SECURITY VULNERABILITY IN ON-LINE APPLICATIONS: ANALYSIS, ANOMALY DETECTION AND PREVENTION OF ATTACK

Thesis submitted to

Cochin University of Science and Technology

in partial fulfilment of the requirements

for the award of the degree of

Doctor of Philosophy

under the Faculty of Technology

by

Thresiamma K. George

Under the guidance of

Dr. K. Poulose Jacob (Research Guide)
Ex. Pro Vice Chancellor,
Professor of Computer Science

Dr. Rekha K. James (Co-Guide)
Professor, Division of Electronics
School of Engineering

Department of Computer Science
Cochin University of Science and Technology
Kochi – 22

July 2017

Security Vulnerability in On-Line Applications: Analysis, Anomaly Detection and Prevention of Attack

Ph.D. Thesis under the Faculty of Technology

By

Thresiamma K. George
Research Scholar
Department of Computer Science
Cochin University of Science and Technology
Kochi, India 682022

Research Advisors

Dr. K. Poulose Jacob (Research Guide)
Ex. Pro Vice Chancellor,
Professor of Computer Science
Cochin University of Science and Technology
Cochin-682022

Dr. Rekha K. James (Co-Guide)
Professor, Division of Electronics
School of Engineering
Cochin University of Science and Technology
Cochin-682022

July 2017

COCHIN UNIVERSITY OF SCIENCE AND TECHNOLOGY
Kochi – 682 022

Certificate

This is to certify that the thesis entitled **"Security Vulnerability in On-Line Applications: Analysis, Anomaly Detection and Prevention of Attack"** is an authentic record of research work carried out by **Mrs. Thresiamma K. George,** under our supervision in the Department of Computer Science, Cochin University of Science and Technology, and further that no part thereof has been presented before for the award of any other degree. All the relevant corrections and modifications suggested by the audience during the presynopsis seminar and recommended by the doctoral committee of the candidate have been incorporated in the thesis.

Dr. K. Poulose Jacob (Research Guide)
Ex. Pro Vice Chancellor,
Professor of Computer Science

Dr. Rekha K. James (Co-Guide)
Professor, Division of Electronics
School of Engineering

Kochi - 22
21st July 2017

Declaration

I hereby declare that the work presented in the thesis entitled **Security Vulnerability in On-Line Applications: Analysis, Anomaly Detection and Prevention of Attack** is based on the original research work carried out by me under the supervision and guidance of Dr. K. Poulose Jacob, Ex. Pro Vice Chancellor and Professor of Computer Science and Dr. Rekha K. James, Professor of Electronics and communication for the award of the degree of Doctor of Philosophy with Cochin University of Science and Technology. I further declare that the content of this thesis in full or in parts have not been submitted to any other University or Institute for the award of any degree or diploma.

Kochi-22 **Thresiamma K. George**

21st July 2017 Reg. No: 3800

Dedicated to

My Family

Acknowledgement

The present study has been a process of Analysis, evaluation, model designing and implementation, which has been a very challenging and reassuring academic exercise. I am grateful to many people who supported me during the research work and preparation of the thesis. First and foremost, I thank God Almighty for the blessings showered upon me to complete this endeavor. I would like to express my sincere gratitude to Dr. Poulose Jacob, Ex. Pro Vice Chancellor, Professor of Computer Science, Cochin University of Science and Technology for his encouragement and guidance throughout the research work. His insightful comments; support and advice have been of immense benefit to me during the most difficult time. I am grateful to my co-guide, Dr. Rekha K.James, Professor, Cochin University of Science and Technology for being a source of support and encouragement. Her sincerity, patience and supportive attitude enabled the successful completion of this work. My sincere thanks are also to Dr. Santhosh Kumar Head of the department of Computer Science, Cochin University of Science and Technology for his support and guidance.

I would like to thank Dr. Sumam Mary Idicula, Dr. Sheena Mathew and all faculty members of Computer Science Department, Librarian, office staff and all research scholars of Department of Computer Science at Cochin University of Science and Technology for providing the much needed help and support in completing the research work. My sincere thanks are also due to all faculty members at Higher College of Technology, Muscat for their timely help and support. I wish to place on record of my sincere thanks to Dr. Vinu, Dr. Sherimon and Dr. Reshmy for their cordiality, support and help. I am thankful to all my teachers at school and colleges for making me what I am today.

Let me express my deepest gratitude to my husband Mr. Sunny Thomas for his encouragement and special care during my hectic schedule and moral support throughout the completion of this work. I specially mention my sons Ashish and

Ashwin for their love, understanding, timely support and encouragement that helped me to fulfill my dream. I also extend my gratitude to my parents, my in-laws, my uncle Dr. K.S Mathew, my brothers and sisters for their encouragement, support and blessings. I also thank my relatives and friends for the concern and support they extended at various stages of my work.

Thresiamma K. George

Abstract

Security vulnerabilities are major threats in web applications which lead to the loss of integrity, confidentiality, and availability of user data. Malicious attacks on on-line applications frequently occur through these vulnerable points and breach the security mechanisms of authentication, authorization, and accountability. The existing techniques and strategies are not sufficient enough to handle most of these vulnerabilities due to its complex nature, and the lack of sophistication in the current input validation techniques. In SQL injection attacks, an attacker attempts to use an application code to access or corrupt database content and it commonly affects database driven online applications. It is one of the most dangerous vulnerabilities which usually occur when data provided by the user is included directly on SQL statements without proper validation. SQL injection attack takes advantage of the poorly coded web application and exploits the sensitive and critical information from backend databases.

In this research work, an efficient Multi-Level Template based Detection and Reconstruction model (MLT-DR) is designed and implemented to detect SQL injection attacks and to reconstruct injected queries. It is a hybrid model with an effective framework to parse and analyse the query with maximum performance and precision. The proposed model validates dynamic queries against the legal query pattern, before redirecting it to the web server. In this approach, the malicious queries are blocked and an alert message generated, if the injection is detected. Only the benign query can access the data from the back-end database server. The Standard Query Template Creator application (SQTC) created within this framework is capable of creating a specific query structure for both legal and injected queries, in the form of tokens. The proposed multilevel template mapper algorithm, which is used in SQTC, describes the procedure to map the injected query with the legal query, to detect the presence of an

injected string. The proposed architecture does not demand any source code modifications and performs detailed analysis at negligible computational overheads without false positives or false negatives. The proposed approach has a fully automated query assessment procedure to build a legal query model and has a validation technique to match the dynamic user query with the design template.

The reconstruction component of the hybrid model reconstructs the queries from the authenticated user by eliminating the injected portion and rebuilds the missing parts of the user query, based on request-id and the type of injection. The model construction algorithm explains the procedure of rejuvenating queries from authenticated users. To reduce the coding complexity and improve the performance of the detection procedure, the SQLI-Shield and SQLI-Rejuvenator package files are deployed. Reconstruction procedures are also validated by using machine learning technique.

MLT-DR is implemented using Java-based application software and MySQL as a back-end database server. The crawler functionality implemented in the web application identifies the hotspot or form field of user input queries. The captured queries are parsed or split into different tokens and stored in a template repository. Malicious queries are logged in and documented for developing the anomaly pattern to have a stronger detection model in the later stage of handling the zero-day vulnerability.

Contents

Acknowledgement i
Abstract iii
List of tables xi
List of figures xiii
Glossary xv
Abbreviations xvii
Notation xix

Chapter 1
INTRODUCTION .... 01 - 14
1.1 Background 02
1.2 Motivation 07
1.3 Problem Statement 10
1.4 Research Objectives 11
1.5 Contributions of Research Work 12
1.6 Outline of the Thesis 13
1.7 Summary of the Chapter 14

Chapter 2
LITERATURE SURVEYON SECURITY VULNERABILITIES .... 15 - 41
2.1 Security Concerns in a Web Application 16
2.1.1 The Web Architecture 17
2.1.2 Data storage and Access Strategies on the Web 18
2.2 Literature Survey and Related Work 19
2.2.1 Common Security Threats and its Consequences 19
2.2.2 Classification of Security Threats and Attacks 20
2.2.3 Vulnerability Assessment Methodology 22
2.2.4 Major Tasks and Steps under Vulnerability Assessment 23
2.3 Application Level Security Vulnerability 25
2.3.1 Major classes of Application Level Vulnerabilities 26
2.3.2 Validation Controls 27
2.3.3 Sanitizing Strategies 29
2.3.4 Tools and Techniques Used in Vulnerability Analysis 30
2.4 Security Models Against Vulnerability Detection 32
2.5 Vulnerability Assessment Reported by Security Organizations 34
2.5.1 Kaspersky Lab B2B International 34
2.5.2 Web Application Security Consortium (WASC) 35
2.5.3 Open Web Application Security Project (OWASP) 36
2.5.4 Acunetix 37
2.5.5 IBM Security and Ponemon Research 37

2.6 SQL Injection Attack- Most Dangerous Attack on Database Layer 38
2.7 Summary of the Chapter 40

Chapter 3
THE MULTILEVEL TEMPLATE BASED DETECTION AND RECONSTRUCTION (MLT-DR) FRAMEWORK .... 43 - 66
3.1 Introduction 44
3.2 Major Attack Categories of SQL Injection 44
3.2.1 Tautologies 44
3.2.2 Logically incorrect query/illegal Queries 45
3.2.3 Union Query 45
3.2.4 Piggy-Backed Queries 46
3.2.5 Alternate Encodings 46
3.2.6 Stored Procedure 46
3.2.7 Inference attack 47
3.3 Standard Queries and its Malicious Pattern 47
3.3.1 Attack category and Signature 51
3.3.2 Classification Strategy 52
3.4 The General Layout of the Proposed Architecture 52
3.5 The MLT-DR Framework 54
3.5.1 Proxy Server 57
3.5.2 Web Server 57
3.5.3 Database Server 58
3.5.4 The Detection Module (TbD) 59
3.5.5 The Reconstruction Module 59
3.5.6 Authentication Checking 60
3.5.7 Log of Denied Queries 60
3.6 Template Design Strategies 61
3.6.1 Strategies used for Storage and Retrieval 62
3.6.2 JavaScript object Notation Format (JSON) 62
3.6.3 JAR file to Retrieve and to Specify the Path Details 63
3.7 Summary of the Chapter 65

Chapter 4
THE MULTILEVEL TEMPLATE BASED DETECTION FRAMEWORK (MLT-D) .... 67 - 96
4.1 The Multilevel Template Based Detection (MLT-D) Framework 68
4.1.1 The server Functionality 69
4.1.2 The Standard Query Template Creator Application (SQTC) 69
4.1.2.1 Model Creation Algorithm 69
4.1.3 The template Repository 71

4.1.4 Token based Query Model Constructor and Parsing Procedure ... 72
4.1.4.1 Token Specification Strategies ... 73
4.1.4.2 Complex Query Evaluation ... 74
4.1.4.3 Query Evaluation Using Tree Structures ... 74
4.1.4.4 Design Specification for Injection Detection ... 75
4.1.4.5 General View of Query evaluation ... 76
4.1.5 The Mapper ... 81
4.1.5.1 Token-Mapper Algorithm ... 82
4.1.6 Validation and Detection ... 83
4.2 Strategies Used in Standard String Matching Algorithms ... 84
4.2.1 Boyer-Moore Algorithm ... 84
4.2.2 Hirschberg Algorithm ... 85
4.2.3 Morris-Pratt Algorithm ... 85
4.3 Experimental Result ... 86
4.3.1 Queries Tested with MLT-DR Framework ... 86
4.3.2 View of Template ID and Storage Format of SQL Query ... 92
4.3.3 Procedure to Detect and block SQL Injection Attack ... 93
4.4 Summary of the Chapter ... 96

Chapter 5
THE RECONSTRUCTION FRAMEWORK ... 97 -118
5.1 Significance of the Reconstruction Framework ... 98
5.2 Components of Reconstruction Framework ... 99
5.2.1 Server Functionality ... 99
5.2.2 Training Data ... 100
5.2.3 Back Propagated-Neural Network model ... 100
5.2.3.1 Multilayer Artificial Neural Network (ANN) for Machine Learning ... 101
5.2.3.2 Major Steps in Back-Propagation Algorithm (BPA) ... 102
5.2.4 Template Store ... 103
5.2.5 Template Mapper ... 103
5.2.6 Template Translation ... 103
5.2.7 SQL Reconstruction ... 106
5.2.8 SQLIA Detection Engine ... 106
5.3 Regular Expression and Comparison for Pattern Matching in SQL Statement ... 107
5.4 Model Construction Algorithm ... 113
5.4.1 Reconstruction Algorithm ... 114
5.5 Experimental Result of Reconstruction Procedure ... 116
5.6 Summary of the Chapter ... 117

Chapter **6**
PROTOTYPE IMPLEMENTATION OF MULTI LEVEL TEMPLATE BASED DETECTION AND RECONSTRUCTION (MLT-DR) FRAMEWORK........ 119 - 134
6.1 System Architecture of the Prototype, MLT-DR..........120
6.2 MLT-DR Training Phase 120
6.2.1 Identification of Hot Spot/ form Field Entry in the Web Pages ... 121
6.2.2 Standard Query Template Creator (SQTC) 122
6.3 Learning Phase of Back Propagated Neural Network Learned Model..........127
6.4 Testing Phase of MLT-DR 128
6.4.1 Template Generator/Parser for User Input Query 129
6.4.2 The Model Mapper.......... 129
6.4.3 SQL Injection Attack Detection Engine.......... 130
6.4.4 Reconstruction Component.......... 132
6.5 Summary of the Chapter..........134

Chapter **7**
PERFORMANCE EVALUATION OF MLT-DR FRAMEWORK... 135 - 154
7.1 Testing Hypothesis 136
7.2 Data Set Used for Testing MLT-DR..........137
7.2.1 Dataset I: Data Available from Cheat Sheets/ URL.......... 137
7.2.2 The extract of Malicious Queries From URL 137
7.2.3 Dataset II: Standard Test Suite Provided by Halfond and Orso 139
7.2.4 Dataset III: Customized School Connect System.......... 141
7.3 Performance Measures of MLT-DR..........142
7.3.1 Process Time Overhead.......... 143
7.3.2 Efficiency of MLT-DR.......... 143
7.3.3 Precision of MLT-DR 145
7.3.4 Effectiveness of MLT-DR.......... 147
7.4 Type I & Type II Error 148
7.5 Receiver Operating Characteristic (ROC) Curve 149
7.6 Storage Overhead & Processing Time for Detection 150
7.7 Comparison of MLT-DR with Other Models..........152
7.8 Summary of the Chapter..........154

Chapter **8**
CONCLUSION AND FUTURE WORK .... 155 - 162
8.1 Summary of the Research Work..........155
8.2 Major Highlights of the Research Work..........156

8.3 Future Directions 159
8.4 Conclusion 161

References .... 163 - 180

Appendices .... 181 - 187

List of Publications .... 188 - 189

List of Tables

Table 2.1 Application level vulnerability-classes and determining factors... 26
Table 3.1 Standard query vs. malicious query 48
Table 3.2 Attack category and signature 51
Table 4.1 Standard Query Template Creator Algorithm (SQTC) 70
Table 4.2 Summary of tokens and values assigned 81
Table 4.3 Token Mapper algorithm for SQL injection Detection (SQLI-D) 82
Table 4.4 Legal queries Vs Injected queries tested in MLT-DR 86
Table 5.1 Symbols and description 104
Table 5.2 Metacharacters used in REGEX 108
Table 5.3 List of functions for string comparison 109
Table 5.4 Basic functions with SQL Regular expression 110
Table 5.5 Model constructors in MLT-DR 111
Table 5.6 Reconstruction Algorithm of Authenticated user Queries (RaAuQ) 115
Table 6.1 Legal queries and Injected queries chosen for BPNN learning .. 123
Table 6.2 Identified attack vectors and signatures 127
Table 6.3 Embedding a Template-ID in a SQLIA-Shield 129
Table 7.1 Data collected from the cheat sheet/URL 137
Table 7.2 Sample vulnerability report with MLT-D detection status "Yes" 138
Table 7.3 The Identified application with hotspots 139
Table 7.4 Test result showing the effectiveness of MLT-DR 140
Table 7.5 SchoolEconnect with attack detection details 141
Table 7.6 Time overhead in SchoolEconnect application 144
Table 7.7 Analysis of false positives in General 146
Table 7.8 Analysis of false positives in SchoolEconnect application 146
Table 7.9 Attack categories and detection rate in MLT-D 148
Table 7.10 Comparison of MLT-DR with other models 153

List of Figures

Figure 1.1 Holistic approach to security 02
Figure 1.2 Vulnerability by paradigm and severity 06
Figure 1.3 Percentage of various vulnerability classes as reported in OWASP 08
Figure 2.1 Typical three tier web architecture 17
Figure 2.2 Major components involved in Assessment 23
Figure 2.3 Processing steps for Vulnerability assessment 24
Figure 2.4 Summary of percentage of attack against the attack categories 35
Figure 2.5 Percentage of vulnerability reported by OWASP 36
Figure 2.6 Security risk Vs Spending 37
Figure 2.7 SQL injection attack on a web application 39
Figure 3.1 General layout of MLT-DR architecture 53
Figure 3.2 Architecture of MLT-DR framework 56
Figure 3.3 Query ID and corresponding JSON format 63
Figure 4.1 Multilevel template based Detection (MLT-D 68
Figure 4.2 Procedure for Query model constructor 74
Figure 4.3 User query evaluation using Tree structure 75
Figure 4.4 Query Evaluation procedure in the hybrid frame work 76
Figure 4.5 Query evaluation using tree structure 78
Figure 4.6 Independent query tokenizing procedure 90
Figure 4.7 Complex query tokenizing procedure 91
Figure 4.8 JSON format of the standard query 92
Figure 4.9 Template ID formats of tested queries 93
Figure 4.10 Evaluation procedure of a dynamic query 94
Figure 4.11 Complex query evaluation procedure with multiple levels 95
Figure 5.1 System architecture of the reconstruction Framework 99
Figure 5.2 BP-NN learning for SQL trained model 101
Figure 5.3 Multilayer representation of neural network 102
Figure 5.4 Status report of NNbR module with a reconstructed query ... 117
Figure 6.1 MLT-DR Prototype in Training phase 121

Figure 6.2 Injected query on the form field of login page 122
Figure 6.3 The unique template ID and template in JSON format 125
Figure 6.4 The generated unique ID for one of the identified web application 126
Figure 6.5 MLT-DR Prototype in testing phase 128
Figure 6.6 Identified Query ID for mapping & JSON format 130
Figure 6.7 Evaluation result of the prototype tool TbD 132
Figure 6.8 Status report -Reconstruction procedure in MLT-DR framework 133
Figure 7.1 Test result showing the effectiveness of MLT-DR 140
Figure 7.2 SchoolEconnect with attack detection details 142
Figure 7.3 Test result showing the efficiency of MLT-DR 144
Figure 7.4 Precision analysis in SchoolEconnect application 147
Figure 7.5 Type I & Type II error rate 149
Figure 7.6 Receiver Operating Characteristic (ROC) curve 149
Figure 7.7 Detection procedure and JSON storage format 150
Figure 7.8 First level Injection detection 151
Figure 7.9 SQL Rejuvenation status report 152

Glossary

Attack	An attempt to destroy or compromise the system
Authentication	Process or action of verifying the identity of a user or process
Authorization	Process of giving permission to do or have access
Benign Query	Harmless or not injected query
Confidentiality	Component of security pillar to keep track of the privacy of information
Countermeasures	An action taken to counteract a security threat
Crawler	Program that systematically browses the World Wide Web
Cyber crime	Criminal activities carried out by means of Internet
Empirical	Verifiable by observation and experience rather than theoretical logic
Firewalls	Part of computer network that is designed to block unauthorized access while permitting outward communication
Form field	Form element used to collect user input
Hotspot	A place of significant activity
Hybrid model	Mix of different performance based model
Injection	Forceful entry or access
Integrity	Component of security pillar to keep track of the accuracy of Information
Malicious code	Any code that is intended to damage the system
Malware	Software that is intended to damage or disable computer system.
Mitigate	Lessen the gravity of mistake
Non repudiation	Assurance that someone cannot deny something
Parsing	Analyze a text or string into logical Syntactic component

Prototype	Model of the system generated
Proxy Server	An intermediary server for request from client
Rejuvenate Query	Reconstruct the malicious query
Sanitize	To make it pure/without injection
Security Breach	Breaking the security and compromising the system
Security Patch	Security patch is intended to correct a vulnerability or viral infection
Spamming	Practice of sending unwanted email messages with large quantity of commercial content
Tokens	Single program element
Vulnerability	Weakness in the system to allow the unauthorized users to compromise the system

Abbreviations

API	Application Program Interface
ANN	Artificial Neural Network
ASCII	American Standard Code for Information Interchange
ASP	Application Server page
B2B	Business to Business
BPA	Back-propagation algorithm
BP-NN	Back Propagated Neural Network
CSRF	Cross- Site Request Forgery
CVSS	Common Vulnerability Scoring System
DAST	Dynamic Application Security Testing tools
DB	Database
DBMS	Database Base Management System
DFS	Depth First Search
DoS	Denial of service attack
FTP	File Transfer Protocol
HTML	Hypertext Markup Language
HTTP	Hypertext Transmission Protocol
IDE	Integrated Development Environment
IDS	Intrusion Detection System
IIS	Internet Information Service
JAR	Java Archieve files
JDBC	Java Database Connectivity
JSON	Java Script Object Notation
LDAP	Lightweight Directory Access Protocol
MLTD	Multi-Level Template based Detection
MLT-DR	Multi-Level Template based Detection and Reconstruction

NNbR	Neural Network based Reconstruction
ODBC	Open database Connectivity
OWASP	Open Web Application Security Project
QUADP	Query Anomaly Detection Process
REGEX	Regular expression
R-Iq	Reconstruction of Input query
ROC	Receiver Operating Characteristic
RSS	Rich Site Summary
SAST	Static Application Security Testing
SQL	Structured Query language
SQLIA	Structured Query language Injection Attack
SQLI-R	Structured Query language Rejuvenator
SQLIV	SQL Injection Vulnerabilities
SQTC	Standard Query Template Creator application
TbD	Token based Detection
TCP	Transmission Control Protocol
URL	Uniform Resource locator
WASC	Web application Security Consortium
XML	eXtensible Markup Language
XSS	Cross Site Scripting

Notation

μ	Micro Symbol (Mu), Index of actual learning pattern μ = 1,…,P
φ	Unicode extended character, to calculate weighted sum of input values
Δ	Increment, a small positive or negative change in a function.
∑	N-ary Summation
∂	Unicode extended character, to check error function
ɛ	Latin small letter Ezh Reversed.
⊕	Exclusive OR Symbol to check the exact mapping of strings
O	Big O (Landau symbol- Time space Complexity)
f	Activation function
ɛμ	Sum-of-squares errors
N_m	Number of elements in layer m
$\Delta^{\mu} wji^{m}$	Change of given weight for pattern μ
φj^{μ}	Weighted sum of input values for element j in layer μ
$\boldsymbol{w_{ii}}^{\boldsymbol{m}}$	Weight between element j in layer m and element I in layer m-1
$ui^{(m-1)\mu}$	Output of element I in layer m-1 for pattern μ
∂^{μ}	Learning error for element j for pattern μ
η	Proportion coefficient

.....༄༅.....

Chapter 1

INTRODUCTION

Contents

1.1 Background
1.2 Motivation
1.3 Problem Statement
1.4 Research Objectives
1.5 Contributions of Research Work
1.6 Outline of the Thesis
1.7 Summary of the Chapter

As web applications have become the primary sources of information dissemination and most preferred way of delivering essential service to customers, it has also become an attractive target for attackers. Security vulnerabilities are major threats in web applications as successful attack through these vulnerable points leads to loss of integrity, confidentiality and availability aspects of a security triangle. The existing techniques and strategies are not sufficient enough to handle most of the vulnerabilities due to the complex nature of vulnerability issues, and the current input validation techniques still require more sophistication. Code injection attacks are one of the most dangerous threats that exploit the application layer vulnerabilities, which top the list due to lack of effective strategies existing for detecting and blocking injection attacks. In recent past, most of the very high-profile organizations are also acknowledging the breaches of their system with data theft, compromises on the database server espionage and service interruptions due to lack of sufficient measures to protect web applications. Most of the cyber-attacks are targeting on the application layer, and there are possibilities of thousands of known vulnerabilities and hundreds of emerging vulnerabilities which we need to tackle as and when it affects the business transactions. The cost of cybercrime is very high, and the toll on the security team after the significant security breach for investigation, analysis and to remediate the damage is too expensive for an organization to bear. Authentication threats and Injection attacks are the standard vulnerability/incidents even in the case of highly secured online financial applications. It is a clear indication that the online applications need to be much more secured and required to be monitored and analyzed closely from various aspects of the security framework.

1.1 Background

Online applications are becoming more and more popular because of its built-in infrastructure and support of diverse technologies, and we require them for our routine activities. The nature of their feature-rich design and their capability to collate process and disseminate information over the internet makes them an attractive target for attack (Johari and Sharma, 2012). The vulnerability is the weakness that makes threats possible. The vulnerability exists within web applications mostly because of poor design, configuration mistakes, and inappropriate coding techniques. Vulnerability scanners help to check vulnerabilities embedded in a web application by testing invalid forms of input query (Aliero and Ghani, 2015).

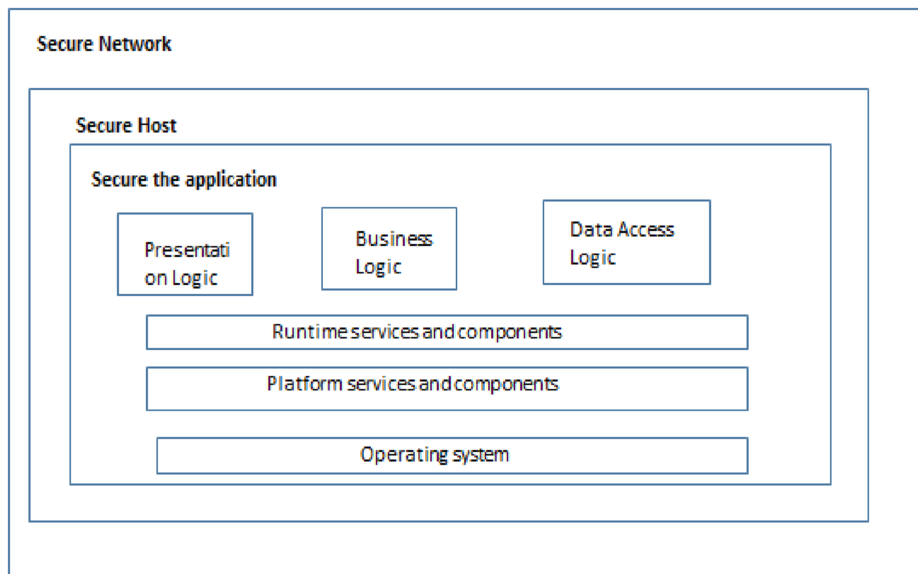


Fig. 1.1 Holistic approach to security

The application server should be configured appropriately to obtain secure network capabilities. A holistic approach to security (Huang and

Kuo, 2005; Microsoft application architecture guide) is required to build a secure web application as shown in Figure 1.1. The diagram indicates that the application layer security is one of the core concerns of the entire web application security other than the Network and Host Security (Awang and Manaf, 2015).

Most of the web applications, supported with backend database servers are susceptible to security attacks (Artzi and Kiezun, 2008). With the advent of web 2.0, the content handling, usability and interoperability of a web application such as e-shopping, e-banking, internet messaging, web communities, etc., have taken up a new phase. They achieved highest possible sophistication to handle the user demands, but at the same pace, it results in the growth of the attack incidents into a dangerous situation (Sahu and Tomar, 2016). Vulnerabilities in online applications prompt malicious users to get an unauthorized entry to compromise critical information stored in the backend database of online applications. Modern web applications or the electronic/digital services are critically affected by various security issues (Alata and Akrout, 2013). Over the period of extensive research, the world witnessed an increased trend of emerging cyber-threats and malware targeting online applications and the traditional security measures adopted by the organizations are not competent enough to deal with the upcoming vulnerabilities and attack spectrum (Li, and Xue, 2014). Although numerous protection strategies have been designed, developed and implemented with the support of vulnerability analysis, detection and prevention mechanism, still there are threats, attacks and compromises on the application layer of security (Bau and Mitchell, 2010). SQL injection is one of the most dangerous vulnerabilities that affect database driven online applications. It

tops the list due to lack of effective strategies for detecting and blocking injection attacks. By exploiting SQL injection vulnerability, an attacker directly interacts with the database server and gain access to the unauthorized data by compromising security (OWASP, 2010; Halfond and William, 2008).

As per the report from the Open Web Application Security Project (OWASP), it rates SQL injection attack as one of the top ten security vulnerabilities targeting backend databases. By exploiting SQL injection vulnerabilities, an attacker directly interacts with the database server and gains unauthorized access to the data by compromising security. Even though there are many methods and strategies developed against SQL injection attack, yet the risk rate of SQL injection is increasing exponentially in most of the online applications as there are fully automated injection tools available with the talented hackers. In a susceptible application, an SQL injection attack uses crooked input that changes the SQL query and establishes an illegal connection to the database (Alfantookh and Abdulkader, 2004).

Monitoring, analyzing, detecting the vulnerabilities and preventing the attacks should be a continuous process so that the severity of the compromises or attempts on attacks can be mitigated. It must also provide a best possible protection strategy by the security team and the system architects. Most of the business organizations are having required protection mechanism regarding firewalls and intrusion detection system as part of their security infrastructure, but still many of the organizations are not having comprehensive tools and practices in place for securing their applications (Jovanovic, C. Kruegel 2006). Hence the hackers continue to attack the

application layer, and the application developers are focusing on the sophisticated features of the application rather than removing the vulnerabilities. Proactive and consistent risk management architecture can efficiently prevent, detect and remediate vulnerabilities in the application layer. As per the security report, it is not an ideal situation to keep track of thousands of known vulnerabilities by a single tool or a human developer. The best coding practices can be another solution to reduce vulnerabilities in the early stage. As per the detailed report by various security organizations, the Static Application Security Testing (SAST) tools can be an ideal choice to identify vulnerabilities in early-stage development by efficiently determining the vulnerabilities in each line of code. But it has an inherently higher false positive rate than the Dynamic Application Security Testing tools (DAST). Most of the developers prefer to compile their code and dynamically test it in a run-time environment iteratively because some of the vulnerabilities will appear only in a run-time environment, in this situation the DAST tools seems to be much more accurate (Akrout and Nicomette, 2014). Appropriate Server Configuration and necessary patches are important strategies for prevention, detection and remediation of vulnerabilities (Borade and Deshpande, 2014).

The severity of vulnerabilities estimated by Common Vulnerability Scoring System (CVSS) in 2015 states that it is essential to analyze the security at all development stages and regularly during operational use. The report of CVSS indicates that more than 63% of applications implemented contain critical vulnerabilities and can lead to confidential data disclosure and system compromise (Calvi and Vigan, 2016). As per the report of Web Application Security Consortium (WASC) for the year 2016; most of the modern web technologies support and solve organizational issues cost

effectively. But well-equipped hacker's attack to compromise servers is the key issue in order to have perfect security (Li Xiaowei, and Yuan Xue, 2011). Acunetix report suggests that web application vulnerabilities are increasingly pose serious threats to organization's overall security hence it is a fundamental requirement for an organization to make application level security as the priority concern. As per the report of Acunetix web application vulnerability security (2016), high-severity vulnerabilities are increasing day by day and present in most of the websites. The near 45,000 sites and almost 5,700 network scans done for a year until March 2016 shows that nearly 55% of websites have one or more high severe vulnerability such as XSS or SQL injection and 84% are susceptible to at least one medium-severity- vulnerability such as CSRF. Almost 8% of the network scanned contained a high security vulnerability. Figure 1.2 shows the Acunetix's reports on vulnerability by paradigm and severity.

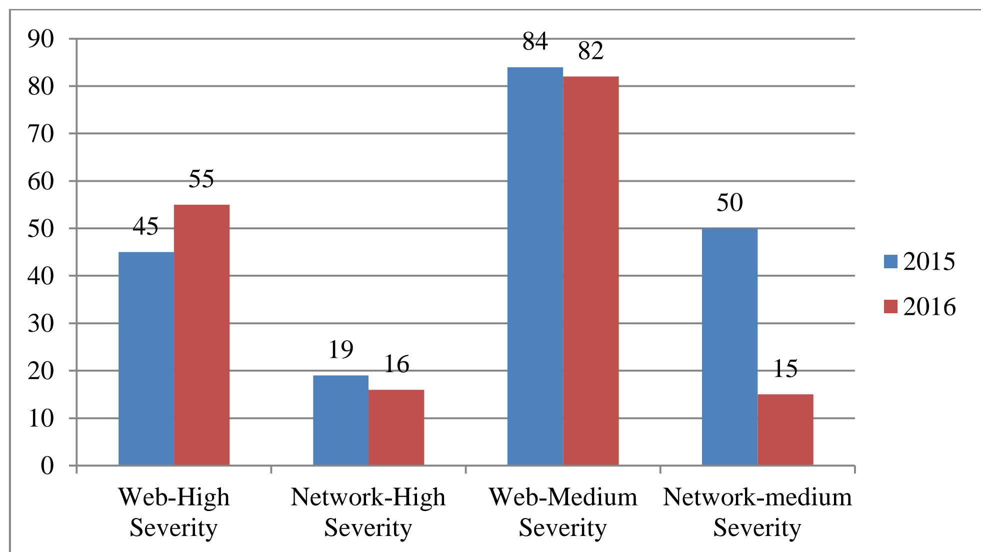


Fig. 1.2 Vulnerability by paradigm and severity

The following points summarize some of the major concerns in this research study:

- Even though there are many prevention and protection techniques developed and implemented against SQL Injection vulnerabilities, code injection attacks and exploiting the confidential data by breaking the authentication logic are common incidents even in the highly secured financial applications.
- There are huge list of threats and attacks documented in the recent past by various security consortiums such as OWASP, WASC, and CVSS, but still the security professionals are giving little attention to web application security.
- In most of the interactive web applications, security vulnerability remains a major issue, and SQL injection still prevails as one of the top-10 vulnerabilities and threat to the online web application with a backend database.

Hence the analysis and development of an appropriate tool or technique as a countermeasure against code injection vulnerability is a major concern for a security person or a developer (Securosis, 2014). An effective and efficient approach to handle code injection vulnerability and to mitigate the denial of service attack (DoS) is yet to be developed (Bhoria and Gharg, 2013).

1.2 Motivation

A less secure web application design may allow crafted injection and malicious update on the backend database. This trend can cause lots of

damages and thefts of trusted users' sensitive data by unauthorized users. In the worst case, the attacker may gain full control over the Web application and ultimately destroy or damage the system. SQL Injection Vulnerabilities (SQLIVs) are one of the open doors for hackers to explore. Hence, they constitute a severe threat for Web application contents (Zhu, 2014; Web Security Threat Classification, 2013). Figure 1.3 shows the percentage of various vulnerability classes as reported in OWASP.

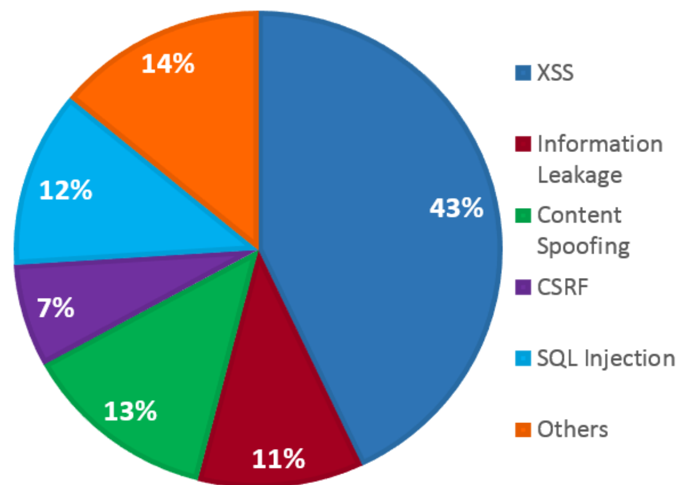


Fig. 1.3 Percentage of various vulnerability classes as reported in OWASP

Input validation is a major concern for application security. Almost 50 to 60 % of security vulnerabilities in 70 % of applications tested is due to poor input validation. Improper Input Validation can result in significant vulnerabilities in the web application, which can be exploited by attackers. Detailed study and analysis of security vulnerability research indicate that the key issues identified at the application layer of the web application are:

- Most of the available online applications with backend database servers are vulnerable to SQL injection attacks.
- Existing solutions against SQL injection attacks have limitations which in turn affect the performance and effectiveness of the application.
- Defensive coding techniques suggested by researchers may not be a complete solution as implementing strict defense coding is expensive and time-consuming in complex systems.
- Most of the defensive coding strategies are labor intensive and may only cover subsets of all possible attack patterns. We can quickly detect critical threats due to SQL injection attacks through dynamic web pages of online application with the implementation of an effective vulnerability scanner.
- Most of the existing scanning tools are not sufficient enough to handle the future security threats, and the scanning tools do not support comprehensive analysis in purchased applications.
- The available tools and techniques require a series of validation in coding practices which severely affect the performance, time and space complexity.

Since this research work focuses on application layer vulnerability, specifically on compromise or attack on the database server, we have narrowed down our research on code injection vulnerabilities as the primary area of research.

1.3 Problem Statement

SQL injection attack on sensitive business applications can easily create a root level attack on the database servers and the other connected network by which there could be a destruction of security attributes such as Confidentiality, Integrity and Availability. Identification of code injection vulnerabilities in the legacy system is hard, and most of the SQLI prevention approach results in false positives (Avireddy and Srinivas, 2012). Denial of service attacks resulted from SQL Injection will decrease the system availability. The existing tools for scanning, validating, protecting and preventing injection attacks on web pages still need further expansion and better strategies to adequately handle the highly automated malicious attacks. Exploiting the code injection vulnerabilities to penetrate the backend database server to steal or disclose the highly sensitive information is one of the most dangerous attacks in a highly confidential web application. The consequences of these types of attacks create an enormous impact on the business applications (Xiaowei and Yuan, 2011; Weinberger and Joel, 2011). Most of the existing SQL Injection-Detection and prevention approaches undergo the following issues:

- They target only a subset of SQLI attack types. A few strategies are developed to handle distinct categories of injections attacks without false positives (Federico, Macro,2016).
- During the dynamic phase, SQLI validations and modification of application code on the online applications are expensive on time and space complexity and results in a bad performance of the web application (Belk,2011).

- The denial of service attack arising out of the SQL Injection is critical in most of the business applications. We must have a high degree of importance to mitigate this type of attack (Seacord and Martin, 2010).

By considering the issues mentioned above, the research problem formulated is to design and implement an effective technique and strategies for detecting and preventing SQL Injection attacks in online applications with better performance, efficiency and reduced denial of service attacks.

1.4 Research Objectives

The objectives of this research are:

- To perform a detailed survey and analysis on existing techniques for detecting SQL injection and counter measures of attacks on web applications.
- To propose a robust hybrid model to detect SQL Injection and prevent attack.
- To suggest multilevel template creator algorithm and template matching algorithm for detection and prevention of SQLI attack.
- To propose an effective Model construction algorithm to rejuvenate the malicious queries from authenticated user.
- To analyze the parameters such as efficiency, effectiveness and precision for a secure web application design and Implementation.
- To propose a strategy or procedure to reduce complexity and better performance of the query evaluation process.
- To develop and implement a prototype to validate the efficiency and effectiveness of the template-based detection model.

1.5 Contributions of Research Work

The proposed MLT-DR is a hybrid model, designed to detect and block SQL Injections without any false positives and with better system availability (George and Jacob, 2018). This model blocks all malicious SQL entries and only the benign query can access the data from the back-end database server. This framework also has the provision to reconstruct the queries from authenticated users at run time, which increases the system availability. MLT-DR, uses an efficient query validation technique which matches the statically generated legal query tokens against the parsed dynamic query tokens. The query mapping and reconstruction procedure are described by the proposed SQL Token Mapper and Model Construction algorithms. A prototype has been designed and implemented using a Java-based application program to test the performance and the effectiveness of the model.

The major contributions are:

- A novel technique MLT-DR, Multi-Level Template based Detection and Reconstruction to parse and analyse the query with high precision rate.
- An effective Standard Query Template Creator application (SQTC), to parse the legal query. If the parsed token has malicious patterns, it can be blocked and stored in a template repository, which can be used as countermeasures in future attack patterns.
- Query reconstruction from authenticated users to increase the system availability.

1.6 Outline of the Thesis

Chapter 1: This chapter introduces this research work which includes the background, motivation, problem statement, objectives and main contributions of this research work.

Chapter 2: This chapter provides a brief survey of web application vulnerability with various works done in code injection vulnerability and its countermeasures. The chapter also discusses distinct categories of SQL injections attacks, prevention and protection tools.

Chapter 3: This chapter presents the proposed SQL-Detection Hybrid model, MLT-DR.

Chapter 4: This chapter explains the key features of SQL-detection, algorithms for token parsing and analysis and procedure to detect and prevent SQLIA.

Chapter 5: This chapter deals with the SQL Rejuvenator module to reconstruct the query. It also explains the architecture of the model, unique features, components and procedures.

Chapter 6: This chapter describes the Design Implementation of Prototype MLT-DR with a customized online application and testing of various categories of queries with appropriate screen shots.

Chapter 7: This chapter describes the performance evaluation of the proposed model, MLT-DR. The detailed empirical analysis is also carried out using sample queries collected from various URLs and shared databases.

Chapter 8: This chapter summarizes the research work by highlighting various contributions made by the proposed model and its significance while comparing it with the other existing models. The chapter also discusses the future directions.

1.7 Summary of the Chapter

This chapter introduces the topic selected for the research work, the security vulnerability in an online application, especially the code injection vulnerabilities. We cover the basic concepts in this research procedure. The chapter also introduces analysis, and implementation details by stating that "*as the popularity of web applications demand sophisticated user interactions of the routine services, the sophistication of attacks is also growing proportionally. And there is an immediate requirement for an effective approach to preventing any exploits on sensitive information through the vulnerable points*". The chapter explains the problem statement, motivation, research objectives and the contributions of the proposed work. We conclude the chapter by showing the layout of the entire documentation of the research analysis and implementation details.

.....ꟹꟹ.....

Chapter 2

LITERATURE SURVEY ON SECURITY VULNERABILITIES

Contents

2.1 Security Concerns in a Web Application
2.2 Literature Survey and Related Work
2.3 Application Level Security Vulnerability
2.4 Security Models Against Vulnerability Detection
2.5 Vulnerability Assessment Reported by Security Organizations
2.6 SQL Injection Attack- Most Dangerous Attack on Database Layer
2.7 Summary of the Chapter

Vulnerability analysis is a process that defines, detects and classifies security vulnerabilities in a system, network or communication infrastructure. It also suggests the countermeasures and the effectiveness of the implementation techniques. The vulnerability exists within a web application if it does not provide a proper validation process for the data entered by the user as input. In the global scenario, the increased amount of dependability of the online application has given way to a heavy traffic in the communication network and risk of security vulnerability. As the popularity of the online and automated processes increases, the chances of vulnerability also increase along with it. As part of the literature survey on vulnerability assessment, we discuss the web architecture, scanning tools, strategies and countermeasures to detect security vulnerabilities. Also, we explain detailed research report on various vulnerabilities, assessment tools and strategies suggested by various security consortiums. We have considered an in-depth evaluation of web application vulnerabilities and the corresponding countermeasures to detect and block the vulnerabilities in the similar fields carried out by researchers. This chapter also summarizes the recent reviews and reports by researchers and the security organizations with due importance given to countermeasures against code injection vulnerability.

2.1 Security Concerns in a Web Application

Web application security is the process of securing confidential data modification and disclosure of information from unauthorized access (Sahu and Tomar, 2016). Security survey by researchers indicate that the security implementation process mostly aims at full filling the confidentiality, integrity, availability and non-repudiation aspects of the security principle (DeMeo and Rocchetto, 2016). Attacks on the online applications have tremendously increased over the years. Most of the web application attacks exploit weak input validation as root vulnerability. Input validation is an important task to protect against almost all of the significant vulnerabilities in the websites. SQL injection vulnerabilities pose a severe threat to online applications because it serves as an open the door to the hacker to explore and eventually compromise the backend database (Maheswari and Anita, 2016). Understanding the right validation approach and techniques for user input filtering are the keys to a secure web application (Kumar and Pateriya, 2012).

The process of security analysis runs parallel with web application development. The group of programmers and developers who are responsible for code development is also responsible for the execution of various strategies, post-risk analysis, mitigation and monitoring. In an online application identifying the type of vulnerabilities present across a web infrastructure is a critical step for providing overall security (Johari and Sharma, 2012). Common issues of web application code vulnerability are the wrong style of code writing and improper server configuration (Awang and Manaf, 2015).Vulnerability scanning is an efficient way to find out the application backdoor, malicious code and other potential threats in the

application. Vulnerability scanners aid in checking vulnerabilities embedded in a web application and has the potential to test invalid forms of input query (DeMeo and Rocchetto, 2016). Cybercriminals use exploits to known vulnerabilities; however, the zero-day vulnerability (which is currently unknown to the manufacturers) are the most dangerous; hackers are actively searching popular programs for hidden security loopholes to create an exploit in them (Zhang and Chen, 2010; Palsetia and Thilagam, 2016).

2.1.1 The Web Architecture

The three-tier web architectures as shown in Figure 2.1, consists of the presentation layer with static or dynamically generated content rented by the browser (front-end), the logic layer with dynamic content processing and generation level application server (middleware) and the Data layer with databases, comprising both data sets and the database management system (back-end) (Li, and Xue, 2014; Bau and Mitchell, 2010). Here each layer can potentially run on a different machine, follows the client- server architecture principles and each tier should be independent.

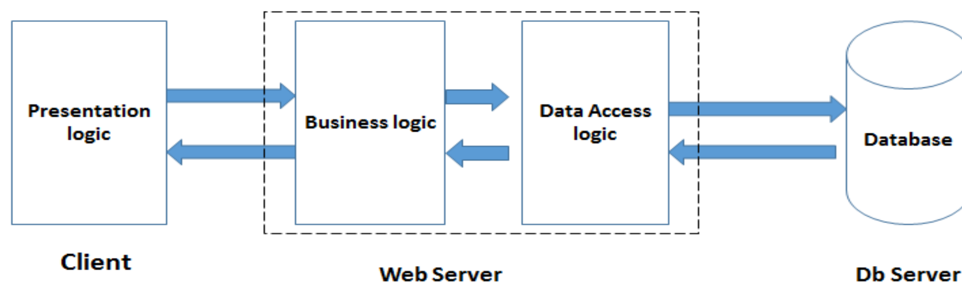


Fig. 2.1 Typical three tier web architecture

Major strengths of a web application are the factors such as ease of access, maintenance and management of up to date information with faster retrieval strategies (Johari and Sharma, 2012). Better format /structure of information retrieval and storage strategies with a user-friendly system across multiple platforms, reaching a broad audience with the concepts of anytime anywhere are also considered as strengths of web applications. If the resources highly automated with all sophisticated features are increasing, there should be equal support and measures to implement the security (Li Xiaowei, and Yuan Xue, 2011). In the latest implementation of a web application, the browser support for updated versions of browser and screen size have to be handled with due importance otherwise there will be technical errors (Anderson and Lane, 2011). SPAM/ Spamming is an issue in most of the applications with difficulty in reaching to the right group of people (Shanmughaneethi and Swamynathan, 2009).

2.1.2 Data storage and Access Strategies on the Web

In web application environment, the size of the data determines the data storage strategies. It can be locally available or placed as remote accessing storage. The locally stored can be a file or local database by using the available application such as Javascript apps or Indexed DB API's. Remote storage of data could be on the cloud or any remote HTTP endpoint that can serve JSON or XML data (Chen and Kalbarczyk, 2006). As proposed by the researchers (Khari and Kumar, 2013; Akrout and Nicomette et al., 2014), web application design and data storage and access strategies should focus on several factors. They are application request processing approaches and pattern, authentication mechanism and authorization process

with exception management, appropriate logging and navigation process, web page rendering and session management (Akrout and Nicomette, 2014).

Data storage within a web application is one of the major concerns of a security professional, as it required strict planning and efficient security measure to ensure the confidentiality and integrity (Chen and Kalbarczyk, 2006). The commonly used remote and local options for an appropriate data storage are web storage, indexed DB, SQLite, API, web services, web API and skyDrive (Zhang and Agarwal, 2008).

2.2 Literature Survey and Related Work

This section highlights recent research works on various strategies to protect web application, factors or components influencing the web security vulnerability analysis, reasons and consequences of code injection attack specifically the SQL injection detection and mitigation strategies. We also considered the works related to string and pattern matching to find out the injection detection and blocking of malicious queries (Su and Wasserman, 2006; Chen and Wu, 2010).

2.2.1 Common Security Threats and its Consequences

There are different classes of threats which can happen through the security holes taking advantage of the security vulnerability (Calvi and Vigan, 2016). As reported by WASC's classification, the primary security threats are occurring mainly due to one or more of the following reasons: (Zhu, 2014; Web Security Threat Classification, 2013).

- Insufficient authentication without appropriate user credentials.

- Insufficient authorization such as user privileges and permissions not verified properly.
- Client side attack during the dynamic interaction of an application.
- Command execution on any website components.
- Information Leakage, the attack discovering the hidden features of any information.
- Logical attacks that use different processes or strategies to get access.

2.2.2 Classification of Security Threats and Attacks

Security threats could be with the intent of stealing confidential information, causing intentional damage. So, the developer needs to take all the precautions to secure the organization's sensitive data by taking necessary steps, or detect security measures or prevent all the security threats to the website (Huang et al., 2004; Xiaowei and Yuan, 2011).

Privilege Elevation is a class of attack in which a hacker is a legitimate user on a system uses his credential to increase his account privileges to a higher level than it was assigned. Through this type of attack, the hacker can gain privileges as high as root on a UNIX system where he will be able to run code with superuser level of rights. The entire system can be effectively compromised and take the advantages of the system (Livshits and Lam, 2005; Konstantinos and Theodoros, 2008). SQL Injection takes advantage of loopholes present in the implementation of web applications that permits a hacker to break the system (Jose and Abirami, 2016). These types of attacks are mainly due to lack of input validations. So, we have to

give appropriate care to input fields like text boxes, comments and executing dynamic queries. It is one of the most common application layer attack techniques used by most of the hackers, where the hacker inserts malicious SQL statements into input field during a dynamic string execution. An attacker can get critical information from the server database. Hence SQL injection attacks are a very dangerous attack. URL Manipulation is the process of manipulating the website URL query strings and capture of the relevant information by hackers (Ali and Javed et al., 2009). Usually, URL manipulation occurs when the application uses the HTTP GET method to pass information between the client and the server. The application passes the information in parameters in the query string. The tester can modify a parameter value in the query string to check if the server accepts it(Halfond and Manolios, 2008). Denial of Service is an explicit attempt to make a machine or network resource unavailable to its legitimate users. A hacker can attack applications in ways that render the application, and sometimes the entire system will become unusable (Felmetsger and Viktoria, 2010). In Data Manipulation, hacker changes data used by a website to gain some advantage or to embarrass the site's owners (Papagiannis and Pietzuch, 2011). Hackers will often gain access to HTML pages and change them to be offensive. Identity Spoofing is a technique where a hacker uses credentials of a legitimate user or device to launch attacks against network hosts, steal data or bypass access controls. Preventing this attack requires appropriate IT-infrastructure and network-level mitigation strategies (Securosis: 2014). Cross-Site Scripting (XSS) is a vulnerability usually happening in the online application. XSS enables attackers to inject client-side script into web pages by other users and trick a user into clicking on that URL (Martin and Lam, 2008). Once executed

by the other user's browser, this code could then perform actions such as completely modifying the behavior of the website, stealing personal data, or performing activities on behalf of the user (Johari and Pankaj, 2012).

To mitigate the classes mentioned above of vulnerability and implement appropriate countermeasures most of the business organizations are putting increased efforts to improve their website security by implementing web vulnerability scanners, penetration testers to protect their sensitive data and block all possible vulnerable points from exploitation (Skrupsky and Bisht et al., 2013). Security professionals are continued to develop and implement the sophisticated technologies to provide user-friendly interfaces to its users with better security capabilities (Kndy and Pathan, 2012).

2.2.3 Vulnerability Assessment Methodology

Web application vulnerability analysis is a strategic process to analyze, identify and classify the security holes in an application's infrastructure thereby forecasting the effectiveness of the countermeasures or the security patches to be implemented (Krugel and Kirda, 2002; OWASP, 2012). Vulnerability analysis also known as vulnerability assessment, is a process that defines, identifies, and classifies the security vulnerabilities on the computer system, communications infrastructure (Cova and Balzarotti, 2007). Vulnerability analysis can also forecast the effectiveness of proposed countermeasures and evaluate their actual effectiveness after they are put into use (Avireddy and Srinivas, 2012). To perform a Vulnerability analysis, the major focus should be on procedures such as:

- Classification of the resources with its significance to business transaction

- Devising countermeasures to deal with probable threats
- Analysis of the consequences of attacks and a recovery mechanism.

To assess the vulnerability of an application, the following procedure or processing cycle is to be completed (Wassermann and Su, 2007). It is required to have a better understanding of the organization's infrastructure and critical processes to get the benefits of the vulnerability assessment (Lebeau and Franck et al., 2013). It is also needed to analyze the application, scan the application to assess the vulnerability and then implement the mitigation process (Lawal and Shakiru, 2016). Figure 2.2 displays the essential components involved in the assessment cycle.

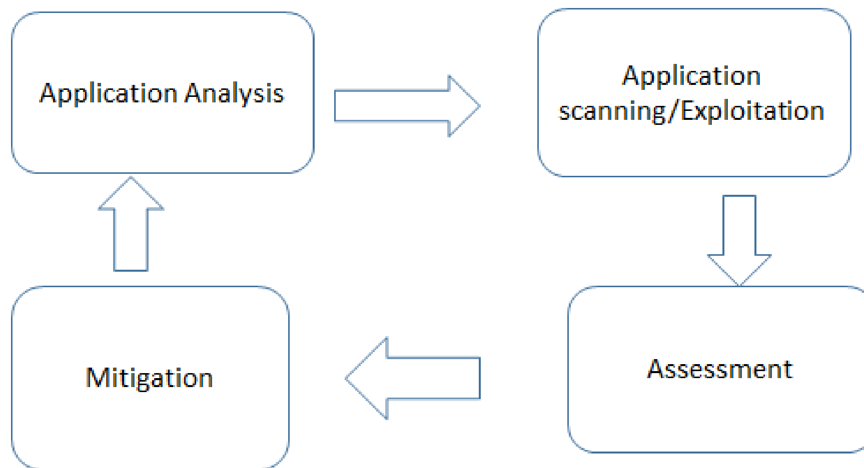


Fig. 2.2 Major components involved in Assessment

2.2.4 Major Tasks and Steps under Vulnerability Assessment

One of the major tasks to be completed in vulnerability assessment is to identify and understand the business process including the terms of compliance, customer privacy and competitors and specify the application

as per the criticality and sensitivity (Xiaowei and Yuan, 2011; Weinberger and Joel et al. 2011). The other main activities or tasks under vulnerability assessment procedure are:

- Identify the data storage devices, servers, hardware devices, network devices, hidden data sources and applications used for securing the application.
- Understand the policies and strategies, and security measures are already in place to protect the application
- Run the vulnerability scanning to understand its significance and understand and evaluate the business risk from the initial phase of vulnerability scanning result.

During the resource classification, we should assign the priority for the resources, and identify the potential threats for each resource along with the strategy to deal with the potential problem. An implementation plan to minimize consequences can be devised based on the severity of the vulnerability (Jovanovic and Kirda, 2006; Doupe and Vigna, 2010). Figure 2.3 shows the processing steps and the flow directions.

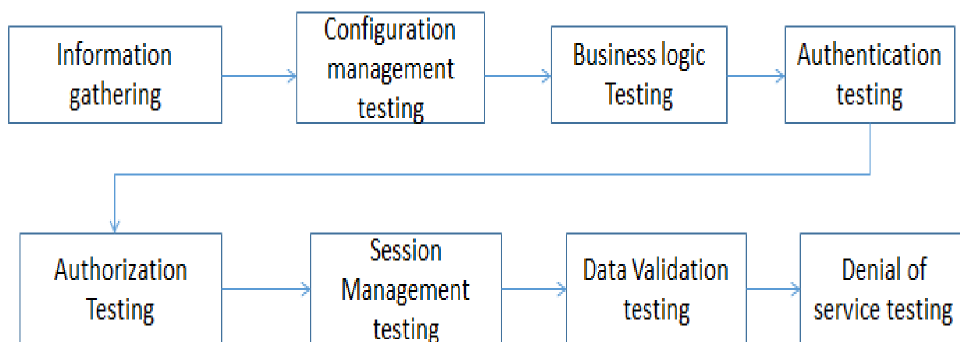


Fig. 2.3 Processing steps for Vulnerability assessment

2.3 Application Level Security Vulnerability

Application level security is the system flaw in an application that could be exploited to compromise the security of an application (Hu et al., 2006; Securosis, 2014; Xiaowei and Yuan, 2011). Here, the attacker uses a tool or method to discover the vulnerability and compromise the server. As reported by Veracode (an application security company), vulnerability is the weakness in the application and is mainly due to the design flow or implementation bug. Hackers with malicious intentions forcefully gain access to sensitive information in the online application and compromise the data stored on the server (Skrupsky and Bisht, 2013; MITRE, 2013). Web application security aims to address and fulfil the four conditions of security, also referred to as principles of security (Bau and Bursztein, 2010; Divya and Deepak et al. 2016):

- **Confidentiality**: States that we should not expose the sensitive data stored in the Web application under any circumstances.
- **Integrity**: States that the data contained in the Web application is consistent and is not modified by an unauthorized user.
- **Availability**: States that the Web application should be accessible to the legitimate user within a specified period depending on the request.
- **Nonrepudiation**: States that the valid user cannot deny modifying the data contained in the Web application and that the Web application can prove its identity to the legitimate user.

2.3.1 Major classes of Application Level Vulnerabilities

The vulnerability assessment can be concluded by conducting risk analysis on data exposure based on the data requirement & criticality (Balduzzi and Balzarotti, 2011). Suggesting appropriate countermeasures are also considered as an important activity during this stage. As per the report suggested by Gartner Security and Risk Management submit in June 2016, Application layer protection is significant, and application layer protection is completed only by focusing on the network layer protection (Halfond, William, 2008; Jovanovic, C. Kruegel, 2006). After classifying the vulnerability occurrence of various components involved in accessing the web application, focus of vulnerability assessment or analysis should be carried out by an emphasis on the critical assets of the organization, classification of legitimate users who may access the data and provide appropriate access privileges to each user (Huang and Hang, 2004). Table 2.1 shows the classification of application layer vulnerabilities and core factors determining the corresponding classification (OWASP, 2010; Halfond and William, 2008).

Table 2.1 Application level vulnerability-classes and determining factors

Classes	Determining factors
Application Specific	• Number of fields Vunerable • Exposed classified Information • Personal information
Server specific	• Maximum server load • Evaluation • Weak and hashed password • Obsolete authentication mechanism
Network specific	• Number of Proxy • Number of open ports • Firewall / proxy rules

A successful attack on web application are happening through network layer and protecting these layers is becoming very sophisticated with Denial of service attack or Distributed DoS (which are very common by flooding the server with an unwanted request). It is challenging to handle the application layer vulnerability only by getting a perfect code or implement a complete coding practice (Manmadhan and Manesh, 2012; Shuo and Zbigniew et al., 2006). Protection through network layer is also critical. Network layer signature for application layer attacks is very hard to develop. We cannot predict it as most of the attacks are blindsided. Attackers use a variety of automated tools or botnet to avoid detection, and the available cyber security solutions are not adequate to handle the fine-grained malicious activity that deviates from applications standard behavior (Kumar and Pranaw, 2012).

2.3.2 Validation Controls

Protecting the web application by patching up the loop holes or vulnerable points are an important task in the security vulnerability assessment. Each web application requires adopting an appropriate validation technique to perform this activity. The primary validation methods are whitelist validation and blacklist validation (Jovanovic and Kruegel, 2006). The whitelist validation process involves checking that the data is one of a set of tightly constrained known good values. We should reject any data that doesn't match (Armando, 2012). During the validation, we should give stage consideration to the following points such as:

- We should strongly type the data always

- Data length should be checked, and fields length should be minimized
- Data Range must be verified in case of numeric input
- Data should be unsigned unless required to be signed

In blacklist validation method data containing “known bad” character and patterns are rejected (Livshits and Michael, 2006; Buchler and Pretschner, 2012). The one mentioned above is a dangerous strategy because the set of possible bad data is potentially infinite. Adopting this strategy will have to maintain the list of “known bad” characters and patterns forever, and you will have incomplete protection(Calvi andVigan,2016; Huang and Hang, 2004). Improper Input Validation can result in significant vulnerabilities in the web application, which can be exploited by attackers. Input validation is a major concern for application security. Almost 50 to 60 % of security vulnerabilities in 70 % of are due to weak input validation (Seacord and Martin, 2010). The extensive use of digital devices in all areas of business, especially on web application give way to the conditions for cyber espionage programs and compromising the critical corporate data. Corporates increasing falls victims of cyber-attack (Belk, M., et al., 2011; Owasp 2010).

There many categories of validation control available as part of the vulnerability assessment (Gerkis 2010). The Compare Validator Control, compares a user's entry against a constant value, against the value of another control (using a comparison operator such as less than, equal, or greater than) or for a data type (Belk, 201). The Range Validator Control, checks that a user's entry is between specified lower and upper boundaries, can

check ranges within pairs of numbers, alphabetic characters, and dates (Gerkis, 2010). The Regular Expression Validator Control verifies that the entry matches a pattern defined by a regular expression and also enables you to check for predictable sequences of characters, such as those in e-mail addresses, telephone numbers, postal codes (Jovanovic and Kruegel, 2006). The Custom Validator control, usually checks the user's entry using validation logic that you write yourself. It is a type of validation that enables you to check for values derived at run time (Huang and Hang, 2004; Bertino and Early, 2007).

2.3.3 Sanitizing Strategies

During this change the user input into an acceptable format instead of accepting or rejecting the input (Khoury and Zavarsky, 2011). There are many sanitizing procedures available depending on the requirements of the application layer (Djuric, 2013). In Sanitize with the whitelist, any characters which are not part of an approved list can be removed, encoded or replaced. Sanitize with the blacklist is to eliminate or translate characters to make the input “safe”. Like blacklists, whitelist approach also requires maintenance (Akrout and Alata, 2014; Weinberger and Joel, 2011). As most fields have a grammar, it is simpler, faster, and more secure to simply validate a single correct positive test than to try to include complex and slow sanitization routines for all current and future attacks. In Comment Sanitization, the comments are the primary source of user input through text areas, JQuery based comment validation plugin can be considered. This plugin checks the syntax of the Author, email and text fields on the client side and reports errors in the syntax to the user (Halfond and Choudhary, 2011).

2.3.4 Tools and Techniques Used in Vulnerability Analysis

The most commonly used strategy to provide security or protect information is to make use of an appropriate scanner to test the application for any malicious entry (Seacord and Martin, 2010). Many scanning, and evaluation tools and techniques are available to deal with vulnerability assessment. It is not possible to suggest a single tool as a standard tool for analyzing all the security vulnerabilities in online application amidst many available tools (Jovanovic and Kruegel, 2006). But each tool has different functionalities. Open source vulnerability assessment technology supports business organization to have a cost effective way to scan application for known vulnerabilities (Bangre and Jaiswal, 2012). It is customized and conveniently bundled in security distribution. We use some of the tools as a general assessment tool, and there are tools used only for specific scanning on servers. If we find security holes as a result of vulnerability analysis, a vulnerability disclosure may be required (Maheswari and Anita et al., 2016). Injected code and critical threats through weak points within an online application can be easily traceable or detected with the implementation of an active vulnerability scanner (Rim, and Eric, 2014). Most of the scanning tools are expensive and do not support comprehensive analysis in purchased applications. It requires a series of changes in coding practices which would take its toll on time and space complexity. HP WebInspect by Hewlett-Packard is designed to perform a complete evaluation of the complete applications security (Federico, Macro et al., 2016). It will check the misconfiguration and precisely vulnerability on the application layer. A large number of both commercial and open sources that scans the web application to look for the known security vulnerability and have their strengths and

weaknesses (Divya and Deepak, 2016). Most of the online applications give priorities on Security testing with an appropriate tool and methodology. Making use of a suitable tool for protection is one of the necessary procedures to provide better security of the online application (Cheon and Lee, 2013). According to the Verizon 2014, Data Breach Investigation Report, 35% of breaches they tracked in 2013 are due to web application attacks, described as exploiting. By analyzing the vulnerability assessment results of more than 30,000 websites under management with WhiteHat Sentinel, the Organizations deploying these technologies can have a closer look at, particularly risk-prone areas. The majority of the available tools can detect vulnerabilities of the requested applications before the deployment (Gould and Devanbu, 2004). Appropriate simulation can be done by the scanners on a malicious activity by attacking and probing and analyzing the unexpected result. Most of the web scanners are a dynamic testing tool and language independent (Djuric, 2013; Shilpa and Priyadarshnini, 2016). As per the listing in OWASP, frequently used, commercial and open source tools /scanners for vulnerability assessment are: Samurai, Safe3, Websecurify, SQLMap, Burp Suite, contrast, Gama Scan, Grabber, Hailstorm, N-Stealth, Proxy app, QualySGuard, Securus, Sentinel, Vega, WebApp360, Web Scan Service, Web security suite and Wikto. Most commonly used active commercial scanners for assessment of security vulnerabilities(Federico and Macro, 2016; Djuric, 2013) are Web Inspect by HP, Rational AppScan by IBM, McAfee Secure by McAfee, HailStorm pro by Cenzic, WVS by Acoustics and NeXpose by Rapid7. A detailed description of the vulnerability scanning and testing tool is given as appendix-1.

Testing tools cannot cover the entire source code of the application, due to dynamic testing approach (Web Application Security Scanner Evaluation Criteria). Finding logical flaws such as the use of weak cryptographic procedures and leakage of information is very difficult for a scanning tool (Bishit and Venkatakrishnan, 2011; Weinberger and Joel, 2011). Most of the free tools are weak against the competent and broad targeting attackers as they are also aware of the free scanning tools. These tools are already equipped with the shortcut for bypassing that type of scanning (Romil, 2012). The scanning tools have a predefined list of attacks and do not generate the attack details for the tested web application (Tajpour and Atefeh, 2010). Most of the tools are limited in their understanding of the behavior of applications because of the dynamic content. Since there are no standard measures to know the strength or the effectiveness of the free tools, it is challenging to get convinced with the stability of the tool (Doupe and Cova, 2010).

2.4 Security Models Against Vulnerability Detection

The SQLR and model suggests a randomized SQL detect and abort queries which contain injected code (Avireddy and Srinivas, 2012). The process is done by modifying a query by appending a random number to it followed by placing a proxy server between the client's web server and application server, and the function of this proxy server would be to receive the request and pass it on to the database server. If we embed the request with SQLIA, it will not recognize the query and will end up with rejection. In CANDID framework, it proposes a test set creation by extracting query structures from every SQL query location in web application source code for avoiding SQLIA and also suggests web app code changes (Bisht and Prithvi,

2012).The verification method concludes with the issuing of the actual query if the test set matches. Authors in Web Application Vulnerability and Error Scanner (WAVES)is adopting a software-engineering technique to design a security assessment tool with some testing method such as black box testing and behavior monitoring(Konstantinos and Theodoros, 2008). In SQLCHECK, a runtime checking algorithm evaluates a real-world web application with a real dynamic attack data as input (Johari and Pankaj, 2012.). It prevents SQL injection attacks without any false positives and false negatives. AMNESIA (Analyzing Monitoring the SQL Injection Attack), a popular model against SQL Injection attack, is a combination of static analysis and dynamic analysis to analyze web application codes and also to monitor dynamically generated queries (Halfond and William, 2005). It is a query building model with all possible queries identified at the hotspot. The runtime monitoring mechanism will reject or report the queries that violate the model. Double-Guard, is a Java-based application, in this research work, to avoid object duplication, lots of statistical analysis and structure mapping is required (McClure and Russell, 2005). POSITIVE TAINTING suggests identifying trusted data by considering trust marked strings and performing syntax aware evaluation (Konstantinos and Theodoros, 2008). Tracking and taint marking should be accurate with a right level of precision. SQL DOM creates class tables and methods for possible operation. Database structure mapping will be done manually to avoid object duplication (Prithvi and Madhusudan, 2007). Webssari approach has a static analysis which uses an automated tool that works on certified inputs (Boyd and Keromytis,2004; Sadeghian and Zamani, 2013). SQL Injection protector for authentication (SQLIPA) technique uses hash value to get

better validation mechanism by using a hash value (Ali and Javed, 2009; Sahu and Tomar, 2016).

2.5 Vulnerability Assessment Reported by Security Organizations

Web security assessment is one of the most key areas to be focused on to have a better security of the critical Information stored in the databases. Incidents of Security attacks in an online application and its analysis to mitigate the attack is reported and documented by various security organizations are explained in this section. There are hundreds of vulnerabilities existing in each web application based on its demand of usage and sensitivity and confidentiality of information stored at the server level (OWASP, 2014). Authentication vulnerability, Authorization vulnerability, Code Injection vulnerability, Configuration vulnerability, error handling vulnerability and Logging and auditing vulnerability are some of the relevant categories of vulnerabilities listed by various security consortiums.

2.5.1 Kaspersky Lab B2B International

As per the survey conducted by Kaspersky Lab B2B International, the hackers attack more than 91 % of companies. And at least once in a year, almost 9% are victims of targeted attack (Kaspersky Lab B2B, 2016). Figure 2.4 shows the summary of the percentage of attack against the vulnerability categories.

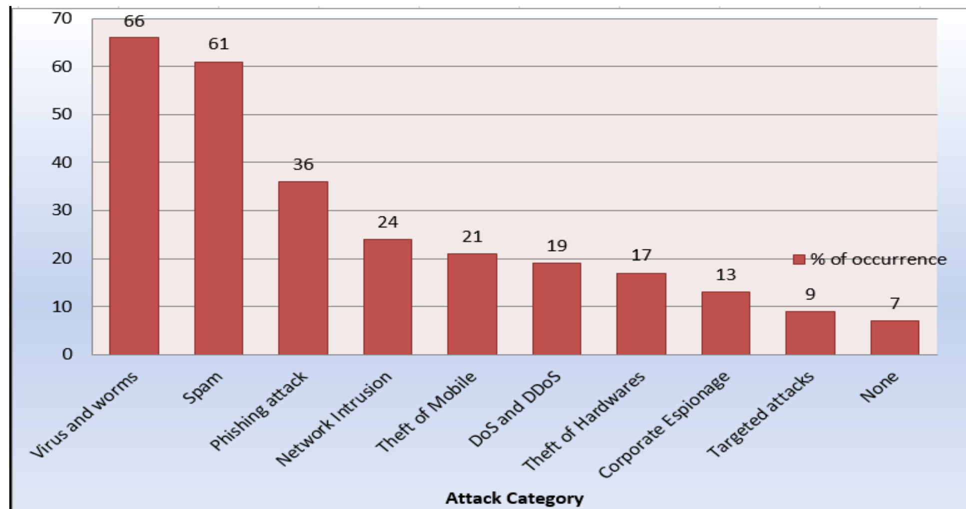


Fig. 2.4 Summary of percentage of attack against the attack categories

Kaspersky Lab B2B International reports also indicates that the top listed application layer vulnerabilities are: SQL Injection, LDAP Injection, Cross-site scripting, Cross-site Request Forgery and Insecure cryptographic storage. Most of the vulnerabilities exist due to the lack of appropriate validation on user input data, authentication mechanism, error handling strategies and efficient configuration of servers or server handling procedures.

2.5.2 Web Application Security Consortium (WASC)

As reported by (WASC TC v.2: Reports in the year 2015) (The severity of vulnerabilities was estimated based on Common Vulnerability Scoring System (CVSS) V.2) indicates that more than 63 % of an application implemented contain critical vulnerabilities and it will lead to sensitive data disclosure and system compromise. This report includes code and configuration vulnerabilities. The percentage of vulnerability based on

the industry are Telecoms (23%), Manufacturing (20%), Mass media 17%, IT (17%), Finance (13%) and government organization (10%). Based on the program code used, it can be classified as Java (43%) and PHP (30%). Based on Server category most common server was Nginx (34%), Microsoft IIS (19%), Apache Tomcat (14%) and Weblogic(14%).

2.5.3 Open Web Application Security Project (OWASP)

Most of the web applications contain security vulnerability indicated as almost 69% of Healthcare 70% of Retail and Hospitality, 76% of Government, 68% of Technology and 65% of manufacturing, 58% of Financial services affected by vulnerability or failure percentage of an application. (OWASP, 2016 Report, Veracode Top 10 Initial risk assessment). Figure 2.5 shows the percentage of the vulnerability reported by OWASP.

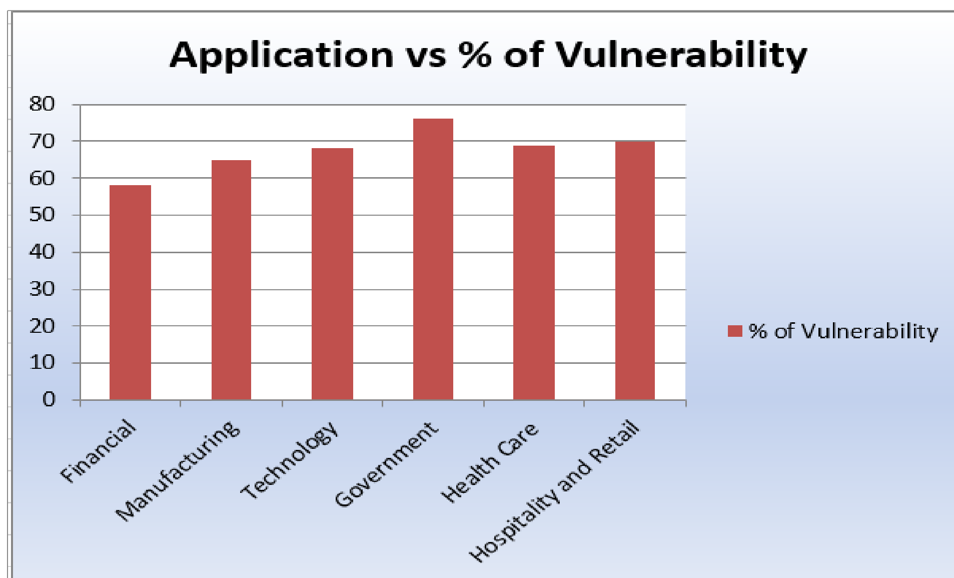


Fig. 2.5 Percentage of vulnerability reported by OWASP

2.5.4 Acunetix

(Acunetix, web application Vulnerability Report 2016) reveals that most of the websites are with severe vulnerabilities and more than 45,000 websites and network scans done on from April 2015 to March 2016. These statistics shows that almost 55% of web sites have one or more high-severity vulnerabilities and growing 9 % more in the upcoming year. The reports conclude by stating that 55 % web application scanned contained, high-severity vulnerability such as XSS or SQL Injection. 84 % were susceptible to a common vulnerability such as CSRF. % of all networks examined contained high security vulnerability.

2.5.5 IBM Security and Ponemon Research

(IBM Security and Ponemon research, 2016) (Align spending with risk): Indicates that majority of risks are happening at the application layer, but spending is mainly focused on network and data layer. Figure 2.6 displays the security risk vs Spending %.

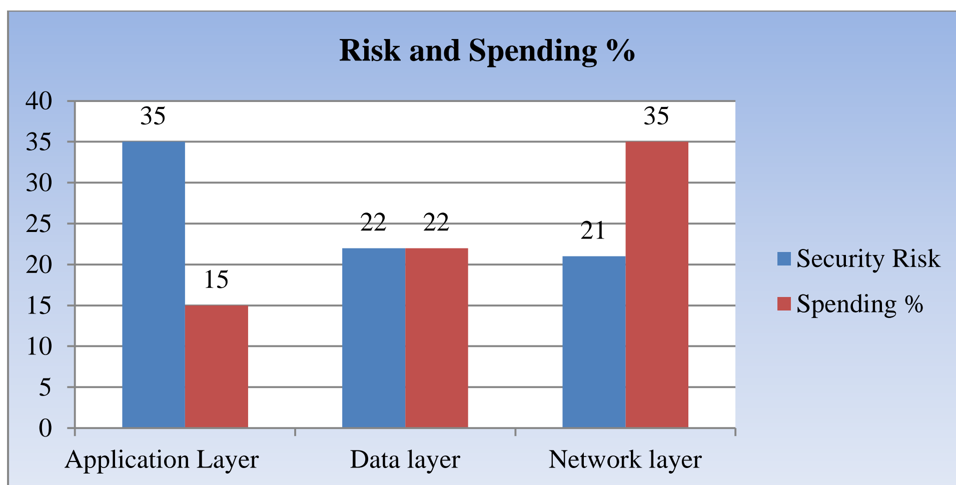


Fig. 2.6 Security risk Vs Spending

Vulnerability reports generated by Federal Bureau of Investigation (FBI) for the period between 2013 and 2014, reveal that the cybercrime is estimated to have cost the global economy more than $445 billion and there are more than 519 million financial records stolen by hackers in the US alone. As the report by, JP Morgan on October 2, 2014, in one of the worst incident, the data of approximately 76 million households and compromise 7 million small businesses. The Stolen data were said to include names, addresses, phone numbers, and email addresses. The hacker accomplishes this data breach through attacks against web applications used by the bank, where the attackers leveraged on vulnerabilities found in the web applications to gain access to the bank's internal network.

2.6 SQL Injection Attack- Most Dangerous Attack on Database Layer

SQL injection has become a popular website hacking tool because almost all the site invites input from dynamic users, who are using web forms to input data, web forms are vulnerable. Risk rate of successful SQL Injection attack is directly linked to the nature and size of business and duration, updates of patches on the application and the implemented security measures (Halfond and William, 2006; Clark, 2009). In SQL Injection Attacks, the crafted codes are directed to the database server, and these codes compromise critical information. SQL Injection attacks may be possible through cookies, server variables and second-order injection attacks (Khan and Khan, 2013). We can categorize most of the SQL-Injection under the first order and second order attacks, more specifically under the category of Tautology, Union Queries, and Piggybacked queries, logically incorrect queries, stored procedure, inferences and alternate encoding (Halfond and

William,2006; Joseph and Jevitha, 2016). Almost 24 million customers at Amazon's Zappos.com became a victim of SQL Injection attack in January 2012, and a few months later Yahoo voice was also hacked through. An SQL Injection Attack (SQLIA) increasingly targets online applications. The traditional security measures adopted by organizations are not sufficient enough to deal with new vulnerabilities and their attack spectrum. As the threat of SQL injection become more advanced, the need for developing a defense against SQL injection is greater than the past. In the hands of a very skilled hacker, a web application code weakness can reveal a root level access to application servers by bypassing firewalls and endpoint defense (Kindy and Pathan, 2011). Figure 2.7 shows an example of an SQL injection attack.

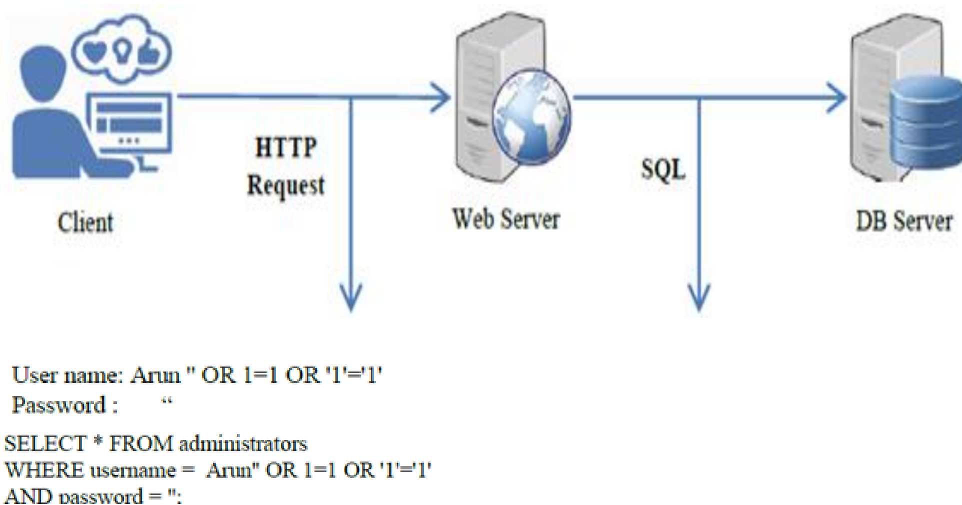


Fig. 2.7 SQL injection attack on a web application

We can consider most of the SQL-Injection under the category of Tautology, Union queries, Piggybacked queries, Logically-incorrect queries,

Stored procedures, Inferences and Alternate encoding (Halford and Orso, 2006; Johari and Pankaj, 2012).

In this attack, a hacker with an administrative privilege compromises the entire database. It takes time to realize the SQL injection attack. Hackers can delete the entire table, and a sophisticated SQL Inject attack can even corrupt the large database and disrupt the backup copies. The consequence of SQL Injection attacks such as authentication bypassing, information discloses, compromised data integrity, compromised availability of data, remote code execution and denial of service attack are considered as very severe attack in the application layer. (Ali and Shahzad, 2009). SQL Injection prevails as one of the top ten vulnerabilities and threats to online businesses targeting backend databases. Most of the cases the business organizations make it as a policy that the user input should never be trusted and must always be sanitized before it is used in dynamic SQL statements (Sadeghian and Ibrahim, 2013). It is observed that SQL injection appears only in a small proportion of applications and are yet making a massive impact on business organizations through data theft or compromising database servers (Swathy and Jevithan, 2016).

2.7 Summary of the Chapter

This chapter gives a summarized view of the literature survey conducted for the research study and analysis. Almost all the papers taken for the study reiterates that *the vulnerability analysis and patching up of these security holes for providing protection of online web application is a never-ending process, the developer and security professionals should come out with innovative strategies to protect the sensitive resources from the*

talented hackers with sophisticated attack tools. Identifying the vulnerability and patching it up at the right time will give lots of benefits to the business organization handling core security measures and will reduce the risk of security compromise. There are hundreds of scanning and detection tools available to mitigate injection vulnerabilities. If we use the open source web vulnerability scanners during the software development then the earlier detection of vulnerabilities, lower security assessment workloads performed before application deployment and decreased handling cost by limiting expensive licensing costs is possible. Just one identified automated tool may not be able to determine all vulnerabilities in a web application, and each one of the tools has its advantages and drawbacks, but some of these identified tools are used as an effective technique to detect and block vulnerabilities in the online applications. As per analysis and reports from various security organizations, we can conclude that application level security is one of the primary concern of web application developer and security professional. Code injection vulnerabilities are the major loopholes to be patched up by analyzing and detecting the vulnerable points. Hence this research study will be focusing on the code injection vulnerabilities, specifically the SQL injection vulnerabilities on a database server.

.....ꙮ.....

Chapter 3

THE MULTILEVEL TEMPLATE BASED DETECTION AND RECONSTRUCTION (MLT-DR) FRAMEWORK

Contents

3.1 Introduction
3.2 Major Attack Categories of SQL Injection
3.3 Standard Queries and its Malicious Pattern
3.4 The General Layout of the Proposed Architecture
3.5 The MLT-DR Framework
3.6 Template Design Strategies
3.7 Summary of the Chapter

Vulnerabilities in web application allow malicious users to get unrestricted access to databases. The literature survey conducted (as explained in chapter 2) during this research study indicates that SQL injection vulnerabilities are the most dangerous threats to the database layer of the web application security. Using these types of vulnerabilities, 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. SQL injection takes advantages of the design flaws in poorly designed web applications to poison SQL statements and bypasses the standard methods of accessing the database content. This chapter mainly deals with the general architecture of Multi Level Template based Detection and Reconstruction (MLT-DR) framework. The proposed multilevel template based framework is a specially crafted vulnerability detection model. It is working on multi-level token based design template, to detect the illegal queries before they are executed on the database server.

3.1 Introduction

The **MLT-DR** hybrid framework is divided into two major modules. We use the first module to detect and block the queries with the support of the proxy server and a software crawler to analyze the application code to make the legal query model and later map it with user input queries at the run time. The second module of this hybrid framework proposes the reconstruction of injected query, and it is developed and implemented with the support of machine learning using back propagated Artificial Neural Network algorithm. Reconstruction of the queries from the authenticated users is carried out by using REGEX functions. Denial of accessing the database by the authenticated users is mitigated with this approach. Logged in malicious queries as indicated in the framework are redirected for further analysis and support for identifying the new pattern of malicious queries. The rest of the sections include the details about the different components involved in the framework, SQL Injection attack categories and pros and cons of attacks.

3.2 Major Attack Categories of SQL Injection

The SQL injection is classified into seven major attack categories based on the injected code or string and procedure or pattern of injection (Halfond and Orso, 2006; Johari and Pankaj,2012).

3.2.1 Tautologies

The hacker injects a code into a conditional statement to evaluate it as true by which the malicious user can then bypass 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';*** the comparison expression uses one or more relational operators to compare the operands and always generate true conditions. The targeted query may return all the rows in the book table. ***The possible signature for this type of SQL injection attack are the string terminator "'", OR, =, LIKE and SELECT***. It is a type of attack in which hackers try to bypass authentication and extract data from database (Halfond and Orso,2006).

3.2.2 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***'. The debugging information shown in the rejected query will reveal the database table information which can later be utilized to conduct further attacks (Sadeghian and Ibrahim, 2013).

3.2.3 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***. Here the attacker joins injected queries to a safe legal query with a word UNION and get details from different tables than the intended ones. Attackers

mainly use this technique to bypass the authentication and extract data (Halfond and Orso,2006).

3.2.4 Piggy-Backed Queries

Attacker tries to inject additional queries along with the original queries, which are said to 'piggy-back' 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***.After executing the first query, the database encounters the query delimiters (;) and executes the second query (Khan and Khan,2013).

3.2.5 Alternate Encodings

In these types of attacks, the char () function and ASCII /hexadecimal encoding can be used. For example:***SELECT accounts FROM users WHERE login="" AND pin=0; exec (char (0x73687574646j776e))***. Hackers use this technique to avoid being identified by secure defensive coding and automated prevention mechanisms by modifying the query using alternate encoding such as ASCII or hexadecimal coding practices (Avireddy and Srinivas, 2012).

3.2.6 Stored Procedure

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

use a shutdown command with which the back-end database will be shutdown (Halfond and William, 2006; Joseph and Jevitha, 2016).

3.2.7 Inference attack

It is a type of attack in which hackers aim to identify the injectable parameters and extract data from databases. It can be further categorized as: Blind SQL injection attack and Timing SQL injection attack. 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 ' and column _ name > 'user')=4) WAITFOR DELAY '00:00:10'***—in this example the hacker tries to work by understanding the behavior of the back-end database by injecting an always true statement and along with a "***WAIT FOR***" keyword (Maor and Shulman, 2004).

3.3 Standard Queries and its Malicious Pattern

Table 3.2 shows the different malicious pattern of the standard and injected queries which are attacking the database server (Halfond and William, 2006; Joseph and Jevitha, 2016). Detailed description of the legal and malicious queries is given in the appendix. Most of the queries shown in the appendix is considered as test bed for MLT-DR framework.

Table 3.1 Standard query vs Malicious query

Legal Query	Injected Query
SELECT deductible FROM policy as p WHERE inputPolicy=$input11 OR id=$input12	SELECT deductible FROM policy as p WHERE inputPolicy=$input11 OR id=$input12 UNION SELECT d.insuredname FROM dependents as d WHERE inputPolicy=$input21 OR id=$input22;
SELECT * FROM users WHERE userid=22;	SELECT * FROM users WHERE userid=22 UNION SELECT body,results FROM reports;
SELECT COUNT(*) FROM reviews WHERE review_author=**'MadBob';**	SELECT COUNT(*) FROM reviews WHERE review_author=**'MadBob'** UNION SELECT IF(SUBSTRING(USER(), 1,4)='root',SLEEP(5),1);
SELECT name,address FROM customers WHERE name like '%a%'**;**	SELECT name,address FROM customers WHERE name like '%a%'union Select NULL,LOAD_FILE('/etc/passwd')#
INSERT INTO users (id, username, password) VALUES (1, 'Jane', 'Eyre');	INSERT INTO users (id, username, password) VALUES (1, 'Olivia' or (SELECT 1 FROM(SELECT count(*),concat((SELECT (SELECT (SELECT concat(0x7e,0x27,cast(users.username as char),0x27,0x7e) FROM `newdb`.users LIMIT 0,1)) FROM information_schema.tables limit 0,1),floor(rand(0)*2))x FROM information_schema.columns group by x)a) or '', 'Nervo');
INSERT INTO users (username, password) VALUES('jack','');	INSERT INTO users (username, password) VALUES('jack',''); DROP TABLE users;

Table 3.1 continued….

SELECT * FROM customers WHERE username = 'timmy'	SELECT * FROM customers WHERE username = '\';SHUTDOWN;
UPDATE users SET password='Nicky' WHERE id=2 and username='Olivia';	UPDATE users SET password='Nicky' or updatexml(1,concat(0x7e,(SELECT concat_ws(':',id, username, password) FROM newdb.users limit 0,1)),0) or'' WHERE id=2 and username='Olivia'
DELETE FROM users WHERE id=2;	DELETE FROM users WHERE id=2 or updatexml(1,concat(0x7e,(version())),0) or'';
DELETE FROM users WHERE id=2;	DELETE FROM users WHERE id=1 or updatexml(0,concat(0x7e,(SELECT concat_ws(':',id, username, password) FROM users limit 0,1)),0) or '';
DELETE FROM users WHERE id=2;	DELETE FROM tablename WHERE id ='x'; Exec(char(0x73687574646f776e));--'
DELETE FROM users WHERE id=2;	DELETE FROM users WHERE id ='2′ UNION SELECT name, cast((mb_free) as varchar(10)), 1.0 FROM haxor;--
UPDATE users SET password='Nicky' WHERE id=2 and username='Olivia';	UPDATE users SET password='Nicky' WHERE id=2 and username='Olivia'; DROP TABLE haxor;CREATE TABLE haxor(line varchar(255) null); INSERT INTO haxor EXEC master..xp_cmdshell 'dir /s c:\';--
UPDATE users SET password='Nicky' WHERE id=2 and username='Olivia';	UPDATE users SET password='Nicky' WHERE id=2 and username='Olivia' UNION SELECT line, '', 1.0 FROM haxor;--
INSERT INTO Favourites (UserID, Friendly Name, Criteria) VALUES(123, 'My Attack','');	INSERT INTO Favourites (UserID, FriendlyName, Criteria) VALUES(123, 'My Attack','''; DELETE Orders;--')

Table 3.1 continued….

SELECT * FROM Products WHERE Product Name = 'abc';	SELECT * FROM Product WHERE ProductName = ''; SHUTDOWN WITH NOWAIT;--
UPDATE users SET password='Nicky' WHERE id=2 and username='Olivia';	UPDATE users SET password='Nicky' WHERE id='2' or username='Oliva'--and username='';
UPDATE users SET password='Nicky' WHERE id=2 and username='Olivia';	UPDATE users SET password='Nicky' WHERE id='2' or 'one'='one' /*;
DELETE * FROM users WHERE id='2' AND username='abc';	DELETE * FROM users WHERE id='2' AND username='' OR 'ab'='a'+'b' #;
DELETE * FROM users WHERE id='2' AND username='abc';	DELETE * FROM users WHERE id='2' AND username='' OR 'ab'='a'||'b' /*;
SELECT * FROM users WHERE userid=22;	SELECT * FROM users WHERE userid='' OR 'ab'='a''b' /*;
INSERT INTO users (username, password) VALUES('jack','');	INSERT INTO users (username, password) VALUES('jack','');SHUTDOWN--
DELETE FROM users WHERE id=2;	*DELETE FROM users WHERE id ='2' UNION* CREATE TABLE haxor(name varchar(255), mb_free int); INSERT INTO haxor EXEC master..xp_fixeddrives;--

3.3.1 Attack category and Signature

The attack categories and their common attack signature are summarized in Table 3.2 (Lee and Wong, 2002; Lebeau, Franck, 2013).

Table 3.2 Attack category and signature

Attack Category	Sample query & signature
Tautology	SELECT * FROM user WHERE id=’1’ or ‘1=1’-‘AND password=’1234’; “or 1=1” ***‘,OR,=,like, select***
Logically incorrect Query	URL:http://www.toolsmarket-al.com/veglat/?id_nav=2234 2) **Invalid conversion(CONVERTTYPE(TYPE)), incorrect login, AND, ORDERBY**
Union Query:	SELECT Name, Phone FROM Users WHERE Id=$id. By injecting the following Id value: $id =1 UNION ALL SELECT credit Card Number, 1 FROM Credit sys Table ***Union, union select***
Piggy-Backed Queries:	SELECT Login_ID FROM users_ID WHERE login_ID=’john’ and password=’’; DROPTABLE users-‘ AND ID=2345 ; ***;***
Stored procedure	*INSERT INTO users (username,password) VALUES('jack',''); SHUTDOWN;* ***SHUTDOWN, XP_cmdshell(), sp-execwebtask()***
Blind Injection	SELECT accounts FROM users WHERE id= '1111' and 1 =0 – AND pass = AND pin=0 SELECT accounts FROM users WHERE login= 'doe' and 1 = 1 -- AND pass = AND pin=0 ***;,and,if else,waitfor***
Alternate Encodings:	SELECT accounts FROM users WHERE login='' AND pin=0; exec (char (0x73687574646j776e)) This example use the char () function and ASCII hexadecimal encoding. **Exec(),char(),ASCII(),BIN(),HEX(),UNHEX(),BASE64(),DEC(), ROT13()**

3.3.2 Classification Strategy

We can classify the SQL injection attacks into various categories based on the exploitation techniques and/ or the injected parameter embedded in the query (Narayan and Mohandas, 2011; Mohosina and Zulkernine, 2012). If the attack category is known, then the identification of parameters/strings used for injection is easier and faster. In most of the cases the attacker uses the following functions or techniques to inject the code into the SQL statement (Pietraszek and Berghe, 2005; Johari and Pankaj,2012):

- Adding Boolean conditions on the SQL Query input.
- Inappropriate use of Union operators.
- Error based techniques / trying with logically wrong queries.
- Out of band techniques or using a different communication device to access the database
- The time-delay in the query and perform further attacks or injections based on the output.

3.4 The General Layout of the Proposed Architecture

The proposed Multi Level Template based Detection and Reconstruction (MLT-DR) framework against SQL Injection attack, is a token based detection and reconstruction approach to detect the illegal queries before they are executed on the database server. In the proposed model, each user input query will be parsed and analyzed using the multi-level template based detection technique before the actual execution at the backend

database server (George and Jacob, 2016). The main objective of this template based framework is to detect and block the SQL injection in web applications without many computational overheads by preserving the efficiency and effectiveness of the given web application.

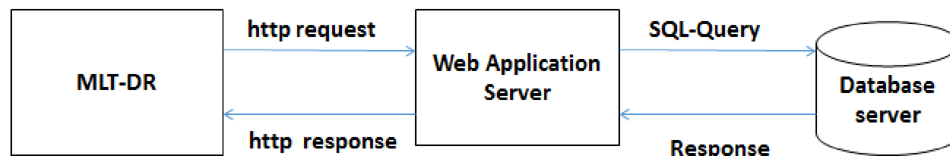


Fig. 3.1 General layout of MLT-DR architecture

The proposed architecture does not demand any source code modifications and can perform detailed analysis at negligible computational overheads without false positives or false negatives. It is an effective token-based model designed and implemented to detect and block SQL Injection attacks by parsing and analyzing dynamic queries against the intended query pattern given in the model. The most significant aspect of this framework is that it prevents all forms of SQL injection attack type and we can implement it in any web application irrespective of the size of the application, type of transactions without much modification on the existing web application.

Most of the code injection detection approaches implemented currently is based on primary areas of content filtering, penetration testing and defensive coding methods (Ezumalai and Aghila,2009; Kindy and Pathan, 2012). Some of the detection approaches encountered during the survey are listed as follows:

- Authentication schemes based on encryption techniques
- Intrusion detection system on networks
- Secure query processing strategies
- SQL injection and detection methods by removing the SQL query attribute values
- Combinational methods for detecting the SQL injection attacks
- Analysis and monitoring for neutralizing SQL Injection attacks

Each method and strategy has its advantage and drawbacks (Wagner and Soto, 2002; Haung and Tsai, 2003). Some of the detection prevention techniques have one or more of the following weaknesses:

- Complex analysis and detection frame work
- Incomplete implementation strategies
- Intensive statical analysis on the application code
- Runtime overheads
- Time-space complexity
- False positives and false negatives

3.5 The MLT-DR Framework

The Multilevel Template based Detection and Reconstruction framework (MLT-DR) is a hybrid model in which module-1 is a Template based Detection (TbD), and module II is a Reconstruction framework. In the proposed MLT-DR framework, the weakness mentioned above are taken

into consideration and could overcome all the issues noted above in the back-end database server of a web application. MLT-DR framework detects/prevents SQL-Injection attack under the categories of tautologies, logically incorrect queries, union queries, piggy-backed queries, stored procedures, timing attacks and another encoding. In this approach, the template of the intended structure of the SQL statements produced by the applications is parsed and stored statically. During the dynamic interactions, these statically stored standard queries are mapped with the dynamic user queries, if there is any violation detected, the dynamic input queries are considered as the malicious queries and blocked for further database access. The following lists the primary tasks or procedures in the SQL Injection detection phase:

- Identify the SQL statements accurately in each web page
- Analyze the query for parsing
- Generate different tokens as per the template specification,
- Lexical analysis/ map the legal queries with the dynamic user input queries
- Validate the token
- Generate appropriate detection result/status message
- Block the SQL query or give the privilege to access the database based on the result

Figure 3.2 Shows the details and components of the proposed framework.

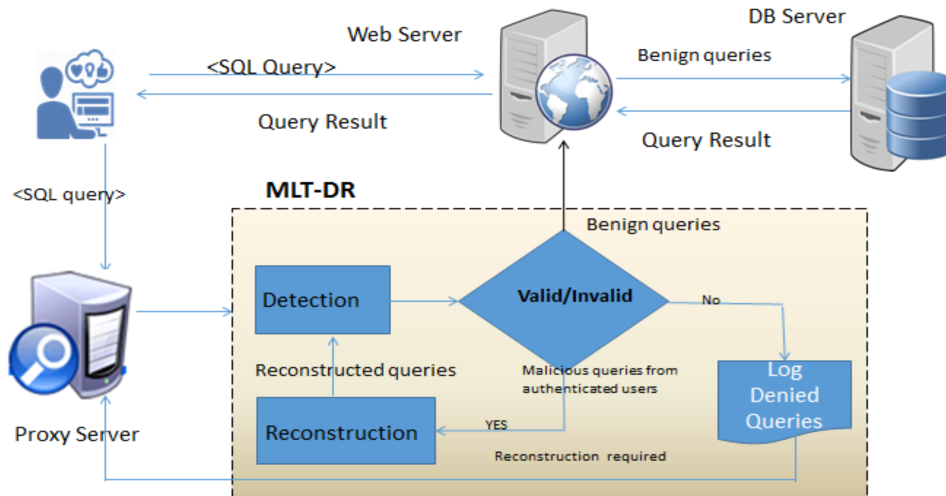


Fig. 3.2 Architecture of MLT-DR framework

It accepts user requests through dynamic web pages and directs the queries to a proxy server, and the proxy server then forwards the queries to MLT-DR module to check for any SQL injections. If SQL injection is not detected, the module redirects such benign queries to the web server and allows them to access the DB Server. But if an injection is detected, such malicious queries are blocked, and if the query is from an authenticated user, we forward it to Reconstruction module for reconstruction using REGEX function. Otherwise, the denied queries are logged in for further malicious query pattern analysis. Reconstruction module receives the malicious query and reconstructs it and forwards the benign (reconstructed query) to the server which in turn allows the query to access the DB server. The proposed framework provides better system availability thereby reducing denial of services. We perform the Reconstruction of queries only when the query is generated from an authenticated user. The following

subsections explain the components of proposed hybrid architecture and strategies.

3.5.1 Proxy Server

In a networked environment, a proxy server acts as an intermediary for accessing the resources from other servers. Most of the organizations are using proxy servers to facilitate security and administrative control. We use proxy servers for monitoring traffic for detecting malicious access or ensuring user privacy. Based on the primary activity it performs, proxy servers can be under the different category. Here, we use the proxy server as an interface between the user browser and MLT-DR framework for detecting the malicious query entered through the user browser and create legal query model. The proxy server receives a query from user browser as form fields and pass it to the anomaly detection framework to the validation process and then to the evaluation engine. So that web server is getting only benign queries and is directed to the database server. One of the primary functions of the proxy server is to create a legal query model to block the malicious entries and protect the database layer of the web application. It can also be used to prevent DoS attack and network intrusion by enforcing secure authentication and authorization strategies to block the malicious entries (Wei and Kothari, 2006; Ntagwabira and Kang, 2010).

3.5.2 Web Server

Web servers are usually designed to serve HTTP content most suitable for handling the static content that interprets user request and forward it to the application server. They store, process and deliver web pages to clients. Web servers are built for hosting and maintaining the website. Most of the

web servers are having the features such as create one or more websites, configure site/ directory security, create FTP sites and virtual directories, configure /nominate custom error pages and specify default documents. In our framework, the web server accepts the user input/SQL queries from the client browser and validates it with the legal query templates from MLT-DR framework, only the benign queries are diverted to get an actual database access.

3.5.3 Database Server

The database server is dedicated to database storage and retrieval. It is supported by a Database Management System (DBMS) and represented in a client- server environment. A database server is also known as SQL engine. Database server controls all database functions. A standard called Open Database Connectivity (ODBC) provides an application programming interface (API) which permits client-side programs to invoke DBMS at the server side and sends back the result to the client program. Once the data is stored in a database, any database application software can be used within the database server. The user requests received through the HTTP and dynamic web pages are executed here. Accessing data on a database server is possible only through SQL statements validated with the support of the proxy server. Database server handles all database access and control functions and hides the DBMS functions from the client. In most of the situations, the database server manages the recovery security services of the DBMS and enforces the constraints specified within the DBMS. A high level of management and security is required at database layer (Xie and Aiken, 2006; Ali and Javed, 2009).

3.5.4 The Detection Module (TbD)

One of the primary objectives 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. In the training phase, the proxy server receives the user queries, and with the support of a software crawler, it locates the hotspots in the application. Then the similar queries are identified and parsed to create Temp_ID and format. The template repository stores the legal query tokens with a unique identity and a particular format. This method uses an efficient query validation technique to match the statically generated legal query tokens against the parsed dynamic query tokens at runtime (Win & Htun, 2014). The major components and working details are explained in Chapter 4 by considering each element.

3.5.5 The Reconstruction Module

Most of the current SQLIA detection and prevention approaches reject the dynamic query if there is an' Injection Attack' in the given input query. Rejecting the queries is directly and denying the authenticated users access to the system and sometimes reduces system availability, especially in the case of false positives. One of the primary objectives of the proposed model is to reconstruct the queries by eliminating injections and rebuilding missing portions of the 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 or it can use the customized legal query model developed in the training phase of the MLT-DR (Lebeau and Franck, 2013). We learn the ideal query model for the seven identified attack categories using the machine learning technique, for further process and reconstruction.

The Neural Network-Reconstruction provides better user access to a web application and reduces the denial of service attack by facilitating the reconstruction option for the authenticated user query. This framework facilitates reconstruction of all types of queries from authenticated users based on the query structure and format that will be as required by the underlying database. Chapter 5 discusses the detailed description and evaluation procedures of reconstruction module.

3.5.6 Authentication Checking

Authentication is the process of verifying the identity of the user and devices to resources or the web application itself, which will provide a trust relationship for further access to the valuable resources. It also enables the accountability of the user to link access to the particular identities (Prabakar and Marimuthu, 2012). Most of the web application is having a strong authentication mechanism (user-id and password verification) to check whether the queries are from a valid user or the authentication credentials are matching with the information stored in the database table. In this research, the web applications use an authentication procedure to ensure that a valid user has requested the query. If the query is from a rightly authenticated user, it will be diverted back to the reconstruction module to remove the injected part of the malicious user query and reconstruct the query as per the requirement.

3.5.7 Log of Denied Queries

We use the query log as a record keeping system; it will be beneficial for conducting a detailed analysis of the malicious pattern of a suspected query. In the hybrid framework, if the query is detected as a malicious query

and identified it as, not from an authenticated user then it will not be proceeded to query reconstruction framework. It will be logged in/blocked for further process. All the blocked queries are documented or kept as a log file, in a data structure server, which can be later utilized in a proxy server to create a new model with the newly injected malicious pattern for analysis and mapping procedure for a legal query.

3.6 Template Design Strategies

Designing appropriate template of a token by extracting it correctly from the given SQL statement (or a multilevel analysis in the case of a complex query) and retrieving it from the store for mapping is considered as one of the primary tasks in this research. During the training phase of this approach, all possible types of legal queries/SQL statements of the standard web application are collected by using a web application crawler tool for the vulnerability analysis, parsed using JSON parser, assigned a unique ID for each query, and stored in the template repository. In the testing phase, the dynamic queries are parsed and validated against legal query template stored in the repository. If there is no match found between the injected and input queries, then the queries executed through the dynamic web pages were flagged as a malicious query and blocked from further execution at the backend database server. We can implement this approach in any web application irrespective of the type of database management system at the application layer. Using this approach, we do the query evaluation and mapping of queries without much computational overhead. One of the most important strategies in a query validation phase is to identify the types of channels/devices used for data access. The SQL-Injection categories are

based on these channels and types of data access (Rawat and Raghuwanshi, 2012). The following categories list the channels:

- In band: Data retrieved using the same channel
- Out-of-band: Data retrieved using different channel
- Inferential or Blind attack: Reconstruct the information by analyzing the database behavior.

3.6.1 Strategies used for Storage and Retrieval

The performance and speed of the web application are significant in the current online business transaction. There should be appropriate protocol, network speed and storage strategies to reduce the page download time of an application. If there is a proper integration of all these technologies, it can reduce the web generated traffic by 50 %.

3.6.2 JavaScript object Notation Format (JSON)

JavaScript Object Notation is a lightweight data-interchange format, an accessible format for humans to read and write and very easy for a machine to parse and generate. While interacting with the website, the JSON feeds can be loaded asynchronously and much more quickly than XML/RSS. It is considered as a universal data structure; almost all programming languages support them in one form or another.

Usually the JSON is built on the following structures:

- A collection of name/value pairs, such as an object, structure, dictionary, hash table, keyed list, or associative array.
- An ordered list of values, such as an array, vector, list, or sequence.

The following Figure 3.3 shows the standard query template tested in this research and the corresponding **JSON** format.

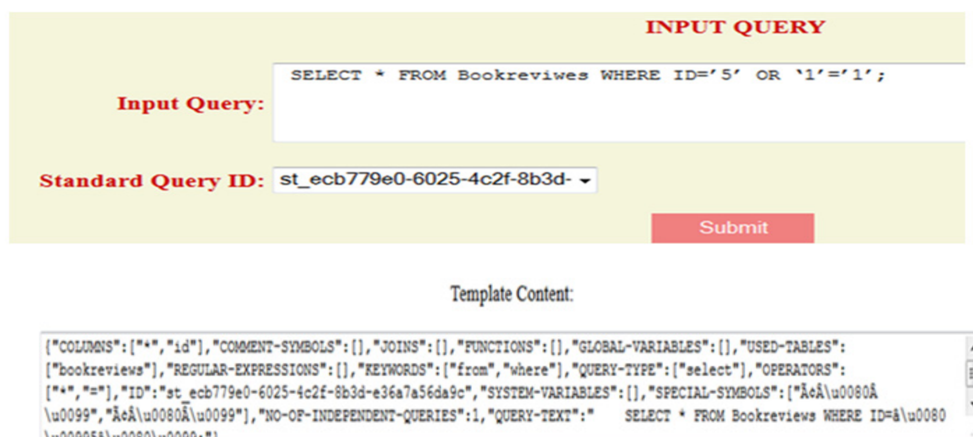


Fig. 3.3 Query ID and corresponding JSON format

There are various built-in functions and operators available in each server application that will enable the application to parse the text, modify the values and transform arrays of JSON object into a table format. Here the queries are converted into JSON objects. It is very cost effective to store the data in JSON format, and easy to access and extract values from JSON format. There are various functions available in each application program to evaluate, extract, validate and manage JSON format.

3.6.3 JAR file to Retrieve and to Specify the Path Details

Java ARchieve (JAR) is a file format based on the popular ZIP file format. It is used for aggregating many files into one. The primary objective of a JAR file is to download Java applets and their requisite components such as class files, images and sounds into browser in a single HTTP transaction, rather than opening it as an individual file. It is also considered

as an effective achieve tool. If the application is using JAR format, it can tremendously improve the speed with which an application can be loaded onto the dynamic web page and start functioning. It also supports compression to reduce the file size and improve download time. The special features of the JAR files are:

- Achieve format that is cross platform
- Only format that handles audio, image files and class files.
- Backward-compatibility with the existing applet code.
- It is written in JAVA, an open standard, fully extendable.
- Most preferred way to bundle the Java applet.

Following are the details of one of the effective JAR files "SQLIAShield" developed for handling several types of query template ID handling in dynamic web pages.

The core of this functionality is a class called "SQLIAShield" which can be instantiated with a parameterized constructor with parameter values as standard query template path and an output folder path. A sample SQLIA shield can be invoked using the following format/path specification:

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

There is also an authentication function available with this type of file, where individual entries in a JAR file can be digitally signed by the author of the applet to keep track of the origin of the authentication.

3.7 Summary of the Chapter

A multi-Level Template based SQL Injection detection and prevention model which has an effective technique to validate SQL queries before it reaches and is executed by the database server can be implemented on a proxy server. It is a novel template based approach to detect and prevent the SQLI attack categories such as Tautologies, Logically incorrect queries, Union queries, Piggybacked queries, stored procedure, Timing and another encoding. This frame work can be implemented either within a proxy server or as an API. The parsing techniques used in the template creator application is challenging to scale up the performance (with negligible false positive and false negative rate) of the proposed model, and the template files stored in the JSON format contribute equally in decreasing the storage overheads.

.....ଛଔ.....

Chapter 4

THE MULTILEVEL TEMPLATE BASED DETECTION FRAMEWORK (MLT-D)

Contents

4.1 The Multilevel Template Based Detection (MLT-D) Framework
4.2 Strategies Used in Standard String Matching Algorithms
4.3 Experimental Result
4.4 Summary of the Chapter

The template based detection framework is an effective framework for SQL injection detection and is a malicious query blocking model. It is designed and implemented by validating dynamic queries against the legal query pattern. In this framework, the proxy server executes user queries and validates it before redirecting it to the web server. It blocks malicious queries and generates an alert message if the injection is detected. Only the benign query can access the data from the back-end database server. Mostly, people design the web applications with multiple pages with multiple data entry fields by the users, which can be the hotspot for SQL injection attack. Frequently used Injection detection tools require source code modification which is a tedious task and will affect the performance of the underlying web application and the storage requirement is also high. The proposed architecture does not demand any source code modifications and can perform detailed analysis at negligible computational overheads without false positives or false negatives. The time/space complexity of verification method is also in proportional to the complexity of query under consideration. The proposed approach has a fully automated query assessment procedure to build a legal query model and has a validation technique to match the design template with the dynamic user query.

4.1 The Multilevel Template Based Detection (MLT-D) Framework

The Multilevel Template based detection framework (MLT-D) is designed and implemented with the support of efficient query template creator and validation strategies. In this detection and blocking model, we analyze the schema or structure of all possible types of queries, validate and then create templates and store it in the template repository, corresponding to each SQL query. The proxy server deployed in this framework has the provision of masking the location of the database server and permits only the authenticated valid queries to access the web server (Panda and Ramani, 2013; Awang and Manaf, 2015). The major components of the Multilevel Template based Detection (MLT-D) module are the parser, detection technique and mapping procedure as shown in Figure 4.1.

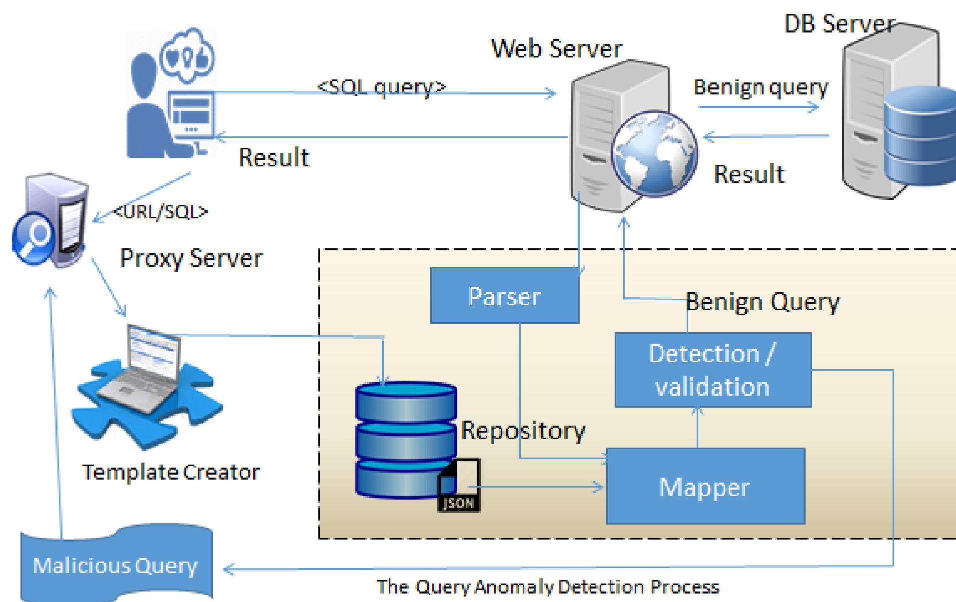


Fig. 4.1 Multilevel template based Detection (MLT-D)

4.1.1 The server Functionality

The web server receives database requests from Query Anomaly Detection Process (QUADP) and forwards it for the execution or accessing the resources from the actual database server. A proxy server is used at the training stage to identify the hot spots of code injection attack and corresponding SQL statement to build the model for further validation. Only benign queries can access the database server thus protecting the database layer. The database server stores the business data that is critical to the online business transaction; the database server has the responsibility of executing and giving the actual result user queries (Khari and Kumar, 2013).

4.1.2 The Standard Query Template Creator Application (SQTC)

One of the most significant components in the template based detection model is the Query Template Creator Application. The standard query template creator module (SQTC) analyzes the web application using a Java application based code analyzer kept at the proxy server, identifies and collects the standard legal queries in the web application. The primary tasks under the template creator component are to parse them into corresponding smaller tokens. In most of the web applications, there are multiple SQL requests (form fields) in each web page as the hotspot for malicious user entries. A web crawler/spider can spot these fields. We split complex queries into several independent queries by using a depth-first tree traversal procedure, and each separate query is tokenized (Valeur and Giovanni, 2005; Sangkatsanee and Charnsripinyo, 2011).

4.1.2.1 Model Creation Algorithm

The identified dynamic queries are parsed or tokenized for further template mapping process. Here the tokens are generated as per the template

specification mentioned in the algorithm given below. We split each SQL query into the predefined template such as query-type, used-tables, columns, system-variables, global variables, functions, joins, special-symbols, operators, comment symbols and keywords. There is a unique ID (Sq-Id) created for each parsed query, and we store the query in JSON format in the template repository. There can be multiple Sq-Ids in each web page, based on the underlying application. The algorithm considers each independent query for validation and creates a unique ID for the respective query (Vigna and Robertson, 2009). It stores the query template in the template repository in JSON format.

Table 4.1 Standard Query Template Creator Algorithm (SQTC)

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

In the algorithm mentioned 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 such as 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 store the corresponding template details in JSON format. During parsing, if there is any extra field in the complex 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. There can be “n” additional columns created based on the input query type.

4.1.3 The template Repository

The template repository holds all possible legal query identities (IDs), template (TI) of the underlying online applications and stored in JSON format. The corresponding ‘ID ‘for each page is arranged and stored in the repository in a unique format by using a jar/package file facility available in the application for each web page. We store the legal query models/templates in a template repository with a unique identity and a specific JSON format.

4.1.4 Token based Query Model Constructor and Parsing Procedure

We test SQL queries against injection detection by splitting them initially into templates based on the number of Independent queries within the full query and the number of tables used. In this research work, the parsing techniques used split the query as per the template specification. The tokens are generated as after a rigorous analysis of queries or SQL statements from the underlying web applications for the analysis. Usually during the parsing, the parser will check the SQL statement for the following conditions of (Buehrer and Gregory, 2005):

- Incorrectly handled escape characters
- Incorrectly handled types
- Insecure Database Configuration

In the proposed approach, the SQL injection attacks are tested by considering the seven different attack categories as:

- Tautologies
- Logically incorrect query/illegal queries
- Union Query
- Piggy-Backed Queries
- Alternate Encodings
- Stored Procedure
- Inference attack

Details of each attack with examples are explained in the chapter 3.

4.1.4.1 Token Specification Strategies

The SQL query is split or parsed into some tokens for the further validation process. We evaluate these tokens closely for malicious pattern or entry. Each legal query is divided into many predefined tokens (Buehrer and Gregory, 2005). The predefined tokens are generated based on query type, used-tables, columns, system-variables, global variables, functions, joins, special-symbols, operators, comment symbols and keywords, based on the schema of the query.

In this proposed approach, the parsing technique has the following advantages:

- Reduce the difficulty associated with the complex SQL statement.
- Provide cost effective and faster evaluation strategy.
- Support of an efficient procedure to validate tokens

To do the vulnerability analysis and to perform mapping of the legal query token with the dynamic query tokens, one of the primary requirements in this research is to parse the query into many distinct tokens based on the structure of the query (Joshi and Geetha, 2014, Liu and Anyi, 2009). Here any complex SQL query can be split into 12 distinct categories of tokens as per the given SQL grammar, which can be done as per the availability of the parser for each web application. We use the parsed token for further evaluation. SQTC is a good option of the parser for the query (in Java based application) and stores it in JSON format. Various categories of functions, special characters, symbols, keywords, reserved words, etc., used for analysis to generate appropriate tokens for the template specification procedure is shown in Appendix.

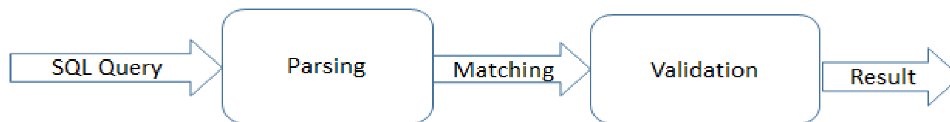


Fig. 4.2 Procedure for Query model constructor

Here the input query is parsed/split into small tokens as per the parsing technique and tools provided by the application. Lexing or Lexical analysis is a process of converting a sequence of characters into a subsequence called tokens (Srivastav and Goel, 2013; Ruse and Michelle, 2010). Parsers and Lexical analyzers are software components for dealing with the input of character sequences.

4.1.4.2 Complex Query Evaluation

One of the primary tasks of the lexical analyzer or parser is to split the complex query into appropriate small tokens. If the query splitting or tokenization cannot give an accurate result, then the evaluation process to produce the mapping function does not produce the correct result. To have an accurate splitting procedure for a complex query, the complex query is split into some independent queries by following depth-first tree traversal strategies. DFS starts at the root and goes down to the left most paths. DFS forest is a collection of one or more DFS tree (Buehrer and Bruce, 2005; Shrivastava and Soni, 2013).

4.1.4.3 Query Evaluation Using Tree Structures

We evaluate the user queries by considering the full query as a tree where the subqueries are on the nodes in various levels of the tree structure as shown in Figure 4.3. To break the query, we follow a tree traversal

procedure, where we consider the input query (injected/ intended) as a forest of queries. If there is more than one independent query, we break into subqueries and place the separate queries on each node of the 'tree'. Then we parse the query into the specified template as per the logic of the given SQL statement after traversing through each node.

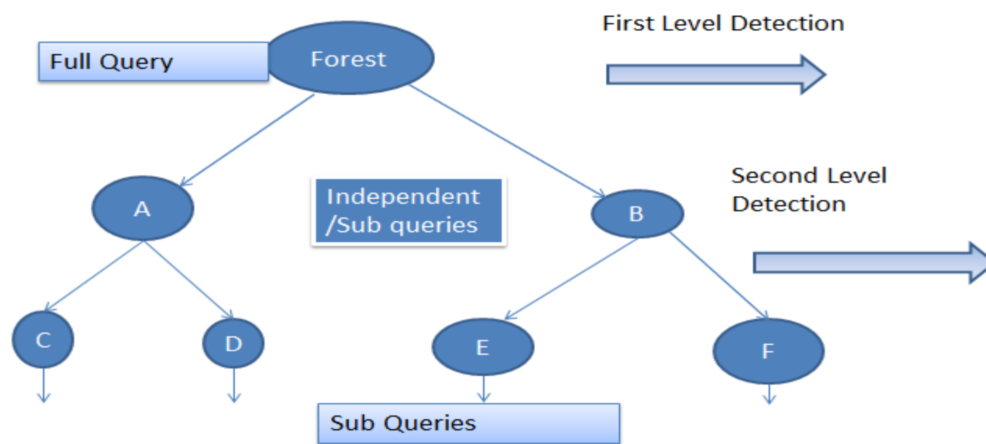


Fig. 4.3 User query evaluation using Tree structure

The nodes in the second level are assigned for extracting the subqueries in a complex query structure where the detection procedure is carried out in two levels.

4.1.4.4 Design Specification for Injection Detection

Usually, a complex SQL query, with multiple subqueries is considered in the form of forest (It is a collection of independent queries). Each independent query is in a tree-like structure (Buehrer and Bruce, 2005). Each node in the tree contains sub-queries. Forest of independent query is represented as, $F = [I_1, I_2, \ldots, I_k]$ where $I_1, I_2, \ldots, I_k$ are independent queries.

4.1.4.5 General View of Queryevaluation

User queries are complex queries and represented as the Forest of trees (F(Q)). We can represent each subquery from q1 to qn. The matching algorithm can then be invoked to check the template followed by analyzing the list of characters in each unit/token of the given template specification. If the evaluation result is 'True' it is considered a legal query, else an alert message stating that it is a 'Malicious query' would be displayed. Figure 4.4 shows the detailed procedure to evaluate the query. The Input query is being checked for several subqueries, tables, fields, etc.

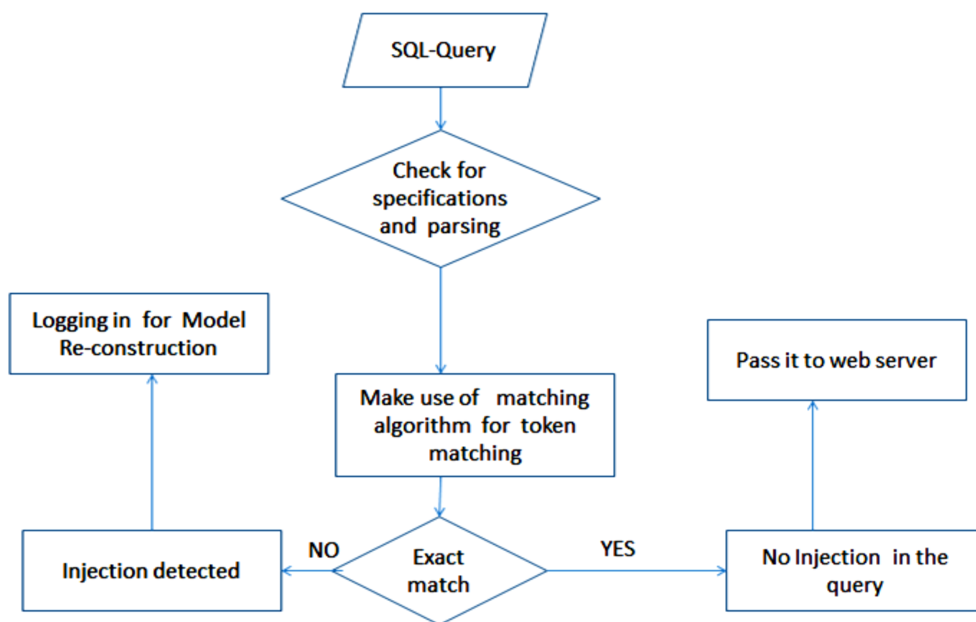


Fig. 4.4 Query Evaluation procedure in the hybrid frame work

The evaluation procedure (tree structure) by using a simple SQL query is explained below:

```
SELECT CustomerID, CustomerName, City, Street, housenum
FROM Customers
WHERE CustomerID IN
    (SELECT a.CustomerID
    FROM Customers AS a INNER JOIN
    (SELECT Country, City, Street, houseno, count (*) AS cn
    FROM Customers
    GROUP BY Country, City, Street, housenum
    HAVING count (*) >1) AS b
    ON (a.Country = b.Country) AND
 (a.City = b.City) AND  (a.Street = b.Street) AND (a.housenum =
b.housenum))
    ORDER BY City, Street, housenum;
    SELECT * FROM department WHERE deptno NOT IN( SELECT
    deptno FROM emp);
```

This complex query can be split into independent subqueries as explained below and shown in Figure 4.5.

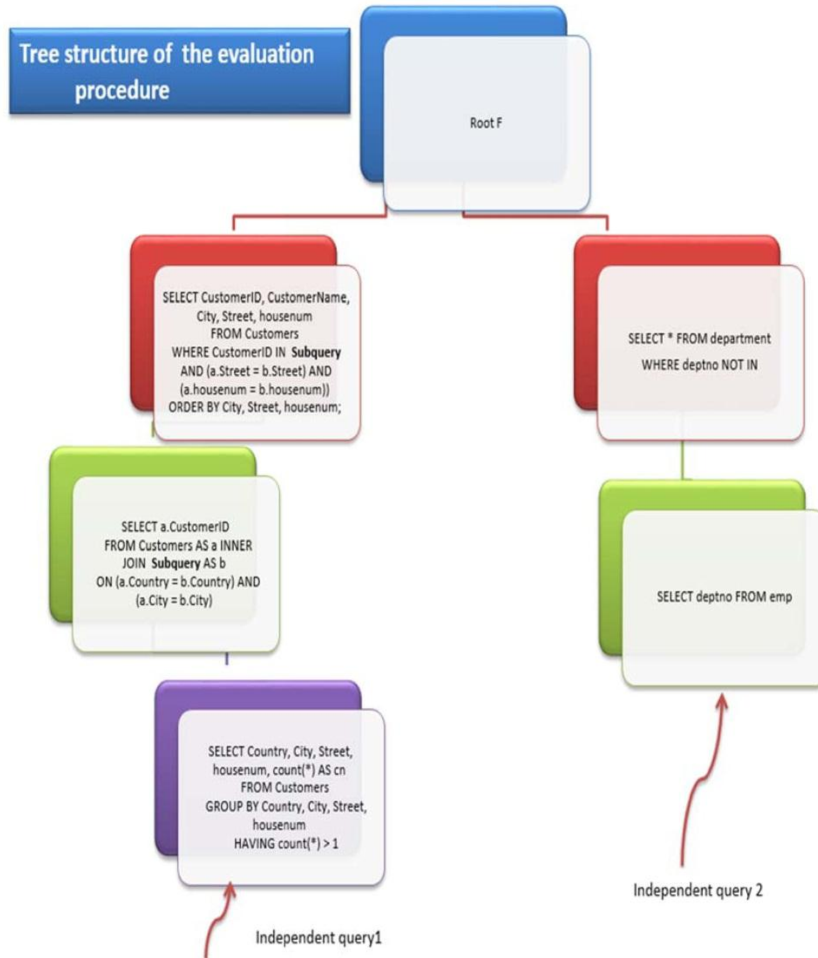


Fig. 4.5 Query evaluation using tree structure

The independent subqueries are numbered and described as shown below:

"SELECT CustomerID, CustomerName, City, Street, housenum FROM Customers WHERE CustomerID IN" ------------------------------- (4.1)

Query (1) is checked for any injection. If it is a valid query , then child nodes are checked. In this case, there are two child nodes. The first child node [Query (2)] is considered next.

"SELECT a.CustomerID FROM Customers AS a INNER JOIN"--- (4.2)

This is a recursive process and it will be continued until the node has no children. So the presence of a subquery is checked in Query(2).

"SELECT Country, City, Street, housenum, count(*) AS cn FROM Customers GROUP BY Country, City, Street, housenum HAVING count(*) > 1" -- (4.3)

Query (4.3) is a simple query, and has no children. Similarly all the nodes in the tree are processed. This Dagger detection process is repeated for all the trees in the forest.

The template creator module makes use of the Standard Query Template Creator (SQTC) algorithm to parse the queries. The number of tokens generated from each query depends on the complexity and structure of the assigned query. Parsing/splitting of the token is performed only for the independent query. The tokens identified are query-type, used-tables, columns, system-variables, global variables, functions, joins, unique symbols, operators, comment symbols and keywords. As per the complexity of the query the number of columns increases or decreases. We perform a similar procedure on each query (multiple queries are on each web page) for generating a unique query identity and corresponding template format and the constructed ID with the format is placed in the template repository. Dynamic query parsing also follows the similar procedure performed for legal query parsing. The dynamic user queries, received through the web server should undergo the tokenizing/parsing process by the parser to have the token mapping against the standard query tokens. The parsing technique used in this approach can handle the tokenizing procedure by analyzing the grammar and structure of the given SQL statements of the dynamic input query by the user.

In most of the database applications, the first stage of SQL statement processing is parsing, separating the pieces of SQL statement into a smaller data structure, which can be processed further by another application. The primary objective of parsing technique is syntax checking and optimization process (Shahriar and Zulkernine, 2012).

If the SQL statements are appropriately parsed, the performance of the query will be efficient. During the parsing procedure, we verify each SQL statement for:

- Syntax
- Table and column definitions checked with the data dictionary
- Access permissions and privileges of user and object
- Locking strategies/security of the relevant object
- Optimal execution plan

All the legal queries identified in the application for model construction and the dynamic queries accepted through the web pages are to undergo the tokenizing procedure given below for further validation and template mapping. During this phase, we parse each SQL query into many tokens as per the designed template specification.

For example: Consider the query to be tokenized is: SELECT * FROM Book reviews WHERE ID='5'; as per the predefined token specification the query is split into different tokens as shown in Table 4.2.

4.1.5 The Mapper

To check the validity, it maps the appropriate standard query from the template repository against the dynamic query with the support of a proposed SQL-Token mapper algorithm, which is shown in Table 4.3. The mapping techniques are carried out on two distinct levels according to the complexity (number of subqueries) of the dynamic queries in each web application. If the match is ‘False’ in any one of the tokens, the alert message in the corresponding field will trigger, and the evaluation engine will indicate ‘First level detection’. Otherwise, it moves onto the next level of detection, and the similar procedure and the evaluation engine will display the result as “Second level detection”. Here the detection process is done at multiple levels based on the complexity of the SQL query within the web pages of the underlying application (Dharam, Ramya, 2012; Lebeau and Franck, 2013).

Table 4.2 Summary of tokens and values assigned

Token specification	Values assigned
Query type	Select
Table	Book review
Number of Independent	1
queries	from , where
Keywords	‘*’, ‘ID’,’5’.
Column values	Null
System-variables,	Null
global-variables	Null
functions,	Null
joins,	‘,’
special-symbols,	=
operators,	Null
comment-symbols	Null

4.1.5.1 Token-Mapper Algorithm

The token mapper algorithm in Table 4.3 shows the SQL Injection detection procedure by mapping the tokens of dynamic query accepted through the browser against the statically stored tokens of legal queries.

Table 4.3 Token Mapper algorithm for SQL injection Detection (SQLI-D)

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) $\oplus$ *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

The mapping techniques are carried out on two distinct levels according to the complexity (number of subqueries) of the queries in each web application (Stuttard and Pinto, 2011). If the received dynamic query is a simple one detection is possible within the first level, else in the case of a complex query, detection will get into the procedure of the second level. Each token is validated for Boolean match function of either 'True' or 'False'. If the length and content of each token are exactly same, that is the Boolean match is “True”, the token is marked with “No injection detected”. Otherwise, it is marked as “Injection detected” in each specified tokens. We repeat this procedure for each token.

In the algorithm mentioned above, we validate 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. We generate the tokens as per the predefined template specification for each standard SQL statement. If each token of the input query and the legal query exactly matches($TL(Sq) \oplus TL(Dq) == 0$), then there is no injection in the query. Otherwise ($TL(Sq) \oplus TL(Dq) \neq 0$) there is injection detected. Sq is the standard/Legal query and Dq is the Dynamic /user-Input query.

4.1.6 Validation and Detection

In this processing stage, we do validation of the dynamic queries against the legal query model. The Query evaluation engine employed at this component displays the alert message based on the detection procedure. If there is an exact match found between the standard and the dynamic

query, then the alert message would be displayed as "Injection not detected" otherwise "Injection detected" message is displayed. The benign queries can move further to access the database server. We log the detected 'malicious queries' for further evaluation and model construction at the later stage after the authentication check and reconstruction guidelines. The 'benign queries', can pass through the web server to access information from the database server.

4.2 Strategies Used in Standard String Matching Algorithms

There are many string comparison and matching algorithms available to validate and compare the strings in the given template or token. These algorithms are rated based on the factors such as complexity, speed/time and storage space/pattern (Lebeauand Franck, 2013;Belk,2011). The proposed SQTC and Detection & mapping algorithms are also evaluated based on time-space complexity and accuracy of detection (Wang and Miner,2004).

4.2.1 Boyer-Moore Algorithm

It is one of the strong string matching algorithms (Buja and Rahman,2014). It will scan the characters of the pattern from rightmost bit to the left and have following features:

- Preprocessing phase in $\boldsymbol{O(m+\sigma)}$ time and space complexity.
- Searching phase in $\boldsymbol{O(mn)}$ time complexity
- There are 3n text character comparisons in the worst case of a non-periodic pattern
- It has $\boldsymbol{O}$(n/m) best performance

If there is a mismatch or a complete match it uses shift functions such as good suffix shift (matching shift) and bad character shift (Occurrence shift). Good suffix shift function is stored in a table of size m+1, and bad character shift function is stored in another table of size $\boldsymbol{\sigma}$**.** Both tables can be pre-computed in time $O(m+\sigma)$ before the searching phase. Searching phase complexity is quadratic. While searching for ab^{m-1} in b^n the algorithm makes only O(n/m) comparisons, which is the best case for any string matching algorithm.

4.2.2 Hirschberg Algorithm

Hirschberg algorithm takes time complexity as O(nm) in the worst case, and space complexity is O(min(nm)) for two sequences. F(i,j)= MAX{F(i-1,J-1) + s(xi-yi),F(I,j-1)+d,F(i-1,j)+d), this equation describes the algorithm (Ezumalai and Aghila, 2009). As per the explanation given, there are three paths in the scoring matrix for reaching a particular position i, j as explained below:

- A diagonal move from position i-1 to j-1 with no gap penalties.
- A move from any position in column j to i, j with gap penalty
- A move from any position in row i to i, j with a gap penalty.

4.2.3 Morris-Pratt Algorithm

The design of the Morris –Pratt Algorithm follows the tight analysis of Brute force algorithm. Here it is possible to improve the length of the shift and consequently increase the speed of the search. It has the following features:

- It performs the comparison left to right
- Preprocessing phase in O(m) space and time complexity
- Searching phase in O(n+m) time complexity
- Performs at most 2n-1 information gathered during the scanning of the text delay bound by m

This algorithm performs at most 2n-1 text character comparisons during the searching phase (Oh and Kim, 2012).

4.3 Experimental Result

The collected queries of various attack categories are tested using the deployed MLT-D framework. The following sections display the details such as type of queries, both malicious and legal queries along with the detection details and results.

4.3.1 Queries Tested with MLT-DR Framework

Following are the list of legal queries, and Injected queries tested in the proposed model and detection field identified with the proposed template creator application.

Table 4.4 Legal queries Vs Injected queries tested in MLT-DR

Legal Query	Injected Query	Detection fields
SELECT name,address FROM customers WHERE name like '%a%';	SELECT name,address FROM customers WHERE name like '%a%' union Select NULL,LOAD_FILE('/etc/passwd')#	Operator, Query Type, Comment.

Table 4.4 continued….

INSERT INTO users (id, username, password) VALUES (1, 'Jane', 'Eyre');	INSERT INTO users (id, username, password) VALUES (1, 'Olivia' or (SELECT 1 FROM(SELECT count(*),concat((SELECT (SELECT (SELECT concat(0x7e,0x27,cast(users.username as char),0x27,0x7e) FROM `newdb`.users LIMIT 0,1)) FROM information_schema.tables limit 0,1),floor(rand(0)*2))x FROM information_schema.columns group by x)a) or '', 'Nervo');	Operator, Query Type, Tables, Fields, Function.
INSERT INTO users (username, password) VALUES('jack','');	INSERT INTO users (username,password) VALUES('jack',''); DROP TABLE users;	Query Type, Number of independent Queries.
SELECT * FROM customers WHERE username = 'timmy'	SELECT * FROM customers WHERE username = '\';SHUTDOWN;	Number of independent Queries, Special characters.
UPDATE users SET password='Nicky' WHERE id=2 and username='Olivia';	UPDATE users SET password='Nicky' or updatexml(1,concat(0x7e,(SELECT concat_ws(':',id, username, password) FROM newdb.users limit 0,1)),0) or'' WHERE id=2 and username='Olivia'	Operator, Function, Special Characters.

Table 4.4 continued….

DELETE FROM users WHERE id=2;	DELETE FROM users WHERE id=2 or updatexml(1,concat(0x7e,(version())),0) or'';	: Operator, Function.
DELETE FROM users WHERE id=2;	DELETE FROM users WHERE id=1 or updatexml(0,concat(0x7e,(SELECT concat_ws(':',id, username, password) FROM users limit 0,1)),0) or '';	Operator, Function.
DELETE FROM users WHERE id=2;	DELETE FROM tablename WHERE id ='x'; Exec(char(0x73687574646f776e));--	Number of independent Queries, Comment.
DELETE FROM users WHERE id=2;	DELETE FROM users WHERE id ='2' UNION SELECT name, cast((mb_free) as varchar(10)), 1.0 FROM haxor;--	Operator, Comment, Fields.
UPDATE users SET password='Nicky' WHERE id=2 and username='Olivia';	UPDATE users SET password='Nicky' WHERE id=2 and username='Olivia'; DROP TABLE haxor;CREATE TABLE haxor(line varchar(255) null); INSERT INTO haxor EXEC master..xp_cmdshell 'dir /s c:\';--	Fields, Number of independent Queries, Query Types, Comment.
UPDATE users SET password='Nicky' WHERE id=2 and username='Olivia';	UPDATE users SET password='Nicky' WHERE id=2 and username='Olivia' UNION SELECT line, '', 1.0 FROM haxor;--	Fields, Operator, Query Types, Comment.

INSERT INTO Favourites (UserID, FriendlyName, Criteria) VALUES(123, 'My Attack',''');	INSERT INTO Favourites (UserID, FriendlyName, Criteria) VALUES(123, 'My Attack','''; DELETE Orders;--')	Tables, Query Types, Fields, Comment.
SELECT * FROM Products WHERE ProductName = 'abc';	SELECT * FROM Product WHERE ProductName = ''; SHUTDOWN WITH NOWAIT;--	Tables, Query Types, Comment.
UPDATE users SET password='Nicky' WHERE id=2 and username='Olivia';	UPDATE users SET password='Nicky' WHERE id=’2’ or username=’Oliva’--and username=’’;	Operator, Comment.
UPDATE users SET password='Nicky' WHERE id=2 and username='Olivia';	UPDATE users SET password='Nicky' WHERE id=’2’ or ‘one’=’one’ /*;	Operator, Comment.
DELETE * FROM users WHERE id=’2’ AND username=’abc’;	DELETE * FROM users WHERE id=’2’ AND username=’’ OR ‘ab’=’a’+’b’ #;	Comment, Operator.
DELETE * FROM users WHERE id=’2’ AND username=’abc’;	DELETE * FROM users WHERE id=’2’ AND username=’’ OR ‘ab’=’a’||’b’ /*;	Comment, Operator.
SELECT * FROM users WHERE userid=22;	SELECT * FROM users WHERE userid=’’ OR ‘ab’=’a’’b’ /*;	Operator, Comment.
INSERT INTO users (username,password) VALUES('jack','');	INSERT INTO users (username,password) VALUES('jack','');SHUTDOWN--	Number of independent Queries.

Table 4.4 continued….

DELETE FROM users WHERE id=2;	DELETE FROM users WHERE id ='2' UNION CREATE TABLE haxor(name varchar(255), mb_free int); INSERT INTO haxor EXEC master..xp_fixeddrives;--	Operator, Comment, Fields, Query Types.
DELETE FROM users WHERE id=2;	DELETE FROM users WHERE id ='2' UNION SHUTDOWN;--	Operator, Comment.
SELECT uname, password FROM users WHERE id = 12	SELECT name, password FROM user WHERE id = 12 UNION SELECT distinct(db) FROM mysql.db--	Query Type, Fields, Comment, Operator.

The screen shot of tokenizing technique of a simple query is shown in Fig.4.6.

QUERY TEMPLATE

Query Format: select playerid from test

Query-Type:	select
Table(s):	tes
Column(s):	1
Column1:	playerid
System Variables:	
Global Variables:	
Functions Used:	
Joins Used:	
Special Symbols:	
Operators Used:	
Comment Symbols:	

SAVE

Fig. 4.6 Independent (simple) query tokenizing procedure

Figure 4.7 shows the screen shot of tokenizing technique for a complex query.

QUERY TEMPLATE

Query Format: select group_concat(b.name),a.teams from (select playerid, group_concat(distinct teamid order by teamid) as teams from test group by playerid) a, player b where a.playerid=b.playerid group by a.teams union select group_concat(c.name order by c.playerid) null from player c where c.playerid not

Query-Type:		select,select,select,select
Table(s):		test,player,test
Column(s):		14
	Column1:	b.name
	Column2:	a.teams
	Column3:	playerid
	Column4:	teamid
	Column5:	teamid
	Column6:	playerid
	Column7:	player
	Column8:	a.playerid
	Column9:	b.playerid
	Column10:	a.teams
	Column11:	c.name
	Column12:	c.playerid
	Column13:	c.playerid
	Column14:	playerid

Fig. 4.7 Complex query tokenizing procedure

4.3.2 View of Template ID and Storage Format of SQL Query

During the tokenizing process, each query is assigned to a unique ID and stored in the template repository. Figure 4.8 displays the JSON format corresponding to the legal query, "SELECT * FROM Book reviews WHERE ID='5';" which is tested using the proposed frame work.

Fig. 4.8 JSON format of the standard query

There are hundreds of queries within each web application and in each web page, there exists multiple complex queries with sub queries. A screenshot of the template repository with the sample template ID created using the MLT-DR framework is shown in Fig 4.9.

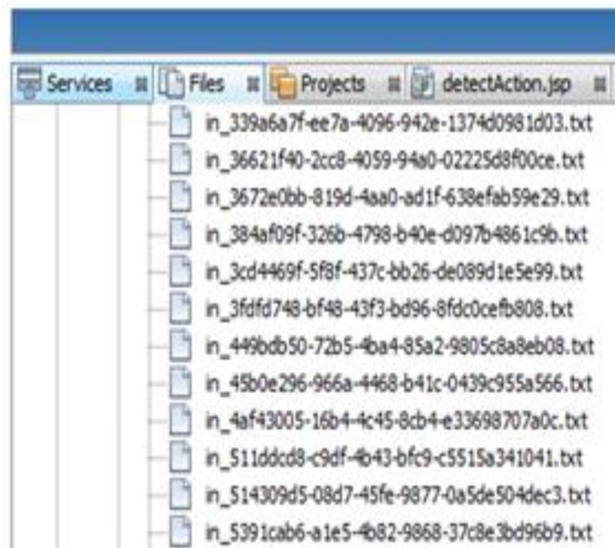


Fig. 4.9 Template ID format of tested queries.

4.3.3 Procedure to Detect and block SQL Injection Attack

At run time, during the user interaction with the database server through dynamic web pages, the input user query is mapped with the standard query from the template store. With the support of the Jar or package file procedure, it is easy to locate and retrieve the Unique ID corresponding to the standard query template file from the repository. Identification and retrieval of the appropriate query identity from the template store and the validation against the injected query are done automatically as per the given procedure. In the template mapping phase, we map the appropriate legal query-ID from the template store with the dynamic query with the support of the template mapping process. If the Boolean match is 'False' in any one of the specification templates/ tokens identified, the alert message in the corresponding field will trigger, and the evaluation engine will indicate 'First level detection'.

INPUT QUERY

Input Query: SELECT * FROM Bookrevives WHERE ID='5' OR '1'='1';

Standard Query ID: st_ecb779e0-6025-4c2f-8b3d-

Submit

Template Content

FIRST LEVEL DETECTION RESULT

No.of Independent Queries:	Not detected
Query-Type(s):	Not detected
Table(s):	Detected
Column(s):	Not 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

Fig. 4.10 Evaluation procedure of a dynamic query

In the case of a complex query, if the injection is not detected in the first level, it will move onto the next level of detection, and the validation procedure is repeated for the second level of detection. Figure 4.10 shows

the detection procedure. The figure shows the dynamic query accepted through the browser, the template ID of the statically stored legal/standard query, JSON format of the parsed query and the first level detection result of the mapping /validation procedure.

QUERY TEMPLATE

Query Format: select group_concat (b.name) ,a.teams from (SUBQUERY_2) a, player b where a.playerid=b.playerid group by a.teams union SUBQUERY_3 ;

Field	Value
Query-Type:	select
Table(s):	subquery_2,player
Column(s):	5
Column1:	b.name
Column2:	a.teams
Column3:	a.playerid
Column4:	b.playerid
Column5:	a.teams
System Variables:	
Global Variables:	
Functions Used:	
Joins Used:	
Special Symbols:	[illegible]
Operators Used:	=,union
Comment Symbols:	

SAVE

SECOND LEVEL DETECTION RESULT

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

No Injection Found after Second Level Detection.

Fig. 4.11 Complex query evaluation procedure with multiple levels

If the query getting evaluated is a complex query, then the detection procedure is performed on multiple levels. Since the following query does not have any injection, the result in the second level shows that there is no detection found on the second level, as shown in Figure 4.11.

4.4 Summary of the Chapter

In this chapter, we explain the SQL query tokenizing algorithm and Query model creation algorithms with examples and screen shots. The SQLIA detection procedure and model creations are implemented using Java based application program. Implementation details indicate that SQL injection is detected without any false positives or false negatives, with the support of the template matching algorithm, Template creator algorithm. Since ID and Query templates placed in the template repository is in JSON format, accessing the query model will not have any storage overhead. Validation of tokens and injection detection will be faster compared to the other available techniques. The SQTC and Template mapper procedure of the proposed technique deliver 100% accuracy and negligible storage overhead.

.....ꙮꙮ.....

Chapter 5

THE RECONSTRUCTION FRAMEWORK

Contents

5.1 Significance of the Reconstruction Framework

5.2 Components of Reconstruction Framework

5.3 Regular Expression and Comparison for Pattern Matching in SQL Statement

5.4 Model Construction Algorithm

5.5 Experimental Result of Reconstruction Procedure

5.6 Summary of the Chapter

The reconstruction framework, introduced in this chapter facilitates the reconstruction of queries from any authenticated users, by ensuring the structure of queries and the underlying database server. The current SQLIA detection and prevention approaches reject the dynamic query if there is any mismatch or additional character found in the given input query. Rejecting the query is directly denying the authenticated users access to the system, and it will reduce system availability, especially in the case of false positives. One of the primary objectives of the proposed reconstruction framework is to reconstruct the queries from the authenticated user by eliminating the injected portion and rebuild the missing parts of the user query, based on request-id and type of injection. This chapter also explains a Back-propagated Neural Network trained (BPNN) query model learned for the seven identified attack categories using the machine learning technique.

5.1 Significance of the Reconstruction Framework

Denial of Service attack and Distributed denial of service attacks are quite common in a database supported web application. It is a type of attack where the hacker attempts to prevent legitimate users from accessing the service by sending an excessive message with invalid address or configuration details causing the server for a long wait or closing the connection (Burkhart and Plattner, 2010). This type of attack usually prevents the authorized user to access the required resource and disrupt the TCP session. There is no single solution to stop these types of attacks. Since there are ample solutions available to detect and prevent the attack, care should be taken to patch and configure database server at the precise intervals and as per the critical requirement of the web application. In most of the web applications, even if there is a simple mismatch of the entry found by the protection mechanism or the scanning module, immediately the query will be discarded or ignored by the application. There will not be any further validation or no re-evaluation of queries are possible with the user queries. Hence in the proposed framework reconstruction of queries are carried out by comparing the BP-NN trained data model (SQTC also can be recommended) against the dynamic query with the support of regular expression and model construction algorithm. The neural network model of the proposed framework includes machine learning approach and can get appropriate trained data model for the legal queries for the underlying web application to be tested with the dynamic user queries (Moradpoor, 2014). Reconstruction of queries can also be done by using the query model implemented during the detection framework, explained.

5.2 Components of Reconstruction Framework

In this framework, there is a provision of reconstructing malicious queries from the authenticated user, to increase the availability of the web application and reduces the denial of service attacks (Sahu and Tomar, 2016). Here the SQL queries are trained using a Back-Propagated Artificial Neural Network (ANN) and this learned/trained model is stored in the template repository. Figure 5.1 shows the system architecture of the reconstruction framework.

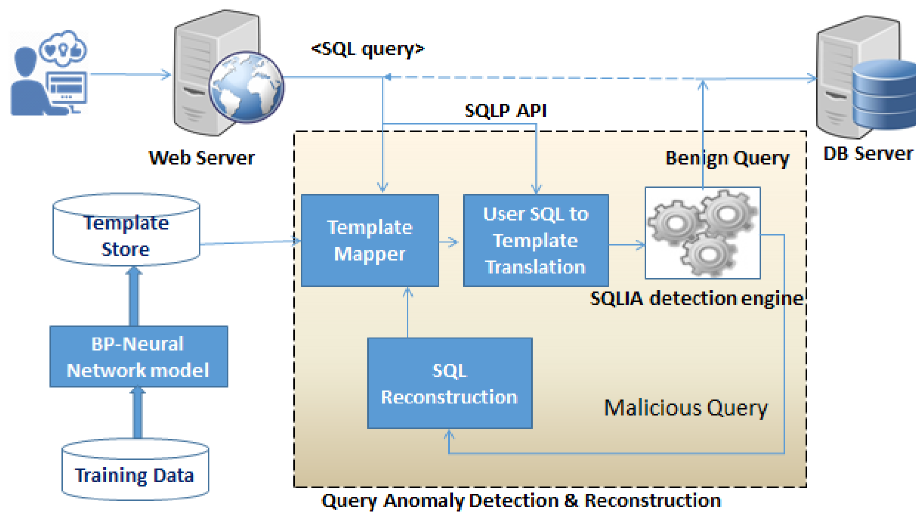


Fig. 5.1 System architecture of the reconstruction Framework

5.2.1 Server Functionality

The web server accepts database requests from the user's browser and directs it towards the Query anomaly detection and reconstruction module for further validation and accessing the actual database server. The detection of mismatch or injected portion and reconstruction of the query (if required) are done within this module by an Application program Interface(API) and

with the support of REGEX function. The database server has the responsibility of executing and giving the accurate result of user queries. The designed API is placed in between the web server and DB server. It will validate the user queries and check for the source of authentication if the correct credentials are found then it is permitted to access the actual server for information access and retrieval. Only the authenticated valid queries or benign queries are redirected to the database server for further access. Chapter 3 explains the details of server functionality.

5.2.2 Training Data

The legal SQL queries were collected and stored, and we train these queries by a Back-propagated Neural Network (BP-NN). The set of queries consists of both malicious and legal queries statically collected from various e-commerce applications and online transactions (Moosa, 2010).

5.2.3 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 (Moradpoor and Naghmeh, 2014). In this approach, we are using pattern recognition task with the assistance of back propagation algorithm to learn the ideal model of legal queries. BP-NN comprises of the training phase and testing phase. Figure 5.2 shows the representation of BP-NN learning for SQL trained model.

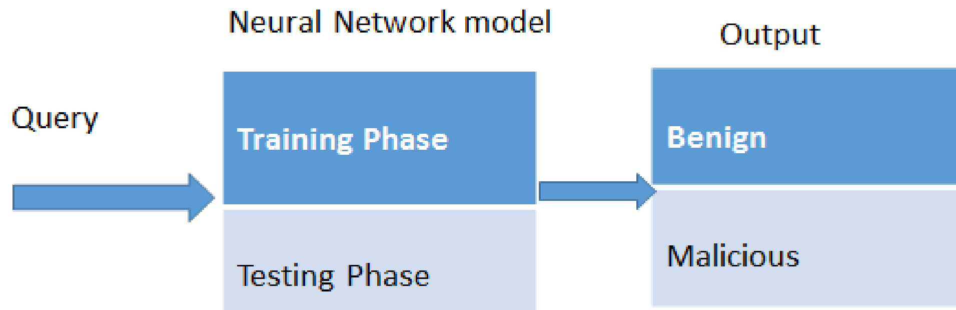


Fig. 5.2 BP-NN learning for SQL trained model

The input layer accepts SQL queries as '11 different tokens' (as specified in the standard query template creator algorithm of TbD module). From the collected queries, we use 75% in training phase, and 25% of queries in the testing phase at the Neural Network (NN) with ten hidden layers. The algorithm classifies the output from the NN model as either 'benign' or 'malicious' query by using the back-propagation algorithm (Moradpoor and Naghmeh, 2014).

5.2.3.1 Multilayer Artificial Neural Network (ANN) for Machine Learning

An artificial neural network can learn through the training process to acquire knowledge and make it available for later use. The artificial neural networks are constructed from the basic building block of an artificial neuron; it is identified with three layers of representation such as input layer, hidden layer, and an output layer. 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 (Haykin and

Simon,2009). This function can produce several 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 5.3 shows the multilayer representation of the neural network.

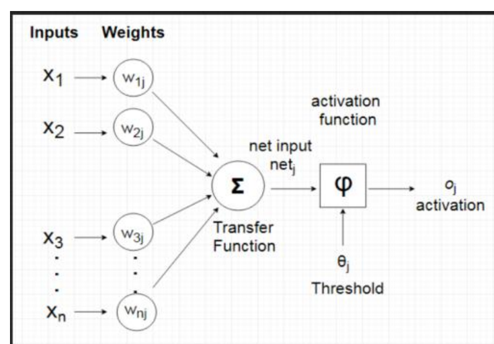


Fig. 5.3 Multilayer representation of neural network

5.2.3.2 Major Steps in Back-Propagation Algorithm (BPA)

ANN must distinguish pattern using the information given to the input without external help. One of the best learning algorithms is Back-propagation algorithm (BPA). It is a supervised learning algorithm for multilayered feed-forward network (Haykin and Simon, 2009; Kieyzun and Ernst, 2009). It is an ill-conditioned optimization problem which consists of minimization of the sum of squares errors, denoted as least squares.

The major steps carried out to train SQL queries in ANN using back propagation for each pattern in the learning set is as follows (Kubo and Shimodaira, 1998; Moradpoor and Naghmeh, 2014):

Step1 : Input the learning vector $\mathbf{u}^{\mu}$ as an input to the network.

Step 2: Evaluate the output value $uj^{m\mu}$ of each element for all layer using the formula

$$uj^{m\mu} = f(\varphi j^{m\mu}) = f(\sum_{t=0}^{nm-1} uji^{m} ui^{(m-1)\mu}) \text{ .... (5.1)}$$

Step3: Evaluate error value $\partial j^{m\mu}$ for the output layers with the support of the formulae

$$\partial j^{m\mu} = f^{'}(\varphi j^{m\mu})\partial j^{m} = f^{'}(\varphi j^{m\mu})(yj^{z\mu} - yj^{\mu}). \text{ .... (5.2)}$$

Step 4: Evaluate sum-of-squares errors $\varepsilon\mu$ from

$$\varepsilon\mu = \frac{1}{2} \sum_{j=1}^{n} (\partial j^{\mu})^{2} \text{ .... (5.3)}$$

Step 5: Perform the back – propagation of output layer error $\partial j^{m\mu}$ to the elements of hidden layers by calculating their errors $\partial j^{m\mu}$ from

$$\partial j^{m\mu} = f^{'}(\varphi j^{m\mu})\, f(\sum_{t=0}^{nm+1} \partial i^{(m+1)\mu} wij^{(m+1)} \text{ .... (5.4)}$$

Step 6: Update the weights of all elements between output and hidden layers and then between all hidden layers moving towards the input layers. Change of the weight can be obtained from

$$\Delta\mu\, wji^{m} = \eta \partial j^{m\mu} ui^{(m-1)\mu} \text{ .... (5.5)}$$

Repeat step 1 to step 6 until satisfactory minimum of complete error function is achieved.

$$\varepsilon = \sum_{\mu=1}^{p} \varepsilon\mu = \frac{1}{2} \sum_{\mu=1}^{P} \sum_{j=1}^{n} (yj^{z\mu} - \varphi j^{\mu})^2$$

The symbols used in the above equation with its description are given in Table5.1

Table 5.1 Symbols and description

Symbols	Description
P	Number of learning pattern
μ	Index of actual learning pattern $\mu = 1,\ldots,P$
M	Index of actual layers $m= 1,\ldots,M$
N_m	Number of elements in layer m
J	Index of actual element $j= 1,\ldots,N_m$
φj^{μ}	Weighted sum of input values for element j in layer μ
f	Activation function
w_{ji}^{m}	Weight between element j in layer m and element I in layer m-1
$ui^{(m-1)\mu}$	Output of element I in layer m-1 for pattern μ
∂^{μ}	Learning error for element j for pattern μ
$yj^{z\mu}$	Expected network output value for element j for pattern μ
yj^{μ}	Actual network output value for element j for pattern μ
$\Delta^{\mu} wji^{m}$	Change of given weight for pattern μ
η	Proportion coefficient

5.2.4 Template Store

The template store has BP-NN trained ideal model, which is available for the API for further mapping and reconstruction of queries. We redirect only the authenticated user queries for reconstruction and check these queries against the legal queries trained and stored in the template store. Chapter 4 deals with the details of template store.

5.2.5 Template Mapper

The Template mapper component retrieves the legal query from template store. The API requires a model template against which we match the dynamic query. In this proposed approach, we create a model file from the training set, and we validate every user query against this model. The template mapper component identifies and locates the model template. If the mapper does not find the appropriate file, there will be incidents of false positives and false negatives.

5.2.6 Template Translation

We translate the dynamic queries to specific pattern matching with the model template storage format and template specification. With the support of 'REGEX pattern matching and model checking techniques',the user query is translated into the similar format of BP-NN trained model query template format for further validation and reconstruction in the later stage (Mukkamala and Sung, 2002; Johari and Pankaj, 2012). The SQLIAShield, a jar file, will speed up the performance by identifying and specifying the path of SQL statement on each page of the web application.

5.2.7 SQL Reconstruction

It is a process performed by the SQL-Reconstruction component, to reconstruct dynamic queries from the authenticated user. During the query reconstruction, the component extracts the complex query as independent subqueries and evaluation is performed only on the separate subqueries. The proposed procedure validates each dynamic query against the trained model template, to detect injection and to reconstruct the required queries from the authenticated user. Table 5.6 shows the reconstruction algorithm (RaAuQ). Reconstruction process takes care of the special task of re-establishing lost portions of the actual query and removing injected part of the query. Using the proposed method, we achieve high accuracy of detection without much loss of efficiency.

5.2.8 SQLIA Detection Engine

SQL Injection Attack (SQLIA) detection engine invokes the matching process against the trained data model from the template repository. The detection engine generates the status report, and it directs the benign queries to the database server for further process. While matching it with the trained model, if the engine identifies a malicious entry in the incoming queries, and if the query is from an authenticated user then the malicious query is redirected to the reconstruction module to be reconstructed. The reconstructed query will be tested again with the mapper and then to the SQL translation process for a further process (Zhang and Hsu, 2011; Dharam and Sajjan, 2012). Instead of the Neural network trained model, we can make use of the query model developed using the web crawler and mapping procedures.

5.3 Regular Expression and Comparison for Pattern Matching in SQL Statement

A regular expression is a powerful tool that gives a concise and flexible way to identify strings of text based on patterns. It can search for any string such as email, IP address or anything that has a pattern. It is widely used in all programming languages to databases and uses its syntax that can be interpreted by a regular expression processor. It is not limited to the usual pattern such as '%' or '(-)' but includes more meta characters to have a flexible pattern (Godefroid and Molnar, 2008; Das and Bhattacharyya, 2010). We use regular expressions to search for complex patterns, but it must be managed carefully. REGEXP handles meta- characters and literals separately during the search function. The primary task performed by the regular expression and pattern matching are:

- Check whether the given sequence is matching with the given pattern
- Replace the subsequence with the alternatives provided in the specific pattern
- Check the occurrence and position of the subsequence in each sequence.

The meta-characters such as: +, ? , *{m}, \, ^, \n etc. identified are used for searching the pattern. Table 5.2 displays the meta characters used and REGEX format and its description.

Table 5.2 List of Metacharacters used in REGEX

Meta character used	Description
^	Matches the position at the beginning of the string
[….]	Matches specified character within the bracket
$	Matches the position at the end of the searched string
*	Matches the preceding characters 0 or more times
{n}	Matches n number of the preceding characters
[\t\r\n]	Regular expression for line breaks
--[^\r\n]*	Single line comments begins and continue the match 0 or more times until return character or new line character found braking the match
/*[\w\W}*?(?=*/)*/	Multiple line comment REGEXP

Table 5.3 List of functions for string comparison

Type of Functions	Sample functions
Numeric Functions	ABS,ACOS,ASIN,EXP,LOG,MOD,POWER,ROUND,SQRT
Character Functions Returning Character Values	CHR, CONTACT,LOWER,NLS-INITCAP,NLS_UPPER REGEXP-REPLACE, REGEXP_SUBSTR, REPLCE, TRIM
NLS Character Functions	NLS-CHARSET_DECL-LEN,NLS_CHARSET_ID,NLS_CHAESET_NAME
Datetime Functions	ADD_MONTHS,CURRENT_DATE,CURRENT_TIMESTAMP,DBTIMEZONE,EXTRACT,FROM-TZ, LAST_DAY, NEW_TIME,LAST_DAY,NUMTODSINTERVAL,ROUND
General Comparison Functions	GREATEST,LEAST
Conversion Functions	ASCIISTR,BIN_TO-NUM,CAST,COMPOSE,CONVERT, DECOMPOSE,RAWTOHEX,NUMTODINTERVAL, ROWIDTONHEX, ROWIDTONCHAR,TO-BINARY_FLOAT, TO_YMINTERVAL UNISTR.
Large Object Functions	BFILENAME, EMPTY_BLOB,EMPTY_CLOB
Collection Functions	CARDINALITY, COLLECT, POWERMULTISET, CARDINALITY, SET.
Encoding and Decoding Functions	DECODE,DUMP,ORA-HARSH,VSIZE
NULL-Related Functions	COALESCE,LENVL,NULLIF,NVL,NVL2
Aggregate Functions	AVG,COLLECT,CORR,COUNT,COVAR-POP,FIRST,GROUP_ID,GROUP_ID,LAST,MAX,MEDIAN, MIN,PERCENTILE_CONT
Analytic Functions	CORR,COUNT,LAG,LAST,LEAD,PERCENT_RANK,ROW_NUMBER
Object Reference Functions	DEREF,MAKE-REF,REF,REFTOHEX,VALUE

Table 5.4 shows some of the basic string matching functions with SQL Regular expression.

Table 5.4 Basic functions with SQL Regular expression

REGEXP	Description
REGEXP_LIKE	Searches a character column within a pattern. Syntax:(source_string, pattern, match-parameters)
REWGEXP_REPLACE	Searches a character column within a pattern and replace the occurrence with the specified pattern. Syntax: (Source, pattern[,replace[,position[,occurrence [,match_parameter]]]])
REGEXP_INSTR	Searches for an occurrence of a string in the pattern Syntax: (Source, pattern[,starting at M[,the Nth occurrence[,return_option[,match_parameter]]]]])
REGEXP_SUBSTR	Return the substring, which matches the regular expression. Syntax: (Source, pattern [,position[,occurrence [,match_parameter]]])

In Table 5.5, the model construction functions and descriptions with MLT-DR are explained.

Table 5.5 Model constructors in MLT-DR

Constructor/Function	Description
SQLRejuvenate(String standardSqlString)	A constructor to initialize the standard SQL string
SQLRejuvenate(String standardSqlString, String[] regularExpression)	A constructor to initialize the standard SQL string and array of regular expressions for inputs
SQLRejuvenate(File trainingDataSetfile, String[] regularExpression)	A constructor to initializes the standard SQL string that is created from training data and array of regular expressions for inputs.
detectSQLIA(String inputSqlString)	A function which detects the SQL injection and returns true if found
validateNoOfIndependentOrSubQueries(String inputSqlString)	A function which checks the number of independent or sub-queries. Returns true if no injection is found.
validateQuerytype(String inputSqlString)	A function which checks query type in the order they appear.Returns true if no injection is found
validateUsedTables(String inputSqlString)	A function which checks used tables in the order they appear.Returns true if no injection is found
validateColumns(String inputSqlString)	A function which checks columns in the order they appear.Returns true if no injection found
validateSystemVariables(String inputSqlString)	A function which checks system variables in the order they appear. Returns true if no injection is found

Table 5.5 continued….

validateGlobalVariables(String inputSqlString)	A function which checks global variable in the order they appear. Returns true if no injection is found
validateFunctions(String inputSqlString)	A function which checks aggregate or built-in SQL functions in the order they appear. Returns true if no injection is found
validateJoins(String inputSqlString)	A function which checks joins in the order they appear. Returns true if no injection is found
validateSpecialSymbols(String inputSqlString)	A function which checks special symbols the order they appear. Returns true if no injection is found
validateOperators(String inputSqlString)	A function, checks operators used in the order they appear. Returns true if no injection is found
validateCommentSymbols(String inputSqlString)	A function which checks comment symbols in the order they appear. Returns true if no injection is found.
validateKeywords(String inputSqlString)	A function which checks keywords in the order they appear. Returns true if no injection is found.
setInputFields(String[] regularExpression)	A function sets regular expression for each input field in the order they appear.
validateAllInput(String inputSqlString)	A function which checks input field values with regular expressions. Returns true if all inputs are valid.
DetectSQLIAWithReconstruction (String inputSqlString)	A function which detects SQL injection and returns reconstructed query with valid input values if any injection is found.

5.4 Model Construction Algorithm

Rejuvenation technique/procedure has the primary objective of reconstructing the query as per the application requirement with the support of a template matching and the model construction algorithm. Most of the research works are focusing on the detection and prevention of malicious queries only; there is not enough work on reconstruction aspects. There are many tools and techniques available to block injected queries, but none of the methods concentrate on the reconstruction of queries from the authenticated users. In this work, there is a module with an appropriate procedure, proposed for reconstructing the authenticated user queries which will reduce the denial of service attacks and increase the system availability for the authenticated user to access the backend database server. The reconstructed queries are validated again by the detection system, and only benign queries can access information from the database server.

In SQL Rejuvenation technique each user query/SQL statement is validated based on the Number of subqueries, Number of tables and fields/keywords/tokens and classified them according to the criteria or evaluation procedure of Token extractor or with the policies of Neural network based trained query model. It can evaluate each input query by mapping it against the specification template or with the trained data implemented on the Application Program Interface (API).

5.4.1 Reconstruction Algorithm

This framework facilitates reconstruction of queries from authenticated users, irrespective of the underlying database. As a prerequisite of reconstruction procedure, we validate each query with authentication credentials of the user and, if it is from an authenticated user then the query is labelled as" reconstruction required". Then we redirect the query 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 process in the training stage. The Query Reconstruction module in the NNbR reconstructs the queries by eliminating injections and also rebuilding missing portions, if any, and removing injected part of the user query which will increase the system availability. Table 5.6 shows the Reconstruction Algorithm of Authenticated User Queries (RaAuQ). Here, if the algorithm detects the input query with injection (additional character/string) and the query is raised from an authenticated user, then it is diverted for reconstruction. Reconstruction algorithm has the main objective of reconstructing the query with the support of a template matching using a regular expression. In SQL rejuvenation/reconstruction technique each user query/SQL statement is validated based on the number of subqueries, number of tables and fields/keywords/tokens and classified according to the criteria or evaluation procedure of the policies of neural network based trained query model. It can evaluate each input query by mapping it against the trained standard query.

Table 5.6 Reconstruction Algorithm of Authenticated user Queries (RaAuQ)

Reconstruction Algorithm of Authenticated user Queries (RaAuQ)

```
Input:  Malicious input token , SQTC(legal query token), Regex functions
Output: Reconstructed queries
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
```

If the malicious user queries are detected at first level (Case I): then we assign 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 clear the content of the injected portion (additional strings/ characters are removed). Then the token is considered as the reconstructed token 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 additional field with the input query, which is considered as the query 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 independent query. Repeat the procedure for each independent query separately and perform reconstruction procedure.

5.5 Experimental Result of Reconstruction Procedure

We verify the authentication of the user credentials and the privileges by the fully automated procedure implemented at the proxy server during the training phase. The 'Authentication ID', Username, Password, User Type, User Status, etc. of each authenticated user are collected and stored. If the query is originated from a user who is having appropriate authentication, then the query will be sent to NNbR module, and the required rejuvenation or reconstruction procedure is carried out. The report displays several benign queries, dynamic (malicious) queries and its reconstructed queries. 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 has to be removed or replaced with a null value. So, the reconstruction algorithm rejuvenates the malicious query and converts it into a benign query. The status report in Figure 5.4 shows the empirical analysis of NNbR module, and the highlighted portion ‘drop table login- -‘ is the injected part.

```
Output
SchoolManagement (run)   GlassFish Server 4.1.1
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
```

Fig. 5.4 Status report of NNbR module with a reconstructed query

5.6 Summary of the Chapter

There are many tools and techniques available to block the queries, but none of the techniques concentrate on the reconstruction of queries from the authenticated users. In this proposed work there is strong model, designed for reconstructing the authenticated user queries, supported with authentication procedure which will reduce the denial of service attack and increase the system availability for the authenticated user to access the

backend database server. The suggested model is an ideal solution for query reconstruction. The proposed reconstruction component will reconstruct the query with the support of regular expression, rather than just rejecting the query. In this approach, reconstructed queries are again diverted to the detection module for comparison and are reconsidered it for further access to the database server.

.....ᘐᘓ.....

Chapter 6

PROTOTYPE IMPLEMENTATION OF MULTI LEVEL TEMPLATE BASED DETECTION AND RECONSTRUCTION (MLT-DR) FRAMEWORK

Contents

6.1 System Architecture of the Prototype, MLT-DR
6.2 MLT-DR Training Phase
6.3 Learning Phase of Back Propagated Neural Network Learned Model
6.4 Testing Phase of MLT-DR
6.5 Summary of the Chapter

The proposed prototype multilevel template based Detection and Reconstruction (MLT-DR), is developed and implemented using Java based application software and MySQL as back-end database server. It is based on the query model and requires access to the queries passed between the server and databases. 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 stronger detection model in the later stage for handling the zero-day vulnerability.

6.1 System Architecture of the Prototype, MLT-DR

The MLT-DR intercept the SQL queries before placing it to the web server, with the intervention of the proxy server placed at the MLT-DR, and it allows only benign queries through the web server. Figure 3.1 (Chapter 3) shows the system architecture of the prototype.

6.2 MLT-DR Training Phase

In this phase, we learn the underlying web application for template creation for the SQL query. It generates the pre-defined token formats at this stage. The significant tasks under this stage is:

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

The MLT-DR accept Uniform Resource Locator (URL) of the web application and user authentication credentials as input and generate many tokens and a standard query template creator (SQTC) model for the template matching. We use SQTC as the input 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 create the Query model, which will be stored in the repository and later used for dynamic query template mapping. Figure 6.1 shows the model creation phase or the training phase.

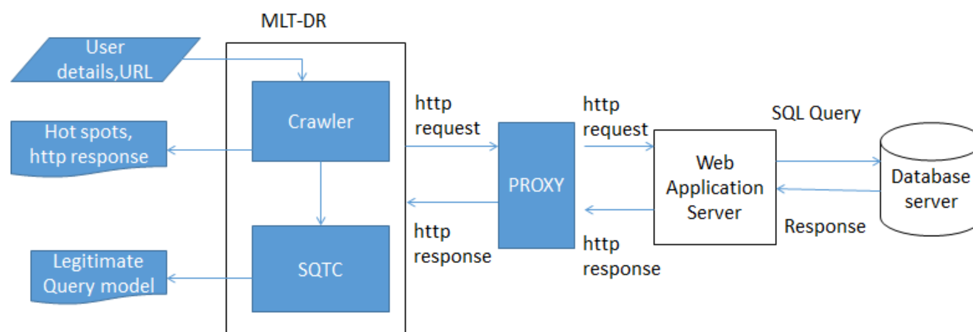


Fig. 6.1 MLT-DR Prototype in training phase

6.2.1 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 required to be filled by the user on each web page.

With the support of the deployed crawler the following benefits can be achieved: -

- Appropriate validation and constraint checking of each form/ input entry field
- Prioritize the security measures in the critical vulnerable points within the web application

For example, Figure 6.2 shows the HTTP request in the form field of login page (an injected query).

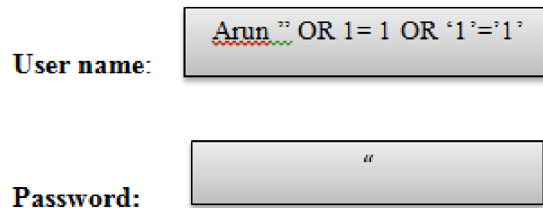


Fig. 6.2 Injected query on the form field of login page

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

The SQL queries identified during this phase are populated with valid input entries and submitted to the application after appropriate analysis of each web page without any missing entries, which increase the accuracy of the Standard Query Template creator (SQTC) model and decrease the false positives.

6.2.2 Standard Query Template Creator (SQTC)

We develop 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. Since we store the template in JSON format, it will substantially decrease the storage overhead. The identified query by the crawler with the support of the proxy server should be tokenized as per the pre-defined tokens designed and developed by analyzing the schema and grammar of the SQL statements.

The similar token specification is applicable for the queries submitted to an Oracle server, MySQL and MS SQL server. In this approach, while submitting the standard query, it is possible to select any one of the databases mentioned above servers as there is no difference in SQL query format in the databases mentioned above.

Table 6.1 lists some of the injected queries and query model learnt.

Table 6.1 Legal queries and Injected queries chosen for BPNN learning

Legal query	Injected Query
INSERT INTO accounts (username, password, mysignature) VALUES ('data', 'data', 'data');	INSERT INTO accounts (username, password, mysignature) VALUES ('data', 'data', 'data',’test’,(select version()))-- -);
SELECT ProductName, QuantityPerUnit, UnitPrice FROM Products WHERE ProductName LIKE'G%';	SELECT ProductName, QuantityPerUnit, UnitPrice FROM Products WHERE ProductName LIKE'G%' UNION SELECT UserName, Password, IsAdmin FROM Users;--
DELETE FROM users WHERE id=2;	DELETE FROM users WHERE id ='x’; CREATE TABLE haxor(name varchar(255), mb_free int); INSERT INTO haxor EXEC master..xp_fixeddrives;--
DELETE FROM users WHERE id=2;	DELETE FROM users WHERE id ='2' UNION SELECT name, cast((mb_free) as varchar(10)), 1.0 FROM haxor;--
UPDATE users SET password='Nicky' WHERE id=2 and username='Olivia';	UPDATE users SET password='Nicky' WHERE id=2 and username='Olivia'; DROP TABLE haxor;CREATE TABLE haxor(line varchar(255) null); INSERT INTO haxor EXEC master..xp_cmdshell 'dir /s c:\';--

Table 6.1 continued….

UPDATE users SET password='Nicky' WHERE id=2 and username='Olivia';	UPDATE users SET password='Nicky' WHERE id=2 and username='Olivia' UNION SELECT line, '', 1.0 FROM haxor;--
INSERT INTO Favourites (UserID, FriendlyName, Criteria) VALUES(123, 'My Attack','');	INSERT INTO Favourites (UserID, FriendlyName, Criteria) VALUES(123, 'My Attack','''; DELETE Orders;--')
SELECT * FROM Products WHERE ProductName = 'abc';	SELECT * FROM Products WHERE ProductName = ''; DELETE Orders;--
SELECT * FROM Products WHERE ProductName = 'abc';	SELECT * FROM Product WHERE ProductName = ''; SHUTDOWN WITH NOWAIT;--
UPDATE users SET password='Nicky' WHERE id=2 and username='Olivia';	UPDATE users SET password='Nicky' WHERE id=2 or 1=1 and username='Olivia’;
UPDATE users SET password='Nicky' WHERE id=2 and username='Olivia';	UPDATE users SET password='Nicky' WHERE id=’’ or 1=1 -- username='’;
INSERT INTO users (username,password) VALUES('jack','');	INSERT INTO users (username,password) VALUES('jack','123’,(UPDATE user SET password=’123’ WHERE username=’abc’;--));
INSERT INTO users (username,password) VALUES('jack','');	INSERT INTO users (username,password) VALUES('jack','123’,(*Exec(char(0x73687574646f776e));*--);
INSERT INTO users (username,password) VALUES('jack','');	INSERT INTO users (username,password) VALUES('jack','123’,(*INSERT INTO Login(id,pass) VALUES (‘2’,’A12’);*--);

For example the SQL query identified by the crawler to be tokenized is:

SELECT COUNT (*) from review where review_authors= "madbob";

The MLT-DR prototype split into the tokens as shown in the following screenshot. Figure 6.3 shows the unique template ID and template in JSON format.

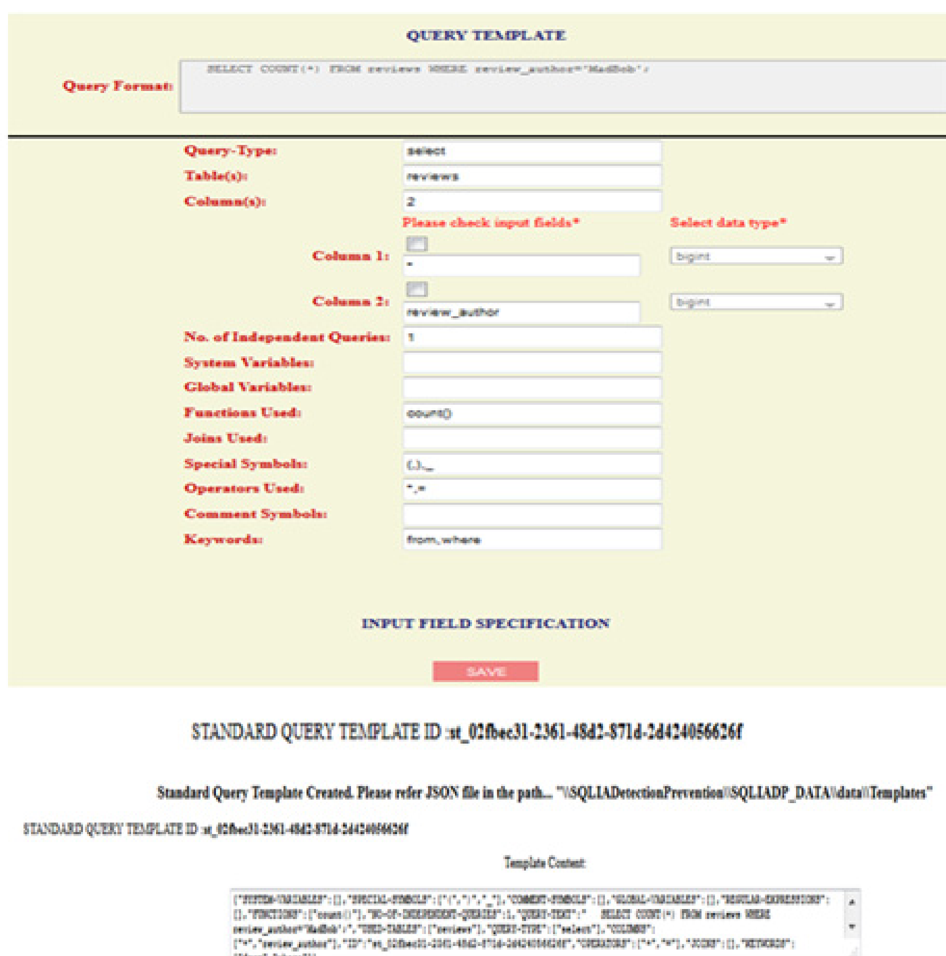


Fig. 6.3 The unique template ID and template in JSON format

Similarly, for all the input entries/SQL statements in the web application are identified by crawling through the entire application and the Unique ID for each query with the corresponding template format is generated and stored in the database structure server/the template repository. Figure 6.4 shows the generated unique ID for one of the identified web application for testing.

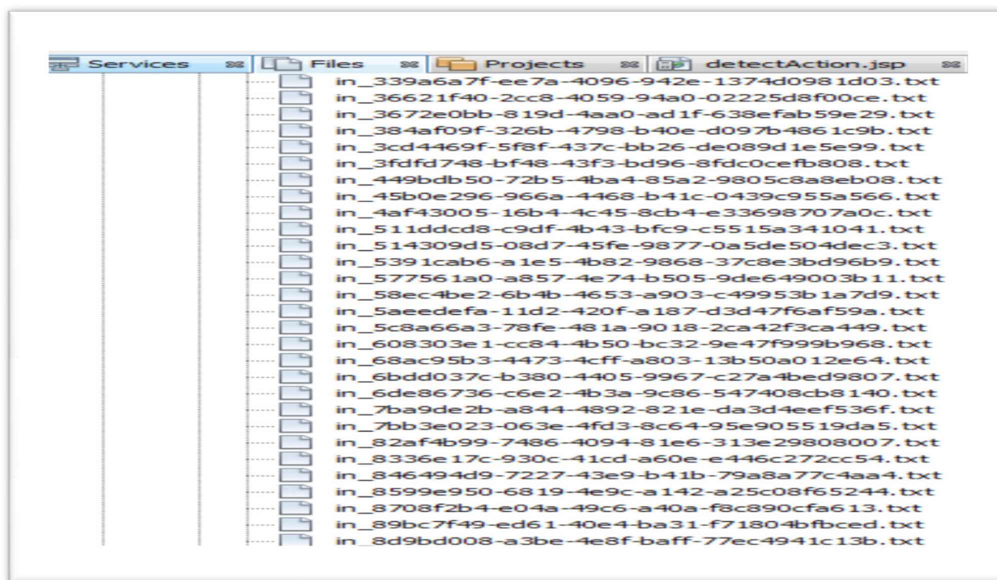


Fig. 6.4 The generated unique ID for one of the identified web application

Algorithm for Query Template Creation (QTC) is supporting the following major task of assigning ID ← get SQL_ query( ), Templat e(ID)←get JSON format(ID) and QTC← get-ID(). The query ID's, which is identified and listed corresponding to each web page is stored in the template repository and used in the testing phase.

6.3 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. The learning phase of SQTC model is already explained in the previous section. The learning phase of neural network based model also takes all the form field in response to the actual user inputs extracted by the crawler and set of SQL queries are collected from the URL of each webpage. We use the generated list of attack signature as input for the back propagated learning procedure. Each collected query is identified with an attack signature. Table 6.2 shows some of the identified attack vectors and signatures.

Table 6.2 Identified attack vectors and signatures

Vectors	SQL injection attack signature
v0	‘
v1	or
v2	=
v3	like
v4	select
v5	covert
v6	int
v7	char
v8	varchar
..	...
v30	Rot13 ( )
v31	*

The Ideal Standard query model corresponding to each page of the web application, which is learned using a back-propagated artificial neural network (details are in Chapter 5) is parsed and stored in the template repository. The legal SQL model is closely monitored to validate the constraint used. The constraints are the Data type used the type of value it can hold and the length of the token.

6.4 Testing Phase of MLT-DR

The testing phase of MLT-DR consists of the following major components for detecting the SQL injection attack: -

- Template generator/parser for user input query
- The Model mapper
- SQL Injection Attack Detection Engine
- Reconstruction component

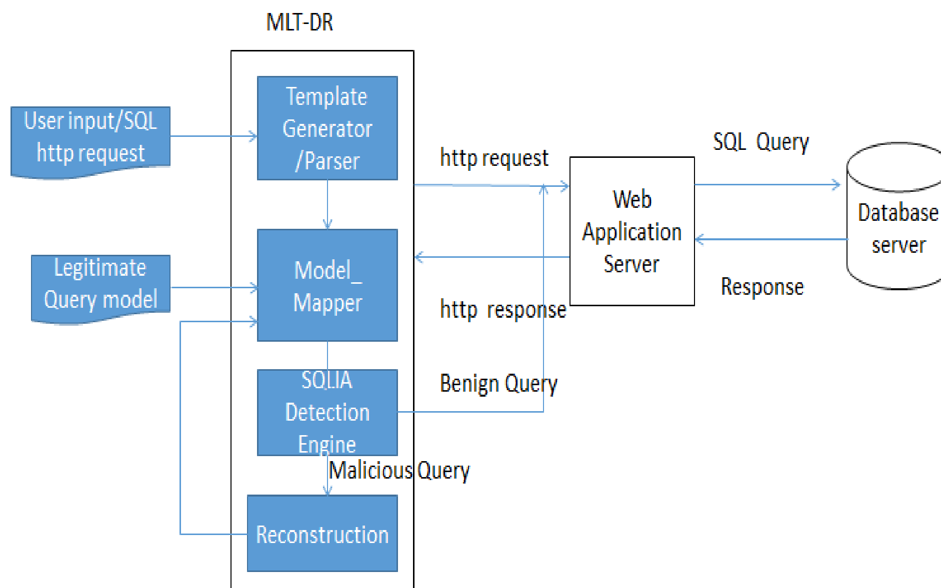


Fig. 6.5 MLT-DR Prototype in testing phase

6.4.1 Template Generator/Parser for User Input Query

In the testing phase of the MLT-DR framework, the template generator or parser / split the user queries accepted through the web pages. We perform splitting of the queries by invoking the template creator procedure of Standard Query Template Creator (SQTC). The parser generates the tokens for the user input query, and the generated/parsed tokens are in the similar format of the standard query tokens. The query is split into tokens by separating tokens such as Query type, Tables, Function, Special symbol, Comment Operator, Columns, Joins, S/m variable, Global variable, Keyword and Subquery (Buehrer and Sivilotti, 2005).

6.4.2 The Model Mapper

The primary task of the model mapper is to locate and retrieve the valid unique ID and template of the SQL query from the template repository, corresponding to the form field entry of the dynamic user queries. We store the legal query model with unique ID and the templates in the repository/ data structure server during the training phase. To have a faster access and retrieval of the appropriate query and to have better performance, we deploy an SQLIA-Shield (JAR file) with this framework. For example, the SQLIA-Shield for accepting user input through the login form can be specified by the path specification as shown in Table 6.3.

Table 6.3 Embedding a Template-ID in a SQLIA-Shield

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

There can be SQLIAShield/ JAR file for each page and each legal query ID and, location/path can be specified as shown above on the login page. We invoke a template mapper algorithm at this stage for mapping the parsed dynamic user queries with the legal query model against Code Injection attack. Figure 6.6 shows the identified standard and Input Query ID for mapping & 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":["Ã¢Â\u0080Â\u0099","Ã¢Â\u0080Â\u0099","Ã¢Â\u0080Â\u0098","Ã¢Â\u0080Â\u0099","Ã¢Â\u0080Â\u0099","Ã¢Â
\u0080Â\u0099"],"NO-OF-INDEPENDENT-QUERIES":1,"QUERY-TEXT":"SELECT * FROM Bookrevives WHERE ID=Â\u0080\u0099Â

Fig. 6.6 Identified Query ID for mapping & JSON format

6.4.3 SQL Injection Attack Detection Engine

SQL injection attacks are detected by matching the parsed user input query against the legal query model developed during the training phase. Detection engine gives status message by displaying the exact token and signature of the attack string. Each attack string is a clear indicator of the type/category of attack (Nguyen and Evans, 2005). The MLT-DR framework also makes use of a back-propagated Neural Network (BPNN) for constructing the legal query model. The user input queries are also matched against this BPN learnt model to detect the injection attack. If user

query input through the form field is not a complex query, then the detection engine displays the message as first level detection. In the case of a complex query, the detection engine has to repeat the procedure, and the detection is possible in the second level. Figure 6.8 shows a sample evaluation result of the prototype tool TbD. The dynamic query is chosen as ***SELECT name, phone from customers where id = 1 UNION all select creditcardnumber,1 from creditcard table***. The SQL Token mapper algorithm checks the match between the dynamic query and the standard query. An exact match is found on the token such as many Independent queries, System variable, Global variables, Joins and Comment symbols. But there is no match on the tokens such as Tables, Special symbols, Operators. So, the injection is detected and is displayed as shown in Figure 6.7. 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 (Mui and Frankl, 2010). Figure 6.7 shows the matched result by the detection engine using Standard Query Template Creator (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.

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

FIRST LEVEL DETECTION >>

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-c1b098841 4be**

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

Fig. 6.7 Evaluation result of the prototype tool TbD

6.4.4 Reconstruction Component

In the proposed MLT-DR framework, reconstruction of queries are carried out by comparing the user input queries with SQTC or BPNN learnt legal model and later with the support of REGEX function. The identified

malicious strings or substrings can be either eliminated or replaced. We invoke the reconstruction procedure and the model construction algorithm only when the HTTP request is re-confirmed that the malicious query is from an authenticated /registered user (Muthuprasana and Kothari, 2006). Figure 6.8 shows the status reports of the reconstruction procedure with several benign queries, dynamic (malicious) queries and its reconstructed queries. For example, consider the dynamic query: -insert into login (Username, Password, User Type, and 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 has to be removed or replaced with a null value. So, the reconstruction algorithm rejuvenates the malicious query and converts it into a benign query (Stolcke and Omohundro, 1993; Shi and Lin, 2012).

```
Output
SchoolManagement (run)   GlassFish Server 4.1.1
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
```

Fig. 6.8 Status report - Reconstruction procedure in MLT-DR framework

6.5 Summary of the Chapter

The implementation phase of the MLT-DR prototype has two primary tasks of learning the system properly with required details and tests it with an appropriate resource with user credentials. The prototype is implemented using Java-based NetBeans IDE and MySQL as the backend server. The empirical evaluation of the system shows that using this prototype, an efficient detection and blocking of SQL injection attack is possible. This chapter also explains the reconstruction of authenticated queries with the support of REGEX functionality. The reconstruction functionality included with this prototype increase the system availability and mitigates the Denial of service attack at a certain level. The deployed SQLIA-shield (JAR file) with appropriate path specification is beneficial for faster retrieval of template ID, corresponding to each SQL statement. The deployed prototype has better performance and reduced time-space complexity.

.....8003.....

Chapter 7

PERFORMANCE EVALUATION OF MLT-DR FRAMEWORK

Contents

7.1 Testing Hypothesis
7.2 Data Set Used for Testing MLT-DR
7.3 Performance Measures of MLT-DR
7.4 Type I & Type II Error
7.5 Receiver Operating Characteristic (ROC) Curve
7.6 Storage Overhead & Processing Time for Detection
7.7 Comparison of MLT-DR with Other Models
7.8 Summary of the Chapter

This chapter deals with various applications considered for evaluating the prototype and, effectiveness of the proposed framework. The proposed multilevel template based Detection and Reconstruction (MLT-DR) prototype is developed and implemented using Java based application software and MySQL as back-end database server. We deploy the Standard Query Template constructor (SQTC) in a data structure server or the template repository. 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 we store the Query template in JSON format, it decreases the storage overhead and to reduce the run time overhead. We assign an SQLIA-shield (JAR file) 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 feasible structure of the required learnt SQL model using Back-Propagated Neural Network (BPNN). All types of SQL Injection attacks are effectively detected and blocked by the implementation of MLT-DR. Apart from evaluating the prototype with the various standard applications and Cheatsheet, a customized school management application is exclusively developed to test the effectiveness and efficiency of the MLT-DR prototype.

7.1 Testing Hypothesis

Performance evaluation of the prototype, MLT-DR is carried out by justifying or answering the following research questions:

Q1: What is the percentage of attacks the proposed prototype is going to handle?

Q2. What percentage of injected code bypasses the MLT-DR, detection mechanism?

Q3. Does the prototype, impose any overhead on the underlying application? It should answer the following sub-questions:

Q3.1 If there is overhead incurred, what is the percentage of overhead?

Q3.2 Is the overhead imposed on the system negligible, while considering the fact that the Protection/ Security of database server has the highest priority than the delay involved in accessing the system?

Q4: Is there any occurrence of false positives or false negatives, while testing the prototype with empirical dataset collected?

Q5: How do you prove that MLT-DR is an efficient approach to detect and block code injection vulnerabilities as compared to the currently available countermeasures against SQLIA?

The empirical analysis carried out on various test bed and the test result of injection attacks shows that the proposed framework can justify all the above research queries. This proves that MLT-DR is much better and an effective approach in handling code injection security vulnerability as compared with the currently available countermeasures against SQL injection attacks. We evaluate the performance of the proposed model by

considering the factors such as process time overhead imposed on query execution, efficiency, effectiveness and precision.

7.2 Data Set Used for Testing MLT-DR

We evaluate the MLT-DR 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.

7.2.1 Dataset I: Data Available from Cheat Sheets/ URL

Table 7.1 shows the data collected from the cheat sheet/URL. The table also represents the application details, number of attack requested listed in the application and the details about the successful detection with corresponding false positives.

Table 7.1 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-sheet/sql-injection/oracle-sql-injection-cheat-sheet	35	35	0

7.2.2 The extract of Malicious Queries From URL

Table 7.2 shows the SQL injection attacks reported by various applications' URL. We test the listed queries with MLT-D frame work and detection status indicates that the "injection attack detected successfully".

Table 7.2 Sample vulnerability report with MLT-D detection status "Yes"

Application(URL)	Original Query	Injected Query
/home/jars/eclipse-workspace/ardilla/experiments/subjectPrograms/SymSchoolmate/index.php	select password from users where username = "1"	select password from users where username = "junk" or 1=1 -- "
/home/jars/eclipse-workspace/ardilla/experiments/subjectPrograms/SymSchoolmate/index.php	UPDATE courses SET aperc = '', bperc = '', cperc = '', dperc = '', fperc = '' WHERE courseid = '1'	UPDATE courses SET aperc = '', bperc = '', cperc = '', dperc = '', fperc = '' WHERE courseid = 'junk' or 1=1 -- '
/home/jars/eclipse-workspace/ardilla/experiments/subjectPrograms/WebChess_0.9.0/mainmenu.php	SELECT * FROM players WHERE nick = '1' AND password = ''	SELECT * FROM players WHERE nick = 'junk' or 1=1 -- ' AND password = ''
/home/jars/eclipse-workspace/ardilla/experiments/subjectPrograms/WebChess_0.9.0/mainmenu.php	SELECT * FROM pieces WHERE gameID = 1	SELECT * FROM pieces WHERE gameID = 5 or 1=1 --
/home/jars/eclipse-workspace/ardilla/experiments/subjectPrograms/faqforge-1.3.2/ index.php	SELECT * FROM Faq WHERE context = '1'	SELECT * FROM Faq WHERE context = 'junk' or 1=1 -- '
home/jars/eclipse-workspace/ardilla/experiments/subjectPrograms/EVE/eveactive/edit.php	SELECT MemberID, Name, Division, Date Joined, RankCorp, Vacation, Comment, Deleted FROM MembersMain WHERE MemberID='1'	SELECT MemberID, Name, Division, DateJoined, RankCorp, Vacation, Comment, Deleted FROM Members Main WHERE Member ID='junk' or 1=1 -'
/home/jars/eclipse-workspace/ardilla/experiments/subjectPrograms/geccBBlite/leggi.php	SELECT * FROM geccBB_forum WHERE id=1	SELECT * FROM geccBB_forum WHERE id=5 or 1=1 --
/home/jars/eclipse-workspace/ardilla/experiments/subjectPrograms/geccBBlite/leggi.php	SELECT id,rispostadel FROM geccBB_forum WHERE id=1	SELECT id,rispostadel FROM geccBB_forum WHERE id=5 or 1=1 --

7.2.3 Dataset II: Standard Test Suite Provided by Halfond and Orso

To evaluate the MLT-DR prototype, we identify the following applications from the test suite provided by Halfond and Orso, a standard test suite used for evaluating the prototype tool AMENSIA. We use a relational database as the backend server for the below-mentioned applications. Table 7.3 shows the identified application and number of hotspots identified in each application.

Table 7.3 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 7.4 clearly indicates the number of forms identified in each application and out of which how many forms are expected to be vulnerable is also clearly indicated. 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 7.4 Test result showing the effectiveness of MLT-DR

Application	#Forms	Expected Vuln_forms	Detected Vuln_forms	F_Positive #	F_Negative #
Book Store	25	21	20	1	1
Events	12	8	8	0	0
EmployeeDirectory	10	7	6	1	0

Vuln_forms: Vulnerable forms ; Positive: False positive; Negative: False Negative

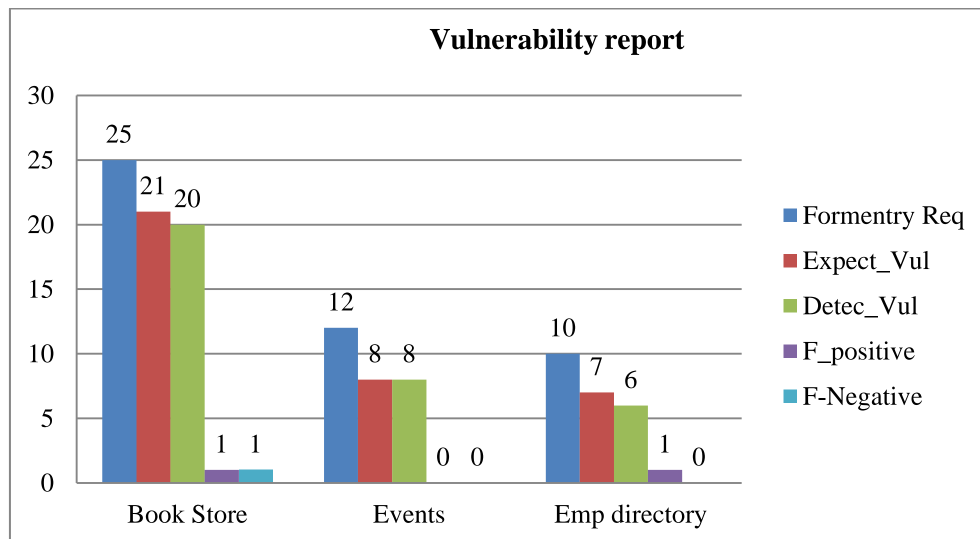


Fig.7.1 Test result showing the effectiveness of MLT-DR

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 MLT-DR 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.

7.2.4 Dataset III: Customized School Connect System

The School Econnect 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 and support staff, online resource management system and event planner for the school activities. Table 7.5 shows the identified hotspots, expected attacks, detected attacks and the corresponding false positive rates.

Table 7.5 School connect 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	27	22	22	0
Event-planner	Management system Event planning system	18	12	11	0

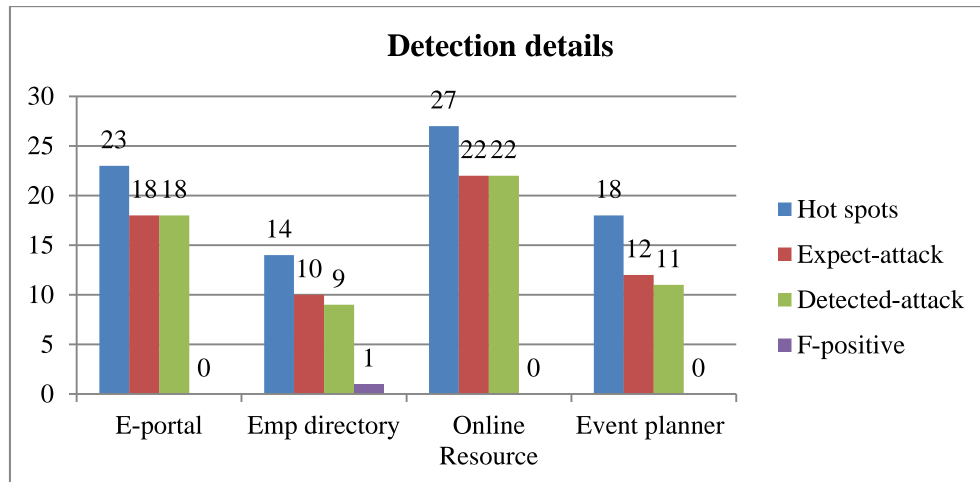


Fig. 7.2 SchoolEconnect with attack detection details

The above details reported in Table 7.5 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 MLT-DR 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.

7.3 Performance Measures of MLT-DR

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. JSON is a logic implementation closer to the database level and simple data storage design. The factors such as efficiency, effectiveness and precision are the base for the evaluation of the proposed model.

7.3.1 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 measure the average CPU time spent for processing the query. Since we use JSON format, the storage and retrieval of standard queries in the MLT-DR 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 a few seconds, and the reconstruction of the queries are also carried out with a negligible delay in response time. Hence the processing time and the overhead involved in executing the query is negligible when comparing it with the response time of a browser in accessing the web application.

7.3.2 Efficiency of MLT-DR

There is a secure and insecure version of the SchoolEconnect application designed, 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 7.6. 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 SchoolEconnect application to analyze and identify the process time overhead. In the secure version, the proxy server with MLT-DR module is implemented to check and block the malicious or injected queries, before it reaches to the backend server of the SchoolEconnect.

Table 7.6 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

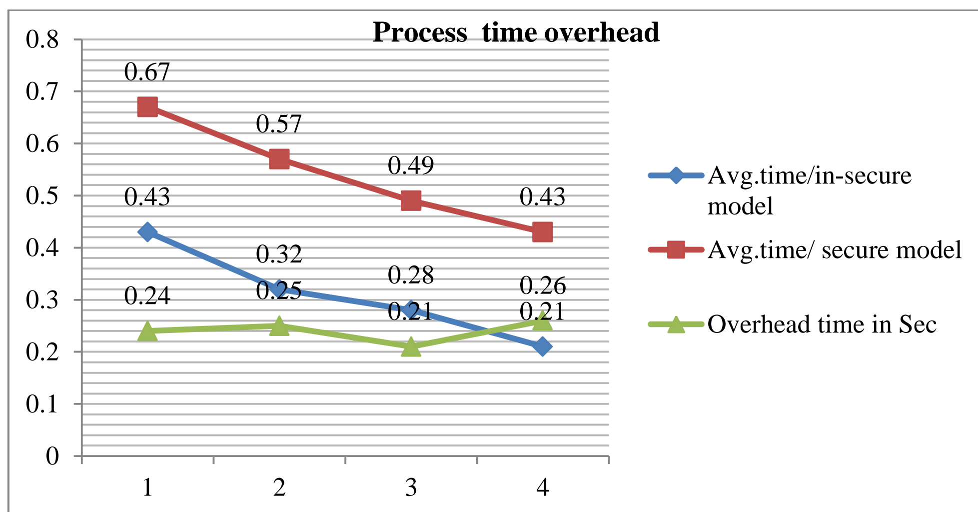


Fig. 7.3 Test result showing the efficiency of MLT-DR

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 the 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 we ignore it. Therefore, the overhead incurred by deploying the secure MLT-DR 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.

7.3.3 Precision of MLT-DR

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 MLT-DR model. The analysis in Table 7.7 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 MLT-DR. We patch this in the trial run.

Table 7.7 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 7.8 shows the empirical analysis of MLT-DR using multiple variations of the queries mentioned above tested with the SchoolEconnect application.

Table 7.8 Analysis of false positives in SchoolEconnect 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.

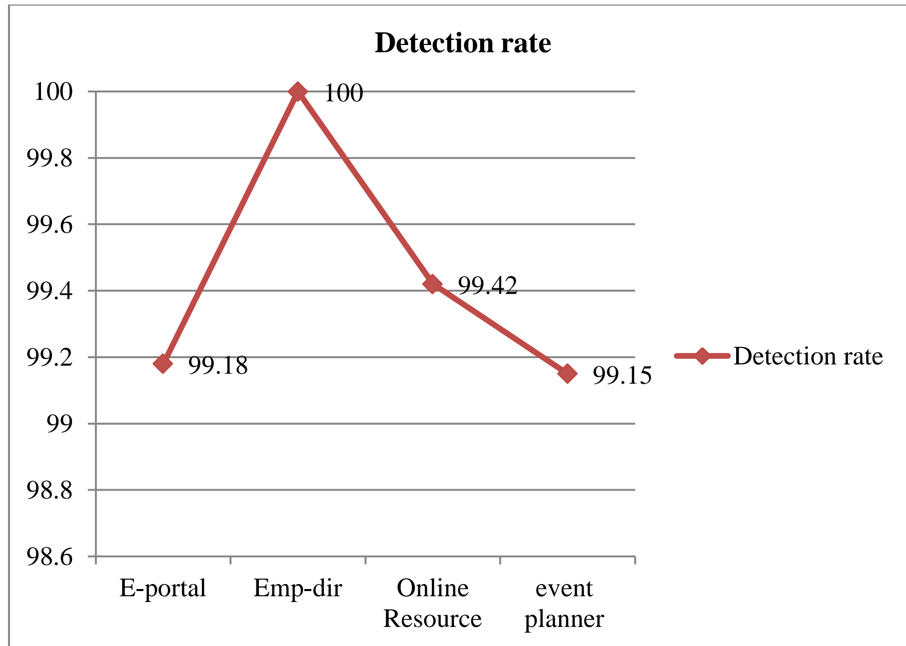


Fig.7.4 Precision analysis in SchoolEconnect application

7.3.4 Effectiveness of MLT-DR

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 7.9 indicates that more than 98% detection is possible based on the occurrence of false negatives and false positives reported.

Table 7.9 Attack categories and detection rate in MLT-D

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.

7.4 Type I & Type II Error

We test the proposed MLT-DR 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 7.5 indicates that the MLT-DR 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.

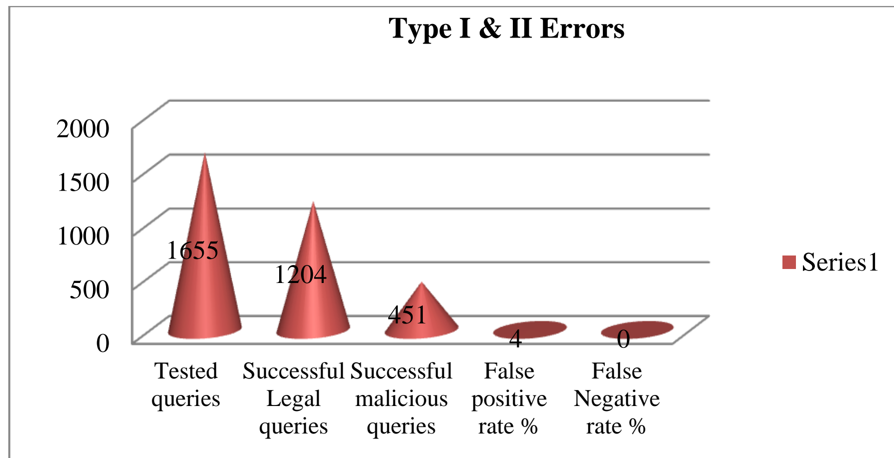


Fig. 7.5 Type I & Type II error rate

7.5 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) shown in Figure 7.6 indicates that the model is achieving 100% result, which is plotted using the values taken from Emp-directory application shown in Table 7.8. The Receiver Operating Characteristic (ROC) curve is plotted between the true positive rate in Y axis (Sensitivity) and the false positive rate in X axis (specificity). The sensitivity is the probability that a test result will be positive when the anomaly is present. The specificity is the probability that a test result will be negative when the anomaly is not present. The Area Under Curve (AUC=1.00) equal to 1 indicates that 100% detection rate is achieved in Emp-directory application

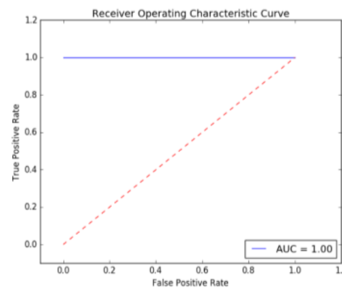


Fig. 7.6 Receiver Operating Characteristic (ROC) curve

7.6 Storage Overhead & Processing Time for Detection

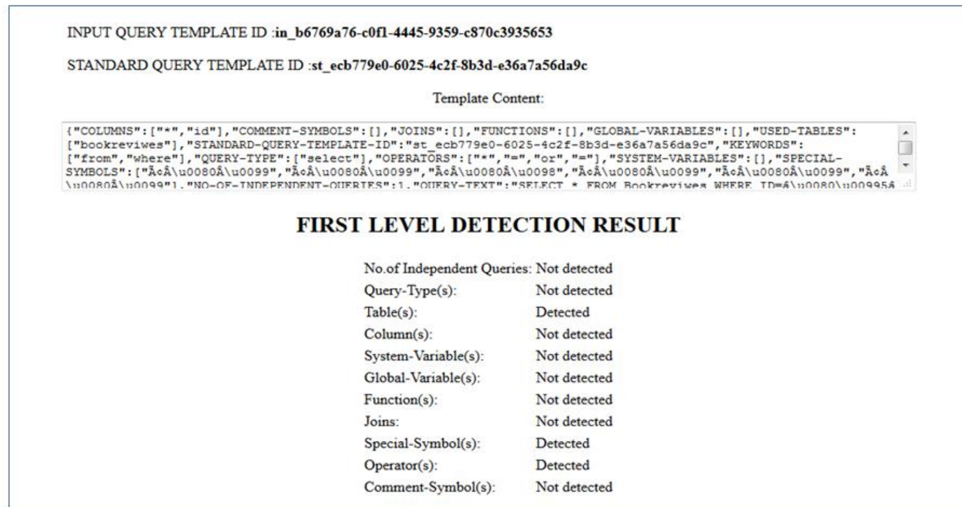


Fig. 7.7 Detection procedure and JSON storage Format

Figure 7.7 shows the MLT-DR model which uses the Unique Template ID and JSON specifications for validation of standard query with user input query. Hence the storage and retrieval of standard queries in the MLT-DR model are easier and faster as compared to the other standard models explained in section 7.8. The proposed model has a strategy for detecting the queries by placing them in distinct categories and complexity levels. The following detection procedure shows that we can identify the less complex queries on the first level.

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

Fig. 7.8 First level Injection detection

The queries which are complex and require traversing technique to split it into a smaller format, only need a bit 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 few seconds as already explained in the previous section. Figure 7.9 shows the status report of Rejuvenation procedure.

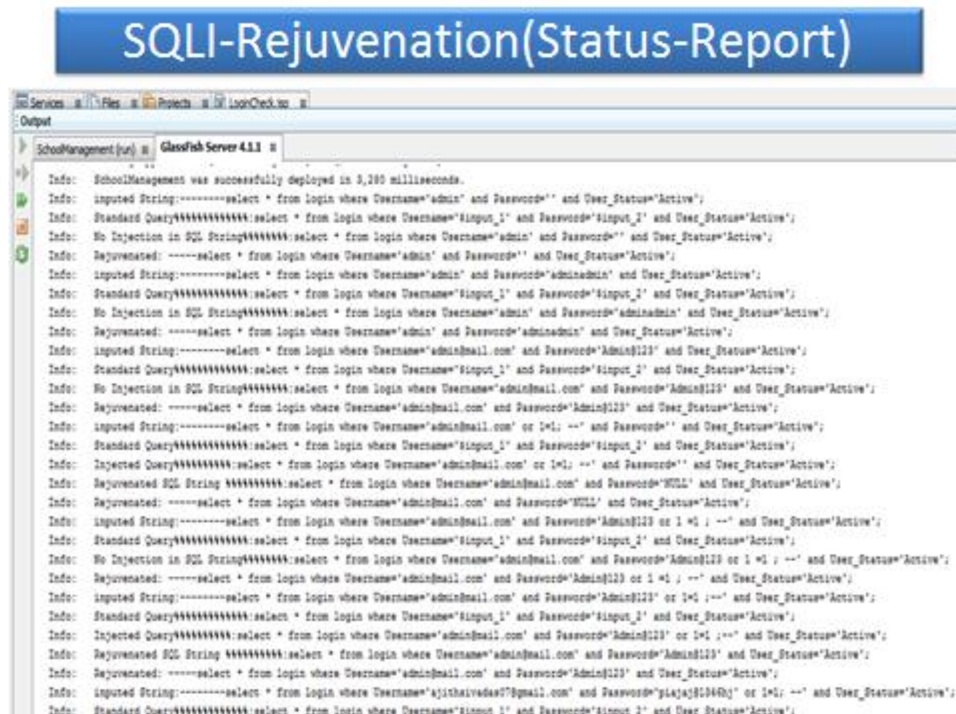


Fig. 7.9 SQL Rejuvenation status report

The report indicates that reconstruction of the queries is also carried out in a few seconds with a negligible delay in response time. Hence the memory (CPU time) and the overhead involved in processing the query is negligible while comparing it with the response time of a browser in accessing the web application.

7.7 Comparison of MLT-DR 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. 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 as shown in Table 7.10 (George and Jacob, 2018; Halfond and William, 2005; Johari and Pankaj, 2012). 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, efficient handling of Time-Space complexity(Bisht and Prithvi, 2012; Moradpoor, 2014).

Table 7.10 Comparison of MLT-DR with other models

Detection/ Prevention method	Tautologies	Union Queries	Illegal Queries	Piggy-back	Inference	Alternate encoding	Stored procedure
AMNESIA	✓	✓	✓	✓	✓	✓	×
IDS	/	/	/	/	/	/	/
SQL Check	✓	✓	✓	✓	✓	✓	×
SQL Guard	✓	✓	✓	✓	✓	✓	×
Tautology checker	✓	×	×	×	×	×	×
JDBC Checker	NA	NA	NA	NA	NA	NA	NA
SQL DOM	✓	✓	✓	✓	✓	✓	×
Proposed MLT-DR	✓	✓	✓	✓	✓	✓	✓

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

Comparison of proposed model with the other prevention techniques shown in the above table, indicates that MLT-DR can effectively handle all seven basic SQL injections, but the other procedures, only partially detect the attacks. The procedure mentioned in AMNESIA, has a strong static analysis and prevention procedure which is carried out in most of the methods shown in Table 7.10 (Halfond and William, 2005). Detection of attack on stored procedures is very difficult to handle by any of the techniques mentioned, except MLT-DR. The techniques focus on queries generated within the application and extending these techniques to other data sets require substantial effort and source code modification. The factors on which the

detection techniques can be evaluated are based on the injection mechanism, deployment requirement and the defensive coding practices. Many prevention methods are based on some conservative analysis, and the major drawbacks are the run time overhead, and the presence of false positives.

7.8 Summary of the Chapter

The proposed prototype MLT-DR, detect and prevent SQL Injections effectively with negligible processing overhead. The Token parsing techniques used in the template creator application, template files stored in the JSON format and Jar files used 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 shows that 100% detection is possible with the proposed framework. MLT-DR blocking the entire tested malicious query without any false negatives indicates that the proposed techniques handle malicious queries effectively. The SQLI-Reconstruction framework with the backward-propagated neural network is a unique approach towards the reconstruction of queries from authenticated users, which will increase the application availability and decrease the Denial of Services for the authenticated users. The empirical evaluation performed on “Secure SchoolEConnect” (customized online application) indicates that the proposed system has only the bare minimum overheads and Time space complexity. The above mentioned empirical analysis of MLT-DR on different data shows that the prototype is an efficient and effective approach towards the detection of SQL Injection and it is blocking or preventing SQL Injections effectively.

.....ഇര.....

Chapter 8

CONCLUSION AND FUTURE WORK

Contents

8.1 Summary of the Research Work
8.2 Major Highlights of the Research Works
8.3 Future Directions
8.4 Conclusion

This Chapter summarizes the research work by highlighting various contributions made by the proposed model and its significance while comparing it with the other existing models. This chapter also discusses the future directions.

8.1 Summary of the Research Work

The proposed Multi-Level Template based Detection and Reconstruction (MLT-DR), is a hybrid model that makes use of a novel technique to detect and block SQL Injection attacks. The essential tasks carried out by this model include parsing and analyzing the legal and dynamic input queries. Standard Query Template Creator (SQTC) procedure, the template mapper algorithm and the model constructor algorithm are additional features that are available with this hybrid model. In the Multilevel Detection framework (MLT-D) of the proposed approach, we analyze all possible intended queries from the given web application with the support of web crawler to

identify the hotspots, and legal queries are collected for further processing of parsing and generating the templates. In this model, template specification with a unique ID is established for each query and stored in the template repository/Data structure server. Dynamic queries are analyzed and parsed using the similar approach and matched against the corresponding template ID of the legal query retrieved from the repository. In the absence of a match between the intended queries and injected queries, the queries are flagged as a malicious query and blocked from further execution at the backend database server. In the Reconstruction framework, we reconstruct the queries from authenticated users by using the regular expression. The model reconstruction algorithm is the core component for rejuvenating queries from authenticated users. The empirical analysis performed using this hybrid model indicates that the proposed model is a novel approach that can efficiently detect and block malicious input entries, irrespective of the underlying database and application software. The proposed architecture does not demand any source code modifications and can perform detailed analysis at negligible computational overheads without false positives or false negatives. MLT-DR prototype is developed using Java based application software and has been implemented to validate the efficiency and effectiveness of the template based detection model.

8.2 Major Highlights of the Research Works

The major highlights of the research work are:

- A detailed survey and analysis on existing techniques for detecting SQL injection and counter measures of attacks on web applications has been done.

- Proposed an effective method of multilevel template creator application to block the malicious injected query. The effective framework Multi-Level Template based Detection and reconstruction framework (MLT-DR) has been developed to detect SQL Injections and to prevent attacks without false positives with maximum possible efficiency. It consists of several sub-tasks to validate SQL queries before it gets executed by the database server. In the proposed approach, the user requests received through dynamic web pages of a web server are directed to the MLT-DR framework to detect the injection, and it directs only benign queries to the database server.

  The different modules in MLT-DR prototype implemented in the proxy server to validate the SQL queries are: -

- **Standard Query Template Creator:** To identify and parse the intended queries of the web application and to get the various tokens/templates specification. The template creator module uses the proposed Standard Query Template creator (SQTC) algorithm to create the template.

- **Template Repository:** Stores all the intended queries assigned with a unique ID. There can be multiple dynamic pages for each web application. Each intended/legal query template is stored in a template repository with a unique ID and in JSON Format.

- **Template Mapper:** To check the validity of the input query the template mapper maps the appropriate intended query from the template repository against the unique input query. Template

mapper algorithm is proposed to perform the mapping operation. Subsequently, the corresponding alert message is sent to the evaluation engine.

- **SQLI- Reconstruction** within the API has feature-rich constructors and string comparison strategies against the Regex function and is also supported with a proposed template match and model constructor algorithms to improve the availability aspect of security pillar and to ensure better accuracy and response time. The important features in the proposed approach is the automatic creation of standard query templates from training dataset, validation of user input against regular expression patterns and reconstruction of injected queries from authenticated users as needed.
- **BPANN–Back-Propagated Artificial Neural Network for** training data set to create a learnt model for legal queries for the web pages, it can be used as an alternative to the SQTC sub-module, and we store it in JSON format.
- **The Query Reconstruction:** The Query Reconstruction module reconstructs the queries by considering the available authentication privileges, eliminating injections and also rebuilds missing portions if any, and removes the injected part of the user query by following the procedure of the Model Construction and REGEX functions.

- Proposed an effective algorithm for Parsing/Tokenizing the query and Multilevel Template Mapper procedure for detection and prevention of SQL Injection attack.
- Proposed an effective Model construction algorithm to reconstruct the malicious queries from authenticated user.
- Proposed Jar files such as SQLI-Shield, SQLI-Rejuvenator JAR to reduce complexity and better performance of the online application.
- Developed and implemented a prototype to validate the efficiency and effectiveness of the template-based detection model.

A hybrid framework of MLT-DR is developed to detect/prevent SQL-Injection attack categories such as tautologies, logically incorrect queries, union queries, piggy-backed queries, stored procedures, timing attacks and alternate encoding.

8.3 Future Directions

SQL Injection will decrease the system availability. The existing tools for scanning, validating, protecting and preventing injection attacks on web pages still need further expansion and better strategies to efficiently handle the highly automated malicious attacks. Exploiting the code injection vulnerabilities to penetrate the backend database server to steal or disclose the highly sensitive information is one of the most dangerous attacks in a highly confidential web application, and the consequences of these types of attacks create a massive impact on the business applications. Most of the

existing SQLI-Detection and prevention approaches undergo the following issues: -

- They target only a subset of SQLI attack types. A few approaches are developed to handle distinct categories of injections attacks without false positives.
- During Dynamic phase, SQLI validations and modification of application code on the online applications are expensive with respect to time and space and results in deficient performance of the web application.
- A root level SQL Injection attack on the database servers can lead to destruction of Confidentiality, Integrity, Availability aspects of the application.
- The denial of service attack resulting from the SQL Injection is critical in most of the business applications, which should be mitigated with high degree of importance.

Even though there are many existing technologies and strategies available to detect and block the SQL injections, yet the SQL injection attack is frequent and it is a major concern of security professionals. Most of the available deployment strategies against injection attacks require significant modification of source code and require additional infrastructure. Hence still there is a requirement for an effective technique to handle the Zero-day attack (to handle the upcoming vulnerability, yet to be identified) with better performance and efficiency.

The future directions of this research are to focus on the areas of code injection attack, other than SQLIA (Vigna and Valeur, 2005; Calvi and Vigano, 2016). The appropriate measure of importance should be given to the injection happening on different database storage of XML databases and storage handling. Injected queries identified during the validation or testing phase of this research work are logged and reused for model re-construction and patching in a later stage. The unknown string/injection pattern, which is blocked during the detection stage, can be stored in a data structure server. These malicious patterns can be learned and later used as an immediate patch against zero-day attack. The query validation procedure adopted in this research can be re-structured for validating emerging attack within the source code (George and Jacob, 2018). The tokenizing procedure or the statistical analysis used in this work can be extended to X-Query injection attack and Parameter Tampering attack. NoSQL injection attack executes in more complex areas of an application than the traditional SQL. In a combination of SQL injection and Parameter tampering attacks, the injected parameters contain malicious SQL/XML commands, which lead to a SQL/Xpath injection. Although sufficient research work has been conducted to find countermeasures to SQL Injection attacks, only limited work has been carried out regarding Parameter Tampering attacks. With the tremendous increase in XML Database by the web applications, XQuery/X-path injections are also becoming a critical vulnerability (Medeiros and Neves, 2016). In the future, this research can be extended to analyze the application vulnerabilities in complex SQL exploitations, such as stored XSS and broken session management.

8.4 Conclusion

The major goal of this research work is to design and implement an effective technique for detecting and preventing SQL Injection attacks in online applications with better performance, higher efficiency and reduced denial of service attacks. To achieve this goal, we implement the proposed hybrid model MLT-DR with techniques to detect SQL Injections and prevent attacks. The parsing method used in the query template creator application was an unconventional approach to scale up the performance of the proposed model. Template files stored in the JSON format contributed equally to decrease the storage overheads. The empirical analysis that was carried out by using data from variously shared insecure-sites showed that almost 100% detection was possible irrespective of the backend database servers with appropriate structure modification on the selected SQL query for testing. The empirical evaluation performed on customized online applications indicates that the proposed system could achieve the optimum performance with negligible false positives and false negatives. We can use the MLT-DR prototype as a scanning tool to detect and prevent SQL Injection attacks in various web applications. Vulnerabilities in web applications could be sanitized, mitigated or controlled by expanding this research study towards various categories of code injection attacks. There are instances where prepared statements, parameterized queries or stored procedures can be used for web-user interactions instead of dynamic SQL statements or form-field entry. Dynamic SQL queries and stored procedures are two of the most important components in an SQL server. Stored procedures are not vulnerable to SQL injection attacks, but it is not effective if the query is complex and requires frequent modification. Parameterized queries cannot be used with dynamic SQL, if a table or column name is passed as parameter. If there is frequent change in table name, columns or number of parameters in the query, dynamic SQL would be the better choice. As dynamic SQL is a significant component which has a key identity in the hierarchy of server security, there should be appropriate measures to ensure the security of the SQL server, which uses dynamic SQL.

.....ഇ.....

.....ജ്ഞ.....

References

[1] Akrout, R., Alata, E., Kaaniche, M., & Nicomette, V. (2014). An automated black box approach for web vulnerability identification and attack scenario generation. *Journal of the Brazilian Computer Society*, *20*(1), 4.

[2] Ali, S., Shahzad, S. K., & Javed, H. (2009). Sqlipa: An authentication mechanism against sql injection. *European Journal of Scientific Research*, *38*(4), 604-611.

[3] Aliero, M. S., Ghani, I., Zainudden, S., Khan, M. M., & Bello, M. (2015). Review on SQL Injection Protection Methods and Tools. *Jurnal Teknologi*, *77*(13).

[4] Alata, E., Kaâniche, M., Nicomette, V., & Akrout, R. (2013, April). An automated approach to generate web applications attack scenarios. In *Dependable Computing (LADC), 2013 Sixth Latin-American Symposium on* (pp. 78-85). IEEE.

[5] Alfantookh, Abdulkader A. (2004) "An automated universal server level solution for SQL injection security flaw." *Proceedings of the 2004 International Conference on Electrical, Electronic and Computer Engineering (ICEEC'04)*.

[6] Anderson, B., Quist, D., & Lane, T. (2011). Detecting code injection attacks in internet explorer. In *Computer Software and Applications Conference Workshops (COMPSACW), 2011 IEEE 35th Annual* (pp. 90-95). IEEE.

[7] Armando, A., Arsac, W., Avanesov, T., Barletta, M., Calvi, A., Cappai, A., & Erzse, G. (2012). The AVANTSSAR platform for the automated validation of trust and security of service-oriented architectures. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems* (pp. 267-282). Springer Berlin Heidelberg.

[8] Artzi, S., Kiezun, A., Dolby, J., Tip, F., Dig, D., Paradkar, A., & Ernst, M. D. (2008). Finding bugs in dynamic web applications. In *Proceedings of the 2008 international symposium on Software testing and analysis* (pp. 261-272). ACM

[9] Avireddy, S., Perumal, V., Gowraj, N., Kannan, R. S., Thinakaran, P., Ganapthi, S., & Prabhu, S. (2012). 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 on* (pp. 1327-1333). IEEE.

[10] Awang, N. F., & Manaf, A. A. (2015). Automated Security Testing Framework for Detecting SQL Injection Vulnerability in Web Application. In *International Conference on Global Security, Safety, and Sustainability* (pp. 160-171). Springer International Publishing.

[11] Awang, N. F., Manaf, A. A., & Zainudin, W. S. (2014). A survey on conducting vulnerability assessment in web-based application. In *International Conference on Advanced Machine Learning Technologies and Applications* (pp. 459-471). Springer International Publishing.

[12] Balduzzi, M., Gimenez, C. T., Balzarotti, D., & Kirda, E. (2011). Automated Discovery of Parameter Pollution Vulnerabilities in Web Applications. In *NDSS*.

[13] Bangre, S., & Jaiswal, A. (2012). SQL Injection Detection and Prevention Using Input Filter Technique. *International Journal of Recent Technology and Engineering (IJRTE) ISSN*, 2277-3878.

[14] Bau, J., Bursztein, E., Gupta, D., & Mitchell, J. (2010). State of the art: Automated black-box web application vulnerability testing. In *Security and Privacy (SP), 2010 IEEE Symposium on* (pp. 332-345). IEEE.

[15] Belk, M., Coles, M., Goldschmidt, C., Howard, M., Randolph, K., Saario, M., & Yonchev, Y. (2011). Fundamental Practices for Secure Software Development. *A Guide to the Most Effective Secure Development Practices in Use Today*, *8*.

[16] Bertino, E., Kamra, A., & Early, J. P. (2007). Profiling database application to detect sql injection attacks. In *Performance, Computing, and Communications Conference, 2007. IPCCC 2007. IEEE Internationa* (pp. 449-458). IEEE

[17] Buehrer, G., Weide, B. W., & Sivilotti, P. A. (2005). Using parse tree validation to prevent SQL injection attacks. In *Proceedings of the 5th international workshop on Software engineering and middleware* (pp. 106-113). ACM.

[18] Bhoria, P., & Garg, K. (2013). Determining feature set of DOS attacks. *International Journal of Advanced Research in Computer Science and Software Engineering*, *3*(5), 875-878

[19] Bisht, P., Madhusudan, P., & Venkatakrishnan, V. N. (2010). CANDID: Dynamic candidate evaluations for automatic prevention of SQL injection attacks. *ACM Transactions on Information and System Security (TISSEC)*, *13*(2), 14.

[20] Bisht, P., Hinrichs, T., Skrupsky, N., & Venkatakrishnan, V. N. (2011). WAPTEC: whitebox analysis of web applications for parameter tampering exploit construction. In *Proceedings of the 18th ACM conference on Computer and communications security* (pp. 575-586). ACM.

[21] Borade, M. R., & Deshpande, N. A. (2014). Web Services Based SQL Injection Detection and Prevention System for Web Applications. *International Journal of Emerging Technology and Advanced Engineering Website: www. ijetae. com (ISSN 2250-2459, ISO 9001: 2008 Certified Journal, Volume 4, Issue 10*

[22] Bosworth, S., & Kabay, M. E. (Eds.). (2002). *Computer security handbook*. John Wiley & Sons.

[23] Boyd, S. W., & Keromytis, A. D. (2004). SQLrand: Preventing SQL injection attacks. In *International Conference on Applied Cryptography and Network Security* (pp. 292-302). Springer Berlin Heidelberg

[24] Buehrer, G., Weide, B. W., & Sivilotti, P. A. (2005). Using parse tree validation to prevent SQL injection attacks. In *Proceedings of the 5th international workshop on Software engineering and middleware* (pp. 106-113). ACM.

[25] Büchler, M., Oudinet, J., & Pretschner, A. (2012). Semi-automatic security testing of web applications from a secure model. In *Software Security and Reliability (SERE), 2012 IEEE Sixth International Conference on* (pp. 253-262). IEEE.

[26] Buja, G., Jalil, K. B. A., Ali, F. B. H. M., & Rahman, T. F. A. (2014). Detection model for SQL injection attack: An approach for preventing a web application from the SQL injection attack. In *Computer Applications and Industrial Electronics (ISCAIE), 2014 IEEE Symposium on* (pp. 60-64). IEEE.

[27] Burkhart, M., Schatzmann, D., Trammell, B., Boschi, E., & Plattner, B. (2010). The role of network trace anonymization under attack. *ACM SIGCOMM Computer Communication Review*, *40*(1), 5-11.

[28] Calvi, A., & Viganò, L. (2016). An automated approach for testing the security of web applications against chained attacks. In *Proceedings of the 31st Annual ACM Symposium on Applied Computing* (pp. 2095-2102). ACM.

[29] Chen, S., Xu, J., Kalbarczyk, Z., & Iyer, K. (2006). Security vulnerabilities: From analysis to detection and masking techniques. *Proceedings of the IEEE*, *94*(2), 407-418.

[30] Chen, J. M., & Wu, C. L. (2010). An automated vulnerability scanner for injection attack based on injection point. In *Computer Symposium (ICS), 2010 International* (pp. 113-118). IEEE

[31] Cheon, E. H., Huang, Z., & Lee, Y. S. (2013). Preventing SQL injection attack based on machine learning. *International Journal of Advancements in Computing Technology*, *5*(9), 967-974

[32] Ciampa, A., Visaggio, C. A., & Di Penta, M. (2010). A heuristic-based approach for detecting SQL-injection vulnerabilities in Web applications. In *Proceedings of the 2010 ICSE Workshop on Software Engineering for Secure Systems* (pp. 43-49). ACM.

[33] Clarke-Salt, J. (2009). *SQL injection attacks and defense*. Elsevier.

[34] Cova, M., Balzarotti, D., Felmetsger, V., & Vigna, G. (2007). Swaddler: An approach for the anomaly-based detection of state violations in web applications. In *International Workshop on Recent Advances in Intrusion Detection* (pp. 63-86). Springer Berlin Heidelberg.

[35] Das, D., Sharma, U., & Bhattacharyya, D. K. (2010). An approach to detection of SQL injection attack based on dynamic query matching. *International Journal of Computer Applications*, *1*(25), 28-34.

[36] De Meo, F., Rocchetto, M., & Viganò, L. (2016). Formal Analysis of Vulnerabilities of Web Applications Based on SQL Injection. In *International Workshop on Security and Trust Management* (pp. 179-195). Springer International Publishing.

[37] Desmet, L., Verbaeten, P., Joosen, W., & Piessens, F. (2008). Provable protection against web application vulnerabilities related to session data dependencies. *IEEE Transactions on Software Engineering*, *34*(1), 50-64.

[38] Dharam, R., & Shiva, S. G. (2012). Runtime monitoring technique to handle tautology based SQL injection attacks. *International Journal of Cyber-Security and Digital Forensics (IJCSDF)*, *1*(3), 189-203.

[39] Djuric, Z. (2013, September). A black-box testing tool for detecting SQL injection vulnerabilities. In *Informatics and Applications (ICIA), 2013 Second International Conference on* (pp. 216-221). IEEE.

[40] Doupé, A., Cova, M., & Vigna, G. (2010). Why Johnny can't pentest: An analysis of black-box web vulnerability scanners. In *International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment* (pp. 111-131). Springer Berlin Heidelberg.

[41] Ezumalai, R., & Aghila, G. (2009). Combinatorial approach for preventing SQL injection attacks. In *Advance Computing Conference, 2009. IACC 2009. IEEE International* (pp. 1212-1217). IEEE.

[42] Felmetsger, V., Cavedon, L., Kruegel, C., & Vigna, G. (2010). Toward automated detection of logic vulnerabilities in web applications. In *USENIX Security Symposium* (Vol. 58).

[43] George, T. K., & Jacob, P. (2015) Analysis On Security Vulnerability In On Line Application, Data Validation Strategies And Testing Tools. *System*, *2*, 3.

[44] George, T. K., & Jacob, P. (2016) A Proposed Architecture for Query Anomaly Detection and Prevention against SQL Injection Attacks.

[45] George, T. K., James, R., & Jacob, P. (2016). Proposed Hybrid model to detect and prevent SQL Injection. *International Journal of Computer Science and Information Security*, *14*(6), 441.

[46] Gould, C., Su, Z., & Devanbu, P. (2004). JDBC checker: A static analysis tool for SQL/JDBC applications. In *Proceedings of the 26th International Conference on Software Engineering* (pp. 697-698). IEEE Computer Society.

[47] Godefroid, P., Levin, M. Y., & Molnar, D. A. (2008) Automated Whitebox Fuzz Testing. In *NDSS* (Vol. 8, pp. 151-166).

[48] Halfond, W. G., Choudhary, S. R., & Orso, A. (2011). Improving penetration testing through static and dynamic analysis. *Software Testing, Verification and Reliability*, *21*(3), 195-214.

[49] Halfond, W. G., & Orso, A. (2005). AMNESIA: analysis and monitoring for NEutralizing SQL-injection attacks. In *Proceedings of the 20th IEEE/ACM international Conference on Automated software engineering* (pp. 174-183). ACM.

[50] Halfond, W., Orso, A., & Manolios, P. (2008). WASP: Protecting Web applications using positive tainting and syntax-aware evaluation. *IEEE Transactions on Software Engineering*, *34*(1), 65-81.

[51] Halfond, W. G., Viegas, J., & Orso, A. (2006). A classification of SQL-injection attacks and countermeasures. In *Proceedings of the IEEE International Symposium on Secure Software Engineering* (Vol. 1, pp. 13-15). IEEE

[52] Haykin, S. S., Haykin, S. S., Haykin, S. S., & Haykin, S. S. (2009). *Neural networks and learning machines* (Vol. 3). Upper Saddle River, NJ, USA:: Pearson.

[53] Huang, Y. W., Yu, F., Hang, C., Tsai, C. H., Lee, D. T., & Kuo, S. Y. (2004). Securing web application code by static analysis and runtime protection. In *Proceedings of the 13th international conference on World Wide Web* (pp. 40-52). ACM.

[54] Huang, Y. W., Tsai, C. H., Lin, T. P., Huang, S. K., Lee, D. T., & Kuo, S. Y. (2005). A testing framework for Web application security assessment. *Computer Networks*, *48*(5), 739-761.

[55] Huang, Y. W., Huang, S. K., Lin, T. P., & Tsai, C. H. (2003). Web application security assessment by fault injection and behavior monitoring. In *Proceedings of the 12th international conference on World Wide Web* (pp. 148-159). ACM.

[56] Hu, W., Hiser, J., Williams, D., Filipi, A., Davidson, J. W., Evans, D. & Rowanhill, J. (2006). Secure and practical defense against code-injection attacks using software dynamic translation. In *Proceedings of the 2nd international conference on Virtual execution environments* (pp. 2-12). ACM.

[57] Johari, R., & Sharma, P. (2012). A survey on web application vulnerabilities (SQLIA, XSS) exploitation and security engine for SQL injection. In *Communication Systems and Network Technologies (CSNT), 2012 International Conference on* (pp. 453-458). IEEE.

[58] Jose, S., Priyadarshini, K., & Abirami, K. (2016). An Analysis of Black-Box Web Application Vulnerability Scanners in SQLi Detection. In *Proceedings of the International Conference on Soft Computing Systems* (pp. 177-185). Springer India.

[59] Joseph, S., & Jevitha, K. P. (2016). Evaluating the Effectiveness of Conventional Fixes for SQL Injection Vulnerability. In *Proceedings of 3rd International Conference on Advanced Computing, Networking and Informatics* (pp. 417-426). Springer India.

[60] Joshi, A., & Geetha, V. (2014). SQL injection detection using machine learning. In *Control, Instrumentation, Communication and Computational Technologies (ICCICCT), 2014 International Conference on* (pp. 1111-1115). IEEE

[61] Jovanovic, N., Kruegel, C., & Kirda, E. (2006). Precise alias analysis for static detection of web application vulnerabilities. In *Proceedings of the 2006 workshop on Programming languages and analysis for security* (pp. 27-36). ACM.

[62] Jovanovic, N., Kruegel, C., & Kirda, E. (2006). Pixy: A static analysis tool for detecting web application vulnerabilities. In *Security and Privacy, 2006 IEEE Symposium on* (pp. 6-pp). IEEE.

[63] Kemalis, K., & Tzouramanis, T. (2008). SQL-IDS: a specification-based approach for SQL-injection detection. In *Proceedings of the 2008 ACM symposium on Applied computing* (pp. 2153-2158). ACM.

[64] Khan, S. A., & Khan, R. A. (2013). Software security testing process: phased approach. In *Intelligent Interactive Technologies and Multimedia* (pp. 211-217). Springer Berlin Heidelberg.

[65] Khan, M. S., & Mahapatra, S. S. (2008). Service quality evaluation in internet banking: an empirical study in India. *International Journal of Indian Culture and Business Management*, *2*(1), 30-46.

[66] Khari, M., & Kumar, N. (2013). User Authentication Method against SQL Injection Attack.

[67] Khoury, N., Zavarsky, P., Lindskog, D., & Ruhl, R. (2011). Testing and assessing web vulnerability scanners for persistent SQL injection attacks. In *Proceedings of the First International Workshop on Security and Privacy Preserving in e-Societies* (pp. 12-18). ACM.

[68] Kindy, D. A., & Pathan, A. S. K. (2011). A survey on SQL injection: Vulnerabilities, attacks, and prevention techniques. In *Consumer Electronics (ISCE), 2011 IEEE 15th International Symposium on* (pp. 468-471). IEEE.

[69] Kindy, D. A., & Pathan, A. S. K. (2012). A detailed survey on various aspects of sql injection in web applications: Vulnerabilities, innovative attacks, and remedies. *arXiv preprint arXiv:1203.3324*.

[70] Kieyzun, A., Guo, P. J., Jayaraman, K., & Ernst, M. D. (2009). Automatic creation of SQL injection and cross-site scripting attacks. In *Software Engineering, 2009. ICSE 2009. IEEE 31st International Conference on* (pp. 199-209). IEEE.

[71] Kosuga, Y., Kono, K., Hanaoka, M., Hishiyama, M., & Takahama, Y. (2007). Sania: Syntactic and semantic analysis for automated testing against sql injection. In *Computer Security Applications Conference, 2007. ACSAC 2007. Twenty-Third Annual* (pp. 107-117). IEEE.

[72] Krügel, C., Toth, T., & Kirda, E. (2002). Service specific anomaly detection for network intrusion detection. In *Proceedings of the 2002 ACM symposium on Applied computing* (pp. 201-208). ACM

[73] Kubo, T., Obuchi, M., Ohashi, G., & Shimodaira, Y. (1998). Image processing system for direction detection of an object using neural network. In *Circuits and Systems, 1998. IEEE APCCAS 1998. The 1998 IEEE Asia-Pacific Conference on* (pp. 571-574). IEEE.

[74] Kumar, P., & Pateriya, R. K. (2012). A survey on SQL injection attacks, detection and prevention techniques. In *Computing Communication & Networking Technologies (ICCCNT), 2012 Third International Conference on* (pp. 1-5). IEEE.

[75] Lawal, M. A., Sultan, A. B. M., & Shakiru, A. O. (2016). Systematic literature review on SQL injection attack. *International Journal of Soft Computing*, *11*(1), 26-35.

[76] Lebeau, F., Legeard, B., Peureux, F., & Vernotte, A. (2013). Model-based vulnerability testing for web applications. In *Software Testing, Verification and Validation Workshops (ICSTW), 2013 IEEE Sixth International Conference on* (pp. 445-452). IEEE.

[77] Lee, S. Y., Low, W. L., & Wong, P. Y. (2002). Learning fingerprints for a database intrusion detection system. In *European Symposium on Research in Computer Security* (pp. 264-279). Springer Berlin Heidelberg.

[78] Liu, A., Yuan, Y., Wijesekera, D., & Stavrou, A. (2009). SQLProb: a proxy-based architecture towards preventing SQL injection attacks. In *Proceedings of the 2009 ACM symposium on Applied Computing* (pp. 2054-2061). ACM.

[79] Livshits, Benjamin, Michael Martin, and Monica S. Lam. (2006) "SecuriFly: Runtime protection and recovery from Web application vulnerabilities."

[80] Livshits, V. B., & Lam, M. S. (2005). Finding Security Vulnerabilities in Java Applications with Static Analysis. In *Usenix Security* (Vol. 2013).

[81] Li, X., & Xue, Y. (2014). A survey on server-side approaches to securing web applications. *ACM Computing Surveys (CSUR)*, *46*(4), 54.

[82] Li, X., & Xue, Y. (2011). BLOCK: a black-box approach for detection of state violation attacks towards web applications. In *Proceedings of the 27th Annual Computer Security Applications Conference* (pp. 247-256). ACM.

[83] Maheswari, K. G., & Anita, R. (2016). An Intelligent Detection System for SQL Attacks on Web IDS in a Real-Time Application. In *Proceedings of the 3rd International Symposium on Big Data and Cloud Computing Challenges (ISBCC–16')* (pp. 93-99). Springer International Publishing.

[84] Manmadhan, S., & Manesh, T. (2012). A method of detecting sql injection attack to secure web applications. *International Journal of Distributed and Parallel Systems*, *3*(6),

[85] Maor, O., & Shulman, A. (2004). Blind SQL Injection. *Imperva.(Online) http://www. imperva. com/resources/adc/blind_sql_server_injection. html*.

[86] Martin, M., & Lam, M. S. (2008). Automatic generation of XSS and SQL injection attacks with goal-directed model checking. In *Proceedings of the 17th conference on Security symposium* (pp. 31-43). USENIX Association

[87] Meucci, M., & Muller, A. (2014). The OWASP Testing Guide 4.0. *Open Web Application Security Project*, 30.

[88] McClure, R. A., & Kruger, I. H. (2005). SQL DOM: compile time checking of dynamic SQL statements. In *Software Engineering, 2005. ICSE 2005. Proceedings. 27th International Conference on* (pp. 88-96). IEEE.

[89] Mohosina, A., & Zulkernine, M. (2012). DESERVE: a framework for detecting program security vulnerability exploitations. In *Software Security and Reliability (SERE), 2012 IEEE Sixth International Conference on* (pp. 98-107). IEEE.

[90] Moradpoor, N. (2014). Employing Neural Networks for the detection of SQL injection attack. In *Proceedings of the 7th International Conference on Security of Information and Networks, September 9-11, Glasgow, UK*. ACM

[91] Moosa, A. (2010). Artificial neural network based web application firewall for sql injection. *World Academy of Science, Engineering & Technology*, *64*(4), 12-21

[92] Mui, R., & Frankl, P. (2010). Preventing SQL injection through automatic query sanitization with ASSIST. *arXiv preprint arXiv:1009.3712*.

[93] Mukkamala, S., Janoski, G., & Sung, A. (2002). Intrusion detection using neural networks and support vector machines. In *Neural Networks, 2002. IJCNN'02. Proceedings of the 2002 International Joint Conference on* (Vol. 2, pp. 1702-1707). IEEE

[94] Muthuprasanna, M., Wei, K., & Kothari, S. (2006). Eliminating SQL injection attacks-A transparent defense mechanism. In *Web Site Evolution, 2006. WSE'06. Eighth IEEE International Symposium on* (pp. 22-32). IEEE

[95] Narayanan, S. N., Pais, A. R., & Mohandas, R. (2011). Detection and Prevention of SQL Injection Attacks using Semantic Equivalence. In *Computer Networks and Intelligent Computing* (pp. 103-112). Springer Berlin Heidelberg.

[96] Nguyen-Tuong, A., Guarnieri, S., Greene, D., Shirley, J., & Evans, D. (2005). Automatically hardening web applications using precise tainting. *Security and Privacy in the Age of Ubiquitous Computing*, 295-307.

[97] Ntagwabira, L., & Kang, S. L. (2010). Use of Query Tokenization to detect and prevent SQL Injection Attacks. In *Computer Science and Information Technology (ICCSIT), 2010 3rd IEEE International Conference on* (Vol. 2, pp. 438-440). IEEE.

[98] Panda, S., & Ramani, S. (2013). Protection of web application against SQL injection attacks. *International Journal of Modern Engineering Research (IJMER) vol*, *3*

[99] Papagiannis, I., Migliavacca, M., & Pietzuch, P. (2011). PHP Aspis: using partial taint tracking to protect against injection attacks. In *2nd USENIX Conference on Web Application Development* (p. 13).

[100] Parker, D. B. (2002). Toward a New Framework for Information Security? *Computer Security Handbook, Sixth Edition*, 3-1.

[101] Palsetia, N., Deepa, G., Khan, F. A., Thilagam, P. S., & Pais, A. R. (2016). Securing native XML database-driven web applications from XQuery injection vulnerabilities. *Journal of Systems and Software*, *122*, 93-109.

[102] Pietraszek, T., & Berghe, C. V. (2005). Defending against injection attacks through context-sensitive string evaluation. In *International Workshop on Recent Advances in Intrusion Detection* (pp. 124-145). Springer Berlin Heidelberg

[103] Prabakar, M. A., Karthikeyan, M., & Marimuthu, K. (2013). An efficient technique for preventing SQL injection attack using pattern matching algorithm. In *Emerging Trends in Computing, Communication and Nanotechnology (ICE-CCN), 2013 International Conference on* (pp. 503-506). IEEE.

[104] Rawat, R., & Raghuwanshi, S. (2012). SQL injection attack Detection using SVM. *International Journal of Computer Applications*, *42*(13), 1-4

[105] Ruse, M., Sarkar, T., & Basu, S. (2010). Analysis & detection of SQL injection vulnerabilities via automatic test case generation of programs. In *Applications and the Internet (SAINT), 2010 10th IEEE/IPSJ International Symposium on* (pp. 31-37). IEEE.

[106] Sadeghian, A., Zamani, M., & Ibrahim, S. (2013). SQL injection is still alive: a study on SQL injection signature evasion techniques. In *Informatics and Creative Multimedia (ICICM), 2013 International Conference on* (pp. 265-268). IEEE.

[107] Sadeghian, A., Zamani, M., & Manaf, A. A. (2013). A taxonomy of SQL injection detection and prevention techniques. In *Informatics and Creative Multimedia (ICICM), 2013 International Conference on* (pp. 53-56). IEEE.

[108] Sangkatsanee, P., Wattanapongsakorn, N., & Charnsripinyo, C. (2011). Practical real-time intrusion detection using machine learning approaches. *Computer Communications*, *34*(18), 2227-2235

[109] Sahu, D. R., & Tomar, D. S. (2016). Analysis of Web Application Code Vulnerabilities using Secure Coding Standards. *Arabian Journal for Science and Engineering*, 1-11.

[110] Seacord, R. C. (2008). *The CERT C secure coding standard*. Pearson Education.

[111] Securosis: (2014) Open source development and application security survey analysis.

[112] Shahriar, H., & Zulkernine, M. (2012). Information-theoretic detection of sql injection attacks. In *High-Assurance Systems Engineering (HASE), 2012 IEEE 14th International Symposium on* (pp. 40-47). IEEE

[113] Shanmughaneethi, S. V., Shyni, S. C. E., & Swamynathan, S. (2009). SBSQLID: Securing web applications with service based SQL injection detection. In *Advances in Computing, Control, & Telecommunication Technologies, 2009. ACT'09. International Conference on* (pp. 702-704). IEEE.

[114] Shaukat Ali, Azhar Rauf, Huma Javed (2009), SQLIPA: An Authentication Mechanism against SQL Injection‟, *European Journal of Scientific Research*, Vol.38 No.4, pp.604-611.

[115] Sheykhkanloo, N. M. (2014). Employing neural networks for the detection of sql injection attack. In *Proceedings of the 7th International Conference on Security of Information and Networks* (p. 318). ACM.

[116] Shi, C. C., Zhang, T., Yu, Y., & Lin, W. (2012). A new approach for SQL-injection detection. *Instrumentation, Measurement, Circuits and Systems*, 245-254

[117] Shrivastava, R., Bhattacharyji, J., & Soni, R. (2013). Sql Injection Attacks In Database Using Web Service: Detection And Prevention–Review. *ASIAN JOURNAL OF COMPUTER SCIENCE & INFORMATION TECHNOLOGY*, *2*(6).

[118] Singhal, D., & Padhmanabhan, V. (2009). A study on customer perception towards internet banking: Identifying major contributing factors. *Journal of Nepalese business studies*, *5*(1), 101-111.

[119] Skrupsky, N., Bisht, P., Hinrichs, T., Venkatakrishnan, V. N., & Zuck, L. (2013). TamperProof: a server-agnostic defense for parameter tampering attacks on web applications. In *Proceedings of the third ACM conference on Data and application security and privacy* (pp. 129-140). ACM.

[120] Su, Z., & Wassermann, G. (2006). The essence of command injection attacks in web applications. In *ACM SIGPLAN Notices* (Vol. 41, No. 1, pp. 372-382). ACM.

[121] Srivastav, A., Kumar, P., & Goel, R. (2013). Evaluation of Network Intrusion Detection System using PCA and NBA. *International Journal of Advanced Research in Computer Engineering & Technology*, *2*(11), 2873-2881.

[122] Stuttard, D., & Pinto, M. (2011). *The Web Application Hacker's Handbook: Finding and Exploiting Security Flaws*. John Wiley & Sons

[123] Stolcke, A., & Omohundro, S. (1993). Hidden Markov model induction by Bayesian model merging. *Advances in neural information processing systems*, 11-11.

[124] Tajpour, A., Heydari, M. Z., Masrom, M., & Ibrahim, S. (2010). SQL injection detection and prevention tools assessment. In *Computer Science and Information Technology (ICCSIT), 2010 3rd IEEE International Conference on* (Vol. 9, pp. 518-522). IEEE.

[125] Valeur, F., Mutz, D., & Vigna, G. (2005). A learning-based approach to the detection of SQL attacks. In *International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment* (pp. 123-140). Springer Berlin Heidelberg.

[126] Vigna, G., Valeur, F., Balzarotti, D., Robertson, W., Kruegel, C., & Kirda, E. (2009). Reducing errors in the anomaly-based detection of web-based attacks through the combined analysis of web requests and SQL queries. *Journal of Computer Security*, *17*(3), 305-329.

[127] Vulnerabilities, C. (2007). Exposures,“The Standard for Information Security Vulnerability Names”. *Common Vulnerabilities and Exposures: The Standard for Information Security Vulnerability Names. url: http://cve. mitre. org*.

[128] Wagner, D., & Soto, P. (2002). Mimicry attacks on host-based intrusion detection systems. In *Proceedings of the 9th ACM Conference on Computer and Communications Security* (pp. 255-264). ACM

[129] Wang, Y., Wong, J., & Miner, A. (2004). Anomaly intrusion detection using one class SVM. In *Information Assurance Workshop, 2004. Proceedings from the Fifth Annual IEEE SMC* (pp. 358-364). IEEE.

[130] Web Security, Inc. The Weakest Link: Mitigating Web Application Vulnerabilities. (2007) Retrieved fromhttp://www.webscurity.com /pdfs/webapp_vuln_wp.pdf.

[131] Web Application Security Consortium. (2010). *WASC threat classification*. Technical report, Jan. 2010. Version 2.00.

[132] Web Application Security Scanner Evaluation Criteria. (2009).

[133] Wei, K., Muthuprasanna, M., & Kothari, S. (2006). Preventing SQL injection attacks in stored procedures. In *Software Engineering Conference, 2006. Australian* (pp. 8-pp). IEEE.

[134] Weinberger, J., Saxena, P., Akhawe, D., Finifter, M., Shin, R., & Song, D. (2011). A systematic analysis of XSS sanitization in web application frameworks. In *European Symposium on Research in Computer Security* (pp. 150-171). Springer Berlin Heidelberg.

[135] Win, W. Y., & Htun, H. H. (2014). A Detection Method for SQL Injection Attacks in Web Applications.

[136] Xie, Y., & Aiken, A. (2006). Static Detection of Security Vulnerabilities in Scripting Languages. In *USENIX Security* (Vol. 6, pp. 179-192).

[137] Zhang, Q., Jones, C., & Agarwal, A. K. (2008). *U.S. Patent Application No. 12/037,211*.

[138] Zhu, J., Xie, J., Lipford, H. R., & Chu, B. (2014). Supporting secure programming in web applications through interactive static analysis. *Journal of advanced research*, *5*(4), 449-462.

[139] Zhang, L., Gu, Q., Peng, S., Chen, X., Zhao, H., & Chen, D. (2010). D-WAV: A web application vulnerabilities detection tool using Characteristics of Web Forms. In *Software Engineering Advances (ICSEA), 2010 Fifth International Conference on* (pp. 501-507). IEEE.

[140] Zhang, K. X., Lin, C. J., Chen, S. J., Hwang, Y., Huang, H. L., & Hsu, F. H. (2011). TransSQL: a translation and validation-based solution for SQL-injection attacks. In *Robot, Vision and Signal Processing (RVSP), 2011 First International Conference on* (pp. 248-251). IEEE.

[141] George, T. K., Jacob, K. P., & James, R. K. (2018). Token based Detection and Neural Network based Reconstruction framework against code injection vulnerabilities. Journal of Information Security and Applications, 41, 75-91.

[142] Uwagbole, S. O., Buchanan, W. J., & Fan, L. (2017, May). Applied machine learning predictive analytics to SQL injection attack detection and prevention. In *Integrated Network and Service Management (IM), 2017 IFIP/IEEE Symposium on*(pp. 1087-1090). IEEE.

[143] Medeiros, I., Neves, N., & Correia, M. (2016). Detecting and removing web application vulnerabilities with static analysis and data mining. *IEEE Transactions on Reliability*, *65*(1), 54-69.

[144] Sheykhkanloo, N. M. (2015). 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*, *9*(6), 1443-1453.

Appendices

APPENDIX-1

Commonly used vulnerability analysis/scanning tools and its descriptions

Tool	Description
IBM Rational AppScan:	It is a commercial tool to assess the vulnerability of a website, especially the SQL injection assessment functionality.
HPScrawlr	It is a free tool/scanner by HP, able to detect regular and blind SQL Injection vulnerabilities
SQLiX	It is a Perl application code able to crawl website and detect injection.
Paros Proxy:	It is a web assessment tool used for manually manipulating web traffic. It is placed as a proxy and checks the request made from the web browser before directing it to the server.
SWAAT	An application analysis tool, to scan source code, it uses regular expression string matching to identify potentially dangerous functions and strings in the code base
The Microsoft Source Code Analyzer	It is for SQL Injection tool is a static code analysis tool to find SQL injection vulnerabilities in Active Server Pages (ASP) code.
CAT.NET	It is a binary code analysis tool that helps you identify common variants of certainprevailing vulnerabilities
Commercial Source Code Analyzer (SCAs)	Commercial Source Code Analyzers (SCAs) are designed to integrate within the development life cycle of an application.
Ounce	The Ounce toolset is a collection of several components. The Security Analyst component Parses source code into what it calls a Common Intermediate Security Language (CISL).

Fortify Source Code Analyzer	It is a Source Code Analyzer is a static analysis tool that processes code and attempts to identify vulnerabilities
Code Secure	Code Secure is available as an enterprise-level appliance or as a hosted software service
OpenVAS:	One of the most advanced vulnerability scanner and general vulnerability assessment tool, scans more than 35,000 vulnerabilities.
Nexpose Community:	Developed by Rapid7, scans almost 68,000 vulnerabilities and makes over 163,000 network checks. It includes automatic vulnerability updates.
Metasploit Framework	A penetration testing framework used to validate vulnerabilities found by Nexpose for patching and mitigation.
Retina CS Community	A web based console that simplify and centralize the vulnerability Management. It includes automated vulnerability assessment for servers, workstations, database web applications. It supports VMware environment
Burp Suite Free Edition:	A software tool kit to carry out security testing of web application.
Nikto	A web server scanner performs comprehensive tests against webservers for multiple (more than 6,700 potentially dangerous files/programs) items. It also checks for server configuration items.
Zed attack proxy Clair	An integrated tool to find out web vulnerabilities, a proxy tool. A specialized container vulnerability analysis service, it extracts all required data to detect known vulnerabilities
Moloch	A large scale IPv4 packet capturing, indexing and database system, it works along with IDS. An analysis tool to handle multiple gigabits/sec of traffic.
PowerFuzzer:	A highly automated open source tool capable of identifying XSS, SQL, LDAP, XPATH and HTTP 500 statuses.

APPENDIX-2

Check lists of functions used:

Type of Functions	Sample functions
Numeric Functions	ABS,ACOS,ASIN,EXP,LOG,MOD,POWER,ROUND, SQRT
Character Functions Returning Character Values	CHR, CONTACT,LOWER,NLS-INITCAP,NLS_UPPER REGEXP-REPLACE, REGEXP_SUBSTR, REPLCE, TRIM
NLS Character Functions	NLS-CHARSET_DECL-LEN,NLS_CHARSET_ID,NLS_CHAESET_NAME
Datetime Functions	ADD_MONTHS,CURRENT_DATE,CURRENT_TIMESTAMP,DBTIMEZONE,EXTRACT,FROM-TZ, LAST_DAY, EW_TIME,LAST_DAY, NUMTODSINTERVAL,ROUND.
General Comparison Functions	GREATEST, LEAST
Conversion Functions	ASCIISTR,BIN_TO-NUM,CAST,COMPOSE,CONVERT, DECOMPOSE,RAWTOHEX,NUMTODINTERVAL, ROWIDTONHEX, ROWIDTONCHAR,TO-BINARY_FLOAT, TO_YMINTERVAL UNISTR.
Large Object Functions	BFILENAME, EMPTY_BLOB,EMPTY_CLOB
Collection Functions	CARDINALITY, COLLECT, POWERMULTISET, CARDINALITY, SET.
Encoding and Decoding Functions	DECODE, DUMP,ORA-HARSH,VSIZE
NULL-Related Functions	COALESCE,LENVL,NULLIF,NVL,NVL2
Aggregate Functions	AVG,COLLECT,CORR,COUNT,COVAR-POP,FIRST,GROUP_ID,GROUP_ID,LAST,MAX,MEDIAN, MIN,PERCENTILE_CONT
Analytic Functions	CORR,COUNT,LAG,LAST,LEAD,PERCENT_RANK,ROW_NUMBER

APPENDIX-3

Appendix - 3 Screen shots of Prototype MLT-DR

(i) Simple query template format

JSP Page
localhost:8080/SQLIADetectionPrevention/splitStandardIndependent.jsp#
Query 1 | SubQuery 1 | SubQuery 2 | SubQuery 3 | SubQuery 4

QUERY TEMPLATE

Query Format: select playerid from test

Query-Type: select
Table(s): tes
Column(s): 1
Column1: playerid
System Variables:
Global Variables:
Functions Used:
Joins Used:
Special Symbols:
Operators Used:
Comment Symbols:

SAVE

(ii) Complex query template format

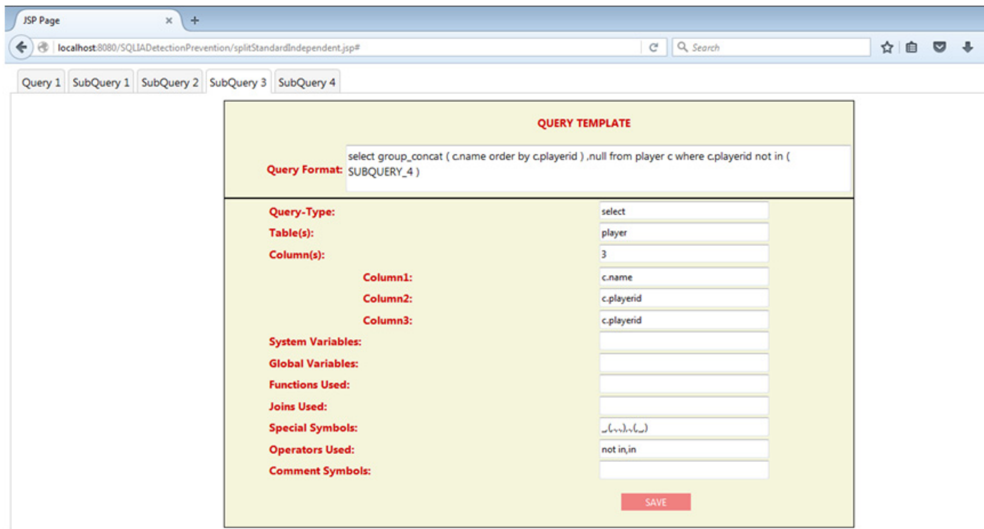
JSP Page
localhost:8080/SQLIADetectionPrevention/splitStandardIndependent.jsp#
Query 1 | SubQuery 1 | SubQuery 2 | SubQuery 3 | SubQuery 4

QUERY TEMPLATE

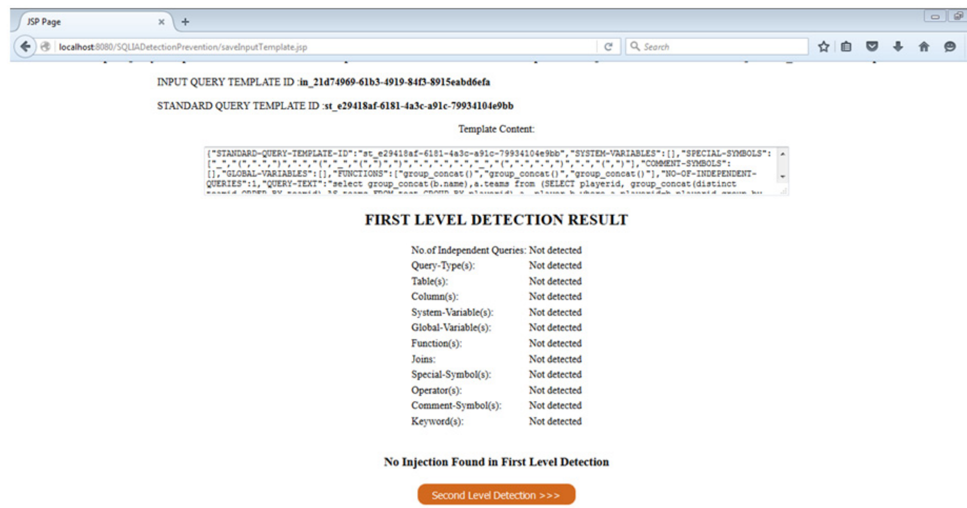
Query Format: select group_concat (b.name) ,a.teams from (SUBQUERY_2) a, player b where a.playerid=b.playerid group by a.teams union SUBQUERY_3 ;

Query-Type: select
Table(s): subquery_2,player
Column(s): 5
Column1: b.name
Column2: a.teams
Column3: a.playerid
Column4: b.playerid
Column5: a.teams
System Variables:
Global Variables:
Functions Used:
Joins Used:
Special Symbols: _(.,).,(_).,.,.,_
Operators Used: =,union
Comment Symbols:

SAVE



JSP Page
localhost:8080/SQLIADetectionPrevention/splitStandardIndependent.jsp#
Search
Query 1
SubQuery 1
SubQuery 2
SubQuery 3
SubQuery 4
QUERY TEMPLATE
Query Format:
select group_concat (c.name order by c.playerid) ,null from player c where c.playerid not in (SUBQUERY_4)
Query-Type:
select
Table(s):
player
Column(s):
3
Column1:
c.name
Column2:
c.playerid
Column3:
c.playerid
System Variables:
Global Variables:
Functions Used:
Joins Used:
Special Symbols:
Operators Used:
not in,in
Comment Symbols:
SAVE



JSP Page
localhost:8080/SQLIADetectionPrevention/saveInputTemplate.jsp
Search
INPUT QUERY TEMPLATE ID :in_21d74969-61b3-4919-84f3-8915eabd6efa
STANDARD QUERY TEMPLATE ID :st_e29418af-6181-4a3c-a91c-79934104e9bb
Template Content:
FIRST LEVEL DETECTION RESULT
No.of Independent Queries: Not detected
Query-Type(s): Not detected
Table(s): Not detected
Column(s): Not detected
System-Variable(s): Not detected
Global-Variable(s): Not detected
Function(s): Not detected
Joins: Not detected
Special-Symbol(s): Not detected
Operator(s): Not detected
Comment-Symbol(s): Not detected
Keyword(s): Not detected
No Injection Found in First Level Detection
Second Level Detection >>>

JSP Page

localhost:8080/SQLIADetectionPrevention/secondLevelDetectionResult.jsp

(",".",".",")",".","(","_",")"],"COMMENT-SYMBOLS":[],"GLOBAL-VARIABLES":[],"FUNCTIONS":[],"QUERY-TEXT":"select group_concat (c.name order by c.playerid) ,null from player c where c.playerid not in (SUBQUERY_4)","USED-TABLES":["player"],"NO-OF-SUBQUERIES":-1,"QUERY-TYPE":["select"],"COLUMNS":["c.name","c.playerid","c.playerid"],"OPERATORS":["not in","in"],"JOINS":[],"KEYWORDS":["order by","from","where"]},"QUERY-TYPE":["select","select","select","select"],"SUB-QUERY-2":{"SYSTEM-VARIABLES":[],"SPECIAL-SYMBOLS":["_","(",")"],"COMMENT-SYMBOLS":[],"GLOBAL-VARIABLES":[],"FUNCTIONS":[],"QUERY-TEXT":"select playerid, group_concat (distinct teamid order by teamid) as teams from test group by playerid","USED-TABLES":["test"],"NO-OF-SUBQUERIES":-1,"QUERY-TYPE":["select"],"COLUMNS":["playerid","teamid","teamid","playerid"],"OPERATORS":[],"JOINS":[],"KEYWORDS":["distinct","order by","as","from","group by"]},"COLUMNS":["b.name","a.teams","playerid","teamid","teamid","playerid","player","a.playerid","b.playerid","a.teams","c.name","c.playerid","c.playerid","playerid"],"SUB-QUERY-4":{"SYSTEM-VARIABLES":[],"SPECIAL-SYMBOLS":[],"COMMENT-SYMBOLS":[],"GLOBAL-VARIABLES":[],"FUNCTIONS":[],"QUERY-TEXT":"select playerid from test","USED-

SECOND LEVEL DETECTION RESULT

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

No Injection Found after Second Level Detection.

<< Back to Home

SQLIADetectionPrevention - NetBeans IDE 8.1

File Edit View Navigate Source Refactor Run Debug Profile Team Tools Window Help

Start Page

Files | Projects | detectAction.jsp | saveInputTemplate.jsp | QueryExtractorClass.java | SubqueryExtractorClass.java | SecondLevelDetecorClass.java | SqlFunctionsList.txt

Inputs
- in_019e1ea5-dd4a-4f24-a35f-d3067d54cd4a.txt
- in_05741339-800f-46d3-966c-b8cad3054795.txt
- in_084ef5ae-935f-4113-85ee-2fbf23b596e3.txt
- in_0d8ac5c1-8c9a-4be8-9911-d173588f8010.txt
- in_145ec0b5-dac6-42bf-af55-5f293be68e43.txt
- in_1464e488-ca6c-49df-ae8c-92cfa58d87fd.txt
- in_183622bb-fa4f-4443-8d11-8f88aa5d6352.txt
- in_1b6deda5-dd3e-4822-b293-e061b232ca18.txt
- in_1d9f1446-b8aa-4c6b-a32b-f17770135 7af.txt
- in_1da3ceb1-bcb6-476d-9b4b-1f4f0daaad52.txt
- in_218cc22f-0163-45e4-86f3-755e4315197b.txt
- in_250e363e-d008-4f29-aa44-7ffa529193f3.txt
- in_25eda7be-4952-414a-9892-6518e2544e38.txt
- in_2acc7dac-6020-4d4d-9dbe-6d7283089a49.txt
- in_2ce0a1b8-fb90-4b42-9049-135751a05381.txt
- in_2ea0f3a8-94a2-4e03-ac4d-df7704c7dc41.txt
- in_339a6a7f-ee7a-4096-942e-1374d0981d03.txt
- in_36621f40-2cc8-4059-94a0-02225d8f00ce.txt
- in_3672e0bb-819d-4aa0-ad1f-638efab59e29.txt
- in_384af09f-326b-4798-b40e-d097b4861c9b.txt
- in_3cd4469f-5f8f-437c-bb26-de089d1e5e99.txt
- in_3fdfd748-bf48-43f3-bd96-8fdc0cefb808.txt
- in_449bdb50-72b5-4ba4-85a2-9805c8a8eb08.txt
- in_45b0e296-966a-4468-b41c-0439c955a566.txt
- in_4af43005-16b4-4c45-8cb4-e33698707a0c.txt
- in_511ddcd8-c9df-4b43-bfc9-c5515a341041.txt
- in_514309d5-08d7-45fe-9877-0a5de504dec3.txt
- in_5391cab6-a1e5-4b82-9868-37c8e3bd96b9.txt
- in_577561a0-a857-4e74-b505-9de649003b11.txt

Standard Query Template Created. Please refer JSON file in the path... "\\SQLIADetectionPrevention\\SQLIADP_DATA\\data\\Templates"

STANDARD QUERY TEMPLATE ID :**st_02fbec31-2361-48d2-871d-2d424056626f**

Template Content:

{"SYSTEM-VARIABLES":[],"SPECIAL-SYMBOLS":["(",")","_"],"COMMENT-SYMBOLS":[],"GLOBAL-VARIABLES":[],"REGULAR-EXPRESSIONS":
[],"FUNCTIONS":["count()"],"NO-OF-INDEPENDENT-QUERIES":1,"QUERY-TEXT":"   SELECT COUNT(*) FROM reviews WHERE
review_author='MadBob';","USED-TABLES":["reviews"],"QUERY-TYPE":["select"],"COLUMNS":
["*","review_author"],"ID":"st_02fbec31-2361-48d2-871d-2d424056626f","OPERATORS":["*","="],"JOINS":[],"KEYWORDS":

<< Back to Home

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

INPUT QUERY TEMPLATE ID :**in_7c11f14d-e4b3-4bcb-9b07-3561a55ef917**

STANDARD QUERY TEMPLATE ID :**st_0a20e1d6-540e-48c4-be6d-8ff223447d08**

Template Content:

{"STANDARD-QUERY-TEMPLATE-ID":"st_0a20e1d6-540e-48c4-be6d-8ff223447d08","SYSTEM-VARIABLES":[],"SPECIAL-SYMBOLS":
["_","_","_","Ã¢Â\u0080Â\u0099","Ã¢Â\u0080Â\u0099","Ã¢Â\u0080Â\u0099","Ã¢Â\u0080Â\u0099"],"COMMENT-SYMBOLS":[],"GLOBAL-
VARIABLES":[],"FUNCTIONS":["password","password"],"NO-OF-INDEPENDENT-QUERIES":1,"QUERY-TEXT":"SELECT
email_id,password,full_name from Login_table\r\n   WHERE email=â\u0080\u0099xâ\u0080\u0099 AND password=â\u0080

FIRST LEVEL DETECTION RESULT

No.of Independent Queries:	Not detected
Query-Type(s):	Detected
Table(s):	Detected
Column(s):	Detected
System-Variable(s):	Not detected
Global-Variable(s):	Not detected
Function(s):	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

<< Back to Home

List of Publications

[1] **Teresa K. George,** K. Poulose Jacob, Rekha K. James**,** *Proposed Hybrid Model to Detect and Prevent SQL Injection***,** International Journal of Computer Science and Information Security (IJCSIS), Vol. 14 No.6, 2016.

[2] **Teresa K. George,** K. Poulose Jacob, *A Proposed Architecture for Query Anomaly Detection and Prevention against SQL Injection Attacks,* International Journal of Computer Applications, Vol. 137 No.7, 2016.

[3] **Teresa K. George,** K. Poulose Jacob, Rekha K. James, *SQLI-Dagger, a Multilevel Template based Algorithm to Detect and Prevent SQL, Injection,* International Journal of Computer Applications Vol. 143, No.6, 2016.

[4] **Teresa K. George,** K. Poulose Jacob, *Fraud detection and mitigation in secure e-payment transaction,* International Journal of Scientific & Engineering Research, IJSER, Vol. 6, Issue 2, 2015.

[5] **Teresa K. George,** K. Poulose Jacob *Analysis on Security Vulnerability in Online Application, Data Validation strategies and Testing Tool*, International Journal of Advanced Research in Data Mining and Cloud Computing, Vol. 3, Issue 3, 2015.

[6] **Teresa K. George,** E. Ben George, N. Balasubrahmanyam *An Innovative Fraud Detection Framework for Secure E-Transactions using Big Data Analytics,* European Journal of Scientific Research Published in Vol.122. No.3, 2014.

[7] **Teresa K. George**, K. Poulose Jacob, Risk *and Vulnerability analysis of E-transactions in the Banking Industry with a specific reference on the common malware type of attack,* International Journal of Computer Science and Information Security, IJCSIS, Vol. 12, No. 6, 2014.

[8] **Teresa K. George,** K. Poulose Jacob, *Security Patches Against the Loopholes in Internet Banking,* International Journal of Engineering Research & Technology (IJERT) Vol. 3, Issue 5, 2014.

[9] **Teresa K. George,** K. Poulose Jacob, *Infrastructure and Security Concerns on Internet Banking in India*, International Journal of Advanced Research in Computer Science and Software Engineering (IJARCSSE) Vol. 4, 2014.

[10] **Teresa K. George,** K. Poulose Jacob, Rekha K. James, *Token based Detection and Neural Network based Reconstruction framework against code injection vulnerabilities*. Journal of Information Security and Applications (JISA) Vol. 41, 75-91, 2018.

.....ꙮ.....