International Journal of Information, Control and Computer Sciences
ISSN: 2517-9942
Vol:9, No:6, 2015

A Pattern Recognition Neural Network Model for Detection and Classification of SQL Injection Attacks

Naghmeh Moradpoor Sheykhkanloo

Abstract**—**Thousands of organisations store important and confidential information related to them, their customers, and their business partners in databases all across the world. The stored data ranges from less sensitive (e.g. first name, last name, date of birth) to more sensitive data (e.g. password, pin code, and credit card information). Losing data, disclosing confidential information or even changing the value of data are the severe damages that Structured Query Language injection (SQLi) attack can cause on a given database. It is a code injection technique where malicious SQL statements are inserted into a given SQL database by simply using a web browser. In this paper, we propose an effective pattern recognition neural network model for detection and classification of SQLi attacks. The proposed model is built from three main elements of: a Uniform Resource Locator (URL) generator in order to generate thousands of malicious and benign URLs, a URL classifier in order to: 1) classify each generated URL to either a benign URL or a malicious URL and 2) classify the malicious URLs into different SQLi attack categories, and a NN model in order to: 1) detect either a given URL is a malicious URL or a benign URL and 2) identify the type of SQLi attack for each malicious URL. The model is first trained and then evaluated by employing thousands of benign and malicious URLs. The results of the experiments are presented in order to demonstrate the effectiveness of the proposed approach.

Keywords**—**Neural Networks, pattern recognition, SQL injection attacks, SQL injection attack classification, SQL injection attack detection.

I. Introduction

DATABASE is a data structure that stores organised information and has multiple tables, which may each include several different fields. For instance, a company database may embrace tables for products, employees, and financial records. Each of these tables would have different fields that are relevant to the information stored in that table. Thousands of organisations store important and confidential information in databases all across the world. The stored data ranges from personal information such as first name, last name, date of birth, student identification number, staff identification number, home address, work address, mobile phone number, national insurance number, email address, and job title to more sensitive information such as username, password, pin code, and credit card information.

CIA triad, which stands for Confidentiality, Integrity, and Availability, is a well-known model that can be used to develop a security policy in any organisation. It was proposed in order to identify problem areas and necessity solutions for computer and information security. Confidentiality is a set of rules that limits or restricts access to certain type of information, Integrity, assures the accuracy of information during its full life-cycle, and Availability certifies that the information is accessible at a required level of performance.

If a given database is attacked the CIA's elements can be violated. For instance, the data in the database can be disclosed to unauthorised users, which is a failure in Confidentiality element of the CIA triad, or in worst-case scenario, it can be modified, which is a failure in Integrity element of the CIA triad, by the hackers or completely wiped out from the database, which is a failure in Availability element of the CIA triad. Therefore, it is important for any organisation to protect their databases in order to prevent any loss to themselves and to their customers.

Nearly all e-commerce sites use databases in order to store information related to customers, products, and financial records with which the data can be easily searched, modified, and updated. Sorting website data in a database provides flexibility as a vital factor for e- commerce sites and other types of dynamic websites.

SQL is a programming language, which is designed for managing data, including data definition, data insertion, data removal, and data modification, in a Relational Database Management System (RDBMS) [18].

SQLi attack is a code injection technique in which hackers try to disclose, modify or remove data from a given database by simply using SQL language and web browser. SQLi attack has been rated as the number-one attack among top ten web application threats on Open Web Application Security Project (OWASP) [13]. OWASP is an open community dedicated to enabling organisations to consider, develop, obtain, function, and preserve applications that can be trusted. SQLi attack takes advantages of inappropriate or poor coding of web applications that allows hackers to inject crafted SQL commands into say a login form in order to gain un-authorised access to data within a given database [14]. If the inputs from user are not properly sanitised, a hacker can generate crafted SQL commands and can inject them into the database in order to pass the login barrier and see what lies behind it. Generally speaking, the SQLi attack is a technology vulnerability that comes from dynamic script language such as Hypertext Processor (PHP), Active Server Pages (ASP), Java Server

Dr Naghmeh Moradpoor Sheykhkanloo is with the School of Science, Engineering, and Technology (SET), Abertay University, Bell Street, Dundee, Scotland, United Kingdom (phone: +44 (0) 1382 308353; fax: +44 (0) 1382 308663; e-mail: n.moradpoor@abertay.ac.uk).

pages (JSP), and Common Gateway Interface (CGI).

In this paper, a pattern recognition neural network model for detection and classification of SQLi attack is proposed in which the proposed model is able to: 1) identify the malicious URLs from the benign URLs and 2) detect the type of SQLi attack for the malicious URLs.

The remainder of this paper is organised as follows. In Sections II & III, seven popular types of SQLi attack as well as related work for SQLi attack detection and prevention are discussed. The proposed neural network-based model for detection and classification of SQLi attacks is discussed and detailed in Section IV. Sections V and VI include the implementations and captured results, respectively followed by conclusions of the work in Section VII, acknowledgments and references.

II. SQL Injection Attack Types

In this section, we discuss the seven popular types of SQLi attack which all are backed up with an example for each. There are countless variations for different SQLi attack types which can be generated by piecing together the different SQLi attacks. However, in this section, in order to give a clear understanding of different types of SQLi attack, we discuss a single representation of each attack along with the related signature(s) and the possible solution(s) to prevent it. A comprehensive classification for SQLi attacks and countermeasures can be found in [1].

A. Tautologies

Tautology SQLi attack is a type of attack in which hackers try to bypass authentication, identify injectable parameters and/or extract data from a given database by simply using WHERE clause conditions which are always be evaluated to true. For instance: “WHERE password = ‘x’ OR ‘x’ = ‘x’” or “WHERE password = ‘x’ OR 1=1”. In the example below, hackers execute a tautology SQLi attack against a back-end database by filling the “Username” textbox with “ ‘or 1=1 --“ in a sample login page:

SELECT “accounts” FROM “users” WHERE username = “‘or 1=1 -- “ AND password =”” AND pin =””

In the above example, as the WHERE clause is always evaluated to true, the database displays the entire rows from the “account” column of the “users” table on the hacker’s screen.

Signature: Given that the tautology SQLi attack always starts with a string terminator (‘) followed by the OR operators and a condition that is always evaluated to TRUE, the possible signatures for this type of SQLi attack are the string terminator “‘”, OR, =, LIKE and SELECT.

Prevention: The tautology SQLi attack can be prevented by strictly validating user inputs on user side and blocking queries containing tautological condition WHERE clauses on database side.

B. Illegal/logically Incorrect Queries

Illegal/logically incorrect queries SQLi attack is a type of attack in which hackers try to identify injectable parameters, perform database finger-printing, and/or extract data from databases by providing illegal/logically incorrect queries. In Illegal/logically incorrect queries SQLi attack, logical errors, type errors as well as syntax errors are the most popular queries that hackers try to generate from a vulnerable login page in an attempt to obtain information about a back-end database. The example below reveals how generated type errors can help hackers to get useful information from a back-end Microsoft SQL Server database.

SELECT “accounts” FROM “users” WHERE username = “” AND password = “” AND pin = “convert (int, (select top 1 name from sysobjects where xtype = ‘u’))”

In the above example, the query tries to extract the first user table name from “sysobjects” where “xtype = ‘u’” and then converts it to integer. By assuming that the back-end database is a Microsoft SQL Server, the generated query contains illegal type conversions therefore it makes the database to throw the following error:

“Microsoft OLE DB Provider for SQL Server (0x80040E07) Error converting nvarchar value ’CreditCards’ to a column of data type int.” This error message can provide the following useful information for hackers. First, it reveals that the back-end database is a Microsoft SQL. This can aid attackers narrowing down all their attempts to a single type of database. Second, the above error message discloses the type of the first defined table in “sysobjects” metadata database, which is “nvarchar”.Third, it exposes the name of the first defined table in “sysobjects” metadata database, which is “CreditCards”.

Signature: There are several ways to perform Illegal/logically incorrect SQLi attack against a given database. This includes employing all the possible incorrect conversions and incorrect logics in SQL world. Therefore, the possible signatures for this type of attack are: invalid conversions (CONVERT (TYPE)), incorrect logics, using AND operator to perform incorrect logics, using ORDERBY, and incorrectly terminating the string using (‘), etc.

Prevention: Strictly validating user inputs on user side and stopping/sanitising the generated error messages such as logical errors, type errors and syntax errors from a given database are the two effective countermeasures for preventing the Illegal/logically incorrect queries SQLi attack.

C. Piggy-Backed Query

Piggy-backed query SQLi attack is a type of attack in which hackers aim to extract data, add or modify data, perform denial of service attack and/or execute remote commands on a back-end database by taking advantages of database misconfiguration where executing multiple statements in a single query is allowed.

The example below shows how the piggy-backed query SQLi can be sent by hackers through a user interface to the related back-end database.

SELECT “accounts” FROM “users” WHERE username = “john1390” AND password = “ ’; drop table users --“ AND pin = “123”

When the above query arrives at the database, it compiles

the first query and then recognises the delimiter ";", which makes the second query runs right after the first query. Running the second query results in dropping the table "users" form the database and thus wipes all the information from the database.

Signature: Given the above example and explanations, the signature for the piggy-backed query SQLi attack is delimiter ";".

Prevention: Strictly validating user inputs on user side and avoiding multiple statement executions on a database by scanning all queries for delimiter ";"on database side are two countermeasures for the piggy-backed query SQLi attacks. However, most databases in the world don't require a delimiter ";" for multiple statement executions therefore, scanning a query for a delimiter cannot guarantee the prevention of piggy-backed query SQLi attack.

D. Union Query

Union query SQLi attack is a type of attack in which hackers try to bypass authentication and/or extract data from a back-end database by merging two separate SQL SELECT queries, which have nothing in common, using UNION SELECT statement. The example below shows how the union query SQLi attack can be sent by hackers through a user interface to the related back-end database.

SELECT "accounts" FROM "users" WHERE username = "" "UNION SELECT "cardNumber" from "CreditCard" where accountNumber = "40654"--" AND password ="" AND pin =""

In the above example, the first "SELECT" query returns the null set, while the second "SELECT" query returns "cardNumber" for account "40654". Thus, after running the above statement, the "cardNumber" for account "40654" will be displayed on the hacker's screen.

Signature: Given the above example and explanations, the signature for the union query SQLi attacks is the UNION and UNION SELECT meta characters of SQL.

Prevention: strictly validating user inputs on user side and blocking multiple query executions in a single statement on database side are the two effective countermeasures for preventing the union query SQLi attacks.

E. Stored Procedures

Stored procedure SQLi attack is a type of attack in which hackers aim to perform privilege escalation, denial of service and/or remote commands using stored procedures related to a given type of database. The example below displays how the stored procedure SQLi attack can be sent by hackers through user interface to the back-end database.

SELECT "accounts" FROM "users" WHERE username = "LIKE '1' or '1'='1'" AND password = " ; exec master.dbo.xp_cmdshell 'dir c:\temp*.sql ' SHUTDOWN; --" AND pin = ""

In the above example, "exec master.dbo.xp_cmdshell 'dir c:\temp*.sql '" statement displays all the file in "c:\temp" directory that have a ".sql" extension to the hacker. This command then follows by executing the "SHUTDOWN" command with which the back-end database will be shutdown.

Signature: Taking into account the above example and explanations, the signature for the stored procedure SQLi attack is as same as the piggy-backed query SQLi attacks, given that both attacks are similar in using delimiter ";" and stored procedure keywords such as (SHUTDOWN, exec, xp_cmdshell(), sp_execwebtask ()) .

Prevention: Strictly validating user inputs on user side, using a low privileged account to run a database on database side, executing stored procedures with a safe interface on database side, and giving proper roles and privileges to stored procedures, which are being used in a user application form, are some countermeasures to block and/or reduce the chance of a successful stored procedure SQLi attack.

F. Inference

Inference SQLi attack is a type of attack in which hackers aim to identify injectable parameters, extract data from a database and/or determine database scheme by testing the possible vulnerabilities of a back-end database when no data returns to an end-user from a slightly secured website.

There are two popular inference SQLi attacks as follows.

1) Inference Blind SQLi Attack

Inference blind SQLi attack is an error-based attack in which hackers try to force a back-end database to throw an error message by asking true-false questions.

The example below reveals how the inference blind SQLi attack can be sent by hackers through a user interface to the back-end database.

SELECT "accounts" FROM "users" WHERE username = "LIKE '1' or '1'='1'; IF SYSTEM_USER = 'sa' SELECT 1/0 ELSE 5; --" AND password ="" AND pin =""

In the above example, hackers try to obtain the back-end database behaviour by sending an "IF ELSE" statement in which a division by zero will be executed on the database if the current user is a system administrator, "sa". As division by zero is undefined and has no meaning, running the above query forces the back-end database to throw an error if the current user is a system administrator, "sa", or else a valid instruction would be executed, "ELSE 5".

2) Inference Timing SQLi Attack

Inference timing SQLi attack is a time-based attack in which hackers employ time delay in order to make difference between true and false responses from a back-end database. For instance, a true response received from a given database means that the time delay was executed successfully while a false response means hackers weren't successful to execute the time delay.

The following example represents how the inference timing SQLi attack can be generated by hackers via a user interface to the related database.

SELECT "accounts" FROM "users" WHERE username = "john1390 and 1>0 WAITFOR 5 --" AND password = "" AND pin = ""

In the above example, hackers try to work out the behaviour of the back-end database by injecting an always true

statement, “1>0”, along with “WAITFOR” keyword, which is a popular keyword for issuing the inference timing SQLi attack.

Signature: Addressing the above example and explanations, the possible signatures for the inference SQLi attack, including both interface blind and interface timing, are: using delimiter “;” with AND operator, using IF ELSE conditional operator, and using WAITFOR.

Prevention: Strictly validating user inputs on user side, carefully crafting error messages return from databases on database side as well as patching/hardening databases can prevent the inference SQLi attacks.

G. Alternate Encoding

Alternate encoding SQLi attack is a type of attack in which hackers try to obscure their injected commands by using encodings techniques such as ASCII, hexadecimal, Unicode character encoding, etc. The following example reveals how the alternate encoding SQLi attack can be generated by hackers and then sent via a user interface to the back-end database.

SELECT “accounts” FROM “users” WHERE username = “john1390; exec (char (Ox73687574646j776e)) --" AND password =”” AND pin = “”

In the above example, hackers try to obscure their attacks by using ASCII hexadecimal alternate of “char (Ox73687574646j776e)”, which is equal to “SHUTDOWN. When the above SQL query bypasses the validation and reaches the back-end database, it will be translated to “SHUTDOWN” and run straightaway on the back-end database. The same technique can be employed by hackers to find replacement for any bad characters, for instance single quote (“ ‘ ”) or “ ' Or 1=1 -- ”, in order to cover their attack and circumvent the validation.

Signature: Given the above example and explanations, the possible signatures for the alternate encoding SQLi attack are: exec (), Char (), ASCII (), BIN (), HEX (), UNHEX (), BASE64 (), DEC (), ROT13 (), etc.

Prevention: Strictly validating user inputs on user side, for instance prohibiting any usage of meta-characters e.g. “Char ()”, etc. and treating all meta-characters as normal characters on database side can prevent the alternate encoding SQLi attack.

All the above types of SQLi attack along with their signatures and preventions are listed in Table I.

Related work for SQLi attacks detection and prevention are addressed in the next section.

III. Related Work For SQL Injection Attack; Detection and Prevention

In this section, we address the earlier work related to the SQLi attack detection and prevention techniques as follows.

Authors in [2] employed the Support Vector Machine (SVM) for SQLi attack classification and detection. The performance of their proposed technique was measured based on the different parameters such as accuracy, detection time, training time, True Positive Rate (TPR), True Negative Rate (TNR), False Positive Rate (FPR) and False Negative Rate (FNR). Based on the captured results their proposed model shows 96.47% accuracy in SQLi attack detection.

Authors in paper [3] proposed a static analysis tool for checking Java Database Connectivity (JDBC) in order to verify the correctness of dynamically generated query strings. JDBC is the JavaSoft application of a standard application programming interface (API) that allows Java programs to first connect to and then access to database management systems. Based on their captured results, their proposed JDBC checker either flags potential errors or verify their absence in dynamically generated SQL queries with low false positive rate.

In order to detect the SQLi attacks, [4] proposed a query tokenisation algorithm assuming that there is no way for someone performing SQLi attack without inserting space, single quote, and/or double dashes in a query. By taking into account this assumption, they designed two arrays: one for the original query and one for the injected query, where each element is a token obtained from the related query. At the end, they captured the lengths of the resulting arrays from two queries and compared them having said that if two arrays have the same length there is no SQL injection otherwise there is injection.

In order to prevent SQLi attack, [5] proposed Random4 encryption algorithm in which user inputs, e.g. usernames and passwords, convert into cipher text using a lookup table. The encrypted keys can then be stored in database and compared with the input values received from users in order to prevent illegal access through SQL injection. For performance evaluation, they used password cracking techniques such as brute force attack and dictionary attacks to crack the keys stored in the database. They have also compared their proposal with the existing algorithms such as AMNESIA [8], SQL rand [10], SQL DOM [9], WAVES [11], and SQL check [12] in terms of encoding, detection and prevention.

Authors in [6] proposed a Service Based SQL Injection Detection (SBSQLID) method for detection of SQL injection vulnerabilities which includes three elements of input validator, query analyser, and error service. The proposed SBSQLID positioned between a given application server and the associated database. The input validator retrieves the user inputs from the web application form and passes them into the set of injection characters for pattern matching. Thus, if the pattern matching returns false, the users would be able to work with the web application otherwise they would be disallowed. After validating the user inputs, the user inputs will be passed to the query analyser for syntactic and semantic structure verification. The last elements of their proposal is an error service with which any error messages form the database server will be generalised and then sent back to the application server in order to stop attackers for receiving any Meta data information from the database.

TABLE I
SQL INJECTION ATTACK TYPES, SIGNATURES, PREVENTIONS

No	Type of SQLi attack	Signature	Prevention
1	Tautologies	‘, OR, =, like, select	-Strictly validating user inputs on user side -Blocking queries containing tautological condition WHERE clauses on database side
2	Illegal/logically incorrect queries	invalid conversions (CONVERT (TYPE)), incorrect logics, AND, ORDERBY, ‘	-Strictly validating user inputs on user side -Stopping and/or sanitising the generated error messages (e.g. logical errors, type errors and syntax errors) from a given database
3	Piggy-backed query	;	-Strictly validating user inputs on user side -Avoiding multiple statement executions on a database by scanning all queries for delimiter “;”on database side
4	Union queries	UNION, UNION SELECT	-Strictly validating user inputs on user side -Blocking multiple query executions in a single statement on database side
5	Stored procedures	;, Stored procedure keywords (SHUTDOWN, exec, xp_cmdshell(), sp_execwebtask())	-Strictly validating user inputs on user side -Using a low privileged account to run a database on database side -Executing stored procedures with a safe interface on database side -Giving proper roles and privileges to stored procedures being used in a user application form
6	Inference SQLi attack	;, AND, IF ELSE, WAITFOR	-Strictly validating user inputs on user side -Carefully crafting error messages return from databases on database side -Patching/hardening databases
7	Alternate encoding	exec (), Char (), ASCII (), BIN (), HEX (), UNHEX (), BASE64 (), DEC (), ROT13 ()	-Strictly validating user inputs on user side, for instance prohibiting any usage of meta-characters e.g. “Char ()” , etc. -Treating all meta-characters as normal characters on database side

Authors in [7] proposed a translation and validation-based solution for SQL injection attacks named as TransSQL, with which SQL requests are automatically translated to the Lightweight Directory Access Protocol (LDAP) equivalent requests. LDAP is a protocol for accessing directories while SQL is a query language for databases. Both queries, SQL and LDAP-equivalent, are then executed on SQL database and LDAP database, respectively. At the end, TransSQL checks the difference in responses from both databases in order to detect and then block SQL injection attempts.

Authors in [8] proposed AMENSIA, Analysis and Monitoring for NEutralising SQL Injection Attacks, to detect and prevent SQLi attacks by combining static analysis and runtime monitoring. The static analysis was used in order to analyse the web-application codes and automatically build a model of the legitimate queries that a given application can generate. Then, the runtime monitoring was employed in order to monitor all dynamically generated queries and check if they are different from the static generated model. At the end, the queries that violate the static model were classified as SQLi attacks and prevented from accessing the database.

Authors in [9] proposed SQL Domain Object Model (DOM) for compile time checking instead of runtime checking of dynamic SQL statements. Their proposed SQL DOM is a set of classes that are shortly typed to a database scheme and is employed in order to generate dynamic SQL statements. Using these classes, the application developer is able to build dynamic SQL statements through manipulation of objects, which are strongly typed to the database, without the need for string manipulations.

In order to detect and prevent SQLi attacks, [10] proposed a randomised SQL query language, SQLrand, with which the SQL standard keywords were manipulated by attaching a randomised as well as a hard to guess integer to them. To achieve probability and security, their proposal consists of a proxy server sits between the client and the database which receives randomised SQL quires from the client and de-randomised them before they pass to the database. Based on the captured results, the latency overhead, that imposed on each query using SQLrand, is negligible thus doesn’t sacrifice the performance.

Authors in [11] proposed a Web Application Vulnerability and Error Scanner (WAVES) as a security assessment tool in order to identify poor coding practices that render web applications vulnerable to attacks such as SQL injections and cross-site scripting. Having said that their goal was to adopt software-engineering techniques to design a security assessment tool, a number of software testing techniques including dynamic analysis, black-box testing, fault injection, and behaviour monitoring was described and took into account in WAVES tool. At the end, WAVES was compared with other tools. The comparison shows that WAVES is a feasible platform for assessing web application security.

Authors in [12] proposed SQLCHECK, which is a runtime checking algorithm, to prevent SQLi attacks. Their proposed algorithm was evaluated on real-world web applications with real-world attack data as input value. Based on the captured results, the SQLCHECK produces no false negative or false positive. It also has low run-time overhead and can be applied straightforwardly to web applications written in different languages.

In [16], we proposed a neural network model for SQLi attack detection where our proposed technique was successful to classify a given URL as either a benign URL or a malicious URL by taking into the account the popular SQLi attack keywords and the popular SQLi attack URL patterns. In this paper, we improve our earlier work [16] by adding another level of intelligence to our previous proposed neural network model in which our new proposal is able to 1) identify the malicious URLs from the benign URLs 2) detect the type of SQLi attack for the malicious URLs.

Our proposed neural network-based model for detection and classification of SQLi attacks is discussed in the next section.

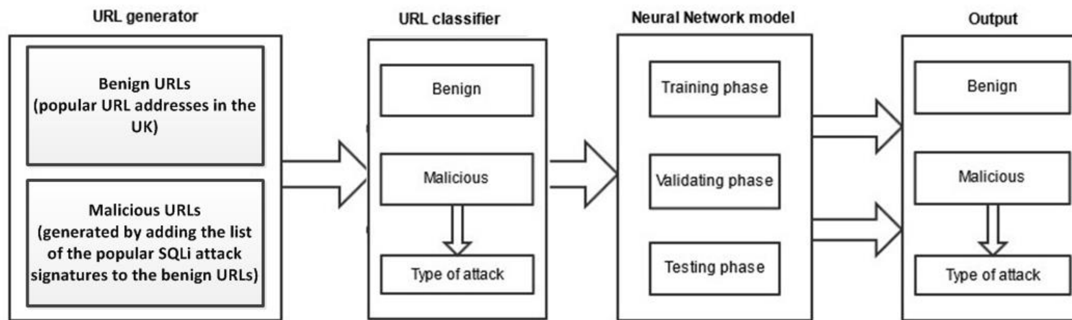

Fig. 1 Components of the proposed neural network-based model for detection and classification of SQLi attacks

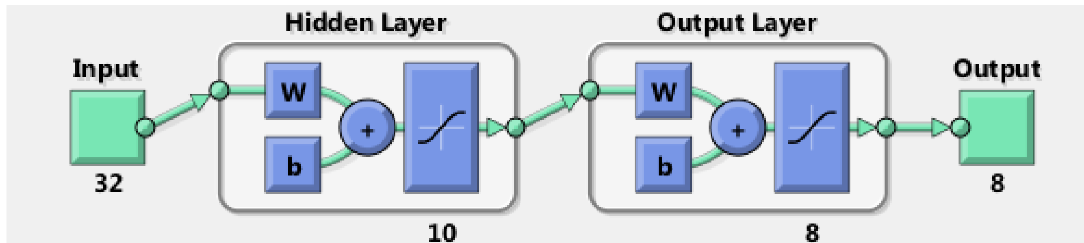

Fig. 2 Network Architecture for the Neural Network component of the proposed model

IV. Proposed Neural Network-Based Model for Detection and Classification of SQL Injection Attacks

Our proposed neural network-based model for detection and classification of SQLi attacks is built from three main elements of a URL generator, a URL classifier, and a neural network model. All the components of the proposed model are depicted in Fig. 1 and detailed as follows.

A. A URL Generator

The URL generator part of the proposed neural network-based model for detection and classification of SQLi attacks has two components of "Benign URLs" and "Malicious URLs", Fig. 1. The benign URLs are the real URLs which are the most popular URL addresses in the UK and haven been captured from [17]. The malicious URLs are generated by adding the list of the popular SQLi attack signatures from Table I to the benign URLs, where seven popular SQLi attacks and their signatures haven been taken into account. As it is detailed in Table I, seven popular SQLi attacks comprising: Tautologies, Illegal/logically incorrect queries, Piggy-backed query, Union queries, Stored procedures, Inference SQLi attack, and Alternate encoding attacks, with different signatures such as: "', OR, =, LIKE, SELECT" signatures for Tautologies, ";" signature for Piggy-backed query, "UNION, UNION SELECT" signatures for Union queries, etc.

B. A URL Classifier

The URL classifier part of the proposed neural network-based model for detection and classification of SQLi attacks has three components of "Benign", "Malicious" and "Type of attack". Therefore, the URL classifier part of the proposed model is responsible to 1) classify each generated URL from the URL generator in the previous stage to either a benign URL or malicious URL and 2) detect the type of attack for each malicious URL. The type of attack for the malicious URLs could be: "Tautologies", "Illegal/logically" "Incorrect Queries", "Piggy-backed Query", "Union Queries", "Stored Procedures", "Inference SQLi Attack", or "Alternate Encoding". Please refer to Table I for more specification for each type of SQLi attack in terms of signatures and prevention techniques.

The URL classifier part of the proposed neural network-based model for detection and classification of SQLi attacks is also in charge of converting the generated URLs, malicious and benign, into strings of logic, 1 as true and 0 as false, where "1" represent the malicious URLs and the type of attack while "0" represent the benign URLs. The mathematical explanations of the above assumptions are detailed as follows.

Let a URL characteristic r_i generated by the URL generator is defined by a random variable R_i:

$$R_i = \begin{cases} 1, \text{if discovered by the SQLi signature detectors} \\ 0, \text{if not discovered by the SQLi signature detector} \end{cases} \quad (1)$$

Let C be a random variable representing the generated URL class, malicious or benign:

$$C \epsilon \{\text{malicious, benign}\}$$

Every generated URL (malicious and benign) is assigned a vector defined by $r^{-} = (r_1, r_2, , \ldots, r_n)$ with r_i being the result of the i-th random variable R_i.

Let a malicious URL characteristic t_i generated by the URL generator is defined by a random variable T_i:

$$T_i = \begin{cases} 1, \text{if discovered by the SQLi attack type detectors} \\ 0, \text{if not discovered by the SQLi atatck type detector} \end{cases} \quad (2)$$

International Journal of Information, Control and Computer Sciences
ISSN: 2517-9942
Vol:9, No:6, 2015

Let D be a random variable representing the type of the malicious URLs: "Tautologies", "Illegal/logically Incorrect Queries", "Piggy-backed Query", "Union Queries", "Stored Procedures", "Inference SQLi Attack", or "Alternate Encoding":

$D\epsilon$ {Tautologies, Illegal/logically incorrect queries, Piggy-backed query, Union queries, Stored procedures, Inference SQLi attack, Alternate encoding}

Every generated malicious URL is assigned a vector defined by $t^{-} = (t_1, t_2,, \dots, t_n)$ with t_i being the result of the *t-th* random variable R_i.

C. A Neural Network (NN) Model

The neural network model part of the proposed neural network-based model for detection and classification of SQLi attacks comprises *n* layers (*n* hidden layers or neurons) with *x* number of input nodes and *y* number of output nodes. The model uses the benign and malicious URLs, which are generated by the URL generator and classified by the URL classifier from the previous sections, Fig. 1, for training, validating, and testing. Thus, as it is depicted in Fig. 1, the neural network model has three components of training, validating and testing. For simplicity these three components are put together in two groups of training components and validating/testing components and described as follows.

1) Training Components

- 'Input' matrix: this matrix includes the data that our proposed neural network model uses in training phase. It comprises all the benign and malicious URLs generated by the URL generator part of our proposed model.
- 'Target' matrix: this matrix includes all the decisions including (malicious or benign) as well as the type of the SQLi attack (Tautologies, Illegal/logically incorrect queries, Piggy-backed query, Union queries, Stored procedures, Inference SQLi attack, Alternate encoding) for each string of data stored in 'Input' matrix.
- Fitness network: this is the neural network with *n* neurons in *n* hidden layer in which the data from 'Input' and 'Target' matrixes will be used for training, validating and testing, consequently.

2) Validating/Testing Components

- 'Sample' matrix: this matrix contains sample data from 'Input' matrix discussed in the previous section. The trained neural network model uses 'Sample' data as input in validation phase.
- 'Output' matrix: this matrix contains output data for the data represented in 'Sample' matrix. The trained neural network model predicts the output value for 'Sample' matrix and stores it in 'Output' matrix. Thus, it predicts a given URL either being a benign URL or a malicious URL. It also predicts the type of the SQLi attack for the malicious URL.
- The implementation of the proposed neural network-based model for detection and classification of SQLi attacks is discussed in the next section.

V. Implementations

In this paper, a neural network-based model for detection and classification of SQLi attacks is proposed which includes three elements of a URL generator, a URL classifier and a neural network model, Fig. 1. All three elements are implemented as follows.

A. A URL Generator

As it was discussed in the previous section, the URL generator element of the proposed neural network-based model for detection and classification of SQLi attacks includes two main components: 1) "Benign URLs" and 2) "Malicious URLs", Fig. 1. The "Benign URLs" include a list of the most popular URLs in the UK while the "Malicious URLs" are generated by adding the SQLi attack signatures to the benign URLs.

For the "Benign URLs", we have taken into account the top 500 popular website addresses in the UK, which has been captured from [17], while the malicious URLs are generated by adding the SQLi attack signatures from Table I to the benign URLs.

For instance, addressing the unique signatures for "Illegal/logically Incorrect Queries" from Table I, a generated malicious URL could be a benign URL that had been combined with word "convert" and word "int", or adding " ', OR, =, like, select" signature to any benign URL could generate a malicious URL which could be identified as a "Tautologies" attack, etc.

The total number of the benign URLs in our implemented scenario is 500. The total number of the malicious URLs comes to 12,500 after considering all the signatures from seven popular SQLi attacks, Table I.

B. A URL Classifier

As it was discussed in the previous section, the classifier part of our proposed neural network-based model for detection and classification of SQLi attacks is in charge of: 1) classifying each URL generated by the URL generator to either a benign URL or a malicious URL and 2) detecting the type of SQLi attack for each malicious URL. These two tasks have been coded and implemented based on the strings of logic, where 1 represents as true/malicious and 0 represents as false/benign, by allocating two vectors, $r^{-} = (r_0, r_1,, \dots, r_{31})$ and $t^{-} = (t_0, t_1,, \dots, t_7)$, to each URL.

For instance, addressing the signature for "Inference SQLi Attack" from Table I, if a URL includes: ";", "and", "if" and "else", it will be classified as a malicious URL with "00000000000101000000110000000000" value for r^{-} in which r_{13} represents ";" r_{11} represents "and" r_{20} represents "if" r_{21} represents "else". Moreover, as it is an "Inference SQLi attack", attack type 6, the value for t^{-} vector is "00000010", etc. The components of the r^{-} and t^{-} vectors have been revealed in Tables II and III, respectively.

C. A Neural Network (NN) Model

As it was discussed in the previous section, the neural network model of the proposed scheme includes three

components of training, validating, and testing. It comprises of *n* layers (*n* hidden layers or *n* neurons) with *x* number of input nodes and *y* number of output nodes and uses the benign and malicious URLs, which are generated by the URL generator and classified by the URL classifier, for training, validating and testing.

TABLE II
ASSIGNED VECTORS TO THE SQLI ATTACK SIGNATURES

Vectors	SQLi attack signatures
r_0	‘
r_1	or
r_2	=
r_3	like
r_4	select
r_5	convert
r_6	int
r_7	char
r_8	varchar
r_9	nvarchar
r_{10}	incorrect logics
r_{11}	and
r_{12}	orderby
r_{13}	;
r_{14}	union
r_{15}	union select
r_{16}	shutdown
r_{17}	exec
r_{18}	xp_cmdshell ()
r_{19}	sp_execwebtask ()
r_{20}	if
r_{21}	else
r_{22}	waitfor
r_{23}	--
r_{24}	ascii ()
r_{25}	bin ()
r_{26}	hex ()
r_{27}	unhex ()
r_{28}	base64 ()
r_{29}	dec ()
r_{30}	rot13 ()
r_{31}	*

TABLE III
ASSIGNED VECTORS TO THE SQLI ATTACK TYPE

Vectors	SQLi attack type
t_0	Benign
t_1	Tautologies
t_2	Illegal/logically incorrect queries
t_3	Piggy-backed query
t_4	Union queries
t_5	Stored procedures
t_6	Inference SQLi attack
t_7	Alternate encoding

In our implementations, we have implemented a neural network model of 10 layers (10 hidden nodes or 10 neurons) in which 70%, 15% and 15% of the total benign and malicious URLs for training, validating and testing, respectively. MATLAB [15], which is popular software for the numerical calculations and formulas with the vast library of functions and algorithms, is used in order to develop, train, validate and test the proposed neural network model. The training and validating components of the proposed model are configured as follows.

1) Training Components

- ‘Input’ matrix: this matrix is a logical *n x32* matrix where the data represented in strings of logics; 1 as true and 0 as false.
- ‘Target’ matrix: this matrix is a logical *nx8* matrix where the data represented in logics; 1 as malicious and 0 as benign.
- Fitness network: this is the neural network with 10 neurons in hidden layer in which 70%, 15%, and 15% of the data from ‘Input’ and ‘Target’ matrixes will be used for training, validating and testing, respectively.

2) Validating/Testing Components

- ‘Sample’ matrix: this matrix is a logical *nx32* matrix contains sample data from the ‘Input’ matrix.
- ‘Output’ matrix: this matrix is a logical *nx8* matrix contains output data for the data represented in ‘Sample’ matrix. The trained neural network predicts the output value for ‘Sample’ matrix, in terms of a URL being benign or malicious and the type of SQLi attack for a malicious URL, and stores it in the ‘Output’ matrix.

The captured results are discussed in the next section.

VI. RESULTS

As it was discussed in the previous section, the proposed neural network-based model for detection and classification of the SQLi attacks includes three main elements of: 1) a URL generator, in order to generate thousands of benign and malicious URLs, 2) a URL classifier, in order to classify a given URL as a malicious URL or a benign URL and also to identify the type of SQLi attack for a given malicious URL, and 3) a neural network model.

In order to capture the performance of the proposed neural network-based model for detection and classification of SQLi attacks, we have considered 13,000 URLs, which includes 500 benign URLs and 12,500 malicious URLs.

The 500 benign URLs are the real URLs which are the top 500 popular website addresses in the UK while the malicious URLs are generated by adding the SQLi signatures from Table I. into the benign URLs using PHP. The 13,000 URLS, including 500 benign URLs and 12,500 malicious URLs, are then classified into either benign URLs or malicious URLs by using the URL classifier part of the proposed model. The classifier also detects the type of the SQLi for each malicious URL based on the seven popular SQLi attack types form Tbale.2. At the end we train, evaluate, and then test the neural network model using MATLAB [15]. In our implementations, the neural network model has 10 hidden layers, 32 input features, 8 output layer, and 8 output features, Fig. 2. The captured results are as follows.

The confusion matrices for training, validating, and testing

are shown in Figs. 3, 4, 5, respectively. In Fig. 6, we have also captured the total confusion matrix in which the training, validation, and testing matrices are all combined.

The numbers of the correct response are shown in green squares while the numbers of the incorrect response are shown in the red squares. The grey squares at the end of each row and each column illustrate the percentages of the accuracies (upper numbers) and inaccuracies (bottom numbers) for output classes and target classes, respectively. The lower-right blue squares illustrate the overall accuracies (upper numbers) and overall inaccuracies (bottom numbers) by taking into account the accuracies and inaccuracies in output classes and target classes.

As they are depicted in Figs. 3-5 and based on the configurations in our implemented scenario, the 13,000 URLs (500 benign URLs plus 12,500 malicious URLs) are scattered in three phases of training, validating, and testing with distribution rates of 70%, 15%, and 15%, respectively. By taking into account the distribution rates in three phases:

- The training phase receives 9,100 out of 13,000 URLs. This includes 0 benign URLs and 9,100 malicious URLs while the former is 0%, and the latter is 96.0% of the total URLs used in this phase. The malicious URLs consisting of 358 URLs from SQLi type2, 1752 URLs from SQLi type3, 349 URLs from SQLi type4, 691 URLs from SQLi type5, 1071 URLs from SQLi type6, 1736 URLs from SQLi type7, and 2775 URLs from SQLi type8.
- The validating phase receives 1,950 out of 13,000 URLs. This includes 0 benign URLs and 1,950 malicious URLs while the former is 0%, and the latter is 96.4% of the total URLs used in this phase. The malicious URLs consisting of 85 URLs from SQLi type2, 374 URLs from SQLi type3, 69 URLs from SQLi type4, 157 URLs from SQLi type5, 202 URLs from SQLi type6, 386 URLs from SQLi type7, and 606 URLs from SQLi type8.
- The testing phase receives 1,950 out of 13,000 URLs. This includes 0 benign URLs and 1,950 malicious URLs while the former is 0%, and the latter is 96.9% of the total URLs used in this phase. The malicious URLs consisting of 57 URLs from SQLi type2, 374 URLs from SQLi type3, 78 URLs from SQLi type4, 156 URLs from SQLi type5, 227 URLs from SQLi type6, 378 URLs from SQLi type7, and 619 URLs from SQLi type8.

Addressing the captured percentages of the incorrect responses in the red squares, Figs. 3-6, as well as the overall percentages of the accuracies and inaccurate in blue squares, we can say that the outputs for all three phases are correct. This includes overall 96% accuracy and 4% of inaccuracy for training phase, overall 96.4% accuracy and 3.6% of inaccuracy for validation phase, and overall 96.9% accuracy and 3.1% of inaccuracy for testing phase. Therefore, the proposed model is trained correct and hence, it performs correct.

The Receiver Operating Characteristic (ROC) curves for training, validating, and testing is shown in Figs. 7, 8, 9, respectively. We have also captured the ROC curve for all three sets of training, validating, and testing in Fig. 10. The colored lines in each axis represent the ROC curves for each of three sets. The ROC curve is a plot of the true-positive rate (sensitivity) against the false-positive rate (specificity).

In our implementations, the true-positive rate (sensitivity) is the percentages of the correct classifications for all 13,000 URLs including benign URLs and malicious URLs, where class1 is a class allocated to the benign URLs while class2 to class8 are allocated to type 2 to type 8 of SQLi attacks, all correspondingly. Additionally, the false-positive rate (specificity) is the percentages of the incorrect classifications for any of 13,000 URLs. A perfect test would show points in the upper-left corner, with 100% sensitivity (i.e. classifying/predicting all the URLs in their correct classes of class1 to class8) and 100% specificity (i.e. not classifying/predicting any URL in any wrong classes).

Addressing Figs. 7-10, the proposed model has a good performance in terms of true positive rate as well as false positive rate which have only 0.1 incorrectness for class 8 (type 8 of SQLi attacks).

Network performance is measured in terms of mean squared error, and shown in log scale. It rapidly decreased as the network was trained. Mean Squared Error is the average squared difference between output and targets. Thus, lower values are better and zero means no error. Fig. 11 shows how the network performance improved for each three sets. The version of the network that did best on is at the end of three phases of training, validation and testing.

Training Confusion Matrix

Output Class									
1	0 0.0%	0 0.0%	0 0.0%	0 0.0%	0 0.0%	0 0.0%	0 0.0%	0 0.0%	NaN% NaN%
2	0 0.0%	358 3.9%	0 0.0%	0 0.0%	0 0.0%	0 0.0%	0 0.0%	0 0.0%	100% 0.0%
3	0 0.0%	0 0.0%	1752 19.3%	0 0.0%	0 0.0%	0 0.0%	0 0.0%	0 0.0%	100% 0.0%
4	0 0.0%	0 0.0%	0 0.0%	349 3.8%	0 0.0%	0 0.0%	0 0.0%	0 0.0%	100% 0.0%
5	368 4.0%	0 0.0%	0 0.0%	0 0.0%	691 7.6%	0 0.0%	0 0.0%	0 0.0%	65.3% 34.7%
6	0 0.0%	0 0.0%	0 0.0%	0 0.0%	0 0.0%	1071 11.8%	0 0.0%	0 0.0%	100% 0.0%
7	0 0.0%	0 0.0%	0 0.0%	0 0.0%	0 0.0%	0 0.0%	1736 19.1%	0 0.0%	100% 0.0%
8	0 0.0%	0 0.0%	0 0.0%	0 0.0%	0 0.0%	0 0.0%	0 0.0%	2775 30.5%	100% 0.0%
	0.0% 100%	100% 0.0%	100% 0.0%	100% 0.0%	100% 0.0%	100% 0.0%	100% 0.0%	100% 0.0%	96.0% 4.0%

Target Class

Fig. 3 Confusion matrix for training

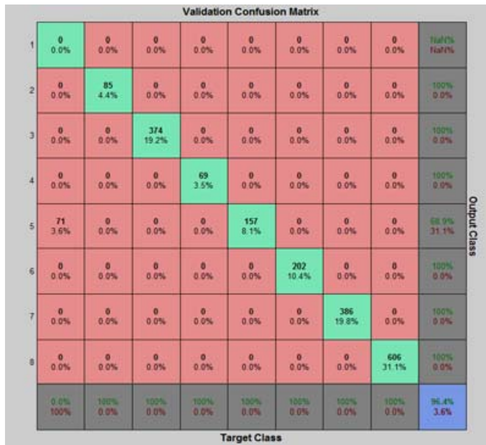

Fig. 4 Confusion matrix for validating

Fig. 5 Confusion matrix for testing

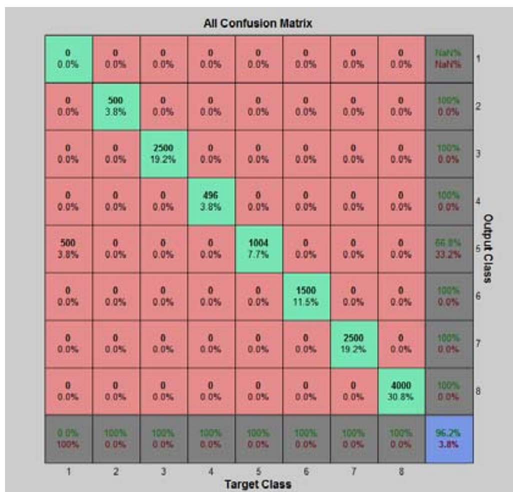

Fig. 6 Confusion matrix for all (training, validating, and testing)

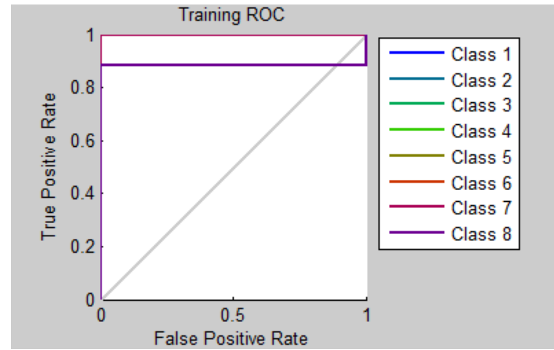

Fig. 7 ROC curve for training

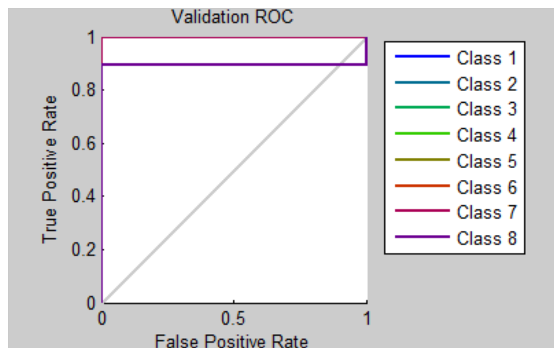

Fig. 8 ROC curve for validation

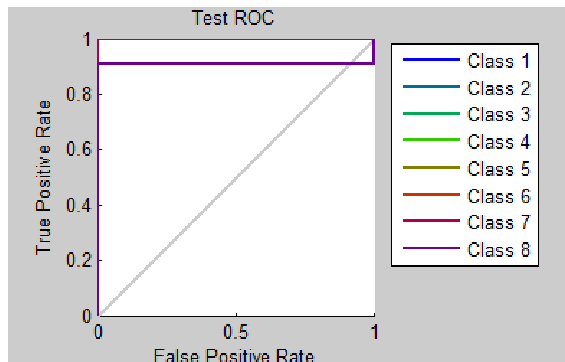

Fig. 9 ROC curve for testing

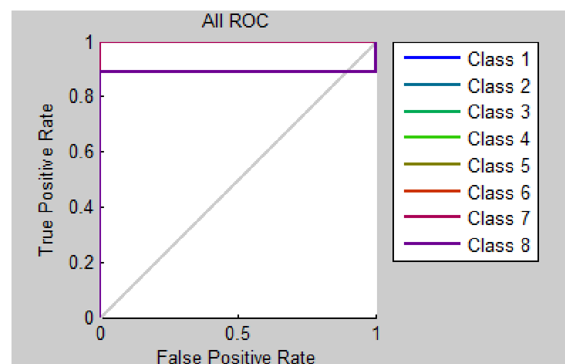

Fig. 10 ROC curve for all ((training, validating, and testing))

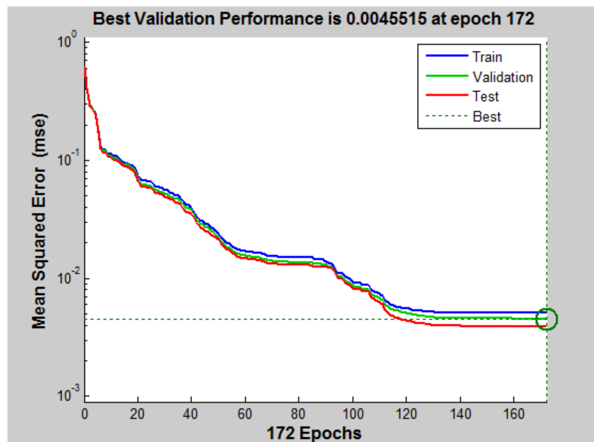

Fig. 11 Network Performance

VII. Conclusion

In this paper, a neural network-based model for detection and classification of SQLi attacks is proposed. The proposed model includes three components of: a URL generator, a URL classification, and a neural network model. The neural network part of the proposed model uses the benign and malicious URLs, which are generated by the URL generator and then classified by the URL classifier into seven popular SQLi attack types, during the three phases of training, validating, and testing.

The proposed neural network model is successful to 1) detect the malicious URLs from the benign URLs 2) classify the malicious URLs based on their signatures into different classes by taking into account the signatures of the seven popular SQLi attacks. Additionally, based on the captured results, the proposed neural network model for detection and classification of the SQLi attack shows a good performance in terms of accuracy, true-positive rate as well as false-positive rate.

Acknowledgment

The author would wish to acknowledge the support of the Abertay University for funding this work.

References

[1] W. G. Halfond, J. Viegas, and A. Orso, “A Classification of SQL-Injection Attacks and countermeasures”, in Proc. of the Internet Symposium on Secure Software Engineering (ISSSE 2006), Mar. 2006.
[2] R. Romil, R. Shailendra, "SQL injection attack Detection using SVM", in International Journal of Computer Applications, V.42, N.13, March 2012.
[3] C. Gould, Z. Su, and P. Devanbu, "JDBC checker: A static analysis tool for SQL/JDBC applications," 2004, pp. 697- 698.
[4] N. A. Lambert and K. Song Lin, “Use of Query Tokenization to detect and prevent SQL Injection Attacks”, IEEE, 2010.
[5] A. Srinivas, G. Narayan, S. Ram, “Random4: An Application Specific Randomized Encryption Algorithm to prevent SQL injection, in Trust, Security and Privacy in Computing and Communications (TrustCom)”, 2012 IEEE 11th International Conference, pp.no. 1327 – 133, 25-27 June 2012.
[6] V. Shanmughaneethi, C. Emilin Shyni and S. Swamynathan, “SBSQLID: Securing Web Applications with Service Based SQL Injection Detection” 2009 International Conference on Advances in Computing, Control, and Telecommunication Technologies, 978-0-7695-3915-7/09, 2009 IEEE.
[7] K. Zhang, Ch. Lin, Sh. Chen, Y. Hwang, H. Huang, and F. Hsu, “TransSQL: A Translation and Validation-based Solution for SQL-Injection Attacks”, First International Conference on Robot, Vision and Signal Processing, IEEE, 2011.
[8] W. G. Halfond and A. Orso, “AMNESIA: Analysis and Monitoring for NEutralizing SQL-Injection Attacks”, In Proceedings of the IEEE and ACM International Conference on Automated Software Engineering (ASE 2005), Long Beach, CA, USA, Nov 2005.
[9] R. McClure and I. Kruger, “SQL DOM: Compile Time Checking of Dynamic SQL Statements”, In Proceedings of the 27th International Conference on Software Engineering (ICSE 05), pages 88–96, 2005.
[10] Stephen W. Boyd, Angelos D. Keromytis, “SQLrand: Preventing SQL injection Attacks”.
[11] Y. Huang, S. Huang, T. Lin, and C. Tsai, “Web Application Security Assessment by Fault Injection and Behavior Monitoring”, In Proceedings of the 11th International World Wide Web Conference (WWW 03), May 2003.
[12] Z. Su and G. Wassermann, “The Essence of Command Injection Attacks in Web Applications”, In the 33rd Annual Symposium on Principles of Programming Languages (POPL 2006), Jan. 2006.
[13] Open Web Application Security Project (OWASP), available at: https://www.owasp.org/index.php/About_OWASP (retrieved: Dec, 2014).
[14] Open Web Application Security Project (OWASP) top 10, available at: https://www.owasp.org/index.php/Top_10_2013-Top_10 (retrieved: Dec, 2014).
[15] MATLAB R2014a, available at: http://www.mathworks.co.uk (retrieved: Dec, 2014).
[16] N. Moradpoor, “Employing Neural Networks for the Detection of SQL Injection Attack”, In The 7th conference on Security of Information and Networks (SIN2014), Sep 2014.
[17] Alexa, Bringing Information into Focus, available at: http://www.alexa.com/topsites/countries/GB (retrieved: Dec, 2014).
[18] Introduction to SQL, available at: http://www.w3schools.com/sql/sql_intro.asp (retrieved: June, 2015).