Username) values('35','7','B','2016-03-19','student2@mail.com'; dr
Info:   Rejuvenated SQL String %%%%%%%%%%%:insert into student_details(Roll_number,Class,Division,Date_of_join,Username) values('35','7','B','2016-03-19','student2@mail
```

Figure 12. Status report of reconstruction procedure in TbD-NNbR framework

6 Evaluation

The factors such as query execution time, efficiency, effectiveness and precision are considered to evaluate the performance of the proposed TbD-NNbR framework. A data structure server or the template repository deploys the Standard Query Template Constructor (SQTC) within the TbD-NNbR prototype. A relational database server is placed at the backend to capture and execute the queries in SQL schema. The prototype is designed and developed for window based operating system. Since JSON format is used to store the query template, it decreases the storage overhead and reduces the run time overhead. It assigns an SQLIAShield to each web page with appropriate path specification for the corresponding SQL statement given in the web application. Various attack categories of queries were analyzed to get the possible structure of the required learnt SQL model using Back Propagated Neural Network (BPNN). Apart from evaluating the prototype with the various standard applications and Cheat sheet, a customized school management application is exclusively developed and implemented to test the effectiveness and efficiency of the reconstruction prototype. The empirical analysis carried out on various test beds, and the test result of injection attacks shows that the proposed framework can identify and detect any injections.

6.1 Data Set Used for Testing TbD-NNbR

We evaluate the TbD-NNbR prototype by using three different data sets collected from several standard open test suites, known vulnerability testing sites and cheat sheets/URL after conducting a detailed survey.

6.1.1 Dataset I: Data Available from Cheat Sheets/ URL

Table 4 shows the data collected from the cheat sheet/ URL. The application type, number of attack requested listed in the application and details about the successful detection are also represented with corresponding false positives.

Table 4. Data collected from the cheat sheet/URL

Cheat sheet/URL	Attack Request	Successful Detection	False Positives
Schoolmate	26	26	0
Webchess	32	32	0
Faqforge	21	21	0
EVE	22	22	0
Geccbblite	32	32	0
http://groups.csail.mit.edu/pag/ardilla	48	48	0
http://pentestmonkey.net/cheat-sql-injection/oracle-sql-injection-cheat-sheet	35	35	0

6.1.2 Dataset II: Standard Test Suites Provided by Halfond and Orso

The following applications in a standard test suite (the test suite used to evaluate AMENSIA tool) provided by Halfond and Orso are used to assess the TbD-NNbR prototype. The below-mentioned applications use a relational database as the backend server. Table 5 shows the identified application and number of hotspots identified in each application.

Table 5. The identified application with hotspots

Application	Description	#Hotspots
Book store	Online book store	25
Events	Event tracking system	12
Employee Directory	Online Employee directory	10

Table 6 indicates the number of forms identified in each application and out of which how many forms are expected to be vulnerable is also clearly shown. The table also shows the detected vulnerable forms along with false positive and false negative incidents. In each of the identified web forms, there can be multiple hotspots which are susceptible. Almost 25 form fields are vulnerable in Book store, 12 in Events application and 10 in Employee directory application.

Table 6. Effectiveness of TbD-NNbR

Application	#Forms	Expected Vuln_form-ent	Detected Vuln_form_ent	F_Positive #	F_Negative #
Book Store	25	21	20	1	1
Events	12	8	8	0	0
Employee Directory	10	7	6	1	0

Vuln_form-ents: Vulnerable form entries; F-Postitive: False positive; F-Negative: False Negative

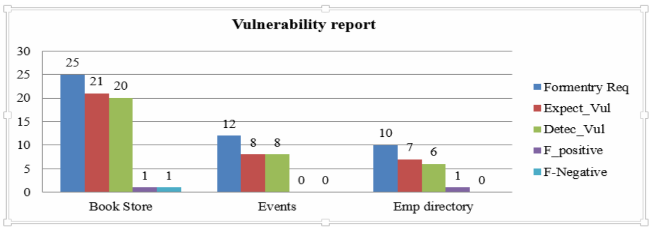


Figure 13. Effectiveness of TbD-NNbR (Vulnerability report)

The analysis from the above table shows that in Book store and Employee directory applications, there is one of each vulnerable form, which is not detected correctly and skipped by the web crawler application. The expected vulnerable forms could not be identified correctly during the training phase. Because of this inaccurate crawling functionality employed in the application, the proxy server functionality at the TbD-NNbR prototype could not block the malicious entry, and it bypasses or skips the model checking and mapping procedure. Hence, there is an incidence of false positives, and false negatives. We can avoid such occurrences if the crawler can identify the vulnerable forms with a better accuracy or with a perfect detection procedure. By considering the above strategies, the detection rate of the prototype on SQL Injection query is 95.66%, which is the best result in comparison with other models.

6.1.3 Dataset III: Customized SchoolEconnect System

The SchoolEconnect web application developed exclusively for testing the prototype can handle the following sub-applications/ modules such as E-learning portal for students, Employee directory for the teaching support & staff, online resource management system and event planner for the school activities. Table 7 shows the identified hotspots, expected attacks, detected attacks and the corresponding false positive rates.

Table 7. Modules of SchoolEconnect with attack detection details

Application	Description	Hotspot Identified	Expected attack	Detected attack	False positives
E-portal	E-learning portal	23	18	18	0
Emp directory	Employee management	14	10	9	1
Online Resource	Online Resource management system	27	22	22	0
Event-planner		18	12	11	0

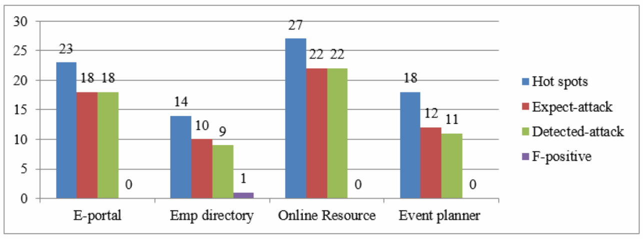


Figure 14. SchoolEconnect with attack detection details

The above details reported in Table 7 indicate that there is only one attack bypassed in the employee directory due to the inappropriate authentication credentials registered and stored in the database server, which has blocked the mapper functionality of the TbD-NNbR framework. Event planner application has a drawback of handling the time and date function. We can quickly rectify it, and with this patching up, 100% detection is possible.

6.2 Performance Measures

We can assess the performance of the proposed model by considering the time overhead or process delay imposed on the prototype at runtime. JSON data at the database level is a valid technique to simplify the data resource implementation cost such as configuration, table handling, filtering, and dynamic query processing. The factors such as efficiency, effectiveness and precision are the base for the evaluation of the proposed model [22-23].

6.2.1 Process Time Overhead

The process time overhead is directly related to the rate at which each web page gets loaded, the number of form fields on each page and the type of database servers assigned for execution of queries. The performance metrics measures the average CPU time spent for processing the query. Since we use JSON format, the storage and retrieval of standard queries in the TbD-NNbR model are easier and faster as compared to the other standard models. The proposed model has a strategy for detecting the queries by placing them in different attack categories and complexity levels. In most of the cases the less complex queries can be identified in the first level but the queries which are complex and require traversing technique to split it into a smaller format, only need a little longer time for detection, which is the second level of the procedure. In both cases of simple and complex query analysis, the time taken for detection is only few seconds and the reconstruction of queries are also carried out with a negligible delay in response time. Hence the processing time and the overhead involved in executing the query are negligible when comparing with the response time of a browser in accessing the web application.

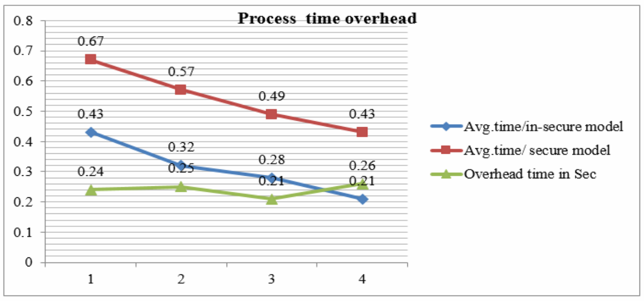


Figure 15. Efficiency of TbD-NNbR

6.2.2 Efficiency

There is a secure and insecure version of SchoolEconnect application designed and deployed as part of this research work. To understand the efficiency of the proposed framework we have empirically analyzed the successful attacks detected as shown in Table 8. We test the queries against the secure and insecure version of the same application designed and deployed by using Java based application software. The performance penalty for the execution of the individual query with the proposed techniques (secured version) is the processing overhead of the queries received. The prototype is evaluated with the secured version and insecure version of the ScoolEconnect application to analyze and identify the process time overhead. In the secure version, the proxy server with TbD-NNbR module is implemented to check and block the malicious or injected queries, before they reach the backend server of the SchoolEconnect.

Table 8. Time overhead in SchoolEconnect application

Application	Successful Detection	Avg.Time Insecure Version	Avg.Time Secure Version	Overhead Time in Sec
E-portal	32	0.43	0.67	0.24
Emp directory	63	0.32	0.57	0.25
Online Resource	45	0.28	0.49	0.21
Event planner	72	21	0.43	0.26

The insecure version of the SchoolEconnect application demonstrates how the injected queries can bypass the authentication logic, but those queries are detected and blocked for further execution with the secured version. The time overhead incurred in query execution of secure SchoolEconnect application is 21 to 24 Seconds. It is negligible when compared with the time taken for loading a page of a web application in the browser or with the time taken for getting a response from a web application. The standard response time to get an answer to the query is a few seconds, which is not a significant overhead and is ignored. Therefore, the overhead incurred by deploying the secure TbD-NNbR is negligible, and, we ignore the processing delay. The proposed architecture is complimentary to many of the available models due to faster detection and low overhead on storage. The model appropriately identifies all the tested queries, and this proved that the proposed system is highly efficient.

6.2.3 Precision

Precision measures the rate of false positives. The dataset of legal and injected query tabulated during the initial stage of the research study, apart from the dataset from the applications mentioned above are being tested to understand the precision measure of TbD-NNbR model. The analysis in Table 9, shows that out of 1655 queries tested, 451 queries belong to malicious categories, and 1204 queries refer to legal queries. There are only 4 cases of false positives due to the inappropriate authentication credentials on the test bed used and crawler functionality at the identification of form entry fields at the training stage of TbD-NNbR. We patch this in the trial run.

Table 9. Analysis of False positives in General

Identified/tested Queries	Legal Queries	Malicious Queries	Successful Detection	False positives	Detection %
1655	1204	451	1651	4	99.75

Table 10 shows the empirical analysis of TbD-NNbR using multiple variations of the queries mentioned above, tested with the SchoolEconnect application.

Table 10. Analysis of false positives in School Econnect application

Application	Queries Tested	Legal Queries (Successful)	Malicious Queries (Successful)	False positives	Detection rate
E-portal	123	100	23	1	99.18
Emp directory	114	80	34	0	100
Online Resource	127	90	37	2	98.42
Event planner	118	90	28	1	99.15

The above analysis implies that the average detection rate of modules listed in the SchoolEconnect is 99.19 %, which is a highly recommended evaluation result.

6.2.4 Effectiveness

We base the effectiveness of the proposed system on the number of false negatives reported during the empirical analysis. From the collected queries, 1655 queries were already tested with other techniques and proved to be malicious queries. The empirical analysis using the proposed method could also detect all the queries, reported as successful attacks. Data shown in Table 11 indicate that more than 98% detection is possible based on the occurrence of false negatives and false positives reported.

Table 11. Attack categories and detection rate in TbD-NNbR

Type of Queries (Attack categories)	Malicious queries	Detected Queries	False Negatives	False Positive	Detection %
Tautology	65	64	0	1	98.46
Union Queries	45	45	0	0	100
Piggy Backed Queries	92	91	0	1	98.91
Logically Incorrect Queries	56	56	0	0	100
Stored procedure	78	77	0	1	98.71
Inference	67	66	0	1	98.50
Alternate Code	48	48	0	0	100

The average detection rate from the above analysis is 99.23% which shows that the effectiveness of the proposed system is rated as the best approach to the SQL injection detection and blocking.

6.3 Type I & Type II Error

We tested the proposed TbD-NNbR with 1204 legal queries and 451 malicious queries collected from an E-learning module and a Student management system (consisting of modules such as course advising, registration, attendance and transcript management system) of a technical college. This empirical analysis shown in Figure 16 indicates that the TbD-NNbR prototype correctly detects all the queries tested with false positive rate 4%, and false negative rate 0%. The above analysis shows that the proposed framework achieved 100% detection.

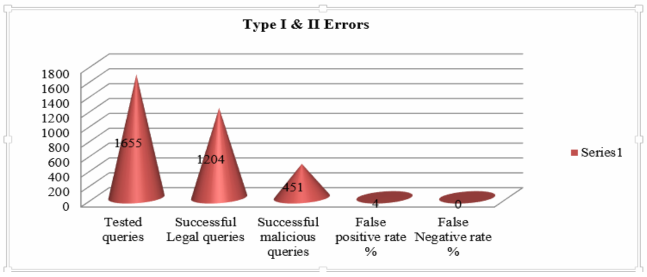


Figure 16. Type I & Type II error rate

6.4 Receiver Operating Characteristic (ROC) Curve

The reconstruction module performs the reconstruction of queries only if the query is from an authenticated user. The Receiver Operating Characteristic curve (ROC) of BP-NN learned model shown in Figure 17 indicates that the trained model is achieving 100% result.

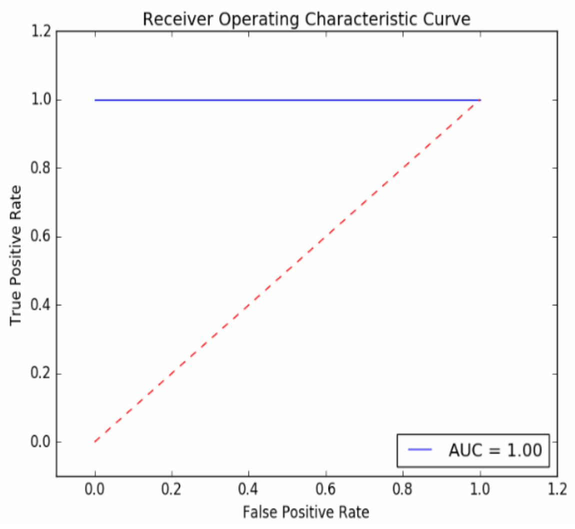


Figure 17. Receiver Operating Characteristic

6.5 Comparison of TbD-NNbR with Other Models

Based on the type of queries, we compared the proposed TbD-NNbR prototype with other standard models such as AMNESIA, IDS, SQL-Check, SQL guard, Tautology Checker, JDBC checker, and SQLDOM [24-25]. The result of the comparative study of TbD-NNbR with other standard techniques indicates that most of the methods fail to detect SQL Injection vulnerabilities under the category of stored procedure, whereas the proposed framework can detect this attack efficiently [26]. The other significant features that differentiate the TbD-NNbR from the other models are faster query processing, perfect detection and blocking of malicious queries, less storage requirement and efficient handling of Time-Space complexity [27].

7 Conclusion

The proposed prototype TbD-NNbR, detects and blocks code injection vulnerabilities effectively with negligible processing overhead and has the novel technique of reconstructing queries. The token parsing techniques used in the template creator application, template files stored in the JSON format and Jar files utilised in the template creator application contribute equally in decreasing the storage overheads. The empirical analysis that was carried out by using data from various shared applications available online

Table 12. Comparison of TbD-NNbR with other models

Detection/Prevention method	Tautologies	Union Queries	Illegal Queries	Piggy-back	Inference	Altemate encoding	Stored Procedure
AMNESIA	✓	✓	✓	✓	✓	✓	×
IDS	/	/	/	/	/	/	/
SQL Check	✓	✓	✓	✓	✓	✓	×
SQL Guard	✓	✓	✓	✓	✓	✓	×
Tautology checker	✓	×	×	×	×	×	×
JDBC Checker	NA	NA	NA	NA	NA	NA	NA
SQL DOM	✓	✓	✓	✓	✓	✓	×
Proposed TbD-NNbR	✓	✓	✓	✓	✓	✓	✓

Legend: ✓-Possible ×-Impossible /-Partially possible NA-Not applicable

shows that an average of 99.23% detection is possible using the proposed framework. The framework detects all the malicious queries without any false negatives which indicates that the proposed technique handles malicious queries effectively. The Neural Network based Reconstruction (NNbR) of queries from authenticated users, increases the web application availability and decreases the Denial of Service attack. The empirical evaluation performed on Secure SchoolEconnect (customized online application) shows that the proposed system has only the bare minimum overheads and Time-Space complexity.

Funding

This research did not receive any specific grant from funding agencies in the public, commercial, or not-for-profit sectors.

References

[1] R. Johari, P. Sharma, A Survey on Web Application Vulnerabilities (SQLIA, XSS) Exploitation and Security Engine for SQL Injection, *2012 International Conference on Communication Systems and Network Technologie*, Rajkot, India, 2012, pp. 453-458.

[2] W.-G. J. Halfond, J. Viegas, A. Orso, A Classification of SQL-injection Attacks and Counter Measures, *Proceedings of the IEEE International Symposium on Secure Software Engineering*, Washington, DC, 2006, pp. 13-15.

[3] N. M, Sheykhkanloo, Employing Neural Networks for the Detection of SQL Injection Attack, *Proceedings of the 7th International Conference on Security of Information and Networks*, Glasgow, UK, 2014, pp. 318-323.

[4] F. Valeur, D. Mutz, G. Vigna, A Learning-based Approach to the Detection of SQL Attacks, *Proceedings of the Second International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment*, Vienna, Austria, 2005, pp. 123-140.

[5] R. Dharam, S. G. Shiva, Runtime Monitoring Technique to Handle Tutology Based SQL Injection Attacks, *International Journal of Cyber-Security and Digital Forensics*, Vol. 1, No. 3, pp. 189-203, November, 2012.

[6] L. Zhang, Q. Gu, S. Peng, X. Chen, H. Zhao, D. Chen, A Web Application Vulnerabilities Detection Tool Using Characteristics of Web Forms, *Fifth International Conference on Software Engineering Advances*, Nice, France, 2010, pp. 501-507.

[7] M. Ruse, T. Sarkar, S. Basu, Analysis and Detection of SQL Injection Vulnerabilities via Automatic Test Case Generation of Programs, *10th IEEE/IPSJ International Symposium in Applications and the Internet,* Seoul, South Korea, 2010, pp. 31-37.

[8] A. Calvi, L. Vigano, An Automated Approach for Testing the Security of Web Applications against Chained Attacks, *Proceedings of the 31st Annual ACM Symposium on Applied Computing*, Pisa, Italy, 2016, pp. 2095-2102.

[9] N. Skrupsky, P. Bisht, T. Hinrichs, V. N. Venkatakrishnan, L. Zuck, TamperProof: A Server-agnostic Defense for Parameter Tampering Attacks on Web Applications, *Proceedings of the third ACM conference on Data and application security and privacy*, San Antonio, TX, 2013, pp. 129-140.

[10] F. Lebeau, B. Legeard, F. Peureux, A. Vernotte, Model-based Vulnerability Testing for Web Applications, *2013 IEEE Sixth International Conference on Software Testing, Verification and Validation Workshops*, Luxembourg, Luxembourg, 2013, pp. 445-452.

[11] M. Burkhart, D. Schatzmann, B. Trammell, E. Boschi, B. Plattner, The Role of Network Trace Anonymization under Attack, *ACM SIGCOMM Computer Communication Review*, Vol. 40, No. 1, pp. 5-11, January, 2010.

[12] S. W. Boyd, A. D. Keromytis, SQLrand: Preventing SQL Injection Attacks, *International Conference on Applied Cryptography and Network Security*, Yellow Mountain, China, 2004, pp. 292-302

[13] W. G. J. Halfond, A. Orso, Combining Static Analysis and Runtime Monitoring to Counter SQL-injection Attacks, *ACM Software Engineering Notes*, Vol. 30. No. 4, pp. 1-7, July, 2005,

[14] W. G. J. Halfond, A. Orso, P. Manolios, WASP: Protecting Web Applications Using Positive Tainting and Syntax-aware Evaluation, *IEEE Transactions on Software Engineering*, Vol. 34. No. 1, pp. 65-81, January/ February, 2008.

[15] Y.-W. Huang, F. Yu, C. Hang, C.-H. Tsai, D.-T. Lee, S.-Y. Kuo, Securing Web Application Code by Static Analysis and Runtime Protection, *Proceedings of the 13th international conference on World Wide Web*, New York, NY, 2004, pp. 40-52.

[16] T.-K. George, R. James, P. Jacob, Proposed Hybrid Model to Detect and Prevent SQL Injection, *International Journal of Computer Science and Information Security*, Vo. 14, No. 6, June, 2016.

[17] M. Moradi, M. Zulkernine, A Neural Network Based System for Intrusion Detection and Classification of Attacks, *Proc. of the 2004 IEEE International Conference on Advances in Intelligent Systems- Theory and Applications*, Luxembourg, 2004, pp. 15-18.

[18] N. M. Sheykhkanloo, A Pattern Recognition Neural Network Model for Detection and Classification of SQL Injection Attacks, *World Academy of Science, Engineering and Technology, International Journal of Computer, Electrical, Automation, Control and Information Engineering*, Vol. 9. No. 6, pp. 1443-1453, June, 2015.

[19] A. Moosa, Artificial Neural Network Based Web Application Firewall for sql Injection, *World Academy of Science, Engineering & Technology, International Journal of Computer and Information Engineering*, Vol. 4. No. 4, pp. 610-619, April, 2010.

[20] D. Balzarotti, M. Cova, V. Felmetsger, V. Jovanovic, E. Kirda, C. Kruegel, G. Vigna, Saner: Composing Static and Dynamic Analysis to Validate Sanitization in Web Applications, *IEEE Symposium on Security and Privacy*, Okland, CA, 2008, pp. 387-401.

[21] K. Kemalis, T. Tzouramanis, SQL-IDS: A Specification-based Approach for SQL-injection Detection, *Proceedings of the 2008 ACM symposium on Applied computing*, Ceara, Brazil, 2008, pp. 2153-2158.

[22] P. Kumar, R. K. Pateriya, DWVP: Detection of Web Application Vulnerabilities Using Parameters of Web Form, *Proceedings of Joint International Conferences on CIIT*, Bhopal, India, 2013, pp. 437-443.

[23] M. A. Lawal, A. B. M. Sultan, A. O. Shakiru, Systematic Literature Review on SQL Injection Attack, *International Journal of Soft Computing*, Vol. 11. No. 1, pp. 26-35, January, 2016,.

[24] R. A. McClure, I. H. Kruger, SQL DOM: Compile Time Checking of Dynamic SQL Statements, *27th International Conference on Software Engineering*, Saint Louis, MO, 2005, pp. 88-96.

[25] S. Ali, A. Rauf, H. Javed, SQLIPA: An Authentication Mechanism Against sql Injection, *European Journal of Scientific Research*, Vol. 38, No. 4, pp. 604-611, December, 2009,.

[26] D. A. Kindy, A.-S. K. Pathan, A Detailed Survey on Various Aspects of sql Injection in Web Applications, Vulnerabilities, Innovative Attacks, and Remedies, *arXiv preprint arXiv: 1203.3324*, March, 2012.

[27] G. Deepa, P. S. Thilagam, Securing Web Applications from Injection and Logic Vulnerabilities: Approaches and Challenges, *Information and Software Technology*, Vol. 74, pp. 160-180, June, 2016.

Biographies

Teresa K. George is a research scholar at CUSAT, India. She is also working at HCT Muscat, Oman. Her teaching interests includes, Data base and Information security, She has over 20 years of teaching experience and published ten research papers in the field of Information Security and web applications.

K. Poulose Jacob is a Professor of Computer Science at CUSAT, Kerala. He is a Professional member of the ACM. His research interests are mainly in Intelligent Architectures, Networks, Information Systems and Artificial Intelligence. He has more than 120 publications to his credit .

Rekha K. James is a Professor at CUSAT, Cochin, India. Her research interests include the design of RNS based arithmetic circuit, Decimal arithmetic, Reversible logic and Low power Design. She has more than 20 publications to her credit.