

Runtime Detection and Prevention for Structure Query Language Injection Attacks

PhD Thesis

Emad Shafie

Software Technology Research Laboratory

Faculty of Technology

De Montfort University

England

This thesis is submitted in partial fulfilment of the requirements for the

Doctor of Philosophy.

May, 2013

Abstract

The use of Internet services and web applications has grown rapidly because of user demand. At the same time, the number of web application vulnerabilities has increased as a result of mistakes in the development where some developers gave the security aspect a lower priority than aspects like application usability. An SQL (structure query language) injection is a common vulnerability in web applications as it allows the hacker or illegal user to have access to the web application's database and therefore damage the data, or change the information held in the database. This thesis proposes a new framework for the detection and prevention of new and common types of SQL injection attacks.

The programme of research is divided in several work packages that start from addressing the problem of the web application in general and SQL injection in particular and discuss existing approaches. The other work packages follow a constructive research approach. The framework considers existing and new SQL injection attacks. The framework consists of three checking components; the first component will check the user input for existing attacks, the second component will check for new types of attacks, and the last component will block unexpected responses from the database engine.

Additionally, our framework will keep track of an ongoing attack by recording and investigating user behaviour. The framework is based on the Anatempura tool, a runtime verification tool for Interval Temporal Logic properties. Existing attacks and good/bad user behaviours are specified using Interval Temporal Logic, and the detection of new SQL injection attacks is done using the database observer component. Moreover, this thesis discusses a case study where various types of user behaviour are specified in Interval Temporal Logic and show how these can be detected.

The implementation of each component has been provided and explained in detail showing the input, the output and the process of each component. Finally, the functionality of each checking component is evaluated using a case study. The user behaviour component is evaluated using sample attacks and normal user inputs. This thesis is summarized at the conclusion chapter, the future work and the limitations will be discussed.

This research has made the following contributions:

- New framework for detection and prevention of SQL injection attacks.
- Runtime detection: use runtime verification technique based on Interval Temporal logic to detect various types of SQL injection attacks.
- Database observer: to detect possible new injection attacks by monitoring database transactions.
- User's behaviour: investigates related SQL injection attacks using user input, and providing early warning against SQL injection attacks.

Acknowledgment

I start my thanks to God who supported me to carry out this work, there were many difficulties and stresses at times but he opened new gates that resolved my mind and personality to look back afresh at my studies again. The encouragement, pushing, and support provided by Dr. Antonio Cau who is my supervisor. I really owe him for his help. We started together doing this research and he guided and supported me at every step of this research. This research under his supervision was interesting and without his support and patience it would have been difficult to achieve. Many thanks for the STRL staff, especially for Prof. Hussein Zedan who is the STRL director and who created this helpful study environment for me and other PhD students. I also express my thanks to Dr. Helge Janicke and Dr. Francois Siewe for their advice and discussions that were helpful. Furthermore, I would like to express my thanks to Dr. Turki Alsommani and Dr. Yaser Alasaleh whose support encouraged me to carry on with this study. I will never forget to express my appreciation to all my STRL friends, even those who have graduated or who are still carrying on in their study especially, Dr. Ali Alzahrani , Dr. Adeeb Alnajjar, Dr. Hani AlQuhaiz, Dr. Meshrif Alruily and Dr Sulaiman Alamru. Our friendship started here at DMU and hopefully will continue forever.

Many thanks also to Lindsey Trent and her mother, Mrs Lynn Ryan, for their friendly help to all STRL students.

Many thanks to the DMU library staff for their cooperation and continuing smiles.

Dedication

To my loving parents

I dedicate this work to my great father, Mr. Abdulrahman Shafie, who has been a permanent source of motivation and endless support throughout my life, and who tried and worked a lot for me to be what I am now. I will never forget my loving mother who gave and still gives me her love, kindness, tenderness and supports me for everything and everywhere in my life.

To my beloved family

I also dedicate this work to my loving wife who has looked after me and my children during this period and was so patient despite me being a long time away from home. This work is also dedicated to my children Abdulrahman, Adnan, Layan, and Lana. I will also never forget my brothers and sisters and this work is dedicated to them as well. I hope that by obtaining this degree I can rejoice with them and add some small pleasure to their life and that I can put a little smile on their faces.

Declaration

I, Emad Shafie, declare that this thesis titled ‘Runtime Detection and Prevention for SQL injection attacks, and its contents are my own and original work and it is submitted for the degree of Doctor of Philosophy, at the Software Technology Research Laboratory (STRL), De Montfort University. This work has never been submitted before in any other university. The work was undertaken by me between October 2009 and May 2013.

Publications

Emad Shafie and Antonio Cau. A Framework for the Detection and Prevention of SQL Injection Attacks. *In Proceedings of the 11th European Conference on Information Warfare and Security ECIW-2012, 2012.*

Contents

Abstract	ii
Acknowledgment	iv
Dedication	vi
Declaration	vii
Publications	viii
Contents	ix
List of Figures	xiv
List of Tables.....	xvi
List of Listings	xvii
Acronyms	xviii
Chapter 1	1
Introduction	1
1.1. Background	2
1.2. Motivation and Research Objectives.....	2
1.3. Research Question	4
1.4. Scope of the Research	4
1.5. Research Methodology	5
1.6. Success Criteria	7
1.7. Thesis Outline.....	7
Chapter 2	9
Background and Related Works	9
2.1. Introduction	10
2.2. Web Applications Review	10
2.2.1. Web Application Architecture	11
2.3. Web Application Security	13
2.3.1. Hacking Definition	14

2.3.1.1.	Hacking Types	14
2.3.1.2.	Hacking Aims	15
2.4.	Hacking Web Application	15
2.4.1.	Web Application Vulnerabilities	16
2.4.2.	Web Application Vulnerabilities Scanning Tools	20
2.5.	SQL Injection	21
2.5.1.	SQL Injection Technique and Examples	21
2.5.2.	SQL Injection Classification.....	22
2.5.2.1.	Tautology Query	23
2.5.2.2.	Piggy-Backed Query	23
2.5.2.3.	UNION Query	25
2.5.2.4.	Logically Incorrect Query	26
2.5.2.5.	Stored Procedures.....	27
2.5.2.6.	Inference Query.....	27
2.5.2.6.1.	Blind Injection Inference.....	28
2.5.2.6.2.	Timing Inference Query	28
2.5.2.7.	Alternate Encoding.....	29
2.5.2.8.	Inline Comments	30
2.6.	Automated SQL Injection Attacks	31
2.6.1.	SQL Injection Tools.....	32
2.6.2.	False Positive and False Negative	35
2.7.	Detection and Prevention Existing Approach	35
2.7.1.	Controlling the User Input	36
2.7.2.	Scanning Tools Using Black Box Testing Approaches.....	36
2.7.3.	Scanning Tools Using White Box Testing Approaches	38
2.7.4.	SQL Randomisation Approach.....	40
2.7.5.	Filtering Input (String Analysis) approaches.....	40
2.7.6.	Taint data Approaches	41
2.7.7.	Static and Dynamic Approaches.....	43
2.8.	Motivation Revisited	46
2.9.	Summary	47

Chapter 3	49
Preliminaries	49
3.1. Introduction	50
3.2. Temporal Logic Background.....	50
3.2.1. Examples of Using Temporal Logic	51
3.3. Interval Temporal Logic.....	52
3.3.1. ITL Syntax.....	53
3.3.2. ITL Semantic	54
3.2.3. Derived Constructs	55
3.2.4. Examples of ITL.....	56
3.3.5. Why ITL?	57
3.4. Tempura.....	58
3.4.1. Tempura Syntax	58
3.5. Anatempura	67
3.6. Using of Anatempura in Our Framework.....	71
3.7. Summary	72
Chapter 4	73
Architecture of Detection and Prevention Framework	73
4.1. Introduction	74
4.2. Overview of Detection and Prevention Framework (DPF).....	74
4.3. Initial Phase (receiving Data)	77
4.3.1. Initial Capture of User Input	77
4.3.2. Users	78
4.3.3. Capturing Data	79
4.4. Checker Phase	79
4.4.1. Input Checker Component.....	80
4.4.2. Database Observer Component.....	80
4.4.3. Output Checker Component.....	83
4.5. Decision Phase	83
4.5.1. Feedback Component	83
4.5.2. User's Behaviours Component.....	84

4.5.3. Example of user's Behaviour	86
4.5.4. Updating of User's Behaviour Component	87
4.5.5. Updates Existing Attack Patterns Component	87
4.6. Summary	87
Chapter 5	88
Detection and Prevention Framework Implementation	88
5.1. Introduction	89
5.2. Implementation Assumptions.....	89
5.3. DPF Components Implementation	90
5.3.1. Capturing Data Component	91
5.3.2. The input Checker.....	96
5.3.3. Behavioural Functions	104
5.3.4. Implementation of Database observer	107
5.3.5. Implementation of Output Checker	109
5.4. Summary	111
Chapter 6	112
Detection and prevention framework Evaluation	112
6.1. Introduction	113
6.2. Evaluation Criteria	113
6.3. Real Web Application Testing	114
6.4. Single Input Checking	116
6.4.1. User Input Samples Testing	116
6.4.2. Input Checker Limitations.....	126
6.5. DB Observer Testing	127
6.6. Output Checker Testing.....	130
6.7. Behavioural Functions Testing.....	132
6.8. Related Work Comparison	138
6.9. Summary	141
Chapter 7	142
Conclusion	142
7.1. Summary of the thesis	143

7.2. Contribution.....	144
7.3. Success Criteria Revisited	145
7.4. Limitations.....	146
7.5. Future work	147
Bibliography.....	149
Appendix 1: Java Code	158
Appendix 2: PHP Code	161
Appendix 3: Tempura Code	164

List of Figures

Figure 2.1 Architecture of Web Application.....	11
Figure 2.2 OWASP Top 10 for 2010	16
Figure 2.3 SQLdict Tool	33
Figure 3.1 Model Checker Technique.....	68
Figure 3.2 Anatempura Technique.....	69
Figure 3.3 General Architecture of Anatempura.....	69
Figure 3.4 The Main framework Architecture	71
Figure 4.1 The Architecture of Detection and Prevention Framework.....	75
Figure 4.2 Initial specifications.....	78
Figure 4.3 Capture Data component	79
Figure 4.4 Database Observer	81
Figure 4.5 Transactions Relation	84
Figure 4.6 Examples of User Behaviour	85
Figure 5.1 Implementation General Architecture	91
Figure 5.2 Http Request Extracting Data	92
Figure 5.3 Preparing Procedures	95
Figure 5.4 The Input Checker	96
Figure 6.1 Web Application Input Sample	114
Figure 6.2 Booting of Java RMI Server and The Result.....	115
Figure 6.3 Tempura Tab and the Analysis Result.....	116
Figure 6.4 The Analysis Result of Safe Input Samples	119
Figure 6.5 The Analysis Result of Tautology Attack Samples	122
Figure 6.6 The Analysis Result for Piggy-back Attack Samples.....	124
Figure 6.7 The Analysis Results for Union Attack Samples	125
Figure 6.8 False Positive of the Checking Result	127
Figure 6.9 Database Observer Rejected Value	128
Figure 6.10 Database Observer Accepted Value Example	129

Figure 6.11 Database Observer Accepted Value Example 2	130
Figure 6.12 Sample of Web Application Page Error	131
Figure 6.13 Sample of Error Handling.....	132
Figure 6.14 Analysis Results of the Behaviour Input Samples.....	135
Figure 6.15 User's Behaviour Results	136

List of Tables

Table 3.1 ITL Syntax	53
Table 3.2 Interval Operations.....	54
Table 3.3 Non Temporal Constructs	55
Table 3.4 Temporal Constructs	56
Table 4.1 Selective User’s Inputs.....	86
Table 5.1 Input String S	97
Table 5.2 Safe Symbols.....	99
Table 6.1 Samples of Good Input	117
Table 6.2 Tautology Attack Samples.....	121
Table 6.3 Piggy-back Attack Samples	123
Table 6.4 Union Query Attack Samples	124
Table 6.5 Behaviour Input Samples	134
Table 6.6 Existing Approaches Comparison with the DPF (1).....	139
Table 6.7 Existing Approaches Comparison with the DPF (2).....	140

List of Listings

Listing 3.1 Java Assertion Point.....	70
Listing 5.1 PHP script: Sending Value from PHP to Java	93
Listing 5.2 Java Code: Checking Input Method	93
Listing 5.3 Tempura Code: inspecting and Sending Data to Java	94
Listing 5.4 Tempura Code: SearchSpecKword function	98
Listing 5.5 Tempura Code: Checking for Double Dash Characters	100
Listing 5.6 Tempura Code: Checking for hexadecimal encoded injection	102
Listing 5.7 Tempura Code: Checking for Alternative Encoded Injection	103
Listing 5.8 Tempura Code: The CheckingModel Procedure	106
Listing 6.1 The Code of Checking Model Test.....	118

Acronyms

SQL	Structure Query Language
DB	Database
TCP	Transmission Control Protocol
URL	Uniform Resource Locator
HTML	Hyper Text Mark-up Language
HTTP	Hypertext Transfer Protocol
LDAP	Lightweight Directory Access Protocol
OS	Operating System
SSL	Secure Socket Layer
TLS	Transport Layer Security
OWASP	Open Web Application Security Project
XSS	Cross Site Scripting
DBMS	Database Management System
DPF	Detection and Prevention Framework
DOM	Document Object Model
SPDL	Security Policy Description Language
BDDs	Binary Decision Diagrams
PQL	Programme Query Language
JSA	Java String Analysis
API	application program interface
ITL	Interval Temporal Logic
CTL	Computation Tree Logic
CTPL	Computation Tree Predicate Logic
LTL	Linear Temporal Logic
MOFTL	Metric First Order Temporal Logic

Chapter 1

Introduction

Objectives

- Present an introduction and the scope of this research.
 - Identify the research problem statement and the motivation of this research.
 - Highlight the research objectives and the success criteria.
 - Provide the adopted methodology for this research.
 - Provide the thesis outline.
-

Chapter 1 – Introduction

1.1. Background

Web based applications are a very important part of the internet because it enables the transfer of data and services such as banking applications and governmental applications via the Internet. However, the big challenge of using these type of applications is how to increase the confidence of using these environments? And one of the most important points is securing these applications against various types of web application attacks. Web application vulnerabilities have been used to exploit and damage these applications, such as SQL injection, insecure cryptographic storage and XSS (Cross Site Scripting) etc. For example, Yahoo has been attacked in July 2012, and more than 400,000 users password and information are stolen (BBC 2012). Another example is that, the hacking of the Nokia developer's network in August 2011, the hacker stole personal information such as email, date of birth etc (BBC 2011). The exploited vulnerabilities in these examples were SQL injection. SQL injection vulnerabilities have been chosen to be investigated in this research. The following highlights the motivation of our selection.

1.2. Motivation and Research Objectives

Web application vulnerabilities are a big area of research as there are various types of them. SQL (Structure Query Language) injection is a common and dangerous example (OWASP 2010, Clarke 2012). This vulnerability type allows the attacker to damage and steal the information of the web application. SQL injection attacks can be done using various techniques, some of them are manual based on the attacker experience in the structure of the web application and use of SQL commands, and the

Chapter 1 – Introduction

other is automated using existing injection tools. This research is one of many researches dealing with the SQL injection problem (Boyd, Keromytis 2004, Huang, Huang et al. 2003, Jovanovic, Kruegel et al. 2006, Kemalis, Tzouramanis 2008, Kieyzun, Guo et al. 2009, Liu, Yuan et al. 2009). The existing approaches focus only on blocking SQL injection attacks using various techniques, such as static analysis that analyses the a source code of web application and determines the access points of application database (Fu, Lu et al. 2007), filtering user inputs that removes the injecting SQL keywords (Shrivastava, Bhattacharyji 2012) or runtime monitoring approach that monitor the user inputs (Halfond, Orso 2006). The existing approaches will be discussed in detail in Chapter 2.

The existing approaches consider SQL injection attacks to consist of a static run of one step, whereas we consider them to be dynamic and consisting of several steps. For example, if the attacker tries to inject a web application using SQL injection that requires at least three steps, the first step determines the database type, and the second step recognizes the database structure and the last step will exploit and damage the application. In addition, the existing approaches have developed detection techniques that can block existing attacks but cannot deal with new attacks. For example, static analysis approaches have been used to determine weak points in the application and this does not protect the application against new forms of attacks despite the protection is more important than detection. Moreover, the dynamic approaches are monitoring the user input looking for existing attacks, some of them check the sequence of SQL statements at runtime and others compare the SQL

Chapter 1 – Introduction

statement structure derived from static analysis with those at runtime. Therefore, the problem statement for this research is:

- The existing detection approaches are static and consider only one step attacks.
- The detection technique should handle new type of attacks.

Thus, the research objectives can be summarized as follows:

- Develop a novel technique to analyse the user input against SQL injection attacks that can deal with new attack patterns.
- Develop a new approach to model the attack behaviour based on the user input.
- Evaluate the result and compare the proposed approach with existing approaches.

1.3. Research Question

The question discussed in this research is as follows:

How to detect new and existing SQL injection attack patterns and how to profile the attacker behaviour?

A research programme has been proposed in section 1.5 to answer this question.

1.4. Scope of the Research

Several attack types can be used for damaging the underlying tier of a web application, these attacks can be done by exploiting one of the existing vulnerabilities of this application like XSS or insecure misconfiguration etc. This research focuses on the detection and prevention of SQL injection attacks that send HTTP requests to the application server. As aforesaid, there are many studies that tackle the problem of SQL injection attack, such static or dynamic analysis. This research focus on SQL injection attack for the following reasons:

Chapter 1 – Introduction

- SQL injection is classified in OWASP 2007 as second common security vulnerability of top ten vulnerabilities, and in 2010 and 2013 OWASP statistics it is classified as the most dangerous one (OWASP 2010, OWASP 2013).
- To deal with the web application vulnerabilities requires focussing on a specific type of web application vulnerabilities.

The development language that is chosen for this research is PHP and the database type is MYSQL. Our section is based on the fact that PHP and MYSQL are free resources and they can be installed using one execution file like “WampServer” (Bourdon. 2013).

1.5. Research Methodology

This research follows a constructive research method (Iivari 1991). We develop a new framework for the detection and prevention of SQL injection attacks that can detect new and existing attacks in addition to monitor ongoing attacks. The monitoring will be based on ITL (Interval Temporal Logic) and will use the Anatempura runtime verification tool. Thus, our research programme consists of the following work packages:

Work package 1: The research background and the related work.

This work package starts with discussing the architecture and security of web applications, highlighting the type of hacking. It provides a summary of web application vulnerabilities. SQL injection vulnerabilities types will be discussed in detail with an illustrative example of each SQL injection type. The SQL injection

Chapter 1 – Introduction

techniques, i.e., manual or automated will be discussed in detail. The existing approaches for detection and prevention of SQL injection attacks will be discussed critically highlighting related work and motivating our approach.

Work package 2: The framework architecture.

This work package provides an overview of the proposed detection and prevention framework and presents the design, discussing its components including the input, the output and the process for each component. The architecture incorporates the monitoring tool Anatepura. Furthermore, the interaction between these components will be discussed.

Work package 3: Implementation.

In this work package the detection and prevention framework will be implemented. The implementation includes the attacks specification, the detection functions, new attacks detection procedures and the behavioural function for dealing with related attacks.

Work package 4: The evaluation.

This work package evaluates the effectiveness of the detection and prevention framework and its components, implemented in work package 3. The evaluation will test the checking components individually. The behavioural functions will be tested using a case study that involves sequences of user input. The testing results will be analysed to measure the effectiveness of our framework. This work package also contains a comparison between the proposed framework and existing approaches.

1.6. Success Criteria

The success of this research will be measured according to its ability of answering the research question, in addition to achieve the research objectives. Thus, the success of this framework and its implementation will be judged according to following criteria:

- The framework can detect and prevent existing SQL injection attack types.
- The runtime verification tool is suitable for monitoring SQL injection attacks.
- The framework can detect new types of SQL injection attacks.
- ITL is suitable to model attack behaviour.

1.7. Thesis Outline

As mentioned in the previous sections, this chapter provides an introduction that discusses the motivation of this research and specifies the research problem and the scope of this research. Moreover, the research aims and the success criteria have been highlighted. This thesis is organized as follows:

- Chapter 2 introduces web applications and gives an overview of their architecture. Furthermore, it discusses the security of these applications and discusses several web application vulnerabilities in general. Moreover, this chapter discusses existing SQL injection attacks techniques. The chapter concludes with an overview of existing approaches for the detection and prevention of SQL injection attacks.
- Chapter 3 provides an overview of temporal logic in general and ITL in particular, showing its syntax and semantics and presents a justification of our

Chapter 1 – Introduction

selection of ITL. Tempura the executable subset of ITL will be discussed using examples. In addition, the Anatempera tool will be reviewed, discussing its features and architecture and its use in our framework.

- Chapter 4 provides and discusses the detection and prevention framework and its architecture and components. Each component will be explained in detail, in addition the interaction between these components will be discussed. This chapter will also give examples of attack behaviour.
- Chapter 5 provides the implementation assumption and the implementation requirements and the reasons of this selection. It also presents the implemented functions and discusses how each component of our framework is implemented.
- Chapter 6 provides the evaluation criteria and the results of testing each component of the framework. The results will be used to measure the effectiveness of each component of the framework. This chapter also contains a comparison of our approach with existing approaches that tackle the problem of SQL injection attacks.
- Chapter 7 summarizes the thesis and discuss the proposed framework illustrating its limitations and strengths. It then revisits the success criteria of our research and then discusses future work.

Chapter 2

Background and Related Works

Objective

- Reviewing web application architecture.
 - Providing a summary of web application vulnerabilities.
 - Explaining the problem of SQL injection attacks in detail.
 - Discussing the existing approaches.
 - Highlighting the related work underpinning the motivation of our approach.
-

Chapter 2 - Background and Related Works

2.1. Introduction

The security of web applications is a concern for many organizations such as banks, universities and other companies. To understand the security aspects of web applications requires being conversant with the basic knowledge of the architecture of web applications and the general process of the transformation of the data in a web application. This chapter provides in general the architecture and the main concept of web application, and it also discusses in detail the web application vulnerabilities, especially SQL injections. This chapter is divided and organized into several sections. Section 2.2 reviews the web application in general and highlights the web application architecture. Section 2.3 highlights the main concept of web application security describing the concept of hacking in general and its aims and types. Section 2.4 defines the hacking of a web application and explains various types of web application vulnerabilities. Section 2.5 describes SQL injection techniques in detail. Section 2.6 discusses SQL injection automated attacks and some of the existing injection tools. Section 2.7 discusses the existing approaches that are proposed to address SQL injection vulnerabilities. Section 2.8 reviews the motivation of this research and highlights the research problem. Section 2.9 concludes and summarizes this chapter.

2.2. Web Applications Review

Due to rapid development of computer software and the Internet communications, the online services have been increased. There are many institutions that have made their services accessible via the Internet. Those institutions have various aims and

Chapter 2 - Background and Related Works

purposes depending on their activity, looking to attract the users to access their webpage to achieve the best return of their availability on the Internet. Consequently, the data and the services are normally placed in a web application. The web application is a software system can be accessed by the user over the Internet (Morley 2008). Another definition of web application is “any software application that depends on the Web for its correct execution” (Gellersen, Gaedke 1999). Therefore, the previous definitions have agreed that a web application is an application or software that depends on the web environment. Accordingly, the features of a web application are similar to features of the web, such as accessibility, availability, and scalability. The next section will specify the web application architecture.

2.2.1. Web Application Architecture

In general the web based application consists of three layers which are as follows:

Logical layer, Middleware layer and Data layer as shown in Figure 1 (Woodger Computing Inc 2012).

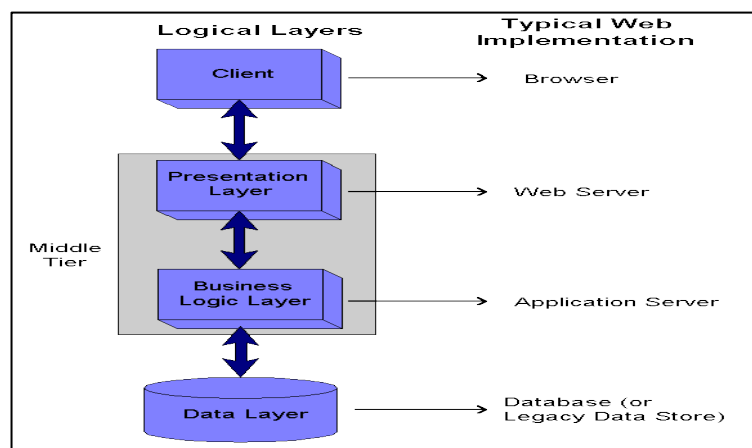


Figure 2.1 Architecture of Web Application

Chapter 2 - Background and Related Works

- Client layer: this layer is running in the user web browser and implements the user interface to allow the user to enter or change data according to the applications needs, and it allows the user to view the content of the web page as well. The client layer uses two general techniques which are as follows:
 - Dump technique: with this technique, the application page has been built by using HTML code only. This type of pages can run with old versions of web browsers. However, there is no validation at these pages which means the data entry will be checked by the middle tier layer. Accordingly, if there is any error caught by the middle layer, the page will post it back to the user browser.
 - Semi-Intelligent technique: here the web page would be built by using Dynamic html and JavaScript in addition to HTML. These pages are more flexible than the dump technique and the developer can design the web page with some options and validations of a user input that can be executed on the client side. As the result, the developed page will be run with better performance.
- Middle Tier: this layer generally consists of two layers which are the presentation layer and the business layer. The first one is for generating the web pages in addition to its dynamic content. The other task of this layer is for decoding the submitted pages that are coming as packets from the client who submitted these pages. Thus, this layer can extract the data that is submitted by the user and send this data to the business layer. The business layer is used to perform the logical part of the application such as the calculation, and user validation. Additionally, this layer is used to manage the application workflow and the access to the data layer.

Chapter 2 - Background and Related Works

- Data layer: this layer organizes and stores the data that is passed from the business layer and retrieves the data that is required from the business layer as well. Moreover, some data manipulation would be done in this layer. For example, the business layer requires specific data from data layer. So, the preparation stage to process the required data can be done in any layer whether business layer or data layer. Similarly, if the data manipulation requires a calculation or collection of data from multiple tables, then the database engine will process this request using the database procedures (Woodger Computing Inc 2012).

The previous paragraphs described in general the web application layers; next we will review the security of web application.

2.3. Web Application Security

Web Applications allow various types of users to access the obtainable services. The permanent availability of web applications will increase the opportunity for everyone who is looking to exploit and damage these applications for illegal purposes. The people who are damaging a web application are commonly known as hacker or attacker, and the technique is called hacking (Morley 2008). The developers are working to implement a functional web application and they neglect the security side (Antunes, Laranjeiro et al. 2009). Consequently, many approaches have been developed to secure the web application against harmful attacks. Each approach is looking for the solution from a special perspective; some approaches are looking to secure the network, and other approaches to secure the application or the application server. Thus, to secure the web application one needs to start finding the problem

that requires a solution. The next sections will highlight the common security problems together with an explanation of the hacking aims and types.

2.3.1. Hacking Definition

Traditionally, the hacker notion was used to call anyone who explores or tries to learn how the computer system works. Currently, the hackers meaning has been changed because the objectives and the behaviour of the hacker has changed. The new meaning of hacker is the person who inserts malicious code to stop the system or to gain unauthorized access for personal or harmful purposes (Beaver 2007).

2.3.1.1. Hacking Types

In general, the hacking types can be classified according to the classification criteria that are used to distinguish between the hacking types. The first classification is from the ethical perspective, and there are two main types which are ethical and unethical, the ethical one is to perform testing for the application to find the weak points by using hacker techniques (Simpson, Backman et al. 2010). The unethical is gaining access for malicious aims such as damaging the application database. Another classification has done by (Beaver 2007) who classified hacking into several types according to the hacking target which are as follows:

- Hacking a server by exploiting a unsecured port in the server.
- Hacking a network by stealing data which is transferring via the network.
- Hacking a personal computer by using unsecured ports or any other vulnerability like exploiting internet explorer vulnerabilities to steal personal information.
- Hacking a web applications starting with exposing the applications vulnerability

and then exploiting it.

Therefore, different types of hacking pose a threat to the web environment. Accordingly, the security of web applications depends on how to secure this application starting from the user computer to the application server.

2.3.1.2. Hacking Aims

The hacker's aim can be predicted from the attacker intent and his target. However, there is no order that can determine who comes first. Thus, the hacking aims are important and can be used to determine the hacking reasons. For example, the hacking of the data layer of a web application is aiming for multiple objectives

- Rigging of the web data either by adding or modifying the data.
- Stealing information by extracting the data.
- Affect the web database performance by running database remote commands (Halfond, Viegas et al. 2006).

Another example is, the network hacking which is a result of insufficient protection of the system network. The target here is the system network and some of the aims are

- Monitoring the user data.
- Stealing important information that is sent by the user.

The mentioned examples show some of the common hacking aims which are related to hacking target. In other words, the attacker's targets determine the attacker's aims.

2.4. Hacking Web Application

As aforesaid, hacking in general is gaining unauthorized access to execute or achieve

Chapter 2 - Background and Related Works

illegal activities. This unauthorized access can be done by exploiting one or more of the web applications vulnerabilities. Therefore, the question here is what is a web application vulnerability? What types of vulnerabilities do exist? The next sections will describe types of web application.

2.4.1. Web Application Vulnerabilities

The common threat against the security of web application is the widespread occurrence of different types of web application vulnerability. A vulnerability is a weak point or gap in the application which allows the malicious attacker to endanger the application stakeholders. The user, the owner and other objects that are depending on the application are considered to be stakeholders (OWASP 2011).

There are several types of web application vulnerability; each one has special properties, such as the vulnerability style, the detection and prevention techniques. Figure 2.2 shows the statistics of OWASP (open web application security project) top ten vulnerabilities which have classified the percentage of the vulnerability that is used in the hacking of web application in 2010.

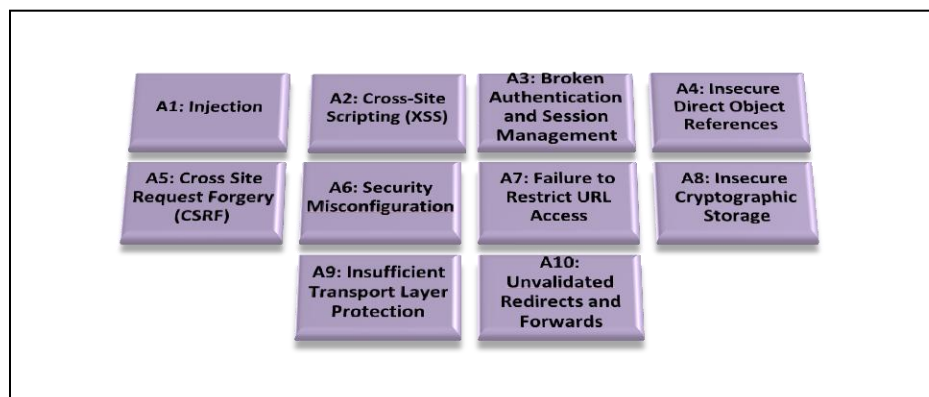


Figure 2.2 OWASP Top 10 for 2010

The statistics have been conducted according to the number of exploiting the same

Chapter 2 - Background and Related Works

vulnerability. Accordingly, the OWASP top ten 2010 vulnerabilities are as follows:

- **Injection:**

This type occurs when the attacker injects the application command or queries by untrusted data. The application interpreter will execute the injected command together with the normal command of the application. In this way, the application data will be affected by unauthorised accesses, as well as the execution of unintended commands. The common example of this type is SQL (structure query language), OS (operating system), and LDAP (Lightweight Directory Access Protocol) injection.

- **Cross Site Scripting (XSS):**

This type happens as a result of poor validation of the untrusted data which is sent via the web application to the web browser. This vulnerability allows a harmful script to run at the victim's computer. Moreover, these vulnerabilities can be classified in two categories which are the first order and the second order attacks. The first order one will be done by inserting script in application page or attract the victim to click on an infected URL that contains a malicious script. The second one is persistent as the attacker can store the malicious script in the application database and can run it permanently. As a result, the attacker can redirect the victim to other malicious sites (Kieyzun, Guo et al. 2009).

- **Broken Authentication and Session Management:**

This vulnerability allows the attacker to hijack the user session or password by compromising it, and using the hijacked information for harmful purposes like exploiting the session as another user. This vulnerability resides in the application as

Chapter 2 - Background and Related Works

a result of poor implementation of the authentication function.

- **Insecure Direct Object References:**

This vulnerability allows the attacker to direct the web application references to be used with other resources. In other words, it allows the attacker to gain unauthorised access of specific resources. This vulnerability is a developer's mistake, because the exposed references of internal object like directory or file are exposed by the developer.

- **Cross Site Request Forgery:**

This type of attack allows the attacker to control the web browser of the victim's computer forcedly, and they can generate requests and send them to the application as if the requests were sent from the victim.

- **Security Misconfiguration:**

This vulnerability is a result of misconfiguration between the system components or neglect of the last update of these components. Therefore, to avoid this type of vulnerability, the system requires a secure configuration for all components. The configuration must be done for system implementation and maintenance (do not use default security option). Moreover, all system software must be up to date starting from the OS to DBMS. For example, if there is a XSS flows in the components. The new update has fixed this problem at the application level but not in application library. These differences can be found easily by the attacker.

- **Insecure Cryptographic Storage:**

The sensitive data in a Web application must be secure enough with suitable hashing

Chapter 2 - Background and Related Works

or encryption techniques to avoid attacks, such as stealing or modifying important data like credit card information or authentication credentials. Thus, if the attacker can gain unauthorized access to a web application databases, he cannot use the stored data as it is encrypted.

- **Failure to Restrict URL Access:**

Some developer lets URL application links point directly to some of the application pages. Normally, the attackers are looking to find the hidden pages by changing the URL address to access it.

For example, if the link of page is <http://example.co.uk/webapp/mainpage> and the attacker will manipulate the URL to http://example.co.uk/webapp/admin_page, then the attacker can gain unauthorised access to other pages. To avoid this vulnerability, the checking of URL access is required for each page of the web application.

- **Insufficient Transport Layer Protection:**

The web application transport protection is important to keep the data transport secure and protected. Many applications have used SSL (Secure Sockets Layer) or TLS (Transport Layer Security) protocols to protect the application data. This vulnerability is a result of weak protection of the transport layer like using expired certificates which are supplied by the SSL provider. For, example, if the network is not secured by SSL the attacker can monitor the network and see the victim's session or cookies then the attacker can used the victim's information through the user session.

- **Unvalidated Redirects and Forwards:**

One of the web browsing features allows the user to move through the web pages by

Chapter 2 - Background and Related Works

redirect or forward. With this moving, validation is required to be sure there is no wrong access for those pages and the redirected and forwarded pages are not changed. The attacker can change the victim's destination pages to other malware sites (OWASP 2010).

The mentioned vulnerabilities are the top ten of 2010, there are other vulnerabilities in a web application such as malicious file execution. Moreover, there are several studies and tools for the detection of the various types of web application vulnerabilities. The next section will highlight some of the scanning tools that are used to expose and determine those vulnerabilities.

2.4.2. Web Application Vulnerabilities Scanning Tools

Due to the increasing security risk in web applications which is the result of the spread of different type of vulnerabilities, there are many tools to scan those vulnerabilities such as Nikto, W3af, Skipfish, Acunetix and Appscan and others (Lyon 2011) ; some of these tools are as follows:

- **Nikto** is a comprehensive solution of web application scanner that can find around 3200 possibly unsafe points. Moreover, it is an open source tool and can be used with multiple types of application server as well as with multiple operating systems like Linux, and Windows. Moreover, this tool is frequently updated to handle the latest vulnerability (Sullo, Lodge 2012).
- (Sullo, Lodge 2012)(Sullo, Lodge 2012)(Sullo, Lodge 2012)(Sullo, Lodge 2012)**Acunetix** is a commercial scanning tool produced by Acunetix Company. This tool has many features in addition to being a web vulnerability scanner, such as

Chapter 2 - Background and Related Works

scanning a web server for unsecure ports. It uses an intelligent and fast crawler that can scan many pages with high performance in addition to detect the type and the application language of the web server automatically (Acunetix 2012).

The mentioned tools are examples of tools that can detect and block various types of web application vulnerabilities. This research will explore one type of vulnerability which is the SQL injection vulnerability. The next section will describe SQL injection vulnerabilities.

2.5. SQL Injection

SQL injection is a common vulnerability used for hacking web application databases by executing a malicious SQL code injected by the attacker. It also has been classified as the first dangerous vulnerability regarding to OWASP statistics (OWASP 2010). Moreover, the problem of this type of attack is that it cannot be handled or controlled by a firewall or other communication security approaches which are used in the prevention of network hacking. Because the attackers using this type of vulnerability can gain access to the web application through the http protocol (Fu, Lu et al. 2007).

2.5.1. SQL Injection Technique and Examples

To serve the user at a website, user information is required. Accordingly, web applications usually provide a login page containing two text fields to allow the user to enter his user name and password. After the user entry, the data will be submitted and the user information will be sent to the web application database to check the user information. By submitting the user data, this data will be sent to the web

Chapter 2 - Background and Related Works

application database using the following SQL statement:

```
Select *from UserTable where username= "user_entry_name" and userpassword  
="user_entry_password"
```

When this SQL statement is executed, the system will return the result of the query. If the user data is valid then the web application permits the user to access other pages at the website or the user input will be rejected and the login page reloads again. However, there is another scenario which is if the user enters the following code at the user name field (user name or '1'='1' - -) then the SQL statement will be like the following:

```
SELECT * FROM UserTable WHERE username = "user name ' or '1'='1' # "
```

At this stage the database engine considers any code after WHERE as a conditional statement, and when the database interpreter find "or 1=1 - ", the check condition is always equal to true. Moreover, any code or condition after the double dash will be ignored. Consequently, the attacker will have unauthorized access to this web application.

This last example is a highlight of one the Tautology query or Tautology based SQL injection attack (Halfond, Viegas et al. 2006, Fu, Lu et al. 2007, Kim 2010). This attack type is done by injecting the web application with a command statement that usually returns true. There are more SQL injection attack types which will be discussed in the next section. There is also a clarifying example for each type.

2.5.2. SQL Injection Classification

According to (Halfond, Viegas et al. 2006) there are different types of SQL injection

techniques. Each type can be done in isolation or in combination. This depends on the attacker's experience, aims and behaviour. In this section various types of SQL injection attacks will be discussed. In addition, for each type there is an illustrative example.

2.5.2.1. Tautology Query

The example in the introduction was a tautology query attack. In this technique the condition statements usually return true or are evaluated to true. When the attacker injects the condition statements of the web application query by malicious code, the attacker is aiming to keep the value of condition statements equal to true. This technique usually uses the login page to inject the login field with "or 1=1".

2.5.2.2. Piggy-Backed Query

The purpose of this type of attack is to inject the original query with an additional query. All queries will be executed in sequence starting with the original one. This attack is different from others because the attackers are not changing or editing the original query, they are just attempting to add new queries and attach them to the original one. Accordingly, the database engine will receive more than one query, the first query will be executed as normal then the second or others will be executed next. Consequently, if the second query was executed successfully, the attackers can execute and inject any SQL command such as stored procedures or any other command. This vulnerability type normally needs a special database engine configuration to allow the attacker to execute harmful SQL commands. In other words, the database engine configuration allows the database system to execute

Chapter 2 - Background and Related Works

single string including multiple command statements. For example, suppose that the following code “ ; drop table UserTable - - ” has been inputted at the login field of the login system page. The scenario will be as follows:

```
SELECT * FROM UserTable WHERE username = ' any ;DROP TABLE UserTable - - 'AND userpassword = ' user_entry_password'
```

After submitting the login page the web application will send this information to the database engine. Then, the database engine will run the login query as routine. As the query is executed the database engine will find the query delimiter “;” or semi comma, so the database will execute the injected code by default. At this stage the user table will be dropped and the system will lose the user data. Another example is, suppose that the database type was an MS-SQL database, and the attacker injects the vulnerable parameter with the SHUTDOWN command. Therefore, the scenario will be as follows:

```
SELECT * FROM UserTable WHERE username = 'user_entry_name' AND userpassword = ' ; SHUTDOWN -- user_entry_password'
```

The database engine will execute the query starting to execute the first part of the query and return null, and then the second part of the query which includes the injected command. Consequently, the injected command will shut down the database (Stuttard, Pinto 2011). One more example is, if the attacker injects the query with a statement to insert user data in above scenario. At this stage the attacker can add wrong information to the database system. Note that there are differences between databases engine to separate the queries. Accordingly, the good way to detect and prevent this type of attack is using an effective technique for validation of the user

Chapter 2 - Background and Related Works

entry at runtime by scanning and analysing queries to find query separations, as well as a correct (safe) database configuration (Lee, Jeong et al. 2012, Kim 2010).

2.5.2.3. UNION Query

The idea behind the union query attack is similar to the other SQL injection types; the attackers are looking for a vulnerable parameter and try to exploit it by changing the data set which is returned for a submitted query. In addition, by using this technique the application will receive different results from the database instead of the one programmed by the developer. This technique starts with injecting the vulnerable parameter using the UNION SELECT keyword, so the attacker can control the second query to obtain the database information. Moreover, data will be available from any table and the attacker can just choose which data he/she wants or from any specific table. Referring to the last example, if the attacker injects the submitting query at student login page as follows:

```
UNION SELECT StudentName ,StudentId,StudentPass from Students where StudentId = 'P07013000'
```

Therefore the submitted query will be like the following:

```
SELECT * FROM StudentTable WHERE StudentName = 'StudentID' AND StudentPass = 'any'  
UNION SELECT StudentName ,StudentId,StudentPass from Students where StudentId = 'P07013000'
```

At this stage, the database engine will execute the first query and return null, and then it will execute the second query and returned the student data including the login information. Consequently, the attacker has unauthorised access to the system and can change or edit any student data. Note that there are previous attack steps

Chapter 2 - Background and Related Works

using other SQL injection attack types to let the attacker know the database structure before starting with this technique such as an illegal query attack (Anley 2002, Spett 2002, Fu, Lu et al. 2007, Halfond, Orso 2006).

2.5.2.4. Logically Incorrect Query

Logically Incorrect Query or illegal query is an SQL injection attack type used at the early stage of an attack to gather information about the database such as database type, table columns and column type or others. In this type of attack the input is a logically false statement to cause a database error response like adding $2=1$ to the condition statement. Therefore, this technique is usually started by injecting the vulnerable parameter of webpage with an incorrect command (logically) to produce an error from the database engine. Moreover, this technique can be used as a blind injection and the attacker can monitor the web application response. Thus, the attacker can obtain the feedback from the database engine according to that error. For example, if the injection code is as follows:

```
SELECT user FROM UsersTable WHERE username='' or 1 = convert ( int, (select top 1 name from sysobjects where xtype='u')) ; -- AND userpass=''
```

The attacker here tampers the input by providing different data type in the condition statement that is not compatible with the system column data type. Thus, if the injected parameter is valuable the database engine responds to this input by returning error feedback message that allows the attacker to do further steps to retrieve data from this database. (Wang, Phan et al. 2010, Spett 2003, Yeole, Meshram 2011, Halfond, Viegas et al. 2006).

2.5.2.5. Stored Procedures

This SQL injection attack technique is used to run or create stored procedures which are used by the database engine. The stored procedure is usually used by the developer or the database administrator to control the database and to take advantage of the database facilities, such as database access or database services. The stored procedures are not similar to each other, i.e., Oracle database are not similar to MYSQL or MS-SQL database. Thus, the attacker needs to determine the database type to exploit this vulnerability. Therefore, the attacker could start with a logically incorrect query attack type to determine the database type, and then the attacker can use the stored procedure attack. For example, if the developer prepares the login condition statement as follow:

```
SELECT @sql_procedure = ' SELECT LoginId , LoginPassword from UserTable  
where LoginId=' + @userlogin + AND LoginPassword = ' + @password + '  
EXEC (@sql_procedure)
```

In this case the use of a stored procedure @ *sql_procedure* provides a way to the attacker to harm the database of the application as the login values have direct access to this database. (Manikanta, Sardana 2012, Santosh 2006)

2.5.2.6. Inference Query

This attack technique is used when the attacker is not able to get any interactive message via an injection command. Therefore, the attackers are looking to find other ways to expose the website vulnerability. The attacker here estimates a web

application response by injecting it with different SQL keywords till he/she gains the required information from the database to start his attack. This type of attack is generally divided into the following sub types.

2.5.2.6.1. Blind Injection Inference

In this technique, the attackers inject the web page with a condition statement to help them to infer the database layout through evaluating the response of the database engine with the inject condition statement, whether the statement is true or false. At this stage, the system will continue working normally if the statement evaluates to true. Consequently, if the injected statement evaluates to false, the web page will not return an error message. However, the web page will not work normally, i.e., there are differences between the page behaviour before the injection statement and after. Therefore, the attacker here will gather the information by comparing the results of the response from queries with true or false injected command injection. (Spett 2003, Tajpour, Masrom et al. 2010)

2.5.2.6.2. Timing Inference Query

In this technique, the attacker injects queries with a malicious command to make a system delay. Then the attacker will observe the reaction from the web application by monitoring the response time and collect information about the database according to this response. If there is a delay then the injected statement or command runs successfully, otherwise the statement execution has failed and the attacker needs to alter the injected statement. Consequently, there are various ways to inject the web application using this type of attack such as using a delay function; the

Chapter 2 - Background and Related Works

next example will clarify the attack technique. If the database type is Ms-SQL and the attacker injects a field of the web application by adding WAITFOR function then the SQL statement will be like the following:

```
SELECT * FROM UserTable WHERE username = 'WAITFOR DELAY '0:0:20'--  
'AND userpassword = 'user_entry_password'
```

Or with MYSQL the attacker can add the following code to the vulnerable variable

```
' union select benchmark( 22500, sha1( 'test' )) ss, ee from test1 where '1'='1
```

If the injected field is vulnerable to injection then the injected code will make a delay for 20 second till the end of function execution. So, the attacker will observe this delay and knows the injected field is vulnerable to injection and usable for other injection attack. The WAITFOR function does not work with Oracle database which has other code to achieve same delay like “*dbms_lock.sleep(20);*”.

Therefore, the attacker will try several attempts considering different database types (Clarke 2012, Yeole, Meshram 2011, Tajpour, Masrom et al. 2010).

2.5.2.7. Alternate Encoding

Normal attack techniques look for known characters or keywords which are usually called bad characters. In this technique, the attackers escape from the normal detection approaches by using injected text that uses alternate encoding. The alternate encoding uses injected text encoded in ASCII, Unicode or hexadecimal. Thus, the attack aims cannot be determined, so the attacker can use more than one encoding technique. Therefore, during the application development the developer should secure the web application against this type of attack by using effective technique that considers various possibilities of malicious encoding text to prevent

Chapter 2 - Background and Related Works

this type of attack. For example, if the attacker injects the user login field with the following string `exec(char(0x73687574646f776e)) - -`, the query statement that is sent for execution by the database engine will be as follows:

```
SELECT * FROM StudentTable WHERE StudentName = 'StudentID ;  
exec(char(0x73687574646f776e))--' AND StudentPass = 'Studentpassword'
```

At this stage, the database engine will execute the mentioned query by using the char function which is built in the database engine. Note that the char function changes the character style of encoding keyword to be in the actual style of character. So, the injected encoded text that mentioned before is working similar to shutdown command, and when the attacker inject the web page by this encoded text the database system will stop working. Therefore, this attack technique is not the same as the attack in previous sections because the effective prevention against this type will need to consider all possible injected encodings that could be harmful to the web application (Howard, LeBlanc 2009, Halfond, Viegas et al. 2006).

2.5.2.8. Inline Comments

This SQL injection attack can be used with all of the previous attacks technique as the attacker can divide the injection command using the inline comment programming feature. This technique can support the attacker to elude from the primitive detection and prevention techniques that are looking for a specific character. For example: if the attacker uses the tautology techniques as follows:

```
Select * from users where username = 'or '1'='1 and password = ' any word '
```

This query can be divided using in line comment as follow:

Chapter 2 - Background and Related Works

*Select * from users where username = 'or /* hi */ '1'='1 and password ...*

Another example if the attacker combine alternate encode with in line comment as follow:

*Select * from users where username = 'or %00 /* hi */ '1'='1 and password ...*

The attacker here injects the null character and in line comment to the original query using the tautology attacks techniques (Clarke 2012, Howard, LeBlanc 2009).

As aforesaid, there are different types of SQL injection attack vulnerability. This classification of SQL injection is useful as it helps the developer to detect and fix the SQL injection vulnerabilities during the application development stage. The other useful way to detect SQL injecting vulnerabilities is determining all possible injection ways to know how these vulnerability types could be exploited. The next section will highlight some of the attack methods that are used with SQL injection.

2.6. Automated SQL Injection Attacks

In general, the injection techniques can be summarised in two main categories, the first one is the manual technique which can be done using the mentioned attacks types that are discussed in the previous section. Success of this injection type depends on the attacker's experience and the security level of the target web application. The detection techniques used to detect this type of attacks depend on the detection of the user input, or in other words it depends on the detection of the injection paths which can be summarised as follows:

- Inputting data by using a parameter
- Inputting data by manipulating URL

Chapter 2 - Background and Related Works

- Inputting data by using hidden field
- Inputting data by tampering the http header
- Inputting data by poisoning the application cookies (Livshits, Lam 2005).

The other type of the injection attacks are automated SQL injection using one of the existed injection tools that are used to attack web application. In the next section some of these tools will be discussed.

2.6.1. SQL Injection Tools

Several automated injection tools have been used for attack, as a tool is easier to use than the manual attack, the attacker just gives the basic information that is required by the tool and waits till the tool retrieves the attack result whether it is successful or not. Many tools have been created; some of them are primitive tools and only can be used to attack specific database or to execute a prepared injection procedure. Other tools can attack any database type and can be used to execute different injection attacks.

One of the primitive tools is **SQLdict** which can be used with MS SQL server only. This tool needs some values to start, the IP address and the SQL account of the victim in addition to loading of a password dictionary. If the injection attack runs successfully, the tool returns the password of this account.

Figure 3 shows an example of how an SQL account 'sa' is attacked by the **SQLdict** tool; the tool has returned the password value of this account.

Chapter 2 - Background and Related Works

The weakness of this tool is that it is limited to one database engine type and it can only search for the password of known SQL accounts in the password dictionary that is loaded by the tool (SQLdict Tool 2008).

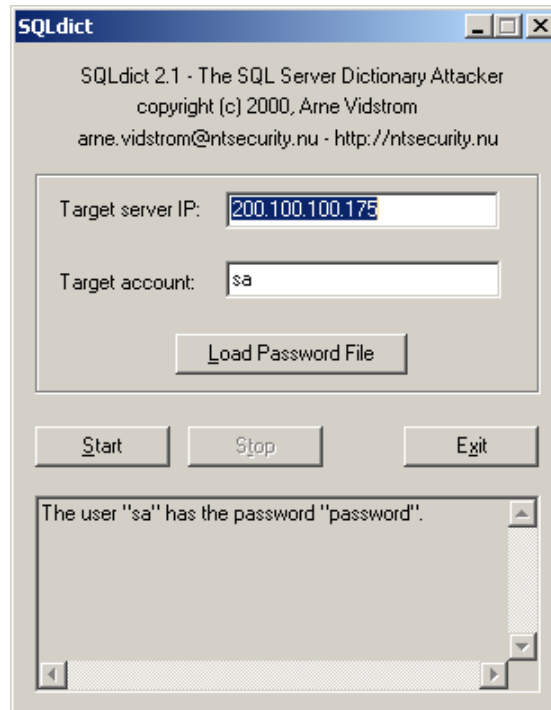


Figure 2.3 SQLdict Tool

Another SQL injection tool is **SQLier** which can be used to attack MYSQL type of database. In general, this tool attacks a vulnerable URL and tries to find out some information about vulnerable components to create an SQL injection template and start exploiting it. The common use of this tool is to find the password of the database based on the Union query attack. **SQLier** runs using the following command:

sqlier [option like -u for username , -o to crack password to file, ..etc] [URL].

This tool is better than SQLdict tool as there is no dictionary to find the password in.

Chapter 2 - Background and Related Works

However, both tools are still primitive as they can only be used for injection of specific database type and execute specific injection attack (SQLier 2006).

One of the more sophisticated SQL injection tools is SQLmap as it has many features that can be summarized as follows:

- Can attack different type of databases like Oracle, MYSQL, etc.
- Support different types of SQL injection techniques such as blind injection, Union query and others.
- Searching for specific database name, table or column and finds the relevant name that contains a string of user name and password.
- Establishing an interaction channel between the attacker pc and the DB server using TCP connection (SQLmap 2012).

Use of the SQLmap tool is similar to the previous tool as it needs some information to starts like the target server address. Then, it can start attacks or test the web application for SQL injection vulnerable components. However, SQLmap has more features and better performance and it is not limited to one database type like the primitive tools.

There are also many other tools like **SQLSmack** for MYSQL and **OracSec** for oracle database, each one has its advantage and limitation depending on the type and environment of use.

The mentioned tools have been produced as result of many studies for the detection of vulnerable components of web applications. Moreover, before discussing these studies an important point should be discussed which is the false positives and false

negatives problem in the detection result. The next section will highlight those points in addition to clarifying the differences between them.

2.6.2. False Positive and False Negative

False positives are “when a tool reports incorrectly that a vulnerability exists, when in fact one does not”. Differently, the false negatives are “when a tool does not report that a vulnerability exists, when in fact one does” (Clarke 2012). Therefore, the most dangerous types of the checking result are false negatives. Some of the existing studies measure the success of their approaches by checking the percentage or the rate of the false positives and the false negatives in their result as one of the evaluating criteria. For example, (Jovanovic, Kruegel et al. 2006) mentioned that there are no false positives produced by their checking model, (Halfond, Orso 2005) said that their approach only produced false positives in two cases and they have specified those cases. Thus, if there is a high rate of false positives or negatives in a specific study comparing with other studies that means the technique of the study that have less numbers of false positives or negatives is more accurate than the other one.

In the next sections the different types of existing detection techniques will be highlighted.

2.7. Detection and Prevention Existing Approach

Many studies have been conducted for the detection and prevention against web application vulnerabilities in general and SQL injection vulnerabilities in particular; these studies have discussed the detection and prevention techniques from different point of views and using different techniques. Some of them used static techniques

Chapter 2 - Background and Related Works

which are used during development time by analysing the web application code to detect the injectable point in the application such as (Xie, Aiken 2006, Fu, Lu et al. 2007, Gould, Su et al. 2004). There are other techniques that use both dynamic and static techniques by monitoring the user input at runtime such as (Halfond, Orso 2006, Huang, Yu et al. 2004). The next sections will discuss these types of detection and prevention techniques.

2.7.1. Controlling the User Input

The available entry fields of a web application can be considered as a gate in front of the attacker. Several suggestions have been proposed to control the user input such as

- Determining the size of text input, if the attacker tries to inject a union attack query in the login field and this field size is ten characters, the attacker cannot inject this field.
- Character replacement: remove some of the common characters that can be used in the injection like semi comma.
- Input validation, by validating the input value that is entered by the user (Hoffmeyer, Wang 2003).

2.7.2. Scanning Tools Using Black Box Testing Approaches

These approaches use two main steps for gathering the information about the weak points in the web application. The first step detects the application workflow using a web crawler to find the vulnerable points. The second step generates an attack and monitors the applications behaviour. This technique has been called black box testing

Chapter 2 - Background and Related Works

as the scanning tools do not examine the source code of web application directly but they try to generate special input and simulate it with this application.

(Kals, Kirda et al. 2006) have developed Secubat which is an open source tool that can scan a web application to detect the vulnerable points. This tool has a graphical user interface that gives the user flexibility to run the testing process. This tool has three components which are a crawler, attack generator, and the analyser. The crawler determines the link tree of the application pages including a web form fields starting from the root web address. The crawler in this approach based on a queued workflow system which improves its efficiency as it can run several concurrent worker threads. Moreover, Secubat tests a web application by injecting single quote for each form field and reports a web application response. The pages response result will be analysed by the analyser.

(Huang, Huang et al. 2003) also have proposed a black box testing technique called the WAVES scanning tool. It is also an open source scan tool based on a web crawler supported by a parser engine that uses a DOM (Document Object Model) parser (W3C 2009) to provide a comprehensive description of the web application components. The attack generator will use the crawler's result to inject a web application fields with a prepared SQL injection pattern. The attack generator's result will depend on a web application response and output. The WAVES tool uses a machine learning technique to enhance and improve its attack generator methodology.

These approaches are useful as they provide a report that shows a web application's

security level, but they have the same problem as other black box testing approaches in that they cannot provide a comprehensive solution as effective as to white box testing.

2.7.3. Scanning Tools Using White Box Testing Approaches

The white box testing or static analysis approaches are based on analysing the internal code of a web application and its structure to detect the vulnerable points at compilation time. Several attempts have been made to check a web application for SQL injection vulnerabilities using the white box testing approach, some of them will be highlighted in the following:

(Gould, Su et al. 2004) has proposed the JDBC Checker which is a tool that can check statically for type correct queries in the SQL statement that are generated dynamically in Java. This technique detects only the SQL injection vulnerabilities that are based on type mismatches like logical incorrect query attacks, because it checks only the syntax of SQL statement for errors, but SQL injection attacks can be syntactically true and it does not return database errors.

(Xie, Aiken 2006) uses an analysis algorithm to analyse open source PHP web applications statically for SQL injection and XSS vulnerabilities. This approach employs analysis to detect and handle vulnerable components of PHP code and other scripting languages that are used to develop the application pages. The authors run the analysis in three steps. The first step converts all application functions into blocks and summarizes these blocks by determining the variables and their location, the block programming language and the variables flow. The second step is an

Chapter 2 - Background and Related Works

intraprocedural analysis to detect the errors and the return set for each block. The third step is an interprocedural analysis to identify block conditions, such as, whether the block has a variable that must be sanitized before running this block. Thus, the vulnerable components will be detected by simulating these blocks using the analysis result. This approach cannot handle inline comment injection attacks and reports a high number of false positives.

The SAFELI framework is one of the white box analysis techniques proposed by (Fu, Lu et al. 2007) to analyse ASP.NET applications. SAFELI consists of several components; one of the main components is MSIL (Microsoft Intermediate Language) Instrumentor which is used to manipulate the application byte code by inserting additional functions for each access point of the application database and replace its variables with symbolic constraints. The output of this component will be scanned with a second component called a symbolic execution engine that maps the whole application pages and its entry points and examines these points for pre collected information about attack patterns called attacks library. Thus, the examination results report the application's vulnerabilities. However, this approach detection is limited as it is based on the existing vulnerabilities that are identified in the attacks library.

In general, static analysis approaches are required to be more accurate for detecting security vulnerabilities, because they report a high number of false positives in the analysis reports (Livshits, Lam 2005). Moreover, applying these approaches for different host languages requires time and extensive effort due to the differences of

the structure of these languages (Bravenboer, Dolstra et al. 2007).

2.7.4. SQL Randomisation Approach

The main idea of this approach is adding numbers to each SQL keyword that are used in the query statement of the application. These numbers are integer numbers generated randomly. Then, during the execution of the application it will rewrite the SQL statements using a proxy filter and by adding a random number to the SQL keyword. Therefore, when the attacker tries to inject the application with any SQL keyword the system will reject them due to the missing random number (Boyd, Keromytis 2004, Kc, Keromytis et al. 2003). However, the problem of this approach is that if the attacker can determine the random number the application can be attacked.

2.7.5. Filtering Input (String Analysis) approaches

This technique is based on filtering from the input data the malicious SQL keywords that can be used to attack the database system. (Scott, Sharp 2002) has developed a proxy filter for the web application that can enforce the validation rule to check user input. Filtering data in this approach uses three components; the first one is the validation constraints specification using SPDL (security policy description language) in addition to the specification of the transformation rule. The second component is a policy compiler which compiles these specifications for execution on a security gateway component. The security gateway validates the specification rules on a web server by checking all http requests before sending it to the application database. However, this approach requires many technical specifications

Chapter 2 - Background and Related Works

to be done by the developer as described in (Scott, Sharp 2002).

(Shrivastava, Bhattacharyji 2012) propose a protection and detection technique based on filtering the user input, they have generated a two level filtration model. The first one is an active guard which builds a susceptibility detector that can block any malicious characters that could be used to attack the web application database. The active guard runs blocking procedures that compare a user input with an existing list of common malicious characters. The second one is a service detector which is used to validate a user input. This approach can block all the existing types of SQL injection attacks using a function called 'killChars'. The drawback of this function is that the function removes several characters that can be used for normal writing without an extra checking of using these characters. Thus, it likely to report a high number of false positives.

2.7.6. Taint data Approaches

These approaches start with a static analysis that identifies hotspots or sensitive points in the web application which are any point that can be used by the application to access the application database. The other step is tracking the data that comes through these hotspots. Examples of these approaches will be highlighted in the following:

(Livshits, Lam 2005) proposed an approach to find Java Tainted Objects. They are using static analysis consisting of two steps. The first step determines the security flow of a web application using a context-sensitive analysis technique (Whaley, Lam 2004) which represents many program contexts using BDDs (Binary decision

Chapter 2 - Background and Related Works

diagrams). The BDDs will be translated using `bddbdb` tool into BDDs-based implementation that can be accessed as a Datalog queries. The second step uses the PQL tool (Martin, Livshits et al. 2005) that can detect the application vulnerable components using the result of first step, and thus reports the application vulnerabilities in addition to its specification using a program query language. The drawback of this approach is that during the information flow analysis, any SQL query that receives data from the user will be considered a false positive vulnerability. For example, the function `executeQuery` is a common sink function used by a Java application to execute an SQL statement and thus retrieves the data from the application database. According to the flow analysis, if the system finds any taint string or data that is passed to this function the system will consider it a unsafe point and thus the application is vulnerable. The problem of this approach is that it reports a high number of false positives.

Also (Jovanovic, Kruegel et al. 2006) have proposed another detection technique implementing by the Pixy tool (Jovanovic, Kruegel et al. 2006) which is a prototype written in Java that can analyse a PHP application statically. This analysis technique is based on data flow analysis to find the taint points of a web application. However, the analysis result shows that there is a rate of 50% of false positives.

(Wassermann, Su 2007) proposed another technique that can analyse a PHP application statically in two steps. The first one uses context free grammars (Thiemann 2005) to specify the syntactic structure for all SQL statements of the application. The second step determine and retain the where SQL query will be

Chapter 2 - Background and Related Works

constructed. The second step results will be labelled to “direct” for the data that comes for the user, or “indirect” if the data comes from another resources like the database. This approach reported low numbers of false positive.

2.7.7. Static and Dynamic Approaches

The main idea of these approaches is finding the sensitive point by analysing the web application code using a static analysis technique to detect the vulnerable components. Then, these vulnerable components will be instrumented with a runtime protection guard to ensure that the submitted data to the application is secure. The following will highlight some of these approaches.

(Huang, Yu et al. 2004) have developed the WebSSARI tool that employs a detection algorithm based on the analysis method of the application information flow to detect the sensitive function that can be tainted in a PHP application. This tool has been supported by a runtime guard that can run an extra checking for sensitive functions that are found by the static analysis. In addition to the static analysis, a runtime guard is added that depends on the annotations that are provided by the user. The runtime guard filters the submitted user input from any SQL Keyword that can be injected in this input. However, the result of the first step static analysis reports a high number of false negatives and false positives (Xie, Aiken 2006) .

(Halfond, Orso 2006) developed AMNESIA (Analysis and Monitoring for Neutralizing SQL Injection Attacks) tool that can be used for the detection and prevention of SQL injection attacks. This tool combines two techniques which are static analysis and runtime monitoring. The static analysis procedure builds an SQL

Chapter 2 - Background and Related Works

query model using JSA (Java string analysis) (Christensen, Møller et al. 2003) that determines the construct queries points which have direct access to the database and specifies the sequence of tokens of that query. Successively, the other step is runtime monitoring which investigate all queries before they are sent to the database. This investigation checks the constructs queries at runtime and compares them against any of the existing attacks. The runtime monitoring specifically checks the sequence of tokens that are specified by SQL query model, thus if the monitoring step finds that the query matches with no previous sequence the query will be prohibited accessing the database. This technique consists of two steps, and the limitation is the monitoring step that depends on the result of static analysis step. For example, in a hard-coded string (like null character %00) there is a mismatch between SQL query model and the runtime monitoring as the last one looks for the original keywords and cannot catch a hard-coded string that is recognized by the SQL query model.

(Kemalis, Tzouramanis 2008) have also proposed a monitoring technique based on a detection algorithm that specifies the syntactic structure for all SQL statements of the application through several phases. These phases describe each SQL statement of the application using a lexical analyser (Kodaganallur 2004) to determine the sequence of SQL keywords in these statements. The monitoring step checks if there is any SQL code injected in a specific SQL statement based on the specification of this SQL statement, and thus blocks unsafe SQL statements from the execution on the database.

(Lam, Martin et al. 2008) improves their previous approach (Livshits, Lam 2005)

Chapter 2 - Background and Related Works

which uses a static analysis technique based on information flow (explained in Section 2.7.6). In their improvement, they add a dynamic error recovery which is a runtime monitoring technique based on PQL specification that is described in the static analysis step. This monitor is added to recover some cases that generate errors during the static analysis. The monitor compares the sequence of query contents of a specific query with its PQL specification, if there is a difference between them this query will be prohibited from the execution on the database.

(Lee, Jeong et al. 2012) use a combination of static and dynamic techniques by removing any of the SQL attribute value of the SQL query at runtime and compare it with a static SQL query. They use Paros (Paros 2004) which is a scanning tool that can perform the static analysis of an application to detect the vulnerable points and describe the syntactic structure of these points. The dynamic step performs the monitoring of the input by applying a detection algorithm that can filter the input from any malicious code based on the static analysis results. However, this static analysis is based on the Paros tool and the last update of Paros was in 2004.

(Manikanta, Sardana 2012) propose a similar technique that starts by analysing all application URL links to detect the vulnerable parameters and the injection points of the application using w3af which is a static analysis tool (Riancho 2012). The next step generate legitimate SQL queries based on the previous step results. The legitimate SQL queries are all valid application queries that can be run. The monitoring step uses GreenSQL (GreenSQL LTD 2012) as a database firewall or front-end to database that can protect the application database against SQL injection.

Chapter 2 - Background and Related Works

GreenSQL monitors legitimate SQL queries and rejects any attacks and reports attack attempts. The author here combines between two existing solutions to achieve the best result of protection system. However, the GreenSQL does not support protection for Oracle database types.

The previous discussed various methods that can detect and prevent SQL injection vulnerabilities. This research is similar to one of the mentioned techniques which are the detection of SQL injection at runtime by monitoring user input. The next section discusses some of the existing approaches including this research and highlights the contribution of this research.

2.8. Motivation Revisited

Many tools have been used to monitor systems at runtime. Some of these approaches have been highlighted in the previous sections. Some of the existing monitoring approaches have checked the order of SQL keywords in a SQL statement at runtime comparing that to the order that is determined by the static analysis using JSA (Halfond, Orso 2006). Other researchers developed a technique using java monitoring to compare the syntactic structure of SQL statements using static analysis with its structure at runtime (Kemalis, Tzouramanis 2008).

Additionally, some of the monitoring do not require static analysis, they just run at runtime only like (Natarajan, Subramani 2012) that propose some specification for detection policies and apply their detection algorithm. As aforesaid, some researchers focus on SQL injection attacks as a static run in one state; so they just try to block the attacker injection attempts (Antunes, Laranjeiro et al. 2009, Fu, Lu et al.

Chapter 2 - Background and Related Works

2007, Lee, Jeong et al. 2012, Kim 2010, Boyd, Keromytis 2004).

However, the attacks are dynamic as they run over several steps such as, finding the vulnerable item, detecting the database type and exploring the database structure. Thus, the detection technique can be improved if there are scenarios that show the injection stages of web application as the detection procedure can predict the next step of the attack. Moreover, some of the existing approaches can only block some of the existing attacks they detect specific injection type because they are not effective to prevent several types like (Natarajan, Subramani 2012), and another one can block all existing types like (Halfond, Orso 2006).

New attacks can be handled in some of the existing approaches like (Halfond, Orso 2006, Boyd, Keromytis 2004) because these approaches block any sequence of SQL keywords that come through the user input. In this research, we focus on two points which are, how to detect new attacks in addition to track the attacker at various stages. Moreover, this research will develop a monitoring technique using the Anatempera tool that runs the detection over several states to find the related attacks and to detect new attacks.

2.9. Summary

This chapter has discussed SQL injection vulnerabilities. It provided an introduction that reviewed the web application architecture, provided a summary of web application vulnerabilities, and explained the problem of SQL injection attacks. It also discussed existing approaches and their detection and prevention techniques and focused on the related work underpinning the motivation of our approach.

Chapter 2 - Background and Related Works

The next chapter highlights the Anatempura tool and the Interval Temporal logic as formal specification language that will be used to specify the monitoring conditions that checks submitted data of the web application against SQL injection attacks.

Chapter 3

Preliminaries

Objectives

- Reviewing temporal logic in general and ITL in detail.
 - Discussing the reasons of our selection of ITL.
 - Describe the Tempura syntax.
 - Providing the architecture of the Anatempura tool.
 - Describe the detection and prevention framework.
-

3.1. Introduction

As aforesaid in the previous chapter, the detection and prevention of SQL injection approaches have been developed using different techniques. Some of these approaches monitor the application at runtime to check the user inputs against any form of SQL injection attacks. This research proposes a new approach to monitor the application at runtime using Interval Temporal Logic (ITL) and specifically using the Anatempura for runtime monitoring.

This chapter introduces Anatempura and its underlying logic ITL. Section 3.2 reviews temporal logic in general and provides examples that show the use of temporal logic. Section 3.3 introduces ITL, specifying its features, semantics, derived constructs, and our justification of using ITL. Section 3.4 describes Tempura (executable subset of ITL) and its syntax providing clarifying examples for each Tempura construct. Section 3.5 introduces the Anatempura tool and its common uses and features. Section 3.6 highlights the framework for detection of SQL injection attacks using Anatempura. Section 3.7 summarises this chapter.

3.2. Temporal Logic Background

The specification of any concurrent program should deal with the execution sequence of these programs in addition to the input and output behaviour of these programs. Temporal logic as a term is being used for expressing program properties involving program conditions, the program execution sequence, and termination etc. (Wolper 1983). In other words, temporal logic can express any system property and structure based on the execution of this system using a sequence of states. There are several

Chapter 3 - Preliminaries

versions of temporal logics such as Computational Tree Logic (CTL), Linear Temporal Logic (LTL), and Interval Temporal Logic (ITL) etc.

In general, these types are similar and focus on analysing the system requirements in addition to the topology of time for these requirements, and they differ from each other in their expressive power. CTL uses a tree-like structure to express the system in terms of its execution paths. Each path describes a specific execution and one of them will be the actual path. LTL expresses the system behaviour as a linear order of execution states of the system. ITL is based on LTL as linear order that can express system requirements. ITL distinguishes from other temporal logic by its use of a chop operator to sequentially compose sequences of states. ITL will be described in detail in Section 3.3. The following section highlights some examples of using temporal logic.

3.2.1. Examples of Using Temporal Logic

Temporal logic is being used in many studies for the specification of requirements, tracking behaviour, and monitoring systems etc. Examples of these studies will be highlighted in the following:

(Holzer, Kinder et al. 2007) have used a verification technique for malware detection based on Computation Tree Predicate Logic (CTPL) which is an extension of CTL. This approach uses a model checker called Mocca that expects as input assembly source code. Mocca has been used to determine whether a security property expressed in CTPL holds or not. The security property that is verified by Mocca is a specification of the behaviour of malicious software.

(Basin, Klaedtke et al. 2010) applied runtime monitoring using MFOTL (metric first order temporal logic) which provide monitoring that can check and enforce system policies like access or usage control policies using an expressive fragment of MFOTL. The fragment consists of formulae that specify system safety requirements. The feature of their proposed monitoring is that it can provide a specification of past and future behaviour using quantitative temporal operators for finite or infinite domain. In their experiment they just show the finite structure that is used for monitoring bank transactions.

(Al Amro, Cau 2012) proposes a virus detection approach that can detect viruses based on the virus behaviour that is specified using ITL. The detection approach uses a tool set that is based on Anatempura (will be discussed later) that can monitor the application program interface (API) calls.

ITL is our selected logic. ITL features and the reasons of our selection of ITL will be described in the following section.

3.3. Interval Temporal Logic

ITL is a formal specification language based on LTL that can express both propositional and first order logic properties and whose key notion is intervals. ITL can describe system requirements and behaviour as a finite sequence of states (interval). In addition, ITL can be used to handle both sequential and parallel composition. ITL has an executable framework Tempura which provides a development and testing environment for ITL specifications (Cau., Moszkowski. et al. 10/2012, Zhou, Zedan et al. 2005, Moszkowski 1994).

3.3.1. ITL Syntax

Table 1 shows the syntax of ITL and provides the syntax of ITL expressions and formulas.

<p><i>Expressions</i></p> $e ::= \mu \mid a \mid A \mid g(exp_1, \dots, exp_n) \mid OA \mid fin A$ <hr style="width: 50%; margin: 10px auto;"/> <p style="text-align: center;"><i>Formulae</i></p> $f ::= p(e_1, \dots, e_n) \mid \neg f \mid f1 \wedge f2 \mid \forall v \bullet f \mid skip \mid f1; f2 \mid f^*$

Table 3.1 ITL Syntax

In Table1, ‘ μ ’ is a fixed integer value, ‘ a ’ is a static variable (variable has a constant value within an interval), ‘ A ’ is a state variable (can change within an interval), ‘ g ’ is function symbol, ‘ p ’ is predicate symbol, ‘ f ’ is formula.

g is a expression that contains mathematical operators like subtraction (-) or addition (+) or others. Atomic formulae are constructed using one or more relational symbols like equal (=) or greater than or equal (\leq) etc. Thus, ITL formulae will be composed using atomic formulae and connectives like ($\neg, \vee, \wedge, \exists, \forall$) in addition to temporal modalities like chop (;), chopstar (*) or skip.

An interval σ is denoted by $\sigma_0 \dots \sigma_n$, an interval consists of one or more states $\sigma_i (i \geq 0)$. Each state of the interval is a mapping from the set of variables to the set of their values. The length of an interval denoted by $|\sigma|$, is equal to the number of

Chapter 3 - Preliminaries

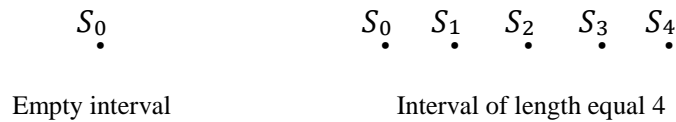
states σ minus one. If the interval $\sigma = \sigma_0\sigma_1\dots\sigma_{n+1}$ then $|\sigma|=n$.

For example, if the length of the interval is 4, the number of states in that interval will be 5. Table 3.2 lists some operations on intervals.

Interval Type	Specification
prefix	$\sigma_0 \dots \sigma_k$ (where $0 \leq k \leq \sigma $)
suffix	$\sigma_k \dots \sigma_{ \sigma }$ (where $0 \leq k \leq \sigma $)
sub	$\sigma_k \dots \sigma_i$ (where $0 \leq k \leq i \leq \sigma $)

Table 3.2 Interval Operations

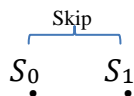
The empty interval is a one state interval. The following example shows a sample of an empty interval and a multi states interval.



3.3.2. ITL Semantic

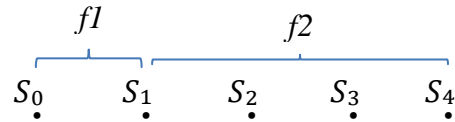
As aforesaid, ITL describes sequences of states (interval). The formula that contains no temporal operator will be called a state formula, and thus will be hold at the initial state of an interval. All ITL formulae will be evaluated over the whole interval. For example, $f_1 \wedge f_2$ will be true over an interval σ iff f_1 and f_2 are both evaluated to true over this interval σ . The following are some useful ITL constructs with their informal semantic:

skip: is a unit interval and its length equal '1'. For example, the following interval has two states ' S_0 ' and ' S_1 ', and its length is equal to '1'.



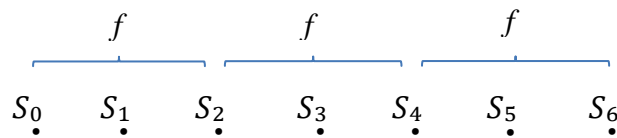
Chapter 3 - Preliminaries

- $f1; f2$: this formula will be true over an interval if that interval can be decomposed into a prefix and suffix interval, and $f1$ holds over the former and $f2$ holds over the latter, and the prefix and suffix interval share a state.



Example of chop construct

- f^* : this formula will be true over an interval if that interval can be decomposed into a number of sub intervals and the formula f is true over all these sub intervals.



Example of Chop star

3.2.3. Derived Constructs

Some frequently used derived ITL constructs are shown in Table 3 and Table 4 (Cau., Moszkowski. et al. 10/2012).

false	$\hat{=}$	$\neg \text{true}$	false value
$f_1 \vee f_2$	$\hat{=}$	$\neg(\neg f_1 \wedge \neg f_2)$	or
$f_1 \supset f_2$	$\hat{=}$	$\neg f_1 \vee f_2$	implies
$f_1 \equiv f_2$	$\hat{=}$	$(f_1 \supset f_2) \wedge (f_2 \supset f_1)$	equivalent
$\exists v \cdot f$	$\hat{=}$	$\neg \forall v \cdot \neg f$	exists

Table 3.3 Non Temporal Constructs

$O f$	\triangleq	$skip ; f$	next
$more$	\triangleq	$O true$	non-empty interval
$empty$	\triangleq	$\neg more$	empty interval
$\diamond f$	\triangleq	$true ; f$	sometimes
$\square f$	\triangleq	$\neg \diamond \neg f$	always
$\textcircled{W} f$	\triangleq	$\neg O \neg f$	weak next
$\diamond_i f$	\triangleq	$f ; true$	some initial subinterval
$\square_i f$	\triangleq	$\neg (\diamond_i \neg f)$	all initial subinterval
$\diamond_s f$	\triangleq	$true ; f ; true$	some subinterval
$\square_s f$	\triangleq	$\neg (\diamond_s \neg f)$	all subinterval
$fin f$	\triangleq	$\square(empty \supset f)$	Final state
$halt f$	\triangleq	$\square(empty \equiv f)$	Exactly in the final state

Table 3.4 Temporal Constructs

3.2.4. Examples of ITL

- The formula $A=5$ means that the variable A has a value 5 in the initial state.
- In the interval, there are two variables A and B, the variable A is always equal 3, the variable B is equal to 0 in the first state and equal to 2 in the next state, the interval will be expressed as follow

$$B=0 \wedge \square (A=3) \wedge O (B=2) \wedge skip$$

$$\begin{array}{cc} A=3 B=0 & A=3 B=2 \\ \dot{s}_0 & \dot{s}_1 \end{array}$$

- In the interval, there are three variables A, B and C. The variable A equals 'R', B is an array of non-accepted characters ['+', '*', '/'], and the variable C will be used to store the comparison result between A and B, C is equal to 'y' if they are similar or 'n' if not. The length of the array is denoted by |B|. The interval can be expressed as follow

skip $\wedge A = 'R' \wedge B = ['+', '*', '/'] \wedge \text{if}(\exists i: 0 \leq i \leq |B| \wedge B[i] = A)$
then $C = 'y'$
else $C = 'n'$

The interval here is a two state interval.

- In the interval, if there is a choice between two formulas that will be executed depending on the value of the propositional variable 'A'. The interval can be expressed as follow

$$(A \wedge f_1) \vee (\neg A \wedge f_2)$$

3.3.5. Why ITL?

Our selection of ITL is based on a number of reasons that make ITL the language of choice for our proposed detection and prevention technique for SQL injection attacks.

- ITL has the Anatempura tool which is a runtime verification tool that can be used for monitoring external applications like web applications. Anatempura has a pluggable architecture that can easily connect to other applications and it also has other features that will be highlighted in Section 3.4.
- The checking procedures that check the user input against SQL injection attacks is a sequence of several checking stages for each transaction. ITL can describe this sequence of stages because of the chop operator.
- The user behaviour will be investigated for all transactions using the checking results. The checking result will be preserved and expressed as ITL intervals. ITL operators like skip and chop are well suited to handle the intervals that contain the

checking results of the web application transactions which are an important factor in the investigation of user behaviour.

3.4. Tempura

Tempura is a programming language based on ITL developed by Ben Moszkowski. Tempura provides an executable framework for development and experimenting of suitable ITL specifications. Tempura variables are similar to ITL as it has 'state' and 'static' variables which can be Integers, Booleans, Floats or Strings, or a derived type like lists. Tempura has most of other conventional imperative programming languages features. For example, Tempura contains iteration construct such as for statement, and it has most of regular operators such as (+),(-) and ,or etc. However, ITL differs from Tempura slightly because Tempura can run only ITL specifications that satisfy three conditions which concern the length of the interval and the variables with their values in addition to that the formula should be deterministic (Moszkowski 1985). The following examples show some different samples of executable and non-executable Tempura code.

$S=0 \wedge \text{skip} \wedge O(S=1)$	is executable.
$\text{len}(5) \wedge \square(S=2)$	is executable.
$\text{empty} \wedge (S=0 \vee S=1)$	is non-executable as there is no unique value for S.
$\text{len}(3) \wedge S \text{ gets } S+5$	is non-executable as there is no initial value of S that can be incremented.

3.4.1. Tempura Syntax

Tempura is an executable subset of ITL. The syntax of Tempura can be classified

Chapter 3 - Preliminaries

into three main categories, Locations, Expressions and Statements.

- **Locations**

Locations which determine where values are stored and can be examined in a specific interval. Thus, if there is an existing variable A, then the location of variable in the next state will be determined by next A. There are various variable types in Tempura such as

Integer number such as 0, 1, 5, -1,-2, or float numbers like \$0.001\$.

List variables which are arrays of fixed or variable length.

There are also other types like Boolean and String variables. The types of expressions in Tempura with a clarifying example will be highlighted in the following.

- **Expressions**

The types of Tempura expressions are, for example, integer, string, list, float or Boolean.

Integer Expression

These types of expressions return an integer value and the operators that can be used with it are: +,-, **,>, or mod etc. For example

exists A , B, C: { A=1 and B=2 and C =A+B and empty }.

C=A+ next B

If (next A) = B then C=2*A else C=2*B

Chapter 3 - Preliminaries

Boolean Expression

These types of expressions return a Boolean value either true or false when evaluated. For example

`A>B`

`next A > 0`

List Expression

List variables are defined using `List(S, n)` or `list(S, n)` commands. The command 'List' defines a list 'S' of size up to n. The command 'list' defines the array 'S' of fixed size n. Moreover, the structure of an existing array 'S' can be fixed over the interval using the command: `stable (structure (S))`. For example

`S[2][1] = 'a'` to assign value a to list `S[2][1]`.

`next S=S+A[2]` to add the value of `A[2]` to list `S` in the next state.

The size of array `S= ['a', 'b', 'c']` is equal to 3. The list size is denoted by `|S|`.

String expression

String expressions have a string as value and these are enclosed in double quotes such as "the mentioned value is a string". String values include most of the C escapes characters. For example, the new line can be introduced using `\n`. The string values also can be appended using the '+' operator like `"aaa" + "bbb" = "aaabbb"`. Moreover, to copy part of a string value one can use `S[i..j]` where `i` denoted the beginning and `j` denoted the end index of the substring. Some examples of using string expressions:

Chapter 3 - Preliminaries

$S3 = S1 + S2$ or $S3 = S1 + \text{next } S2$

$S1 [2..4] = [S[2], S[3], S[4]]$

Let $S1 = \text{"abcd"}$ then the substring of $S1[3..4] = \text{"cd"}$

Float expression

Float expressions have a floating point value. Tempura denotes float values by enclosing them between two \$ characters. Float expressions can be printed using %F.

For example

```
format("the value: %F ", Var)
```

$A = \text{itof}(B) * \$100\$$ 'itof' to change the variable type from integer to float.

$A = \tan(\$70\$)$

- **Statements**

Statements are a subset of ITL in addition to Tempura system commands that can be used to define simple or compound statements.

Example of Simple Statements

$A = B$ assignment in current state.

$A \text{ gets } e$ to assign value of 'e' to variable 'A' throughout the interval.

$\text{stable}(A)$ variable 'A' has a constant value for all states in the interval.

empty interval of length zero.

skip interval of length 1.

The next operator is used to refer to the variable in next state as long as the next

Chapter 3 - Preliminaries

state exists. For example,

$$\text{next}(A) = A+1 \quad \text{or} \quad A := A+1$$

A in the next state is assigned the current value of A plus 1.

$\text{len}(5)$ has interval length 5.

Example of Compound Statements

Choice

Choice in Tempura is denoted by the if statement.

If (A=B) then output A else {output A and output B}

if A then f_1 and if B then f_2

if A then f_1 else f_2

Loop

Loop in Tempura is denoted by for, while and repeat.

while $I \leq |S|$ do { $I := I+1$ and output $S[I]$ and skip }

repeat { $S := S + [|S|+1]$ and skip } until ($|S|=5$)

*for $i < |S|$ do { $S[i] := S[i]*2$ and skip }*

Forall

This operator is a universal quantification *forall $i < n$: {...}*. It corresponds to indexed concurrency.

forall $i < |S|$: { $S[i] := S[i] + 2$ }

all list elements are assigned a new value concurrently.

Chapter 3 - Preliminaries

Keep

keep A=4 and len(6)

A is 4 in all state of the interval except the last state. The last state will be determined using another operator i.e, fin and halt.

Keep (x=4) and len (5) ; (x=5 and empty)

The X variable in the first interval has no value at the last state on the interval.

System commands

There are two system commands which will be used for executing Tempura code such as, 'run' for executing a program, 'load' for loading a program. There are also other system commands such as, the 'input' command to request a value from the user. The 'output' command prints a variable value or any text using double or single quotes and it can print to file or on screen. format is an output command that has feature C programming language style string characters.

<i>set outfile="stdout"</i>	set output to be on screen.
<i>set infile="input file"</i>	input data from file.
<i>format("the number is: %2d \n",Var)</i>	print a text with a variable value.
<i>output(Var)</i>	print the variable value.
<i>output ('any text')</i>	print a string.
<i>input (X)</i>	input the variable value.
<i>/* run */ define name() = { tempura code}.</i>	define a program with a specific name.

Chapter 3 - Preliminaries

load "../library/exprog".

load a program file.

Sequential and Parallel execution

The sequential execution uses the chop operator that ends the current interval execution and starts a new interval. For example if there are two procedures P1 and P2 as follows

define P1(L) = { exists X: { X=L and X gets X+1 and len(5) and fin (output X) } }.

define P2(L) = { exists X: { X=L and X gets X+2 and len(5) and fin (output X) } }.

Sequential example

/ run */ define Test1 () = { exists A,B: { A=3 and P1(A)};{ B=2 and P2(B) } }.*

The first interval has a length of 5 and the P1 procedure gets executed, the second interval has also a length of 5 and the P2 procedure is executed. Thus, the overall interval length is 10.

$$\begin{array}{cccccccccccc} A=3 & A=4 & A=5 & A=6 & A=7 & A=8 & B=2 & B=4 & B=6 & B=8 & B=10 & B=12 \\ \dot{s}_0 & \dot{s}_1 & \dot{s}_2 & \dot{s}_3 & \dot{s}_4 & \dot{s}_5 & \dot{s}_6 & \dot{s}_7 & \dot{s}_8 & \dot{s}_9 & \dot{s}_{10} & \dot{s}_{11} \end{array}$$

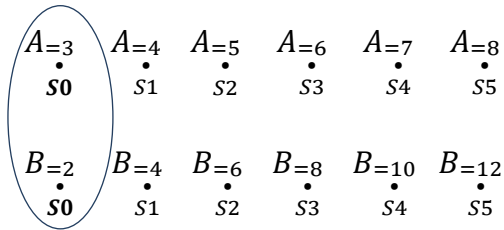
Note that state number 5 is a shared state the two intervals are fused at this state. It contains both of A and B variables. The state 5 ends the prefix interval of the P1 procedure (for A variable), and starts a suffix interval for the P2 procedure (for B variable).

Parallel example

/ run */ define Test () = { exists A,B: { A=3 and B=2 and P1(A) and P2(B) } }.*

In this example, P1 and P2 are executed in parallel, and the overall interval length is 5.

Chapter 3 - Preliminaries



Halt

Halt operator is being used to define the interval termination condition.

$A = 3$ and A gets $A+1$ and halt ($A=6$)

In this example, A is 3 in the initial state, and A will be incremented with 1 in the next state until the value of A equal 6 as the interval will be terminated by halt condition $A=6$.

Fin

There are two versions of fin operator such as, $fin(A)$ which means the value of variable 'A' at the end of the interval (a location). The other version is $fin(A=5)$, i.e., a statement, which means variable 'A' must have 5 at the end of the interval.

Both halt and fin statements concern the last state of the interval. The difference between them is that halt terminates as soon as the termination condition has been reached whereas fin only requires the final state to satisfy the final condition.

Always, Sometimes

$always f$ means that there is a formula f holds for every suffix interval.

$sometime f$ means that there exists suffix interval for which f holds.

$always (output(A))$ print the value of A in every state of the interval.

Chapter 3 - Preliminaries

$\{ A=0 \text{ and } len(3) \text{ and } A \text{ gets } A+1 \text{ and sometimes } (A = 2 \text{ and output } A)\}$.

In this example, A is 0 in the initial state, and the interval length is determined to be 3. A will be incremented with 1 in each next state, and when A reached the state with a value equal 2 then the value of A will be printed.

Exists

This operator is being used to introduce the local variables either static or state.

$exists A,B:\{ A=0 \text{ and } B=1 \text{ and skip and } B:=A \}$ “B:=” means next B

Functions

Tempura functions are denoted by, $define F(v_1, \dots, v_n) = \{e\}$. A parameter will be passed to a function via ‘call by reference’.

$define F(A) = \{A+2\}$.

The function is called: $B=F(A)$

Following example shows how to call the function using more than one parameter.

$define Power(A,B) = \{A ** B\}$.

Function call: $C = Power(A,B)$

Procedure

Following concerns procedure calls.

$define CallbyValue(X,A) = \{$
 $exists B:\{B=A \text{ and } B:=B+3 \text{ and skip and } fin(X=B)\}\}$.

Variable X is used to return the procedure result. This procedure can be called using

Chapter 3 - Preliminaries

the following Tempura code.

```
define TestCallbyValue () = {  
  
exists C,D: {  
  
C=3 and testValue(D,C) and fin(output(D))}  
  
}.
```

The mentioned syntax of Tempura can be executed using the Anatempura tool which will be explained in the next section.

3.5. Anatempura

Anatempura is a tool that can execute Tempura code. Anatempura is also a runtime verification and validation tool that can be used to monitor and check specific application conditions at runtime (Al Amro, Cau 2011, El-kustaban, Moszkowski et al. 2012). Moreover, Anatempura has a visualisation component with a graphic interface that can simplify the tracking and the analysis of the monitoring results. It has a pluggable architecture that can easily connect to other applications. The first C-version of the Tempura interpreter is by Roger Hale, after that further development has been done by Ben Moszkowski and Antonio Cau.

In addition, runtime verification as used in Anatempura does not suffer from the state explosion problem such as experienced by model checkers (Cimatti, Clarke et al. 2002, Holzmann 1997). The explosion problem happens as a result of the huge number of states that are used to specify the system behaviour. The specification will determine all possible system sequence states that can be reached and uses by the

model checker to check the system property.

The state transition used to describe system behaviour, will be determined using two factors which are the current state and the transition relation. The corresponding finite automaton of the system is intersected with the automaton of the negation of the property (Valmari 1998). If this intersection is empty the system satisfies the property. If the intersection is not empty it will constitute a counter example why the system does not satisfy the property. If there are three processes and the system has 10 variables, thus the number of states will be $10^3 = 1000$. Thus, if the system has thousands of variables the number of states grows too big to be practical. Figure 3.1 shows the technique that is used by the model checker to check the system properties.

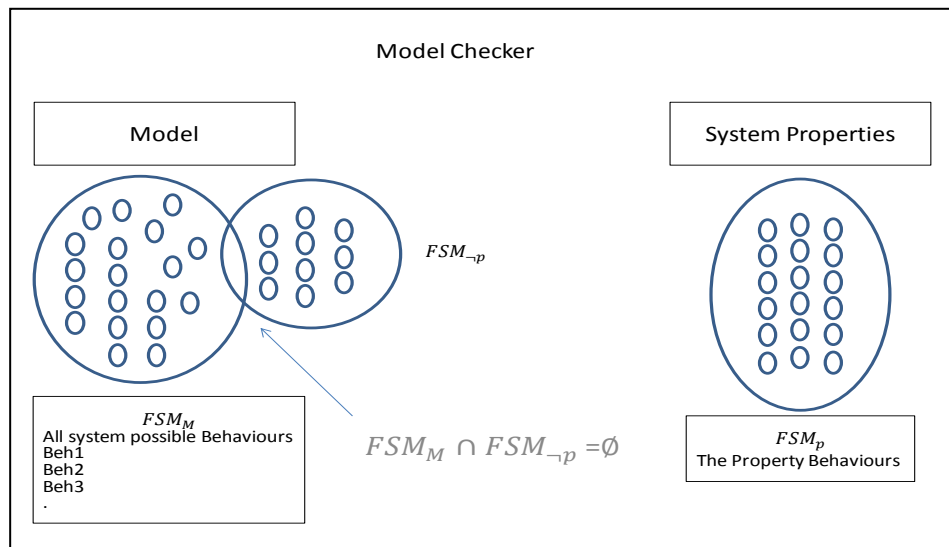


Figure 3.1 Model Checker Technique.

In Anatempura a new state is computed on the fly using rewrite rules as follows:

$f = w \wedge @ f'$ where 'w' is the current or initial state, and that means the next state will be accepted only if formula f' is valid from the next state on ward. This reduces the number of the states to be only the current one that is used during the runtime

Chapter 3 - Preliminaries

validation process. So there is no need to compute the automaton with its states as is done by model checkers. Figure 3.2 the Anatempura Technique

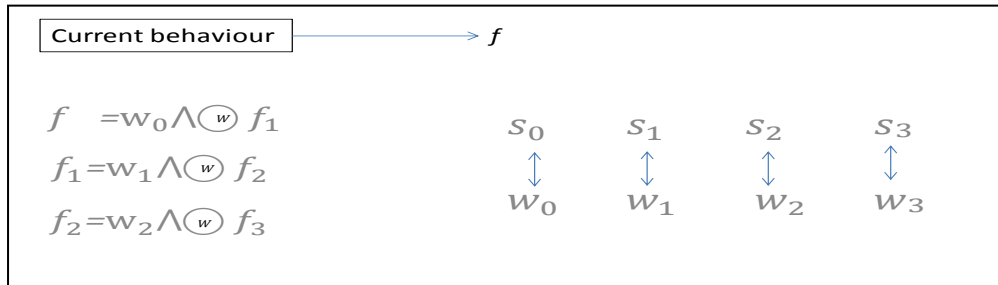


Figure 3.2 Anatempura Technique.

Figure 3.2 shows how Anatempura reduces the number of states using the weak next operator. Therefore, if S_0 is not the same as w_0 then we know that the system behaviour so far will not satisfy the property, otherwise we continue with S_1 and we need to check whether it is the same as w_1 . Again if they are not the same we know that the system behaviour so far will not satisfy the property.

The general architecture of Anatempura is as follow:

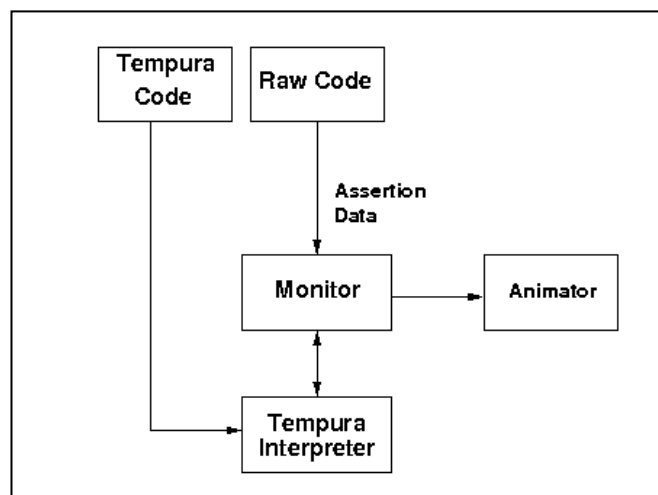


Figure 3.3 General Architecture of Anatempura

Figure 3.3 shows the main components of the Anatempura tool which are Tempura

Chapter 3 - Preliminaries

interpreter, the monitor and animator.

The Tempura interpreter executes Tempura code in addition to validating the monitor conditions. The monitoring conditions of Anatempura are the assertion points which can be easily inserted in the application source code. These points will be used to generate a sequence of states as the application runs like values of variable and the timestamp. This sequence represents a behaviour of the application and can be used by Tempura. Listing 3.1 shows an example of assertion point in Java.

```
System.out.println ("!PROG: assert Var:"+Val+":time!\n");
```

Listing 3.1 Java Assertion Point

Listing 3.1 is a Java print command that contains “!PROG:assert Var:"+Val+":time!” as an assertion point. This assertion point is captured by the monitor as Val is a value of Var when the ‘print’ command is executed.

For example, the following java code contains an assertion point as follow:

```
Date Time = new Date();
```

```
System.out.println ("!PROG: assert Name:"+Name+": "+ Time.toString()+"!\n");
```

This assertion point sends a variable called Name, its value and the system current time to Anatempura. In this research, the assertion points will be added manually to the application source code to monitor the user’s entries and behaviour, the assertion points can be added automatically using JIE tool (Tromer 1999).

Anatempura will be used to perform two tasks. The first task is to monitor the application’s submitted data and check whether this data contains any form of SQL

injection attack. The second task is to investigate related attacks using the checking result of the submitted data.

3.6. Using of Anatempura in Our Framework

As the main goal of this research is the detection and prevention against SQL injection attacks, we need to develop a new security framework that uses runtime verification as shown in Figure 3.2.

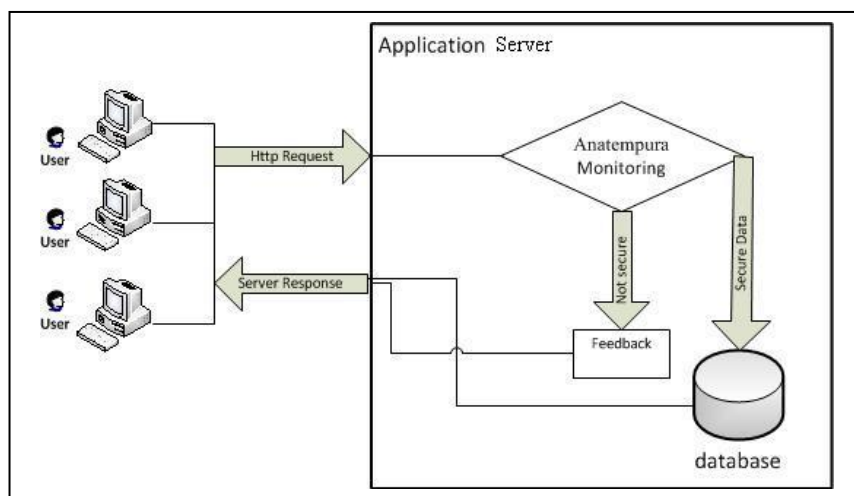


Figure 3.4 The Main framework Architecture

This framework shows the main components of the proposed detection technique. It is based on the specification of SQL injection attacks and the monitoring tool Anatempura which checks these attacks. Anatempura will perform the monitoring of all web application transactions and checks each transaction input according to the SQL injection attacks specifications. As aforesaid in Chapter 2 on the SQL injection attacks techniques, there are some characters that are used frequently in SQL injection like a single quotation character. Thus, if a single quotation character is blocked that means the injection that is based on this character will be blocked as

well.

Thus, key to this detection technique is producing an accurate ITL specification of attacks. The detection results will be used in the investigation of the attack behaviour. Therefore, Anatempura keeps track of each transaction as it checks the current transaction and investigates the related attacks for existing transactions. The following chapters (4 and 5) will describe in detail how the detection and tracking is implemented based on Anatempura.

3.7. Summary

This chapter reviewed temporal logic in general and described various types of them and discussed the differences between them. An overview of the selected logic ITL with its executable tool Tempura has been provided together with its syntax and semantics.

Moreover, Tempura operators and their common use with clarifying examples for each operator have been provided. This chapter is concluded with highlighting the Anatempura tool, showing its general architecture. The next chapter provides details of our proposed detection and prevention framework which is based on the Anatempura.

Chapter 4

Architecture of Detection and Prevention Framework

Objectives

- Provides an overview of the detection and prevention framework.
 - Describe the framework phases.
 - Describe the components of the architecture.
 - Give examples of user behaviour.
-

4.1. Introduction

Chapter 2 has introduced SQL injection attacks describing the various techniques that are used to attack a web application. Chapter 2 also discussed the existing approaches that are developed to tackle this problem and discussed existing solutions to block these types of attacks. Chapter 3 has reviewed the Anatempura tool that is chosen as a basis of our framework showing its architecture and describing the underlying logic ITL used by this tool and its executable engine Tempura. Anatempura checks the input data for existing SQL injection attacks by monitoring the web application's inputs using the attack specifications expressed in Tempura. This chapter explains in detail our Detection and Prevention Framework (DPF).

The contents of this chapter are organized as follows: Section 4.2 describes the DPF architecture and provides an overview of the detection and prevention framework. Section 4.3 discusses the initial phase of the DPF. Section 4.4 provides the details of the checking phase that is used in the analysis of the user entry data. Section 4.5 describes the decision phase in detail by specifying the user behaviour and the feedback component. Finally, Section 4.6 provides a summary of this chapter.

4.2. Overview of Detection and Prevention Framework (DPF)

The main aims of DPF is the monitoring and blocking of SQL injection attacks that are used to gain unauthorised access of web applications and their databases. DPF uses Anatempura as runtime monitoring and verification tool to block malicious users inputs. DPF is initialized by specifying existing attack patterns using Tempura. Anatempura will be connected to a web application server to monitor the users input

using the attacks specifications. Figure 4.1 shows the main architecture of DPF.

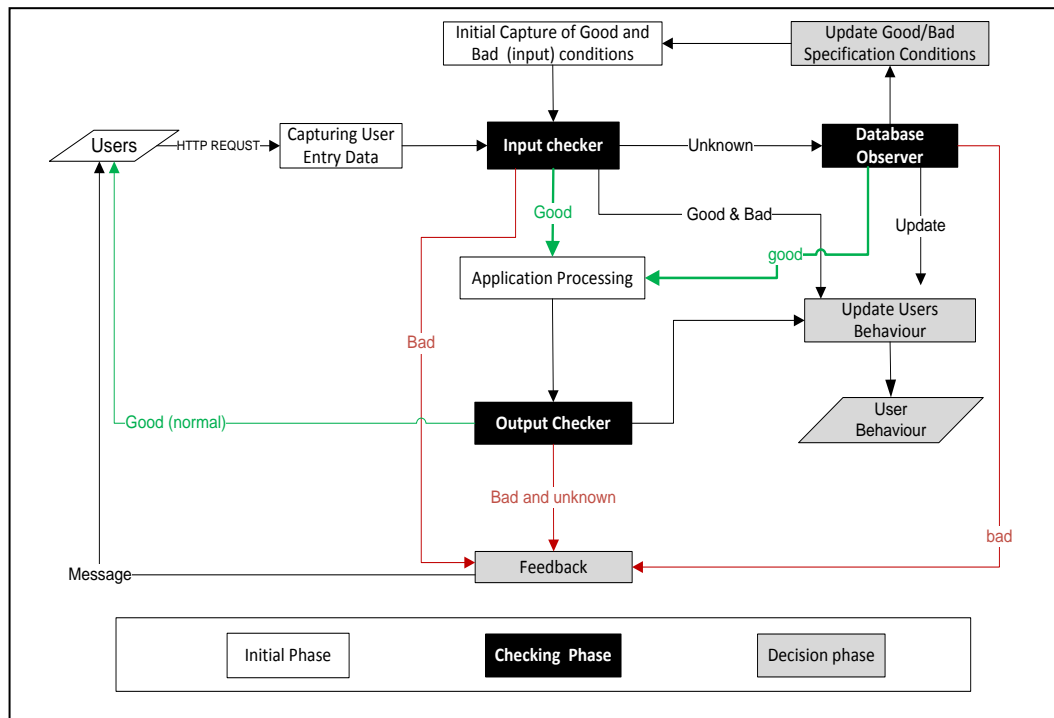


Figure 4.1 The Architecture of Detection and Prevention Framework

DPF starts when the user enters data and submits the web application page, the data will be sent from the client machine to the web server using the HTTP protocol. Moreover, DPF extracts the submitted data in the capture user input component. Extracting the data depends on the hotspots of a web application, which maps the submitted data to its variables in the web application. The hotspots can be detected using one of the existing static analysis tools such as the pixy tool for web applications that are developed using PHP (Jovanovic, Kruegel et al. 2006). Therefore, the hotspots will be assumed to exist. The extracted data will be transformed into Anatepura assertion point format (variable, value, timestamp) so that the Tempura interpreter can use this data to check against existing attack patterns using the input checker component. The result determines whether the input is good,

bad or unknown and this information will be used for the investigation of related attacks.

DPF has three checking components which are input checker, output checker and database observer, and each one of them uses a different technique for checking the user input. The first checking stage uses the input checker to analyse the user input using two steps, the first step will analyse the user input to check whether it is a good input (see Section 4.3.1), if the first step determines that the user input is not good then the second step will compare those inputs with the existing attack patterns.

Thus, if the first step considers the user input to be good then it will be sent for processing by the application processing component. If the user input matches an existing attack pattern, the input will be rejected and the user will be informed by the DPF feedback component as a part of the decision phase (see Section 4.5). In addition, the DPF decision phase has another component that updates the user behaviour database with information that a bad input has been used to attack the web application. Note that the user behaviour will be updated in both cases whether the input was good or bad, and it will be used for investigating related attacks (see Section 4.5.2).

If the input checker cannot determine whether the user data is good or bad then DPF will run the database observer to determine what the effect of the user input is on the database engine. Thus, if the user input is accepted (see Section 4.4.2) by the database observer then DPF will send the user input to application processing component, if the user input is not accepted (see Section 4.4.2) then DPF will reject

the user input and inform the user using the DPF's feedback component, and then DPF will also update both the user's behaviour database and the existing attacks pattern with information that a new injection attack has been used on the web application.

The last step of the checking phase is the output checker which is used to determine whether the message that is communicated from the database engine to the user contains any information about the database type or structure or not (see Section 4.4.3). DPF consists of three phases which are the initial phase, checking phase and decision phase. These phases will be discussed in detail in the following section.

4.3. Initial Phase (receiving Data)

The initial phase consists of several steps that need to be done before data can arrive at the input checker component. The first step in this stage provides the Tempura formula that is used to analyse the user's input. This formula is based on the specification of SQLlib (SQLlib-tool 2007) which is an open source tool. The following specifies the Initial phase and describes its components:

4.3.1. Initial Capture of User Input

The initial capture of user input step component needs to determine the good input and the bad input at character level as shown in Figure 4.2.

Good input should not include any bad symbol like single quotation and double quotation or star, or the good input should not contains any of the SQL keywords that can be used to attack the web application database. Furthermore, the bad input will be specified by describing some of the existing attack patterns like union query,

piggyback query, and tautology etc.

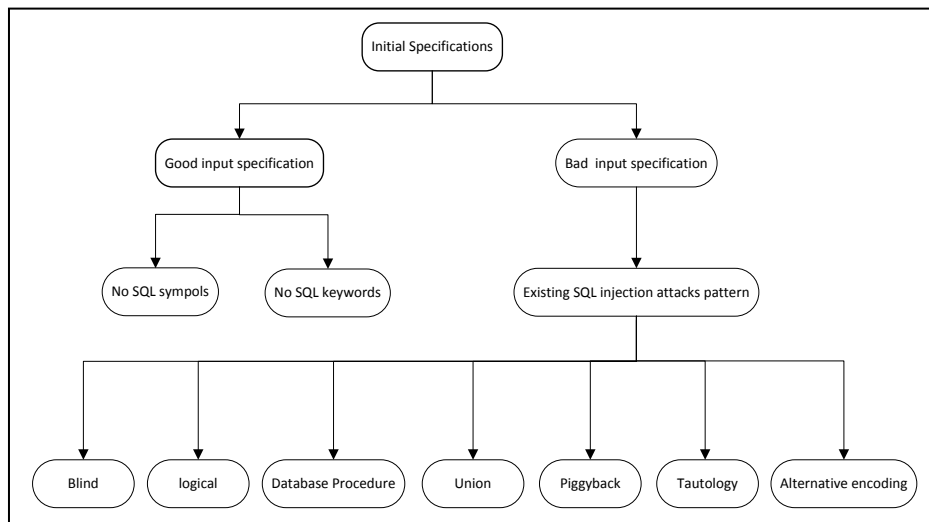


Figure 4.2 Initial specifications

The existing attacks specifications will be merged for any attack that uses the same injection character which means that the specification formula does not describe each type separately.

For example, the blind attacks can be done using a single quotation similar to a piggyback attack, so the specification of the attacks that involve a single quotation do not need to be in a separate detection formula, and both attacks can be detected using a specification that covers the attacks that use a single quotation. Therefore, the initial bad/ good specification will be used by Anatepura for initializing the DPF.

4.3.2. Users

In DPF the users are considered to be a part of the initial phase because any transaction will be started from the user. Thus, the user is anyone who submits an HTTP request to the web application. So, the user would be good, bad or unknown. Therefore, there are no beforehand assumptions proposed for users behaviour.

4.3.3. Capturing Data

At this stage the system will analyse the HTTP request to extract the user input data, then the extracted data will be transformed into Anatempura's assertion point format. Moreover, Anatempura normalizes this data using two procedures that transform it in a style without extra spaces and only lower case characters. This normalization step will be useful to block any attacker who tries to use extra spaces or upper and lower characters as attack methods.

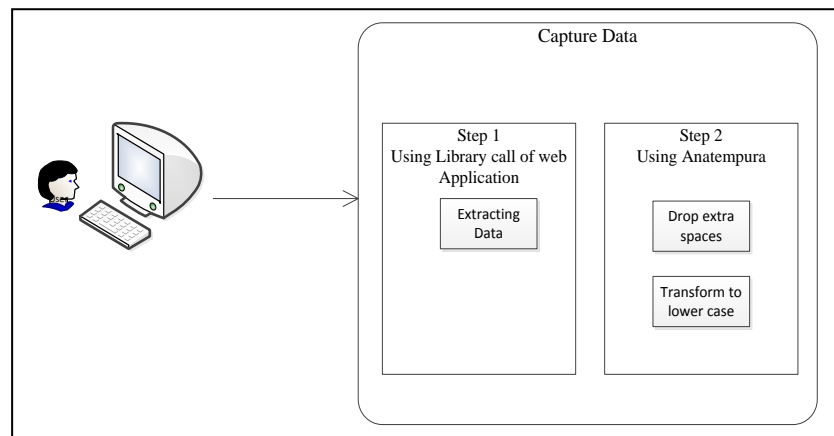


Figure 4.3 Capture Data component

The capturing Data component consists of two steps which are the extract data and normalize data steps. This extract data step does not involve Anatempura but is performed at the web application side. In other words, the data will be extracted by using a library call offered by the programming language that is used to develop the web application.

4.4. Checker Phase

In this phase, the system will analyse the data that comes from the capturing data component. The phase will use three checking components and these are as follows:

4.4.1. Input Checker Component

This component is the heart of the DPF as it determines the next step of DPF whether to proceed to the application normal processing component or the database observer component. Moreover, the input checker component will use the existing attack patterns that are prepared by the initial capture of user input component. The input checker component will analyse the user entry against existing attack, and the result is one of three possibilities:

The entry data is good, which means that the user input does not contain any SQL symbols or keywords that are used in existing SQL injection attacks, so the data will be passed to the application server for normal processing, and the user's behaviour will be updated. The other possibility is that the entry data is bad, then the data will be rejected and the user's behaviour will be updated and a message will be prepared to be sent to the user via the feedback component.

The last possibility is that the entry data is unknown; in this case the database observer component determines whether the entry data is bad or good according to effect these data have on the database. The database observer component checks the entry data by validating four conditions that determines whether the entry data is safe or not. The database observer component will be discussed in detail in the next section.

4.4.2. Database Observer Component

This component will check unknown entry cases which are not caught by the input checker component. The purpose of the database observer component is to determine

what exactly will happen to the database on the transaction of user input, and this will be done by monitoring the outcome of each database transaction.

The monitoring of the database transaction needs the web application developer because the database observer component needs the developer to specify the expected result of each transaction that is run by the web application such as, the table name, running command type, number of the expected records, and the user type.

The expected result of a database transaction will be compared with the runtime result. The comparison between the expected result and the runtime result is used to ensure that the database transaction is safe (if the runtime result is similar to what the developer expected) as shown in the Figure 4.4. In case of a unsafe transaction the database will be rolled back to the state before this unsafe transaction.

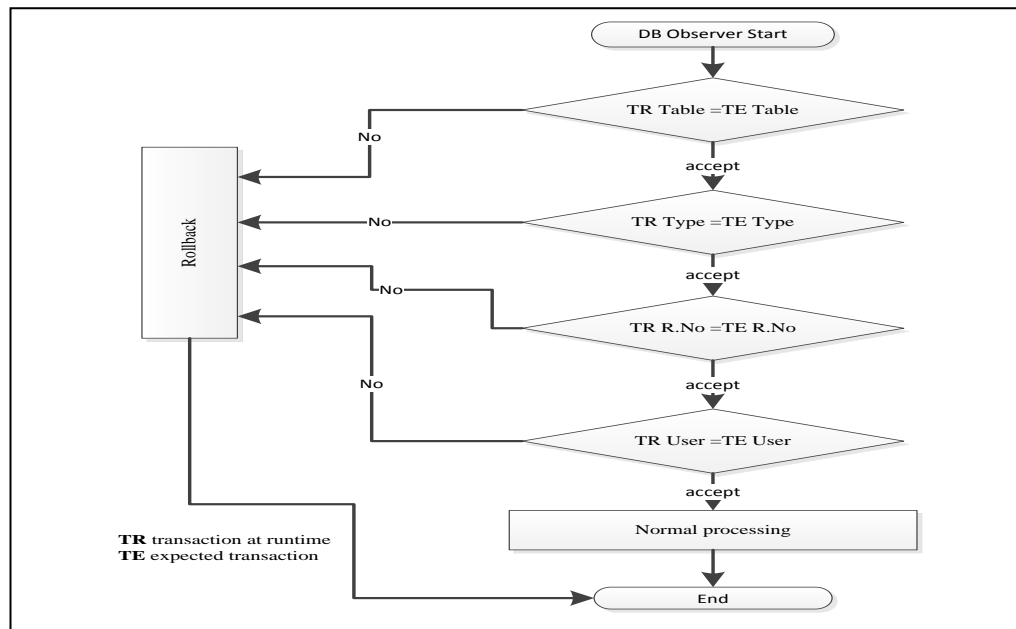


Figure 4.4 Database Observer

Therefore, the database observer has to monitor four conditions for each transaction;

those conditions are specified by the web application developer as follows:

- Transaction type at runtime is the same as the expected one specified by the developer. For example, if the transaction type at runtime is “Select” and the expected one is “Select” as well, then the database observer component will accept this transaction and continue. If they are different the database observer component will do a rollback and prepare a feedback message for the user and update the user’s behaviour.
 - The transaction table name at runtime is the same as the expected one that is specified by the developer. For example, if the transaction table name at runtime is “users” and the expected table name is also “users” then the database observer component will accept this transaction and continue to the next step. If they are different then the database observer component will do a rollback and prepare a feedback message for the user and update the user’s behaviour.
 - The transaction record number at runtime is the same as the expected number that is specified by the developer. For example, the login page normally returns one record with the select statement, so if at runtime the select statement returns the same number (one record), then the database observer component will accept this transaction, otherwise it will be rejected and the database observer component will do a rollback and prepare a feedback message for the user and update the user’s behaviour.
 - Transaction user type at runtime is the same as the expected one that is specified by the developer. For example, the user tries to change the password at the change
-

password page then the user type should be same to the expected one. However, if the user tries to change another user's password then the database observer component will catch this, then this transaction will be rejected and the database engine will do a rollback and prepare a feedback message for the user and update the user's behaviour.

Note the database observer can only deal with recoverable transactions so no DDL (Data Definition Languages) commands like create, drop, and alter table, because the injected DDL commands cannot be recovered by a rollback command, so these transactions should already be rejected by the input checker component.

4.4.3. Output Checker Component

This component checks whether the message sent to the user is safe or not. The output checker will not analyse the response in the same way as the input checker, as it will block any message that contains details about the database structure or type because these types of messages are not safe. Moreover, the output checker will block unsafe messages using the library calls in the programming language which is employed during the development of the web application.

4.5. Decision Phase

This phase of DPF depends on the results of the input and output checking phases, and consists of the feedback and user's behaviour components.

4.5.1. Feedback Component

The feedback component prepares the message that will be sent to the user regarding the cases of bad entry data. If the user entry is bad and the input is caught as unsafe

then DPF will respond to this entry by using a prepared message depending on the type of badness.

4.5.2. User's Behaviours Component

SQL injection attacks usually consists of several steps, this component focuses on the main point of this research which is how to investigate the user behaviour according the history of entry data. DPF will track each transaction in the system and detects the type of the transaction, i.e, whether it is a good, bad or unknown transaction. The tracking information will be used to model the user behaviour. Therefore, the user behaviour depends on the result of the input and output checker components in addition to the result of the database observer component. The DPF will use user information like IP address, user status (good, bad), attacking technique (primitive, advance), and time stamp of the transaction to build the user behaviour as shown in

Figure 4.5.

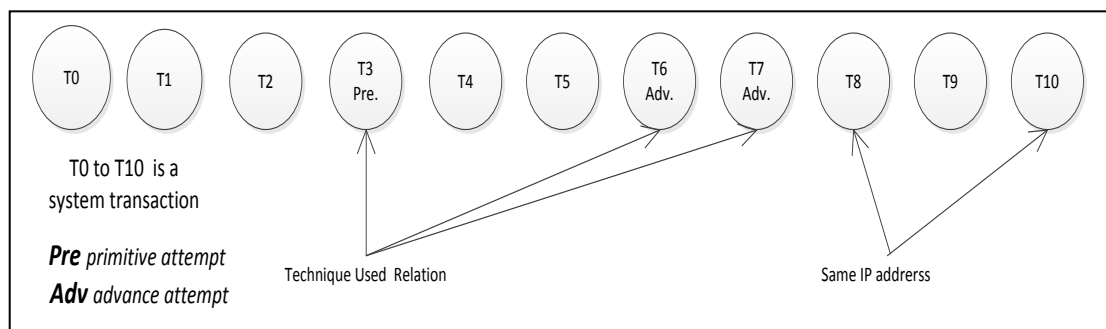


Figure 4.5 Transactions Relation

Figure 5 shows a sample of a web application transaction that contains both good and bad attempts as determined in the checking phase. Transaction 3, 6 and 7 are related as T3 is start SQL injection attacks using a specific technique, and T6 and T7 are

Chapter 4 - Architecture of DPF Framework

based on T3 as they cannot be executed if T3 does not inject successfully. T8 and T10 are related as they have come from the same IP address. The following ITL formula determines whether two bad inputs are related by IP address:

$$\exists IP. \diamond (Status(Input) = Bad \wedge IP(Input) = IP); \diamond (Status(Input) = Bad \wedge IP(Input) = IP)$$

This means if the IP of the user in a certain state is equivalent to the IP of the user in a previous state then these inputs are related. Another example of related attacks is shown by the following ITL formula which determines whether two bad inputs are related “by stored procedure “.

$$\exists command. \diamond (Declare (Input) = command); \diamond (EXEC (Input) = command)$$

This means if there is an execution of a stored procedure command in a certain state and the declaration of that stored procedure in a previous state then these inputs are related.

Moreover, the DPF can determine user behaviour according to the following three criteria: the percentage of transactions, the sequence of transaction type, and the transaction types as shown in Figure 4.6.

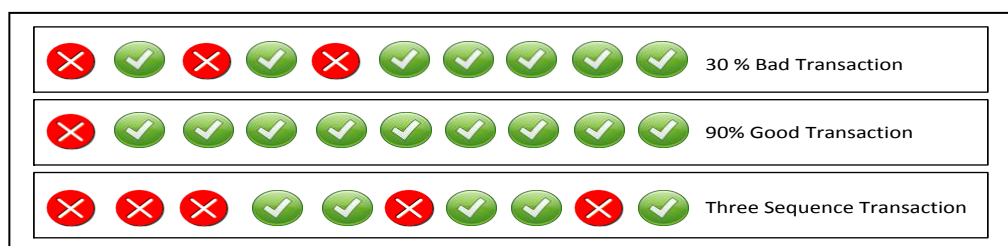


Figure 4.6 Examples of User Behaviour

Figure 4.6 shows several user behaviour graphically. Therefore, user behaviour can be used as a quick way to view the transactions status and the proportion of hacking

attempts that have been done so far. It can act as an early warning system.

4.5.3. Example of user's Behaviour

This example illustrates how attacker's information can be used to model user behaviour. This example describes an input scenario and assumes that the status of each input is already determined by the DPF checking phase as mentioned in the checking phase of Section 4.4. Table 4.1 shows a particular input scenario that involves several sample user inputs.

Seq.	User IP	The input	status
1	146.168.255.12	Normal	g
2	146.168.255.13	Normal	g
3	82.164.254.12	' or '1'='1	b
4	82.164.254.12	';drop table users;--	b
5	212.164.254.14	Normal	g
6	212.164.254.16	Normal	g
7	212.164.254.14	any'; declare @NewStoreProcedure char(80);	g
8	67.164.254.14	Normal	g
9	146.164.2.46	' ;	b
10	212.164.254.14	Normal	g
11	182.164.254.23	any'; EXEC (@NewStoreProcedure);	b
12	212.164.254.14	Normal	g
13	212.164.254.16	Normal	g

Table 4.1 Selective User's Inputs

In Table 4.1, input 3 and 4 are marked as bad, those attempts are one step attacks because they do not retrieve any information from the database and just try to inject the harmful code in the web application fields. However, those attempts have the same IP address which means there is a relation between them because both attempts have been done by the same user. Input 7 and 11 can be classified as related as well, because the attacker here declares the stored procedure in the first attempt and in the

second attempt he/she uses it. So there is a relation between these hacking attempts and this justifies the use of monitoring user behaviour.

4.5.4. Updating of User's Behaviour Component

The user's behaviour will be updated continuously within DPF according to the result of the checking phase. The updating will involve all of the user's inputs types (good, bad). Thus, this component will update behaviour according to the three criteria used to investigate the sequence of bad transactions: the percentage of the checking result type, related IPs and related techniques.

4.5.5. Updates Existing Attack Patterns Component

This component receives new attack patterns from the database observer component. When the database observer finds any unsafe input, it will send the input to this component to update the existing attack patterns. Note the updating of the attack patterns library will be manually, because the library that is used by input checker is specified in ITL, thus the updating uses manual translation into ITL by anyone who expert in ITL.

4.6. Summary

An overview of the architecture of our framework DPF has been presented in this chapter. The framework phases and the components of each phase have been discussed. This chapter also describes the task of each component in detail and how these components will interact with each other. The user's behaviour has been discussed in detail using a clarifying example. The following chapter will present the implementation of the DPF.

Chapter 5

Detection and Prevention Framework

Implementation

Objectives

- Provide the reasons of selection of tools for the implementation.
 - Provide the architecture of selected components.
 - Present the implementation of each component.
-

5.1. Introduction

The previous chapter has described the main structure and processes of DPF that are proposed to check the submitted data that comes to the application server through http requests. This chapter describes in detail how the DPF is implemented and it is organized as follows, Section 5.2 describes the DPF implementation assumptions in order to realize DPF. Section 5.3 describes the implementation of all the components of the DPF. Section 5.4 gives the summary of this chapter.

5.2. Implementation Assumptions

The existence of different types of programming languages and DBMS that can be used for creating and developing web application is a reason for choosing a specific environment to implement our framework. The implementation is used to determine the interaction and the compatibility between the components and to know exactly the effectiveness of Anatempura with this environment as a runtime monitoring tool. Additionally, SQL injection attacks normally depend on the type of DBMS that is used as application repository, because some of the SQL commands work only for a particular DBMS. For example, a MSSQL database can be injected using single quotation, or semicolon, or double dash --, /* ... */ and xp_ (for stored procedure catalog name) characters (MSDN 2008).

Thus, the development language that is chosen is PHP and the DBMS is MYSQL. This selection is based on the fact that PHP and MYSQL are free resources and they can be installed together using one execution file like 'WampServer' (Bourdon. 2013). Our choice of MYSQL means that we focus on the injection possibilities that

affect this database type which are restricted as follows (Matsuda, Koizumi et al. 2011, Clarke 2012):

- Using a semicolon.
- Using a single quotation as a character data and string delimiter.
- The comment delimiter is either hash mark ‘#’ or inline comment `/* ...*/`.
- The encoded character using ‘0x’ for executing hexadecimal code and ‘%’ percentage.

Therefore, the implementation will focus on these as they are key to SQL injection attacks. In addition, the implementation is created to test the SQL injection attacks for web application, thus we assume that hotspots of the web application have been determined before using existing tools.

5.3. DPF Components Implementation

DPF has several components which were already described in the previous chapter. The following will be implemented first: capturing Data component. This component will be used to extract submitted data and send them to the input checker using a library call in the PHP development language. The second step will be the usage of the extracted data which are a user IP address, submission time stamp, and a user input. This information will be sent to Anatempura using the PHP-Java Bridge (Bökemeier., Koerber. 2012) as the Anatempura cannot communicate directly with an PHP application. The bridge supports transportation of data between PHP and Java application at runtime and sends the extracted data to a Java application and this

Java application sends them to the Anatempura tool. The Java application that communicates with the Anatempura is implemented using Java RMI (Remote Method Invocation) (Oracle. 2012) as shown in Figure 5.1.

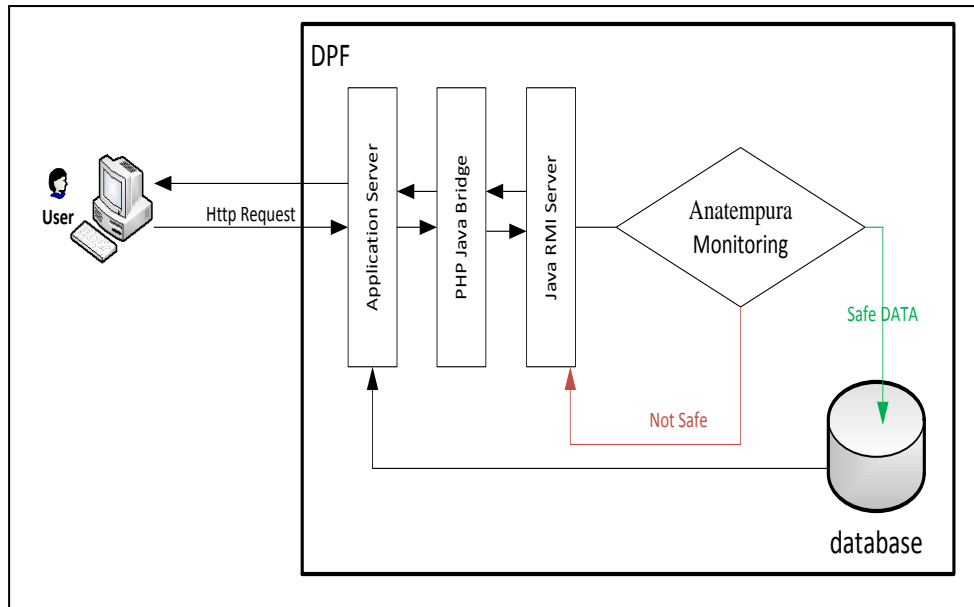


Figure 5.1 Implementation General Architecture

This Java application is running as a server that can communicate with a PHP application using the bridge. Thus, the extracted data will be analysed by the input checker that is executed by Anatempura to detect the input status and decide which data is safe or not before passing it to the application database. Additionally, there are other processes like the user behaviour and database observer which will be described later on. The implementation of all of the mentioned components will be explained in detail in the following Section.

5.3.1. Capturing Data Component

This component prepares the data to be analysed by the Anatempura tool, so the submitted information that comes to the server via HTTP requests will be

reformatted in Anatempura format which is (IP address, submitted data, submission time) as shown in the Figure 5.2.

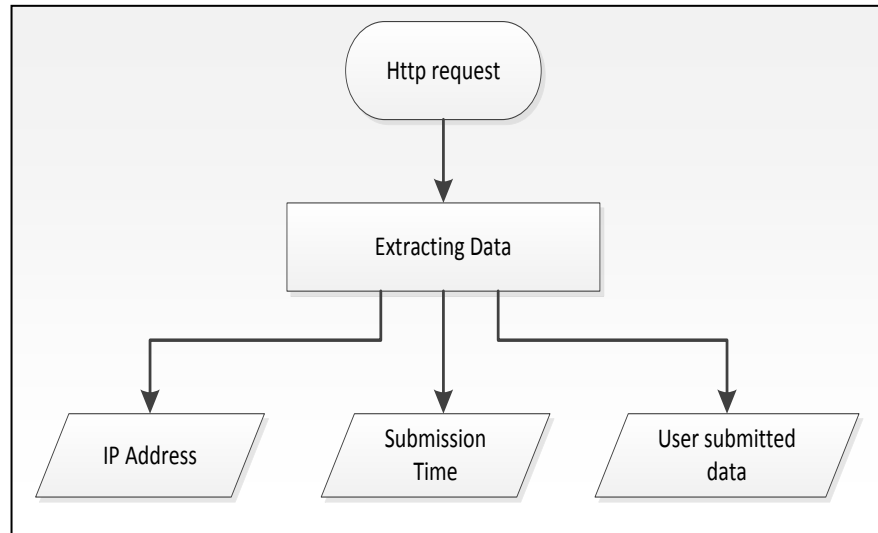


Figure 5.2 Http Request Extracting Data

The discussed information will be extracted using the following PHP code to extract the value for a specific variable in a HTTP request:

```
Value= $_REQUEST ["username"]
```

The value here for a variable called 'username'. The submission time will be extracted using the following:

```
Time=Date ("h:i:s A")
```

The user IP address can be extracted by:

```
UserIP = $_SERVER ["REMOTE_ADDR"]
```

As aforesaid, the extracted information will be sent to the Java application and then to Anatempura using the PHP-Java Bridge. The java application (RMI) that transfers

Chapter 5 - DPF Implementation

the data between Java and PHP consist of three files which are ServerImpl for the implementation of the server methods, Server to define the RMI server method, and Client to communicate with PHP. After installing the Bridge, PHP initializes an instance of 'Client' and calls it by the value that will be checked using the script as shown in Listing 5.1.

```
$VarJava = new Java("Client");  
$Result = $VarJava ->CheckValue($Value_for_Checking);
```

Listing 5.1 PHP script: Sending Value from PHP to Java

Thus, the object 'VarJava' can invoke the 'Client' class methods. The Client class calls the checking method of the RMI server called CheckInput() that is implemented as shown in Listing 5.2.

```
public String CheckInput(String sentvalue) throws RemoteException  
{ try {  
    Console c = System.console();  
    String in ;  
    System.out.println("!PROG:assert V_check:"+sentvalue+":Timestamp :!\n");  
    in = c.readLine("read it \n");  
    System.out.println("Tempura Result: "+in);  
    return in;  
}
```

Listing 5.2 Java Code: Checking Input Method

The CheckInput() contains an assertion point that is used to communicate with Anatempura, and thus the input data will be transferred to Anatempura for checking.

Chapter 5 - DPF Implementation

Anatempura receives the inputs using the *get_var* procedure. The input will be inspected using the *CheckingModel* procedure, and the result will be returned to the application Java using the *prog_send_ne(X)* procedure as listed in Listing 5.3.

```
while (Loop = 1)
  do {
    {get_var("V_check ", NewValue) and output(NewValue) and
    CheckingModel(X,NewValue) and stable(NewValue)};
    {prog_send_ne(X) and skip and
    if (NewValue ='ex1') then Loop := 0 else Loop :=1 }
  }
```

Listing 5.3 Tempura Code: inspecting and Sending Data to Java

The reason of using the Java RMI application is that Anatempura can start and monitor the Java application and it will communicate with the web application server (server to server).

The web application's variables and hotspot points are known as the application has been implemented for testing the effectiveness of Anatempura in monitoring submitted data against SQL injection Attacks. So, the Anatempura tool can be used to monitor an existing application as long as the application's variables and hotspots are known beforehand. The detection of these variables and hotspots can be done by using an existing analysis tool like the Pixy tool (Jovanovic, Kruegel et al. 2006) for PHP applications.

Therefore, extracted data is received by Anatempura and can be analysed by the input checker. Anatempura receives the data in array style and analyses the submitted data only and determines the status, i.e., whether it is safe or not. The analysis of the

submitted data is based on the initial capture component that prepares the data before the analysis stage using two procedures as shown in Figure 5.3.

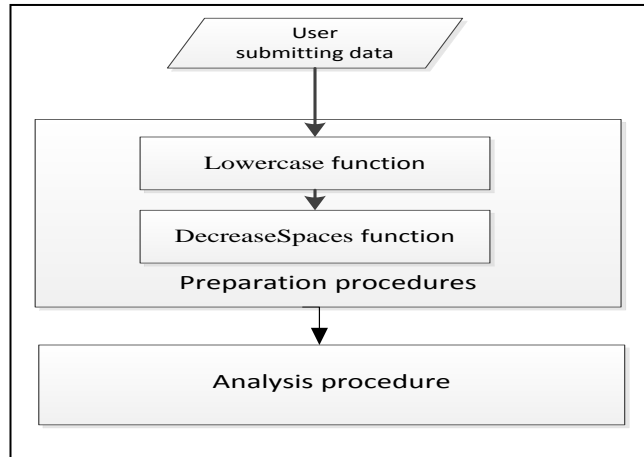


Figure 5.3 Preparing Procedures

There are two preparation procedures which are:

- **Lowercase** procedure
- **DecreaseSpaces** procedure

These procedures run in sequence for every transaction, and they utilize predefined functions like `unascii` which is used to give the corresponding character an ASCII value. For example, if the function called like `unascii (57)` the function will return the character 'W', and this function will be used in the lowercase function in the preparation procedure.

Lowercase is one of the preparation functions and it is used to transform the user input into lower case. Normally, when this function is called it starts with an input string and converts this string into a string of lower case characters using the ASCII code of input string characters.

DecreaseSpaces is a preparation function that is used to remove any extra spaces in the user input. This function has two steps, the first one determines all non space characters in the string, and the next step is to restructure a sequence of characters in such a way that the extra spaces will be converted to one space character, and then the function will return this restructured string. The code of these functions is in the Appendix 3.

5.3.2. The input Checker

The input checker receives the prepared data from the capturing data component so that it can be analysed. The analysis functions inspect the content of the inputs and determine if those inputs contain any form of SQL injection attacks. The first step of the input analysis splits the input tokens and it is followed by other procedures like **SearchGenKeywords**, **GoodEntry** and **BadEntry** as shown in Figure 5.4.

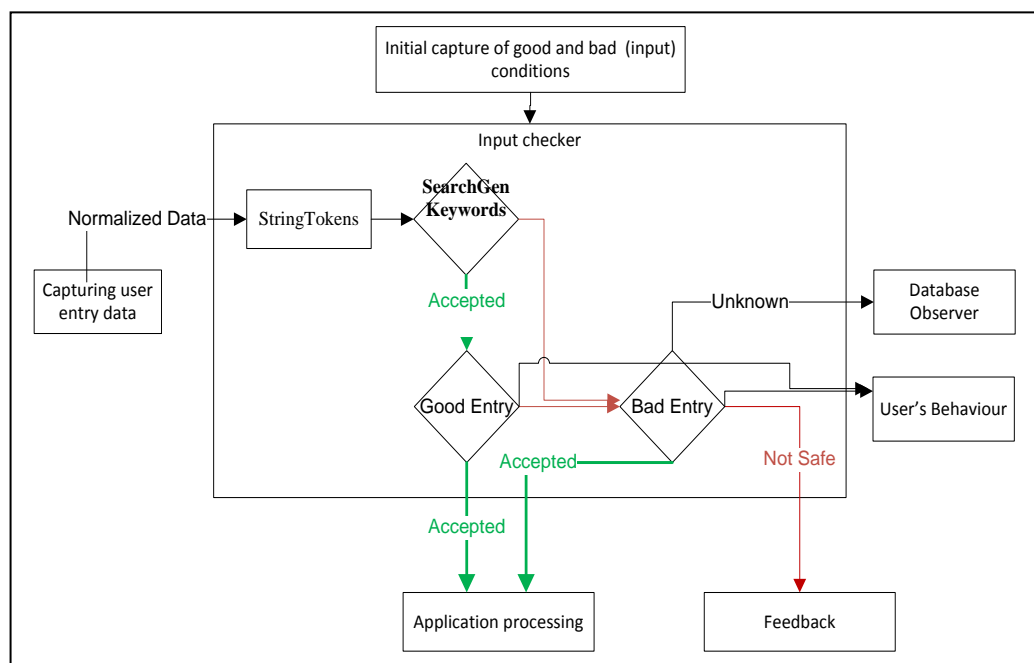


Figure 5.4 The Input Checker

Chapter 5 - DPF Implementation

The **StringTokens** function has two steps, the first step determines a non space character of the input string, and the second step divides the input string in an array of tokens which can be used to search for specific SQL Keywords. This function uses the same technique as the DecreaseSpaces function. However, StringTokens differs in the second part during the restructuring of the parts of the string. Table 1 shows an example of the first part of the function.

'		u	n	i	o	n		s	e	l	e	c	t		*		f	r	o	m		t	a	b	l	e
0	2	3	4	5	6	8	9	10	11	12	13	15	17	18	19	20	22	23	24	25	26					

Table 5.1 Input String S

The other part of the function transforms a string S into an array of keywords called 'A1' that involves all tokens of string S. The result of the example will be

A1= ["", "union", "select", "*", "from", "table"] .

The **SearchGenKeywords** function is used by the main two functions that analyse the user input which are GoodEntry and BadEntry procedures. When this function is called it searches for the most common SQL injection keywords that come after an SQL injection symbol like semi colon and single quotation. The function starts by transforming it into an array of token using the **StringTokens** function. The returned array of string tokens will be compared with common SQL keywords that can be used in a SQL injection attack, like ["select", "drop", "update", "delete", "alter", "create", "union", "declare", "exec", "insert"] as part of the analysis of the user input. The comparison will be done between two arrays which are a string token array and a

common SQL keywords array and the result will be either 'y' for yes or 'n' for no.

The **SearchSpecKword** function is used by the **BadEntry** function to compare the token that comes after the SQL symbol with a special token that is normally used after this symbol, like union after a single quotation. In other words, this function checks the input against existing attacks patterns that use a single quotation. The function has two input variables S and D. Variable D is an array of the comparable tokens that will be used to check what comes after the SQL symbol. The S variable is the part of the user input that comes after the SQL symbol. So, if the input is `input = " any ' or '1' ='1 "`, the S variable will be `S = or '1'='1`. This part of the user input is transformed into a list of tokens using the Q array. Listing 5.4 shows the checking part of the function which compares the first token of Q with D that contains a possible injected keyword that can come after this SQL symbol.

```
stable(Q) and I=0 and J=0 and A='n' and G=0 and {
while I < |D|
do{ if (D[I]=Q[0])
then{I:=|D| and skip and A:='y' and G:=|Q[0]|}
else { I:=I+1 and skip and A:=A and G:=G}
```

Listing 5.4 Tempura Code: SearchSpecKword function

This function returns as result an array $X = [A, G]$. The G variable is used to return the length of list of the token Q that can be used to determine the remaining part of a string S, and A is a variable used for returning the comparison result. Moreover, the

result of this function is determined to be 'n' or 'y' in the same way as the SearchGenKeywords function. The result will be used in the next step of the input checking, namely the CheckingModel procedure (Listing 5.8).

GoodEntry function is one of the main functions for analysing the user input and it is the first analysis stage that checks whether the user entry contains any SQL keywords that can be used in SQL injection attacks or not. The function has as input a normalized string and first checks for a hex encoded SQL injection technique by comparing the user input on the character level with a key of this type of injection like '0x ...', if it is not matched the analysis process will continue checking for input characters in the range of '0 to 9' and 'a to z' using an ASCII code. Additionally, there are some symbols that are considered safe, because they are not used as a key in this attack, the safe symbols are shown in Table 5.2.

Seq.	Symbol	Character ASCII code
1.	space	32
2.	!	33
3.	\$	36
4.	.	46
5.	:	58
6.	?	63
7.	_	95
8.	£	163

Table 5.2 Safe Symbols

The choice of the above characters is because they can be used as part of user input when using a web application. The analysis steps will be executed sequentially for each character, thus if any of the mentioned steps has a positive match the function

will return 'n' which means the input is not safe.

The **BadEntry** function is the second step of the analysis that checks user inputs and compare it against existing attack patterns. The function has several parts, one checks for the SQL injection keys such as a single quotation as a character delimiter, semi colon as query delimiter, hash symbol as comment delimiter and a back and forward slash with a star as a comment delimiter in addition of other encoded injection attacks.

The part that checks if the input contains a double dash characters is as follows:

```
if ((S[I..I+2] = "--" and (I+2 <= |S|)) or
    (S[I..I+3] = "- -" and (I+3 <= |S|)))
then {I:=|S| and A:='b'}
```

Listing 5.5 Tempura Code: Checking for Double Dash Characters

The variable S contains normalized user input and the variable I is the index of the current user input character. The code shows that the checking of this part is based on comparison between the sequences of characters in S with double dash characters with or without space in between the double dash. If there is a matching then it will return 'b' which means the input is bad. Note in the loop the value of I should not exceed the length of S, so the last possibility to find the double dash comment will be |S| minus 3 or |S| minus 2 and that depends on the form of the double dash.

The second part will check for SQL injection attacks using a single quotation using several steps; the first step is to check for a single quotation, the second step will check the token that comes after a single quotation, and the last step checks what

comes after that token.

The code below shows the second step to find the token written after a single quotation:

```
SearchSpecKword(X,S[I+1..|S|],["union","or","and","group","order"])
```

This code checks for common tokens used to inject web applications after a single quotation. If SearchSpecKword returns that there is no matching, the loop will resume from the 'I+1' character of string S to check if there is another single quotation. If SearchSpecKword returns that there is a matching then what comes after this token will be checked using another string index D that starts from I+1 to |S| to save the value of I if there is no matching.

```
While (D < |S|)  
do {if (S[D]= ";" or S[D]="=" or S[D]="-" or S[D]="'" or S[D]=">" or S[D]="<"  
or S[D..D+6]="select" or S[D..D+3]="all")
```

Note, the code above is checking for characters that can come after the injecting token and if there is a matching the entry will be considered as an attack and it will be rejected, because the mentioned characters and token are used in SQL injection attacks.

The second part will check whether the input includes a semi colon and check for attacks that can be done using a semi colon. The process of the checking will be done in several steps, the first step checks for a semi colon, the second step will check the token that comes after a semi colon, the last step checks what comes after that token.

Chapter 5 - DPF Implementation

The code below shows the second step to check the token after a semi comma using SearchSpecKword:

```
SearchSpecKword(X,S[I+1..|S|],  
["select","drop","update","delete","alter","create","declare","bigen","exec"])
```

The code above checks for tokens used to inject web applications after a semi colon. If SearchSpecKword returns that there is no matching, the loop will resume the checking process from the 'I+1' character of string S. If SearchSpecKword returns that there is a matching then it will check what comes after this token via string index D that starts from I+1 to |S| to save the value of I if there is no matching as shown in the following:

```
while (D < |S|)  
do {if (S[D]= ";" or S[D]= "#" or S[D]= "--" or S[D]= "" or S[D]= "*" or  
S[D..D+5]="table" )
```

The code above checks for the character that comes after the injecting token and if there is a matching with any of those possible characters and token, the entry will be considered as an attack and it will be rejected, because the mentioned characters and token are used in SQL injection attack.

The last part will check whether the user input contains any of the alternative encoded injection techniques using the most widely used encoding.

```
while I < |S|  
do{ if ((S[I..I+3]= "0x" or S[I..I+3]= "(0x" or S[I..I+4]= "( 0x" ) and  
(((ascii(S[I+3])>= 48 and ascii(S[I+3])<=57) or (ascii(S[I+3])>= 97  
and ascii(S[I+3])<=102)) and ((ascii(S[I+4])>= 48 and  
ascii(S[I+4])<=57) or (ascii(S[I+4])>= 97 and ascii(S[I+4])<=102))))
```

Listing 5.6 Tempura Code: Checking for hexadecimal encoded injection

Chapter 5 - DPF Implementation

Listing 5.6 shows the code that is used to detect attacks that use a hexadecimal encoded injection. The idea of the mentioned code is to block strings that starts with zero followed by character x and after it the hex encoded SQL command. This checking step inspects the input for '0x' and two characters that come after it, if the inputs characters have the ascii code from 48 to 57 ('0' to 'f') or from 97 to 102 (A to F) which are the hexadecimal injection key the inputs will be rejected and it will be considered as a bad input.

Additionally, there are other types of alternative encoded injection and these will be checked using the following:

```
if (S[L..I+6]= " char(" or S[L..I+7]= " char(" or S[L..I+5]= "char(" or S[L..I+6]= " exec("
or S[L..I+5]= "exec(" or S[L..I+6]= "select(" or S[L..I+7]= "select (" or S[L..I+3]= "%00"
or S[L..I+3]= "%2f" or S[L..I+3]= "%2a" or S[L..I+5]= "%252f" or S[L..I+5]= "%252a" )
```

Listing 5.7 Tempura Code: Checking for Alternative Encoded Injection

The code above checks for SQL injection used in:

" char(" which is concatenating characters in ASCII code. Note, there are several forms of the mentioned characters to consider different cases of spaces that can be used to elude this checking.

"select(" to catch 'select' keyword that is entered with bracket as one token.

"exec(" to catch 'exec' keyword that is entered with bracket as one token.

"%00" is a null character that is used to confuse the checking process of the position of the token.

"%2a" and "%2f" which are used to encode the inline comment character using Unicode style to escape from the normal checking of the character '*' or '/.

5.3.3. Behavioural Functions

These functions investigate the relation between previous user inputs by checking the list of results that is created during the checking process. The list 'H' contains the submitted input, submission time, an IP address and status whether it is good, bad, or unknown. The behavioural functions implement the four possible relational attacks that are defined in Chapter 4. The code below shows the implementation of the first possibility which is the percentage of the bad, good or unknown inputs.

```
T=0 and I=0 and for i<|H| do {  
    if H[i][1] = 'b'  
    then {T:=T+1 and I:=I }  
    else {T:=T and if H[i][1] = 'OB' then I:=I+1 else I:=I}
```

The variables 'T' and 'I' have been used to count the percentage of bad and unknown inputs, and the good inputs can be calculated using these values.

```
G=itof(T)*$100$/itof(|H|) and  
D=itof(I)*$100$/itof(|H|) and  
format("the percentage of bad inputs is: %F \n ",G) and  
format("the percentage of unspecified inputs is: %F \n",D) and
```

The second behavioural function will check for repeated IP addresses that are used by the attacker who inject a malicious code in the application. The first step will

determine a bad input attempt using the information of array result 'H':

```
for i<|H| do {  
    if H[i][1] = 'b'  
    then{HH:=HH+[i] and D:=D+[H[i][0]]}  
    else{HH:=HH and D:=D}
```

The variable D is an array used to register the IP of each bad input attempt, and the variable HH is an array used to register the array index of those attempts. The next step will remove the redundant IP addresses that are registered more than one time in the array D using the Filter () function which will return an array X which is an array D without duplicates. The X and HH arrays will be used to create the result that shows in order the IP and the submission time of the input.

```
while I < |X| do{  
    for j<|HH| do { stable (G) and stable I and  
        if X[I]=H[HH[j]][0]  
        then {T:=T+[H[HH[j]][3]]}  
        else {T:=T} and skip};
```

The code above uses several variables and two loops to create the 'G' array that is used to gather the result of related IPs. The first loop uses 'X[I]=H[HH[j]][0]' condition to check if the IP of array X is equal to the IP of array the 'HH' using the H result array of the analysis. The matching IP address will be collected in array T grouped by IP address. The result of this part is as follows:

Chapter 5 - DPF Implementation

```
[IP1,[transaction times],IP2,[transaction times],...,IPn,[transaction times]
```

The third behavioural function will find a sequence of user inputs that is considered to be bad. This part uses the index of result array H to detect a sequence of three bad inputs and it will show the IPs and their submission time.

```
if (H[i][1] = 'b' and H[i+1][1] = 'b' and H[i+2][1] = 'b' and i~=0)
    then {i>0 and if (H[i-1][1] = 'b' and H[i][1] = 'b' and H[i+1][1] = 'b')
        then {G:=G+[i+2]}
        else {G:=G+[i]+[i+1]+[i+2]}}
```

The CheckingModel procedure is the sequential composition of all of the discussed functions as shown in Listing 5.8.

```
define CheckingModel(X,S) = {
    exists D,R,F,T,M:{
        {stable(S) and LowerCase(T,S)};
        {stable(T) and DecreaseSpaces(R,T)};
        {stable(R) and SearchGenKeywords(D,R)};
        {stable(R) and stable(D) and GoodEntry(F,R)};
        {stable(R) and stable(D) and stable(F) and
        {if (D='g' and F='g')
            then {M='g' and empty}
            else {BadEntry(M,R)}}} and fin(X=M)}
    }.
}
```

Listing 5.8 Tempura Code: The CheckingModel Procedure

5.3.4. Implementation of Database observer

This component deals with unknown input cases to exactly determine the response of the database engine regarding this type of user entry. The DB observer is developed to determine new attacks using the application development language to observe those unknown cases. The database observer monitors the transaction to check four conditions as follow:

- Transaction type at runtime is the same as the expected one specified by the developer.
- The transaction table name at runtime is the same as the expected one that is specified by the developer.
- The transaction record number at runtime is the same as the expected number that is specified by the developer.
- Transaction user type at runtime is the same as the expected one that is specified by the developer.

So, the DB observer will be implemented using the PHP language to monitor the conditions of DB observer that are explained in Chapter 4. To implement this part, each transaction should be enclosed by transaction delimiters as follows:

```
function Begin()
{
mysql_query("BEGIN");
}
```

The Begin function will be used to start the DB transaction.

Chapter 5 - DPF Implementation

```
function commit()
{
mysql_query("COMMIT");
}
```

The Commit function will be used to end the DB transaction in addition to save the data into the database.

```
function rollback()
{
mysql_query("ROLLBACK");
}
```

The Rollback function will be used to end the DB transaction and to cancel the execution of a database transaction.

The mentioned functions will be used for the monitoring of each database observer condition.

In this part we implement only two of the conditions as the other two can be done in a similar way. The chosen conditions are the condition of user type and the condition of record number. The recorded number of the execution can be checked using two PHP functions which are `mysql_num_rows()` and `mysql_affected_rows()`. The difference between those functions is that the `mysql_num_rows()` function will be used with the selection command whereas the other will be used with the modification command like delete, update etc.

The user type will be run as a PHP session that starts when the user log in to the system, so at the login page there no need to check this condition because the other

three condition are enough to check the transaction at this page, and it will be used in the other page of the application. The session can be initialized at the login page using the following script.

```
$_SESSION['username'] = $_post["username"];
```

5.3.5. Implementation of Output Checker

This component is used in the last step of the checking of the user input and it will block any message that contains details about the database structure or type. This component is implemented using the PHP language. The PHP program returns the error message of database using the `mysql_error()` function, and the error message number is returned by the `mysql_errno()` function. So, the mention functions will be used to block any unsafe messages as follows:

```
if (mysql_errno() == 1061)
{
    echo "There is a duplicated entry ";
}
```

This code illustrates how to replace a MYSQL DB engine message with one that does not indicate the DB type. For example, if the user enters a duplicate value in the DB, in this case the DB will respond to the user input by ‘ Duplicate entry ‘**User_a**’ for key ‘**UName**’ ‘ and that shows the column name ‘UName’ of the users table. Using above code will change the message into ‘There is a duplicated Entry’. Moreover, the unchanged message indicates a MYSQL database engine. If the message appears like ‘**unique constraint (%s) violated**’ that means the database

Chapter 5 - DPF Implementation

engine is Oracle. In this case the output checker will hide the database message and replace it with another one to block any information that can indicate the DB type. Another scenario is if the attacker tries to gather some information about the database type (if possible), some wrong cases of SQL syntax will be used. Thus, the database engine will respond to those cases with a SQL syntax error. For example, if the input is “ or 1 = 1-- ” and the database type is MYSQL, this input is illegal syntax but it will cause a DB error as follows:

“You have an error in your SQL syntax; check the manual that corresponds to your MySQL server version for the right syntax to use near 'any" at line 1”. Error no: 1064

Therefore, this type of message shows the DB type and it can be blocked by choosing another message as follows:

```
if (mysql_errno() == 1064)
{
    echo “Wrong entry, try again! ”;
```

One way to handle the different types of the DB error cases is by using the PHP switch statement as follows:

```
switch (mysql_errno())
{
    case 1064:
        echo “Wrong entry try again! ”;
        break;

    case 1061:
        echo “There is a duplicated entry”;
```

```
        break;

    default:
        echo "Wrong entry try again";}
```

5.4. Summary

This chapter presented the implementation of the DPF components and gave a detailed explanation of the implementation of each component and the relation between each of the DPF components. The selected implementation tools and programming languages are discussed with a justification of our choice. Moreover, the implementation of the Checking Module procedure in Tempura is explained in detail. The next chapter will discuss the feasibility of our implementation discussing the strengths and limitations of DPF.

Chapter 6

Detection and prevention framework

Evaluation

Objectives

- Discuss the evaluation criteria.
 - Provides several samples of user input and its expected result.
 - Presents and discussed the evaluation result of each DPF component.
 - Compare the DPF approach with the existing approaches.
-

6.1. Introduction

The previous Chapters 4 and 5 have discussed the research framework and the implementation of our approach DPF. Thus, the success of the goal of the research depends on the evaluation results and the successful execution of DPF components. This chapter will discuss the evaluation of DPF that checks the DPF effectiveness and the interaction between DPF components in addition to evaluate each component individually. The evaluation will be done using the Anatepura tool with a sample of user input. This chapter is organized as follows: Section 6.2 discusses the evaluation criteria that will be used to evaluate DPF components. Section 6.3 tests the interaction of different software used to test the DPF framework. Section 6.4 presents the results of input checker examination and discusses various samples of input data as well. Section 6.5 discusses the testing result of database observer component. Section 6.6 shows the testing results of the output checker component. Section 6.7 tests the behavioural functions by investigating related attacks and discusses their results. Section 6.8 compares the DPF evaluation results with some of the existing approaches. Finally, Sections 6.9 summarises this chapter.

6.2. Evaluation Criteria

Focusing on a special environment is required to evaluate the effectiveness of this research, and using specific criteria that specifies the successful measurement. Accordingly, we are focussing to choose same environment for the evaluation of our framework that is mentioned in the implementation chapter, i.e., using PHP as development language and MYSQL as database engine. In this part the effectiveness

Chapter 6 - DPF Evaluation

of DPF will be measured for each DPF component using various samples of user input and we are looking to check the following:

- The implementation feasibility.
- The success of input checker using as measurement the rate of false positives and false negatives.
- Handling of new attacks.
- Matching of attacks behaviour.

6.3. Real Web Application Testing

This part tests the interaction of different software used to test the framework on a web application. The testing starts by booting the Java RMI server using Anatempera code as explained in Chapter 5. The Wamp server is also up and running. The testing will be done by submitting a sample of login page that contains two login fields. Figure 6.1 shows an example of a web application page that contains a user input sample.



Figure 6.1 Web Application Input Sample

Chapter 6 - DPF Evaluation

The submitted data is being sent via PHP page to the Java RMI server and then to Anatempura. The checking process for this page will run two times as this page contains two fields. Anatempura checks the user name field first and the password field second. The user name field has any user name, and the password is injected by SQL injection and thus these entries will be rejected and a feedback message will be sent to the user. Therefore, different types of inputs have been considered to check the leverage of the input checker component in catching the common SQL injection attacks. The examination checks also the input checker ability for determining the safe input as there are several types of safe input that can be used by the user as normal entries. The data that is submitted from the web application will appear in the external tab of the Anatempura interface. Figure 6.2 shows the external tab page of the Anatempura interface. This tab page contains the RMI server starting messages, and the analysis results of submitted data.

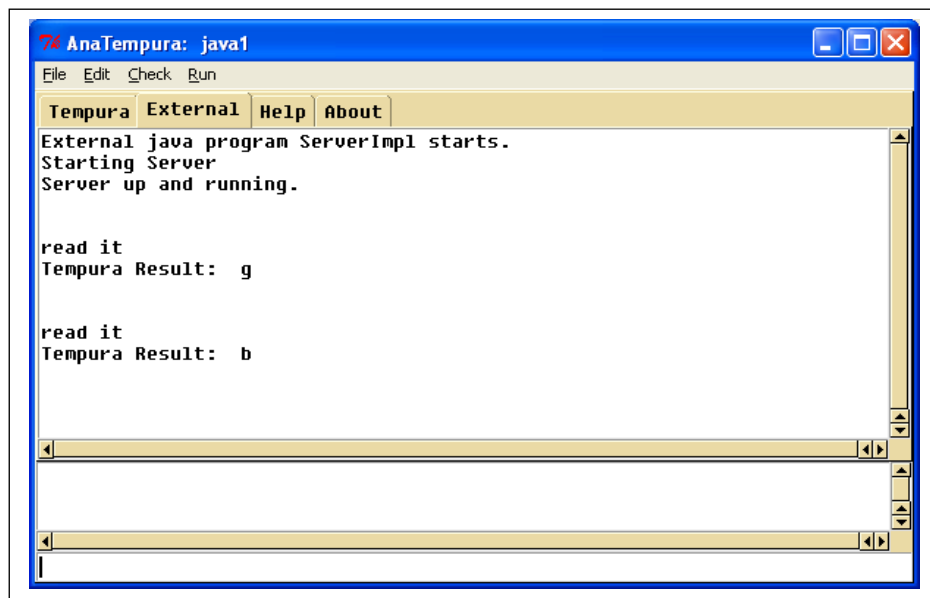


Figure 6.2 Booting of Java RMI Server and The Result

Note there are two values that are read as shown in Figure 6.3, which are the username and the password values. Therefore, each submitted value will be analysed separately.

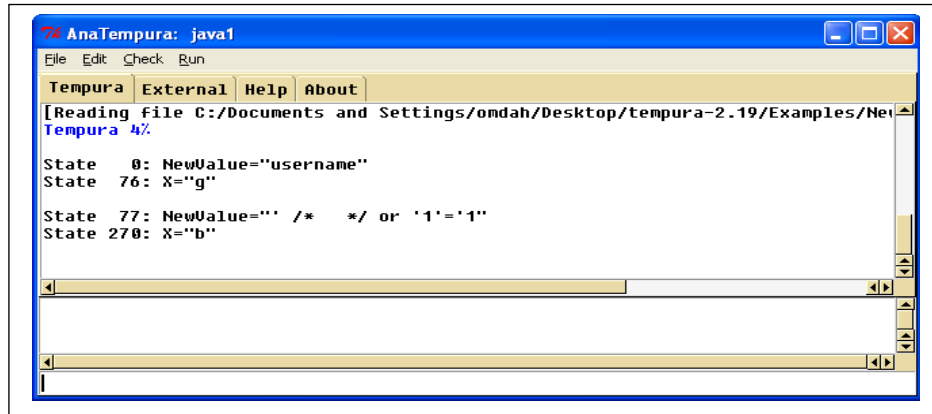


Figure 6.3 Tempura Tab and the Analysis Result

Note in the case of good inputs the value will be hidden, but they appear here because of testing the application.

6.4. Single Input Checking

In this part the effectiveness of the input checker will be examined to measure its ability to analyse the user inputs. The evaluation is divided into three sections, the first section will check safe inputs and how the input checker deals with them. The second section discusses the ability of blocking attacks and the last section will discuss the limitation of the input checker.

6.4.1. User Input Samples Testing

The input checker component checks if the input contains any form of SQL injection attack. In addition, the user input can be also a safe input. The first testing involves samples that show possible types of safe input and shows the result of the input checker component during the analysis of these samples.

Safe Inputs

The following table contains input samples and the expected analysis result. The expected result is added with each input sample.

Seq.	Input sample and Expected result
1	["g", "which better 'I don't know or 'may be'?? "].
2	["g", "I don't know , or it's will be fine. "].
3	["g", "0"].
4	["g", "to apply visit http://www.dmu.ac.uk/home.aspx "].
5	["g", "user_input"].
6	["g", "user+input"].
7	["g", "us/er+09=2"].
8	["g", "username"].
9	["g", "I want (£5) or (7.5 \$) to buy this ben."].
10	["g", "UserName"].
11	["g", "Daived starts his game at '6:30' am, and Ali normally start it at 7:00. "].
12	["g", "why you Do not send the paper?"].
13	["g", "emad_ss@hotmail.com"].
14	["g", "fahed@yahoo.com"].
15	["g", " declare @nn"].
16	["g", "12/01/2012"].
17	["g", "12/feb/2012"].
18	["g", "12.12.2012"].
19	["g", "12-12-2012"].
20	["g", "exec @nn "].
21	["g", "1234567890"].
22	["g", "I do not know!! what is the reason of this?"].
23	["g", "hani.doody@hotmail.com "].
24	["g", "the 'dmu' website is: http://www.dmu.ac.uk/home.aspx "].
25	["g", "I got 50% in my math, and Aric Got 60%."].
26	["g", "input"].
27	["g", "my address is: flat2, 20 garden avenue Leicester post code: le2 4ee "].

Table 6.1 Samples of Good Input

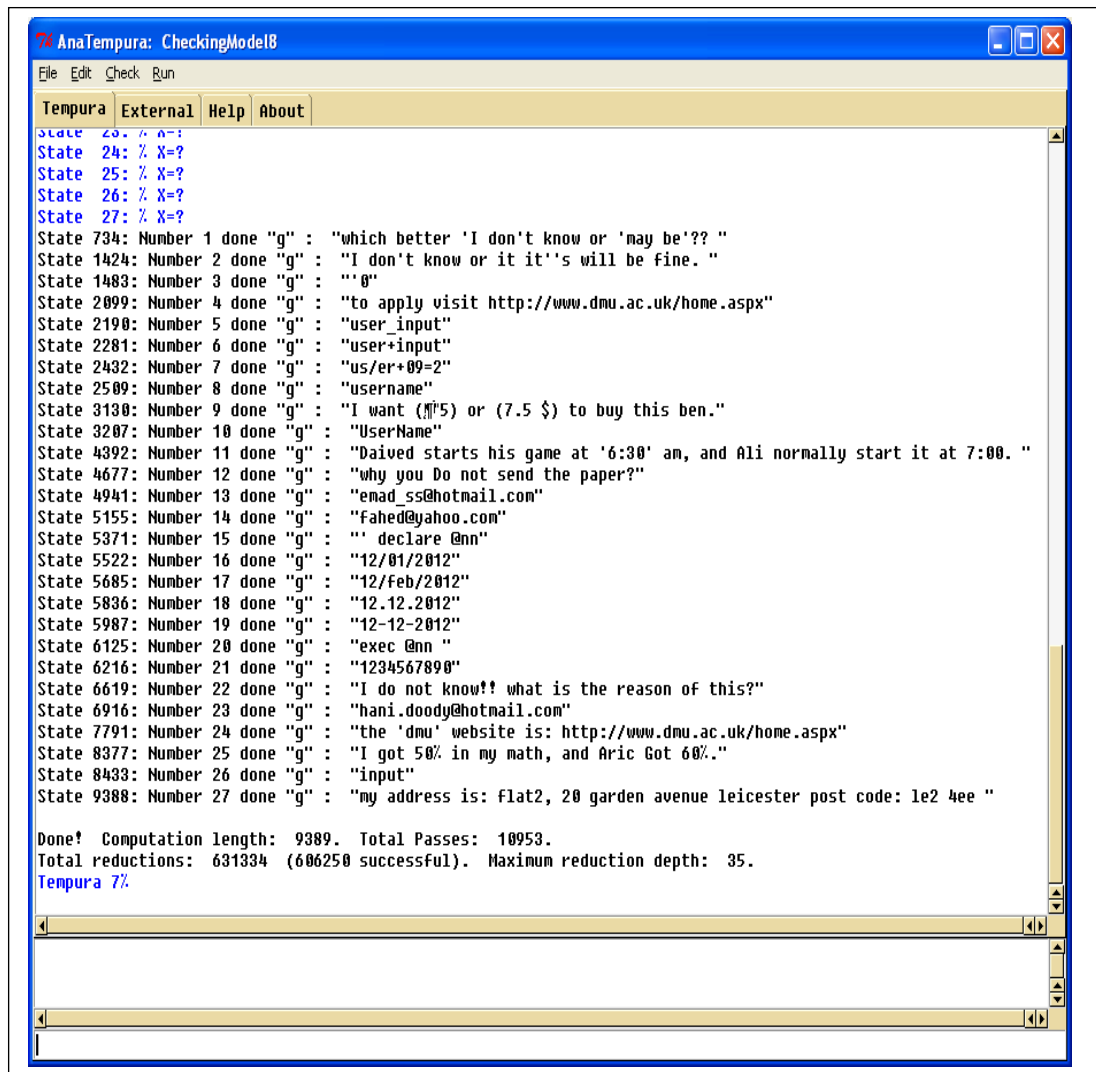
Table 6.1 shows 27 samples of safe input, these samples contain in some cases some of the SQL injection keys such as semicolon and single quotation. The expected result for each sample is denoted by the 'g' character. These samples are saved in a file called GoodSample. The simulation has three steps; the first step determines the source file of the sample using 'infile' system command. The second step reads the file records and save them to an array using an extendable array as Anatepura reads the file recorded one by one. CheckingModelTest is a Tempura application that read these values and glues the expected result as shown in Listing 6.1.

```
/* run */ define CheckingModelTest () = {
exists X,S,H:{
{set outfile="stdout" and set infile="GoodSample"
and list(H,0) and input X and while (X ~= 0) do
{extend_list(H,X) and skip and next input X } };
{stable(H) and for i<|H|
do {{CheckingModel(S,H[i][1]); {
if S = H[i][0]
then{ format("\n Number %d done %t : %t \n",i+1,S,H[i][1])}
else {format ("Conflict ! Number %d is %t Expected %t for: %t
",i+1,S,H[i][1],H[i][0]) }
and skip}}
}
}}.
```

Listing 6.1 The Code of Checking Model Test.

Listing 6.1 shows that the values will be saved to an array called H that can be easier and more flexible to be accessed and checked.

CheckingModelTest analyses the sample value and compares the result with the expected result. If the result matches with a CheckingModelTest result, the application prints 'done' or if not the application print 'Conflict' and shows the analysis result. A good sample has been simulated and the result is shown in Figure 6.4.



```
7 AnaTempura: CheckingModelB
File Edit Check Run
Tempura External Help About
State 23: % X=?
State 24: % X=?
State 25: % X=?
State 26: % X=?
State 27: % X=?
State 734: Number 1 done "g" : "which better 'I don't know or 'may be'?? "
State 1424: Number 2 done "g" : "I don't know or it it's will be fine. "
State 1483: Number 3 done "g" : ""g"
State 2099: Number 4 done "g" : "to apply visit http://www.dmu.ac.uk/home.aspx"
State 2190: Number 5 done "g" : "user_input"
State 2281: Number 6 done "g" : "user+input"
State 2432: Number 7 done "g" : "us/er+09=2"
State 2509: Number 8 done "g" : "username"
State 3130: Number 9 done "g" : "I want (¥5) or (7.5 $) to buy this ben."
State 3207: Number 10 done "g" : "UserName"
State 4392: Number 11 done "g" : "Daived starts his game at '6:30' an, and Ali normally start it at 7:00. "
State 4677: Number 12 done "g" : "why you Do not send the paper?"
State 4941: Number 13 done "g" : "emad_ss@hotmail.com"
State 5155: Number 14 done "g" : "Fahed@yahoo.com"
State 5371: Number 15 done "g" : "' declare @nn"
State 5522: Number 16 done "g" : "12/01/2012"
State 5685: Number 17 done "g" : "12/feb/2012"
State 5836: Number 18 done "g" : "12.12.2012"
State 5987: Number 19 done "g" : "12-12-2012"
State 6125: Number 20 done "g" : "exec @nn "
State 6216: Number 21 done "g" : "1234567890"
State 6619: Number 22 done "g" : "I do not know!! what is the reason of this?"
State 6916: Number 23 done "g" : "hani.doody@hotmail.com"
State 7791: Number 24 done "g" : "the 'dmu' website is: http://www.dmu.ac.uk/home.aspx"
State 8377: Number 25 done "g" : "I got 50% in my math, and Aric Got 60%."
State 8433: Number 26 done "g" : "input"
State 9388: Number 27 done "g" : "my address is: Flat2, 20 garden avenue leicester post code: le2 4ee "

Done! Computation length: 9389. Total Passes: 10953.
Total reductions: 631334 (606250 successful). Maximum reduction depth: 35.
Tempura 7?
```

Figure 6.4 The Analysis Result of Safe Input Samples

Figure 6.4 shows the actual result produced by the input checker component. The

input checker can deal with different types of safe user input and the word done which means the expected results matches the actual result.

According to the result in Figure 6.4, the samples with the number 16, 17, 18 and 19 are date style and they are considered safe inputs despite their contents containing forward slash, dot and dash characters because there is only one sequence dash character and the forward slash is suspicious only if it is followed by star.

Samples number 1 and 3 contain a single quotation and they are consider as safe input, as the single quotation is not followed by any of SQL injection keywords. Sample number 15 is slightly different to 1 and 3 because it is similar to a SQL injection attack. The single quotation here is followed by the declare command and the danger of this command will be achieved if it comes after a semicolon. Moreover, samples 13 and 14 are an example of email address and they contain the 'at' character that can be used with a stored procedure attack. They are considered as safe inputs because the 'at' character is not preceded with a SQL command like declare or exec. Therefore, the checking model checks the sequence of each character to avoid the cases of false positives in the analysis results.

Tautology Attacks

The second test of the checking model, checks its ability to catch and block various forms of SQL injection attacks. This test involves several samples of these forms. Some of these samples have more than one form of SQL injection as the SQL injecting attack can consist of one or more injection type. The first SQL injection

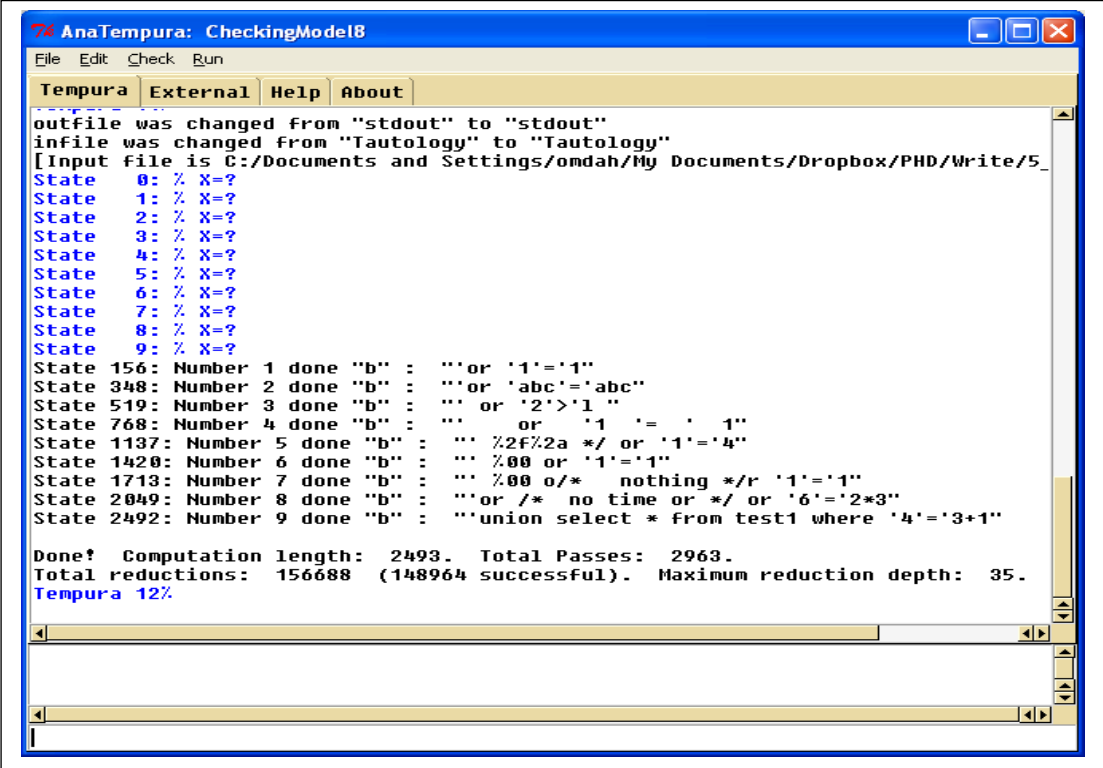
attack that will be checked is a tautology attack. The following sample illustrates the most common tautology attacks.

Seq.	The input sample
1	["b","or '1'='1"].
2	["b","or 'abc'='abc"].
3	["b"," or '2'>'1 "].
4	["b"," or '1 '= ' 1"].
5	["b"," %2f%2a */ or '1'='4"].
6	["b"," %00 or '1'='1"].
7	["b"," %00 o/* nothing */r '1'='1"].
8	["b"," */ /* no time or */ or '6'='2*3"].
9	["b","union select * from test1 where '4'='3+1"].

Table 6.2 Tautology Attack Samples.

Table 6.2 shows samples of using tautology attack. The differences between these samples are by adding other characters that can change the attack form but they still will be dangerous and executed. The test run using the same scenario of simulation that is used with the good sample in the previous section as the sample value will be read from a file that has the input sample with its expected result. Figure 6.5 shows the actual result that is produced by the checking model, the result is matched with the expected one and thus the checking model successfully caught this SQL injection type of attack. The result shows that sample number 1,2 and 3 are just a tautology attacks that starts with a single quotation and followed by the 'or' keyword and true condition statement. Sample number 4 is a tautology attack with an extra space. The extra space will be removed and the injection code will be caught by the checking

model. As aforesaid in Chapter 5, the checking model inspects the input looking for a single quotation and then checks what comes after this character. Sample 5 is a logical incorrect query attacks mixed with alternative encoding attacks. Samples 6, 7 and 8 are mixed tautology and alternative encoding attacks. The checking model in this case blocks the injection based on the alternative encoded key character and it checks the input to find a percentage character followed by number and the slash that is followed by the star character.



```
7 AnaTempura: CheckingModel8
File Edit Check Run
Tempura External Help About
outfile was changed from "stdout" to "stdout"
infile was changed from "Tautology" to "Tautology"
[Input file is C:/Documents and Settings/omdah/My Documents/Dropbox/PHD/Write/5_
State 0: % X=?
State 1: % X=?
State 2: % X=?
State 3: % X=?
State 4: % X=?
State 5: % X=?
State 6: % X=?
State 7: % X=?
State 8: % X=?
State 9: % X=?
State 156: Number 1 done "b" : "'or '1'='1'"
State 348: Number 2 done "b" : "'or 'abc'='abc'"
State 519: Number 3 done "b" : "' or '2'>'1 '"
State 768: Number 4 done "b" : "' or '1'='1'"
State 1137: Number 5 done "b" : "' %2f%2a */ or '1'='4'"
State 1420: Number 6 done "b" : "' %00 or '1'='1'"
State 1713: Number 7 done "b" : "' %00 o/* nothing */r '1'='1'"
State 2049: Number 8 done "b" : "'or /* no time or */ or '6'='2*3'"
State 2492: Number 9 done "b" : "'union select * from test1 where '4'='3+1'"
Done! Computation length: 2493. Total Passes: 2963.
Total reductions: 156688 (148964 successful). Maximum reduction depth: 35.
Tempura 12%
```

Figure 6.5 The Analysis Result of Tautology Attack Samples

Samples number 9 is a union attack based on a tautology attack. Samples 8 and 9 have true condition statement that is based on integer expression. The checking model will catch this attack as a union query attack because the injection starts with a single quotation and followed by the union keyword. Thus, the result shows that the

checking model is blocking these attack type successfully.

Piggy-back Query Attacks

The following sample will be used to check the effectiveness of the checking model for blocking piggy-back query attacks.

Seq.	The injection sample	The effect
1	["b","';select 1,2,3 from users into dumpfile '/temp/anyfilename;# "].	SQL injection of select statement that can retrieve all user information.
2	["b","';drop table users cascade constraints;# "].	Injection of drop command to delete the user table.
3	["b","'; insert /* New user */ into users values(2, 'u123','123 ');# "].	Adding new user to the application permitted users in addition to in line comment.
4	["b","'; %00 update users /* all user */ set u_password= '123';# "].	Changing the users passwords.
5	["b","'; alter table users drop u_password;# "].	Changing the structure of the user table.
6	["b","'; create user /* database user */ dbu1 identified by 'p123';# "].	Add new user to database system.
7	["b","'; delete from /* any */ users where u_name like '%Emad%' ;# "].	Delete the admin user from the user table.
8	["b","";declare @sql_procedure ;# "].	Defines a stored procedure.

Table 6.3 Piggy-back Attack Samples

Table 6.3 contains a sample of SQL injection attacks that are based on piggy-back attacks. Figure 6.6 shows the analysis result of the checking model for these samples. As aforesaid in Chapter 2, the idea of these attacks that adding a new query to the original query, so a semicolon character has been used to perform these type of attacks. Samples 1 to 7 are injected using a semicolon. The checking model checks what comes after this character, if it is one of the existing attacks keywords then it will be blocked. As the result in Figure 6.6 is matched with the expected result that

means the checking model can block these types of attacks successfully.

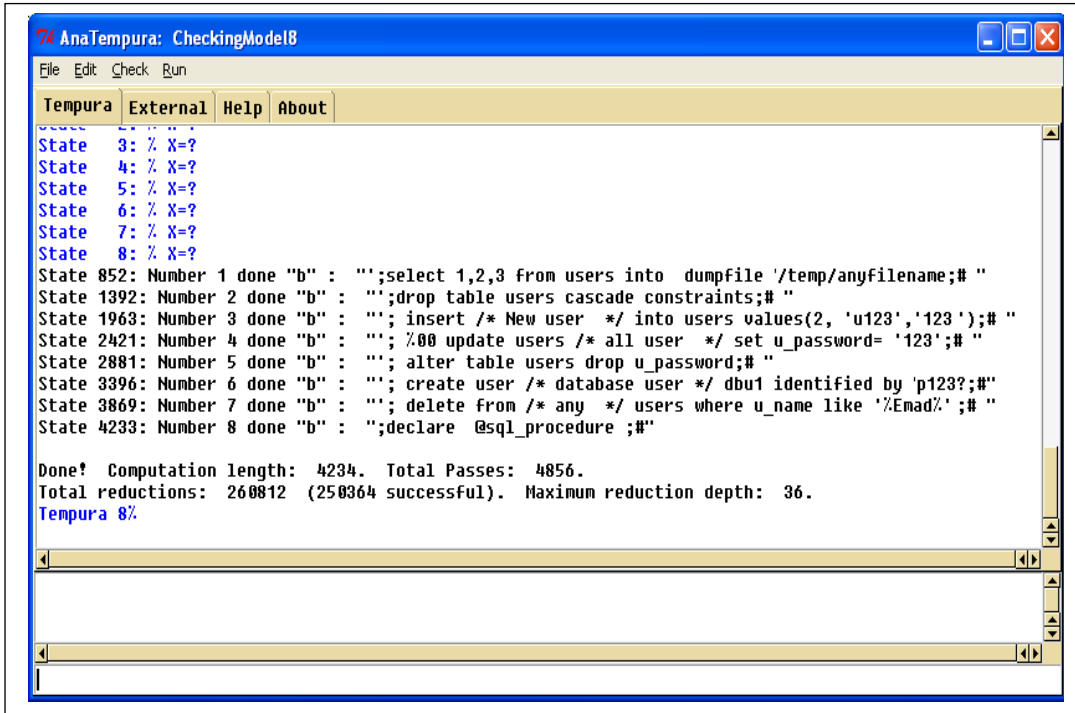


Figure 6.6 The Analysis Result for Piggy-back Attack Samples

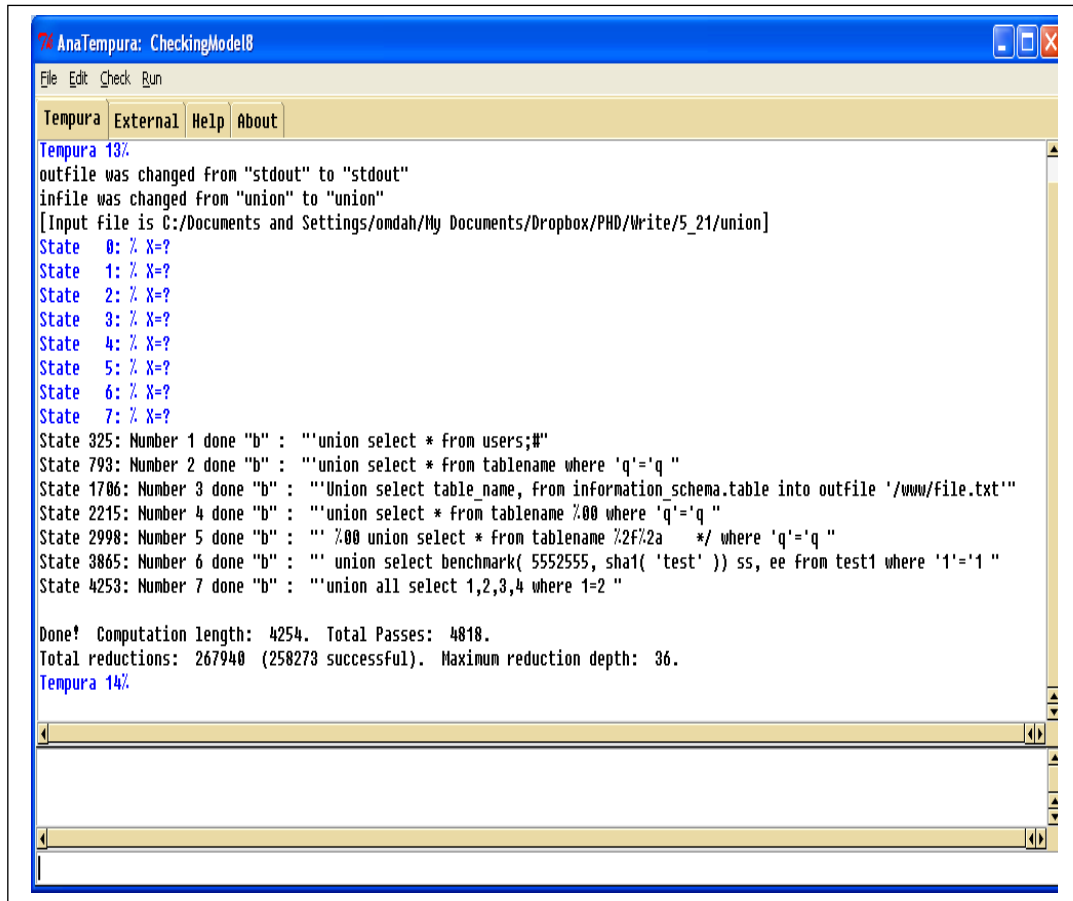
Union Query Attacks

Seq.	The injection sample
1	["b","'union select * from users;#"].
2	["b","'union select * from tablename where 'q'='q '"].
3	["b","'Union select table_name, from information_schema.table into outfile '/www/file.txt'"].
4	["b","'union select * from tablename %00 where 'q'='q '"].
5	["b","' %00 union select * from tablename %2f%2a /* where 'q'='q '"].
6	["b","'','' union select benchmark(5552555, sha1('test')) ss, ee from test1 where '1'='1 '"].
7	["b","'union all select 1,2,3,4 where 1=2 '"].

Table 6.4 Union Query Attack Samples

Chapter 6 - DPF Evaluation

Table 6.4 shows various styles of using the union keyword in SQL injection attacks. This injection type uses a single quotation as value delimiter and use the union command followed by the injection code. The checking model has tested the samples of Table 6.4 and the result is shown in the Figure 6.7.



```
AnaTempura: CheckingModel8
File Edit Check Run
Tempura External Help About
Tempura 13%
outfile was changed from "stdout" to "stdout"
infile was changed from "union" to "union"
[Input file is C:/Documents and Settings/ondah/My Documents/Dropbox/PHD/Write/5_21/union]
State 0: % X=?
State 1: % X=?
State 2: % X=?
State 3: % X=?
State 4: % X=?
State 5: % X=?
State 6: % X=?
State 7: % X=?
State 325: Number 1 done "b" : "'union select * from users;#"
State 793: Number 2 done "b" : "'union select * from tablename where 'q'='q '"
State 1706: Number 3 done "b" : "'Union select table_name, from information_schema.table into outfile '/www/file.txt'"
State 2215: Number 4 done "b" : "'union select * from tablename %00 where 'q'='q '"
State 2998: Number 5 done "b" : "' %00 union select * from tablename %2f%2a */ where 'q'='q '"
State 3865: Number 6 done "b" : "' union select benchmark( 5552555, sha1( 'test' )) ss, ee from test1 where '1'='1 '"
State 4253: Number 7 done "b" : "'union all select 1,2,3,4 where 1=2 '"

Done! Computation length: 4254. Total Passes: 4818.
Total reductions: 267940 (258273 successful). Maximum reduction depth: 36.
Tempura 14%
```

Figure 6.7 The Analysis Results for Union Attack Samples

Union attacks as shown in Figure 6.7 can consist of more than one attack type like sample 7 that is similar to logical incorrect query. Samples 2, 4 and 5 are union and tautology attack including code of alternative encoding attacks. Sample 6 is a blind injection that causes a delay if injected successfully. The checking model has

detected these types of attacks, and the result matches the expected result.

6.4.2. Input Checker Limitations

The input checker component has been checked using various types of user input and multi forms of SQL injection attacks. The result matches with those of the expected results. However, these samples are not enough for the evaluation of the input checker component as they do not cover all possible input. Thus, there is a possibility of false negatives and false positives. False negatives were not found during the testing of this component despite using different attack types. However, the alternative encoding attacks have unlimited possibility. In the detection algorithm, the common alternative encoding attacks have been handled based on the key of these attacks. For example, the function char is used to combine the character decimal code. The function char and the codes that start with '0x' are being blocked in addition to the other alternative encoding attacks that are discussed in Chapter 5.

Another limitation is that the checking component produces some false positives if the injection code contains the SQL injection key or the input starts with one of the common SQL injections. For example, if the input contains a single quotation followed by the union keyword and then a semi colon, the detection algorithm considers this code as an injection where it actually does not affect or damage the database. Thus, this case and other similar cases that start with an injection are blocked as they can cause a database error.

Another example is if the input has a part that is similar to the SQL injection form like using a single quotation followed by the 'or' keyword and then anything, in this case,

the input checker will consider it an attack as shown in Figure 6.8.

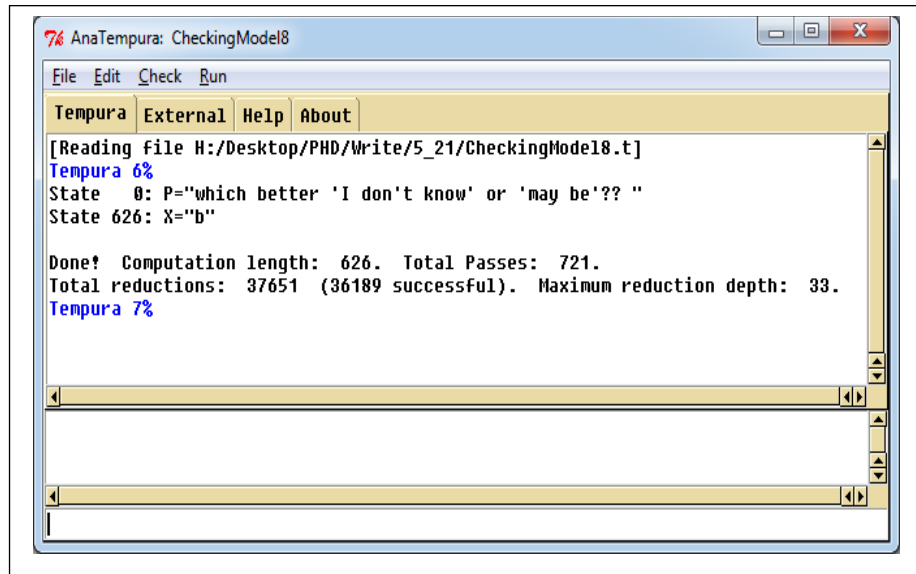


Figure 6.8 False Positive of the Checking Result

6.5. DB Observer Testing

As aforesaid in Chapter 4, the database observer is detecting unknown SQL injection attacks. Chapter 5 discusses the techniques that control the session using three functions which are Begin, Commit and Rollback. Thus, any unknown session will be started with the Begin function and if it matches the database observer condition the function Commit will end the session, otherwise the session will end with the Rollback function. To check effectiveness of this component, we discuss a database sample and show how the database observer will deal with unknown injections. The results show the database condition values during the injection and without the injection. Figure 6.9 shows a result sample of the record number condition during the injection of the web page.

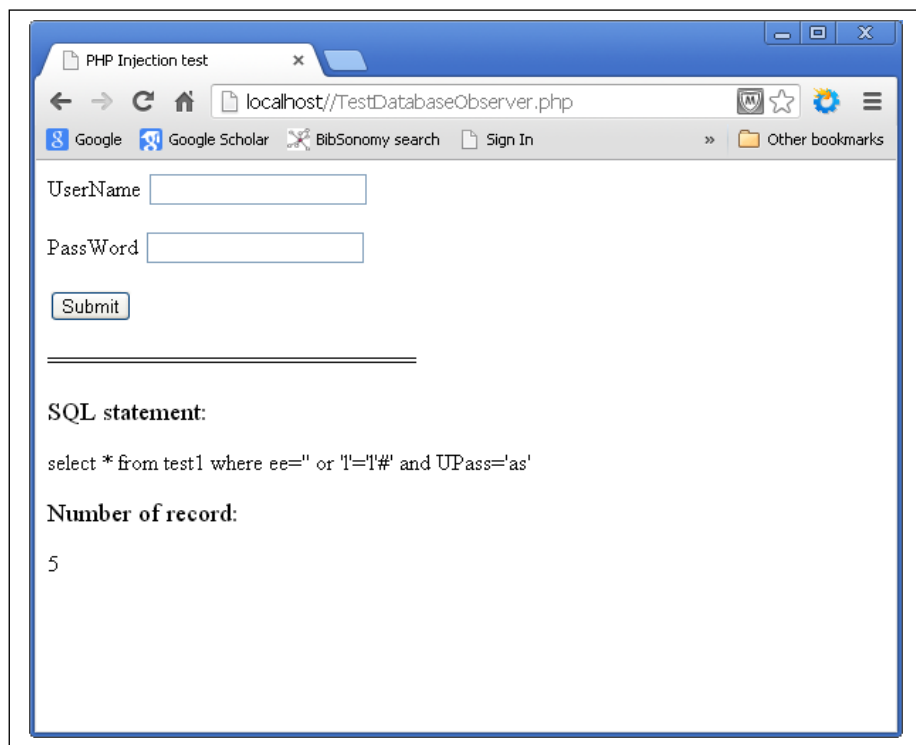


Figure 6.9 Database Observer Rejected Value

Figure 6.9 shows an example of a login page that contains the SQL statement after the page is submitted. The page also contains the number of records that are affected using the submitted data which is similar to the database observer checking technique for the first checking condition. In this example, the submitted page returned 5 as record number which means the SQL injection code has run successfully. Thus, this data will be rejected and the initial capture specification will be updated. Note that the page has no checking of the submitted data as this example is for testing the database observer.

Figure 6.10 and Figure 6.11 show examples of submitted pages that are accepted by the database observer component.

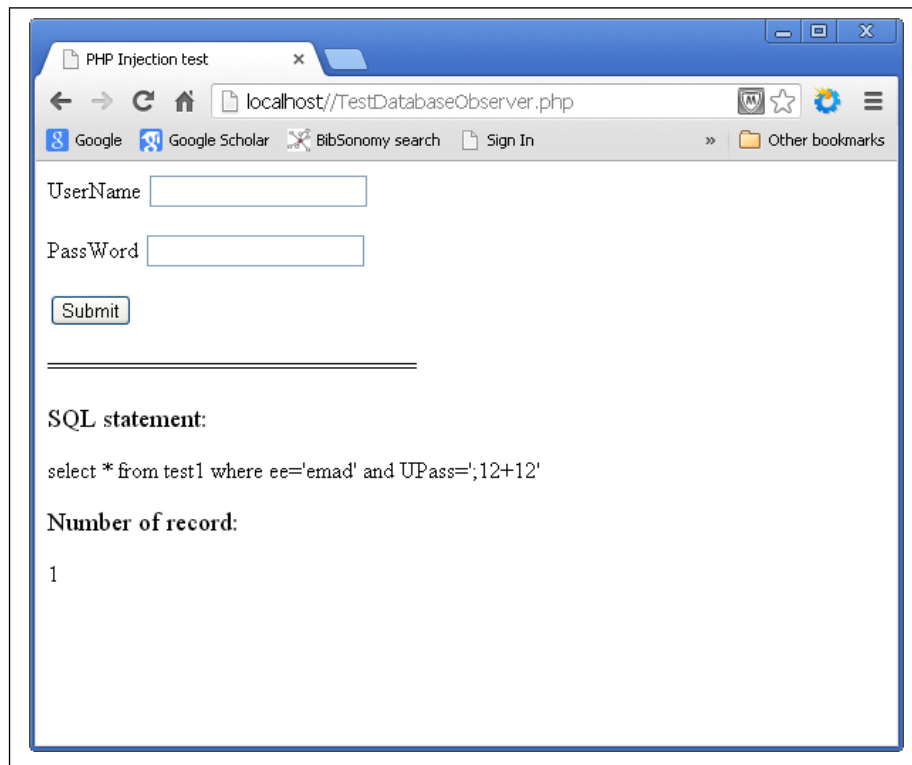


Figure 6.10 Database Observer Accepted Value Example

Figure 6.10 shows that the SQL statement has a value for the Pass field containing semicolon and plus characters. The value is not matched with existing attack patterns and it cannot be considered good because of its contents. However, the number of matching records is 1 which means that the SQL statement matches one record of the test1 table. Thus, the data is compatible with the first condition of the database observer.

Figure 6.11 shows an example of a SQL statement that has a value of password field that contains a semicolon followed by a bracket. However, the number of matching records is 0 which means that the SQL statement does not match with any record of the user table and therefore no records are affected by executing this statement.

Chapter 6 - DPF Evaluation

Thus, the data is secure and does not break the first condition of the database observer.

In the examples of Figure 6.10 and Figure 6.11, the new attacks will be analysed, than the new attacks specification will be added to the initial capture. Thus, the initial capture will be updated manually to deal with new cases that are found by the database observer.

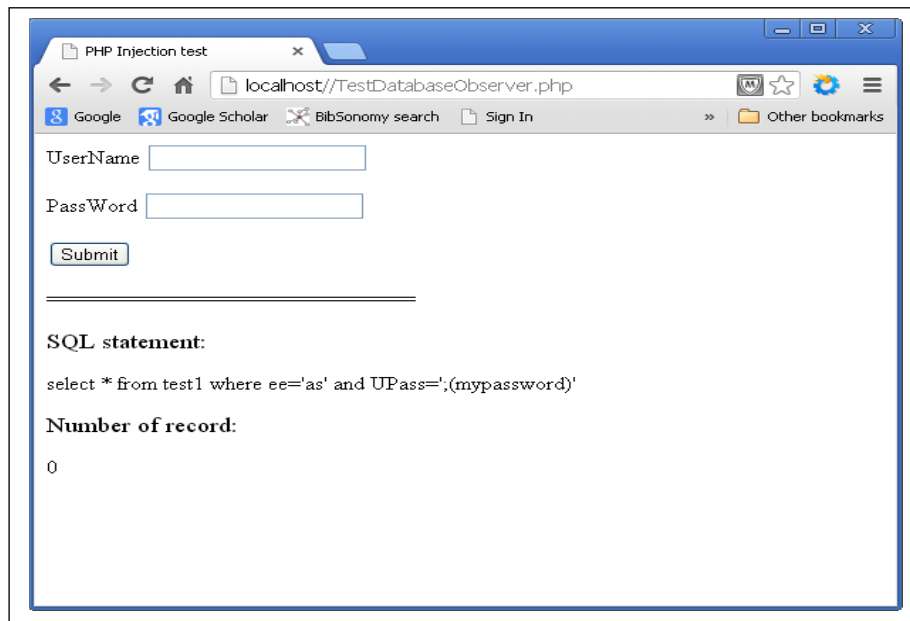


Figure 6.11 Database Observer Accepted Value Example 2

Similarly, the evaluation of the user type condition will be done. For example, if a web application has several user types and the user type is created when the user login as explained in Chapter 4, then the user cannot access and use any page without available session. In other words, the user will be blocked if (s)he tries to change the data that is not related to the session type.

6.6. Output Checker Testing

This component replaces the messages that come from the database with one that

Chapter 6 - DPF Evaluation

does not contain any information about the database type or structure. Figure 6.12 shows an example of the database response to the injection code that is not executed successfully.

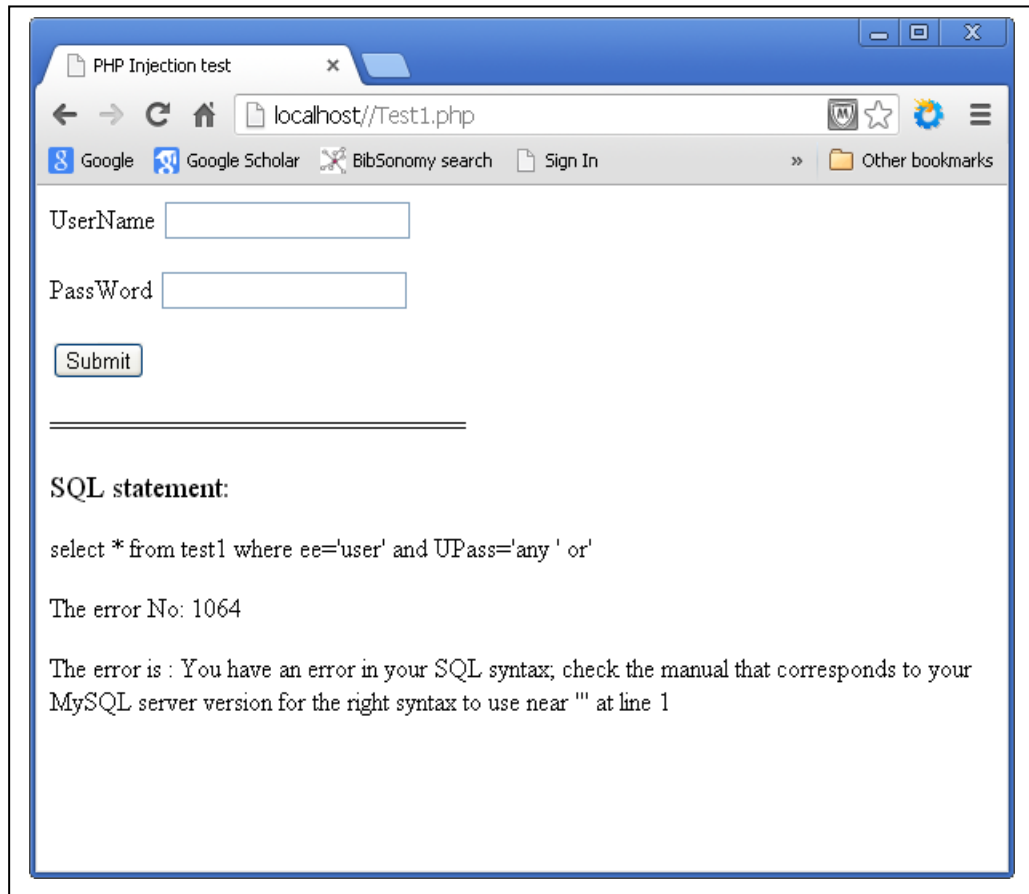


Figure 6.12 Sample of Web Application Page Error

The injected code here is a single quotation followed by an 'or' SQL keyword. The error message shows information about the database type which is MYSQL. In Chapter 4, the replacement of the messages type has been discussed and implemented. Figure 6.9 shows another message for same input.

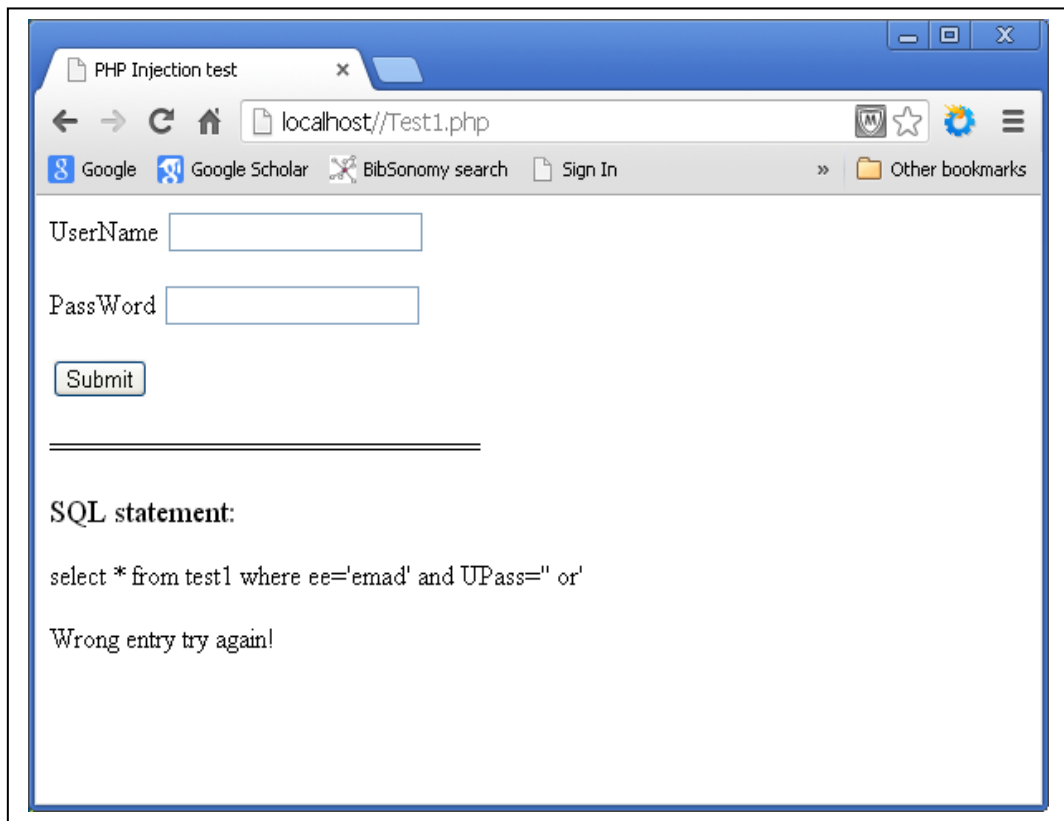


Figure 6.13 Sample of Error Handling

Thus, the error has been replaced with another message and therefore the output checker has run successfully.

6.7. Behavioural Functions Testing

Testing of the user's behaviour will be done by simulation in Anatepura only. The reasons of the simulation are because of the difficulty to collect the SQL injection data from various machines and to check the success of all user behaviour. Thus, the investigation simulates related attacks using a file that contains various types of user input. Each record in this file contains four values which are the IP address, the input sample and the time stamp of the submitted data that come from application server.

Chapter 6 - DPF Evaluation

The investigation of related records will be done in several steps. The first step uploads the file that contains the sample records to Anatempura using an extendable array as Anatempura reads the file records one by one. The input of each record of the sample in this array will be analysed using the input checking component. The analysis result of each record will be added to the record and replace the expected result. Thus, the record will contain four values as follows

```
["IP Address "," Input Sample", "Time", " The analysis result"]
```

As aforesaid in Chapter 4, the investigation of related user behaviour will be done according to four criteria which are the sequence of SQL injection attacks, the percentage of the transaction type, the related IP address, and the related store procedure attacks. The following sample shows sample inputs:

Seq.	Input Samples
1	["192.168.1.2","which better I don't know, or say 'may be'?? ","Time 1:1","g"].
2	["192.168.1.1","; declare @SQLProcedure","Time 1:2","b"].
3	["192.168.1.6","; drop table ","Time 1:3","b"].
4	["192.168.1.12"," or 1=1 ","Time 1:4","b"].
5	["192.168.1.1","12/feb/2012","Time 1:5","g"].
6	["192.168.1.8","union select @SQLProcedure ","Time 1:6","b"].
7	["192.168.1.6","I don't know or it it's will be fine. ","Time 1:7","g"].
8	["192.168.1.1","to apply visit http://www.dmu.ac.uk/home.aspx ","Time 1:8" "σ"]
9	["192.168.1.1","user_input","Time 1:9","g"].
10	["192.168.1.1","user+input","1:10","g"].

11	["192.168.1.2","; exec @SQLProcedure ","Time 1:11","b"].
12	["192.168.1.6","us/er+09=2","Time 1:12","g"].
13	["192.168.1.6","username","Time 1:13","g"].
14	["192.168.1.6","I want (£5) or (7.5 \$) to buy this ben.,"Time 1:14","g"].
15	["192.168.1.1","emad_ss@hotmail.com","Time 1:15","g"].
16	["192.168.1.6","fahed@yahoo.com","Time 1:16","g"].
17	["192.168.1.1",""or '1'=1","Time 1:17","b"].
18	["192.168.1.6","" %00 or '1'=1","Time 1:18","b"].
19	["192.168.1.4","" %2f%2a */ or '1'=1","Time 1:19","b"].
20	["192.168.1.7","" %252f%252a */ or '1'=1","Time 1:20","b"].
21	["192.168.1.1","" or exists (select * from test1) and "="","Time 1:21","b"].
22	["192.168.1.1","" or not equal to zero","Time 1:22","b"].
23	["192.168.1.2","12/01/2012","Time 1:23","g"].
24	["192.168.1.6","UserName","Time 1:24","g"].
25	["192.168.1.5","12.12.2012","Time 1:25","g"].
26	["192.168.1.1","12-12-2012","Time 1:26","g"].

Table 6.5 Behaviour Input Samples

The test has two parts, the first one uses the input checker component because the creation of the behaviour runs in sequence and starts after the checking component, and the second one run uses the behavioural procedure. Figure 6.14 shows the analysis results of the sample inputs and Figure 6.15 shows the related behaviour of these samples.



Figure 6.14 Analysis Results of the Behaviour Input Samples

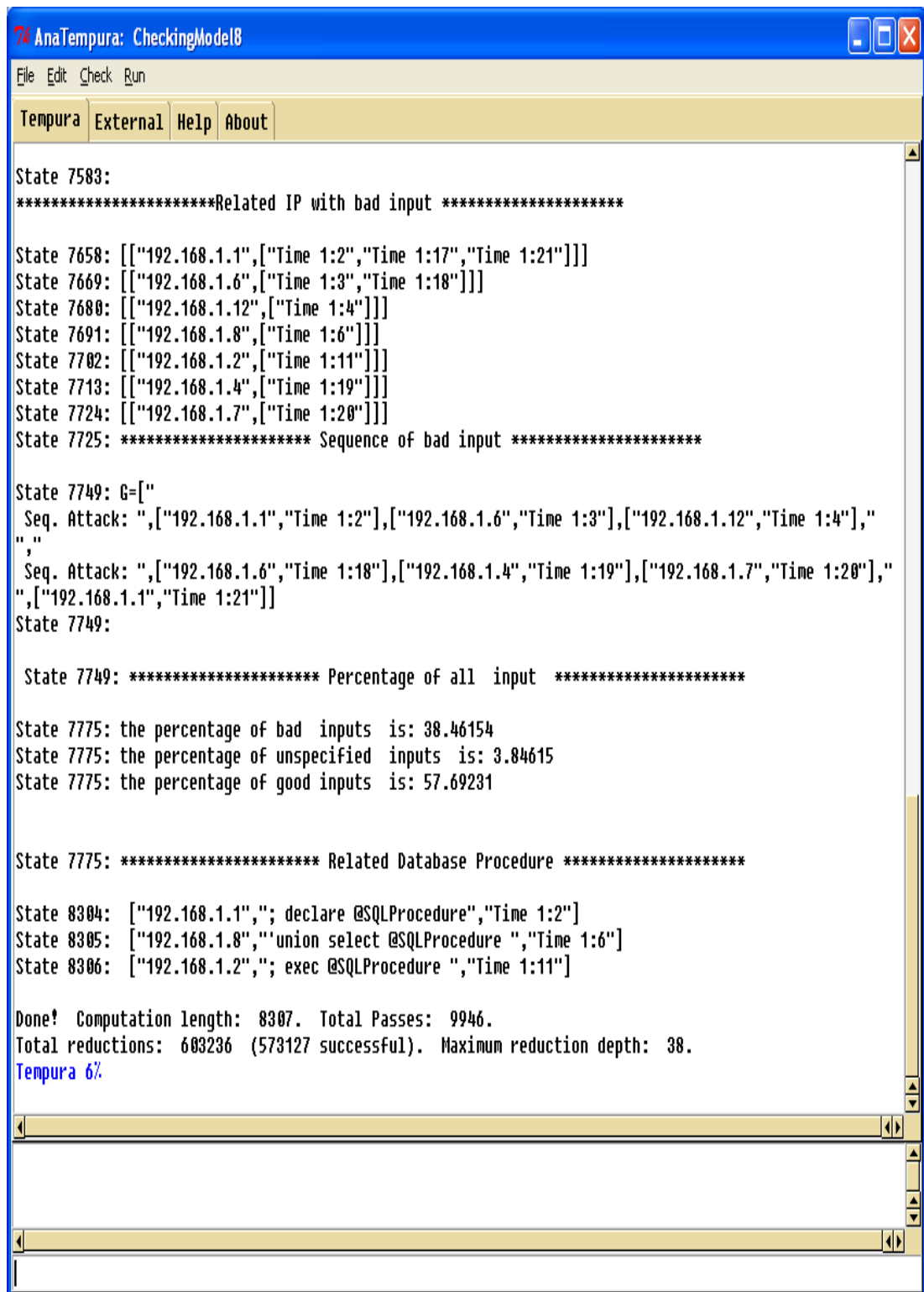


Figure 6.15 User's Behaviour Results

Figure 6.14 shows 26 cases of user input and the analysis result for these cases, the IP address for each case and the transaction time. These values have been investigated using the behavioural function and the result is shown in Figure 6.15 in 4 parts. The first part contains the related bad cases that come from the same IP address. The result of this part displays the IP address followed by the sequence transaction times that are arranged in ascending order.

The result of the second part shows the related attacks that run sequentially, and the behavioural function here checks for sequences of three attacks or more because the SQL injection attacks require at least three attempts to run the injection. The first attempt detects the database type. The second one finds the database structure, and the third one starts the exploit and attack the database. The result shows that the inputs number 2, 3 and 4 are related as they run sequentially. Another sequence is for the inputs number 18,19,20,21, and 22. Note the inputs in these attempts come from several IP addresses.

The third part shows the percentage of each transaction type, i.e., good, bad, and unknown. The result shows the percentage of the three types of transaction whereas the bad transaction has a percentage of 38.46154, the unknown transaction equal 3.84615 %, and the good transaction percentage 57.69231 which is not an acceptable value for any web application if that data is collected from a real web application, because if the percentage of the bad transaction is high then it means that this application is under attack. Thus, the transaction percentage statistics can be used to produce an early warning for the system administrator about the attacks. So, if the

percentage of good transactions is 95% that it can be considered as nearly safe because the 5 % involves false positives cases and some unknown cases that can be handled with database observer in addition to some rejected cases of bad input. Therefore, the determination of the percentage of the system transactions is required to know the system states. And these percentage needs to be determined by the application administrator.

The result of the last part shows the related attacks that are based on using the same stored procedure. Input number 2 is used to declare a stored procedure and input number 6 and 11 are used these procedure.

Therefore, the result shows that the behavioural functions can detect related attempts of bad input such as presented in Chapter 4 and it has run successfully. The following section discusses some of the existing approaches and compares them with our approach.

6.8. Related Work Comparison

There are many studies and web application vulnerabilities scanning tools that tackle the problem of the SQL injection. Some of these studies are discussed in Chapter 2. DPF will not compared to the web application scanning tools like Nikto or Acunetix because they uses black box testing techniques and they deal with various of web application vulnerabilities. In this section, the DPF technique and its checking results will be discussed and compared with other studies that are proposed to block SQL injection attacks. The comparison will be based on the following criteria:

- Blocking all attacks type

Chapter 6 - DPF Evaluation

- Using static analysis
- Modifying code
- Developer specification level
- Producing false positives and false negatives
- Runtime underlying logic

In addition to the comparison criteria, the DPF differs from existing approaches as it can track attacks using behavioural functions, and it also block new attacks by following database transactions using the database observer. The comparison will be divided in two tables because the information of comparison criteria is not available in some studies. The following table show the comparison result of some of the mentioned criteria.

Approaches	Using static analysis	Attacks specification	Block exist attacks	Tracking Attacks
(Halfond, Orso 2006)	Fully	Automated	All	No
(Wassermann, Su 2007)	Fully	Automated	All	No
(Shrivastava, Bhattacharyji 2012)	No	Manual - Filter	All	No
(Natarajan, Subramani 2012)	yes	Automated	Some *1	No
(Manikanta, Sardana 2012)	Fully	Automated	All	No
(Lee, Jeong et al. 2012)	Fully	Automated	All	No
DPF	partly	Manual	All	yes

Table 6.6 Existing Approaches Comparison with the DPF (1)

Table 6.6 shows some the existing approaches and the comparison information according to the criteria: ‘using a static analysis’, ‘attacks specification’ and ‘block

Chapter 6 - DPF Evaluation

existing attacks’. Some of the existing approaches analyse the code and simulate to find vulnerable contents, and others do not require the static analysis stage because they are based on filtering the inputs. The DPF assumes that static analysis is used to determine the hotspots of the application. The DPF attacks specification will be done manually because the detection specification needs to be specified in ITL.

The second comparison information is shown in the following table.

Approaches	Modifying code	False Positive	False negative	Runtime monitoring	Database Observer
(Boyd, Keromytis 2004)	Yes	No	No	No	No
(Halfond, Orso 2006)	No	Low	No	Yes java based on N DFA	No
(Wassermann, Su 2007)	No	low	No	No	No
(Shrivastava, Bhattacharyji 2012)	No	N/A	N/A	No	No
(Natarajan, Subramani 2012)	No	N/A	Yes	Yes Java monitoring	No
(Manikanta, Sardana 2012)	No	No	No	Yes using DB Firewall	No
DPF	Yes	Low	No	Yes using Anatemपुरa	Yes

Table 6.7 Existing Approaches Comparison with the DPF (2)

Table 6.7 shows another comparison which is based on the criteria: ‘modifying code’, ‘false positives’, ‘false negatives’ and ‘using of runtime monitoring’. Some of the existing approaches modify the application code to apply their approach like (Boyd, Keromytis 2004) as they need integrated software that can initialize and recollect the random number of each SQL keyword. The DPF requires little code

modification because of the assertion points that will be added to a web application code for each hotspot of runtime monitoring. The most dangerous type of checking result is false negatives and false positives. False positives are limited as discussed in the evaluation of the input checker section. So according to the criteria using Anatempura as runtime monitoring is recommended.

6.9. Summary

This chapter presented the evaluation of the DPF components and discussed the evaluation result of each component. The evaluation of the input checker components had several stages of testing for most common and existing attacks which were using Anatempura and run time testing. The DPF component database observer and the output checker also have been evaluated and discussed. The behavioural functions are evaluated using a case study. The evaluation result shows that DPF has been implemented successfully. The next chapter will conclude this thesis and discuss the strengths and limitations of DPF in addition to the future work.

Chapter 7

Conclusion

Objectives

- Summarise the thesis.
 - Discuss the research limitations.
 - Highlight the contributions to knowledge of this research.
 - Discuss future work.
-

7.1. Summary of the thesis

This thesis presented a new framework for the detection and prevention of SQL injecting attacks (DPF) that can detect existing and new attacks and investigate related SQL injection attacks at runtime. The DPF framework is based on ITL using its executable engine Tempura and its runtime monitoring tool Anatempura. The framework's components are discussed showing how these components interact with each other to detect and prevent SQL injection attacks. Furthermore, the DPF consists of three checking components, i.e., the input checker component, the output checker component and the database observer component. The input checker monitors the user inputs for existing SQL injection attacks that are specified using ITL. The database observer checks database transactions of inputs that are tagged as unknown. The output checker is used to check if the database messages contain any information about the database structures or type. Therefore, the checking process can deal with various types of user input. The investigation of related attacks uses also Anatempura and can construct the user behaviour via behavioural functions using the web transactions information like, user inputs, submission time and the IP address.

The framework implementation is programmed using Tempura, Java, and PHP. Tempura is used to implement the detection formula and the behavioural functions. PHP is used to implement the database observer and the output checker. Java is used as a bridge to transfer submitted data between the PHP application and the Anatempura tool.

Chapter 7 - Conclusion

The testing of the feasibility of DPF and the effectiveness of DPF components are done in several stages. The input checker is tested in two stages, the first one at runtime to check the interaction of various components (Section 6.3).

The input checker is tested using various samples of user input. The samples contain examples of safe input and existing attacks patterns like tautology, piggy-back query and union attacks. The effectiveness was measured by simulating sample inputs, using the Anatempura tool, and the simulation results were discussed. The effectiveness of the input checker in detecting existing attacks pattern was shown (Section 6.4). The database observer and the output checker were tested using different PHP pages that show various user input and the way these components deal with these cases was discussed (Section 6.5, Section 6.6).

The user behaviour was tested using a case study that contains information about real transactions like IP addresses, the submission time and the input. This testing is performed using runs simulating the input behaviour with Anatempura. The behavioural function testing results showed that the investigation criteria of related attacks are successful. Finally, the DPF framework is compared with existing approaches that are proposed to detect SQL injection attacks.

7.2. Contribution

This research makes the following contributions:

- New framework for detection and prevention of SQL injection attacks (Section 4.2).

Chapter 7 - Conclusion

- Runtime detection: using a runtime verification technique based on Interval Temporal logic detecting various types of SQL injection attacks (Section 4.4.1).
- Database observer: to detect possible new injection attacks by monitoring database transactions (Section 4.4.2).
- User's behaviour: investigates related SQL injection attacks using user input, and providing early warning against SQL injection attacks (Section 4.5.2).

7.3. Success Criteria Revisited

Success criteria have been proposed in Chapter 1 to judge the success of the research.

The following will revisit those criteria to measure the success of this research.

- The framework can detect and prevent existing SQL injection attack types.

The framework architecture has been discussed in Chapter 4 and there are three checking components that can check the user inputs. Chapter 5 discussed the implementation of these components. Chapter 6 has discussed several samples of user input that contains good and bad inputs. The result showed the input checker's ability to detect and prevent existing SQL injecting attack types. Thus, this framework has been successful in detecting and blocking the existing SQL injection attacks.

- The runtime verification tool is suitable for monitoring the web application.

In Chapter 3, the Anatempura and its features have been discussed. An overview of using Anatempura in the DPF framework is discussed as well. Section 6.3 highlighted a runtime example that shows how the Anatempura deal with web transactions. Anatempura can monitor attacks specification, and there are several

Chapter 7 - Conclusion

samples of user input that were discussed, and that show the effectiveness of Anatempura in the detection of various types of inputs (Section 6.4). Therefore, using Anatempura is recommended for monitoring a web application.

- The framework can detect new types of SQL injection attack.

One of the contributions in this research is that it uses the database observer which can detect a new SQL injection attack by monitoring the unknown input and determining whether the input is safe or not. Chapter 6 has discussed several cases that use a database observer showing the effectiveness of this component. The result of these cases showed the database observer's ability to catch and block new attacks.

- ITL is suitable to model attack behaviour.

Attacks behaviour component tracks the user over several states and determines related states. The web transaction data, such as the user input and others, have been used to investigate this relation. Chapter 4 discussed the conditions that are proposed to investigate this relation, and Chapter 5 showed the implementation of the behavioural function using Tempura. Chapter 6 discussed the result that is achieved based on a real web application. The results supported our choice for ITL to model attacks behaviour as all proposed conditions were successfully realized.

7.4. Limitations

As aforesaid in Chapter 6, the evaluation results of the proposed framework are similar to the expected result of each test sample. Thus, the framework can detect the existing and new SQL injection attacks, in addition to modelling attacks behaviour. However, the framework has the following limitations.

Chapter 7 - Conclusion

- Production of false positives and the possibility of false negatives.

This part is discussed in Section 6.4.2, and the reason for the false positives is that the injection code contains the SQL injection key or starts with one of the common SQL injections. False negatives can be produced because of the alternative encoding attacks having unlimited possibilities. Furthermore, to discuss the rate of false positives and false negatives would require a comprehensive benchmark that includes all existing SQL injection attacks techniques in addition to possible safe inputs. Such a benchmark does not exist at the moment.

- Manual addition of the specification of new attacks.

The initial capture component that contains the attacks specification is now manually updated because of the new attacks that needs to be analysed first then its specification will be added to the detection formula. However, Anatempura can update the attacks specification automatically but that requires further research.

7.5. Future work

As stated in Chapter 2, the detection of SQL injection attacks is based on the DBMS type that is used within a web application because the SQL injection code should be compatible with the DBMS type to run the injection successfully. Currently, the detection technique is tested for the MYSQL database type and the testing results showed the effectiveness of the checking components. The limitation of the input checker component is discussed showing some examples of false positives and the reasons for these cases. In addition, because the alternative encoding technique is unlimited, so, there is a possibility of false negatives. Thus, the future work will

Chapter 7 - Conclusion

focus on the following:

- Improve the detection technique and develop the ability to check the SQL injection attacks for other database types.
- Improve the detection formula to reduce the false positive cases, and do more investigation of the alternative encoding attacks to check that the input checker component can detect more of these types of attacks.
- The related attacks can now be investigated based on four conditions; further research can establish other conditions.
- Improve the investigation of related attacks so that the input checker component can predict the next steps of attacks.
- Improve the updated of detection formula of the existing attacks to be run automatically.
- Further research to specify XSS attacks and the way to add its specification to the detection formula.
- Check the DPF ability to detect and protect the SQL injection vulnerabilities that are mentioned in CVE entries (MITRE 2013).

Bibliography

Bibliography

- ACUNETIX, 2012-last update, Web Application Security with Acunetix Web Vulnerability Scanner. Available: <http://www.acunetix.com/vulnerability-scanner/> [10/18, 2010].
- AL AMRO, S. and CAU, A., 2012. Behavioural api based virus analysis and detection. *International Journal of Computer Science and Information Security*, **10**(5), pp. 14-22.
- AL AMRO, S. and CAU, A., 2011. Behaviour-based virus detection system using Interval Temporal Logic, *Risk and Security of Internet and Systems (CRiSIS), 2011 6th International Conference on 2011*, IEEE, pp. 1-6.
- ANLEY, C., 2002. Advanced SQL injection in SQL server applications. *White paper, Next Generation Security Software Ltd*, .
- ANTUNES, N., LARANJEIRO, N., VIEIRA, M. and MADEIRA, H., 2009. Effective Detection of SQL/XPath Injection Vulnerabilities in Web Services, *Services Computing, 2009. SCC'09. IEEE International Conference on 2009*, IEEE, pp. 260-267.
- BASIN, D., KLAEDTKE, F. and MÜLLER, S., 2010. Policy monitoring in first-order temporal logic, *Computer Aided Verification 2010*, Springer, pp. 1-18.
- BBC, 12 July 2012, 2012-last update, **Yahoo investigating exposure of 400,000 passwords** [Homepage of BBC], [Online]. Available: <http://www.bbc.co.uk/news/technology-18811300> [05/05, 2013].
- BBC, 29 August 2011, 2011-last update, **Nokia's developer network hacked** [Homepage of BBC], [Online]. Available: <http://www.bbc.co.uk/news/technology-14706810> [05/05, 2013].
- BEAVER, K., 2007. *Hacking for dummies*. John Wiley & Sons.
- BÖKEMEIER., J. and KOERBER., J., 2012-last update, PHP -Java Bridge. Available: <http://php-java-bridge.sourceforge.net/pjb/index.php> [12/010/2012, 2012].
- BOURDON., R., 2013-last update, Wamp server. Available: <http://www.wampserver.com/en/> [11/15, 2010].
- BOYD, S. and KEROMYTIS, A., 2004. SQLrand: Preventing SQL injection attacks, *Applied Cryptography and Network Security 2004*, Springer, pp. 292-302.
- BRAVENBOER, M., DOLSTRA, E. and VISSER, E., 2007. Preventing injection attacks with syntax embeddings, *Proceedings of the 6th international conference on*
-

Bibliography

Generative programming and component engineering 2007, ACM, pp. 3-12.

CAU., A., MOSZKOWSKI, B. and ZEDAN, H., 10/2012-last update, Interval Temporal Logic. Available: <http://www.cse.dmu.ac.uk/STRL/ITL/> [1/17, 2010].

CHRISTENSEN, A., MØLLER, A. and SCHWARTZBACH, M., 2003. Precise analysis of string expressions. *Static Analysis*, , pp. 1076-1076.

CIMATTI, A., CLARKE, E., GIUNCHIGLIA, E., GIUNCHIGLIA, F., PISTORE, M., ROVERI, M., SEBASTIANI, R. and TACCHELLA, A., 2002. Nusmv 2: An opensource tool for symbolic model checking, *Computer Aided Verification 2002*, Springer, pp. 241-268.

CLARKE, J., 2012. *SQL injection attacks and defense*. Syngress Publishing.

EL-KUSTABAN, A., MOSZKOWSKI, B. and CAU, A., 2012. Formalising of transactional memory using interval temporal logic (ITL), *Engineering and Technology (S-CET), 2012 Spring Congress on 2012*, IEEE, pp. 1-6.

FU, X., LU, X., PELTSVERGER, B., CHEN, S., QIAN, K. and TAO, L., 2007. A static analysis framework for detecting SQL injection vulnerabilities, *Computer Software and Applications Conference, 2007. COMPSAC 2007. 31st Annual International 2007*, IEEE, pp. 87-96.

GELLERSEN, H.W. and GAEDKE, M., 1999. Object-oriented web application development. *Internet Computing, IEEE*, **3**(1), pp. 60-68.

GOULD, C., SU, Z. and DEVANBU, P., 2004. JDBC checker: A static analysis tool for SQL/JDBC applications, *Proceedings of the 26th International Conference on Software Engineering 2004*, IEEE Computer Society, pp. 697-698.

GREENSQL LTD, 2012-last update, Database Security Solutions | GreenSQL. Available: <http://www.greensql.com/> [09/12, 2012].

HALFOND, W.G.J. and ORSO, A., 2006. Preventing SQL injection attacks using AMNESIA, *Proceedings of the 28th international conference on Software engineering 2006*, ACM, pp. 795-798.

HALFOND, W.G.J. and ORSO, A., 2005. AMNESIA: analysis and monitoring for NEutralizing SQL-injection attacks, *Proceedings of the 20th IEEE/ACM international Conference on Automated software engineering 2005*, ACM, pp. 174-183.

HALFOND, W., VIEGAS, J. and ORSO, A., 2006. A classification of SQL-injection attacks and countermeasures, *Proceedings of the IEEE International Symposium on*

Bibliography

Secure Software Engineering 2006, IEEE, pp. 65-81.

HOFFMEYER, C.C. and WANG, J., 2003. Protecting Web Services from Interpretive-Language Injection Attacks.

HOLZER, A., KINDER, J. and VEITH, H., 2007. Using verification technology to specify and detect malware. *Computer Aided Systems Theory–EUROCAST 2007*, , pp. 497-504.

HOLZMANN, G.J., 1997. The model checker SPIN. *Software Engineering, IEEE Transactions on*, **23**(5), pp. 279-295.

HOWARD, M. and LEBLANC, D., 2009. *Writing secure code*. Microsoft press.

HUANG, Y.W., HUANG, S.K., LIN, T.P. and TSAI, C.H., 2003. Web application security assessment by fault injection and behavior monitoring, *Proceedings of the 12th international conference on World Wide Web 2003*, New York, NY, USA, pp. 148-159.

HUANG, Y.W., YU, F., HANG, C., TSAI, C.H., LEE, D.T. and KUO, S.Y., 2004. Securing web application code by static analysis and runtime protection, *Proceedings of the 13th international conference on World Wide Web 2004*, ACM, pp. 40-52.

IIVARI, J., 1991. A paradigmatic analysis of contemporary schools of IS development. *European Journal of Information Systems*, **1**(4), pp. 249-272.

JOVANOVIC, N., KRUEGEL, C. and KIRDA, E., 2006. Pixy: A static analysis tool for detecting web application vulnerabilities, *Security and Privacy, 2006 IEEE Symposium on* 2006, IEEE, pp. 6 pp.-263.

JOVANOVIC, N., KRUEGEL, C. and KIRDA, E., 2006. Precise alias analysis for static detection of web application vulnerabilities, *Proceedings of the 2006 workshop on Programming languages and analysis for security 2006*, ACM, pp. 27-36.

KALS, S., KIRDA, E., KRUEGEL, C. and JOVANOVIC, N., 2006. Secubat: a web vulnerability scanner, *Proceedings of the 15th international conference on World Wide Web 2006*, ACM, pp. 247-256.

KC, G.S., KEROMYTIS, A.D. and PREVELAKIS, V., 2003. Countering code-injection attacks with instruction-set randomization, *Proceedings of the 10th ACM conference on Computer and communications security 2003*, ACM, pp. 272-280.

KEMALIS, K. and TZOURAMANIS, T., 2008. SQL-IDS: a specification-based approach for SQL-injection detection, *Proceedings of the 2008 ACM symposium on*

Bibliography

Applied computing 2008, ACM, pp. 2153-2158.

KIEYZUN, A., GUO, P.J., JAYARAMAN, K. and ERNST, M.D., 2009. Automatic creation of SQL injection and cross-site scripting attacks, *Software Engineering, 2009. ICSE 2009. IEEE 31st International Conference on* 2009, IEEE, pp. 199-209.

KIM, H.K., 2010. Frameworks for SQL Retrieval on Web Application Security, *Proceedings of the International MultiConference of Engineers and Computer Scientists* 2010.

KODAGANALLUR, V., 2004. Incorporating language processing into java applications: A JavaCC tutorial. *Software, IEEE*, **21**(4), pp. 70-77.

LAM, M.S., MARTIN, M., LIVSHITS, B. and WHALEY, J., 2008. Securing web applications with static and dynamic information flow tracking, *Proceedings of the 2008 ACM SIGPLAN symposium on Partial evaluation and semantics-based program manipulation* 2008, ACM, pp. 3-12.

LEE, I., JEONG, S., YEO, S. and MOON, J., 2012. A novel method for SQL injection attack detection based on removing SQL query attribute values. *Mathematical and Computer Modelling*, **55**(1), pp. 58-68.

LIU, A., YUAN, Y., WIJESEKERA, D. and STAVROU, A., 2009. SQLProb: a proxy-based architecture towards preventing SQL injection attacks, *Proceedings of the 2009 ACM symposium on Applied Computing* 2009, ACM, pp. 2054-2061.

LIVSHITS, V.B. and LAM, M.S., 2005. Finding security vulnerabilities in Java applications with static analysis, *Proceedings of the 14th conference on USENIX Security Symposium* 2005, pp. 18-18.

LYON, G., 2011-last update, **SecTools.Org: Top 125 Network Security Tools**. Available: <http://sectools.org/tag/web-scanners/> [02/10, 2011].

MANIKANTA, Y.V.N. and SARDANA, A., 2012. Protecting web applications from SQL injection attacks by using framework and database firewall, *Proceedings of the International Conference on Advances in Computing, Communications and Informatics* 2012, ACM, pp. 609-613.

MARTIN, M., LIVSHITS, B. and LAM, M.S., 2005. Finding application errors and security flaws using PQL: a program query language, *ACM SIGPLAN Notices* 2005, ACM, pp. 365-383.

MATSUDA, T., KOIZUMI, D., SONODA, M. and HIRASAWA, S., 2011. On predictive errors of SQL injection attack detection by the feature of the single character, *Systems, Man, and Cybernetics (SMC), 2011 IEEE International*

Bibliography

Conference on 2011, IEEE, pp. 1722-1727.

MITRE, October 02, 2013, 2013-last update, Common Vulnerabilities and Exposures. The Standard for Information Security Vulnerability Names. Available: <http://cve.mitre.org/> [10/18, 2013].

MORLEY, D., 2008. *Understanding computers in a changing society*. Course Technology Ptr.

MOSZKOWSKI, B., 1994. Some Very Compositional Temporal Properties, *Programming concepts, methods and calculi: proceedings of the IFIP TC2/WG2. 1/WG2. 2/WG2. 3 Working Conference on Programming Concepts, Methods, and Calculi (PROCOMET'94), San Miniato, Italy, 6-10 June 1994* 1994, North-Holland, pp. 307.

MOSZKOWSKI, B., 1985. Executing temporal logic programs, *Seminar on Concurrency* 1985, Springer, pp. 111-130.

MSDN, L., 2008-last update, SQL Injection in SQL server [Homepage of MSDN], [Online]. Available: [http://msdn.microsoft.com/en-us/library/ms161953\(SQL.105\).aspx](http://msdn.microsoft.com/en-us/library/ms161953(SQL.105).aspx) [11/02, 2012].

NATARAJAN, K. and SUBRAMANI, S., 2012. Generation of Sql-injection Free Secure Algorithm to Detect and Prevent Sql-Injection Attacks. *Procedia Technology*, **4**, pp. 790-796.

ORACLE., C., 2012-last update, **Remote Method Invocation Home** [Homepage of Oracle Corporation], [Online]. Available: <http://www.oracle.com/technetwork/java/javase/tech/index-jsp-136424.html> [03/11, 2011].

OWASP, 13 April 2013, 2013-last update, **OWASP Top 10 for 2013**. Available: https://www.owasp.org/index.php/Top_10_2013 [05/10, 2013].

OWASP, 2011-last update, Category:Vulnerability - OWASP. Available: <https://www.owasp.org/index.php/Category:Vulnerability> [11/18, 2012].

OWASP, 2010-last update, Top 10 2010-Main - OWASP. Available: https://www.owasp.org/index.php/Top_10_2010-Main [11/15, 2012].

PAROS, 2004-last update, Web Application Security Assessment. Available: <http://www.parosproxy.org/> [02/18, 2011].

RIANCHO, A., 2012-last update, w3af - Web Application Attack and Audit

Bibliography

Framework. Available: <http://w3af.sourceforge.net/> [12/9, 2012].

SANTOSH, K., 2006-last update, Are stored procedures safe against SQL injection? : Palisade. Available: <http://palizine.plynt.com/issues/2006Jun/injection-stored-procedures/> [12/10, 2012].

SCOTT, D. and SHARP, R., 2002. Abstracting application-level web security, *Proceedings of the 11th international conference on World Wide Web 2002*, Citeseer, pp. 396-407.

SCOTT, D. and SHARP, R., 2002. Developing secure Web applications. *Internet Computing, IEEE*, **6**(6), pp. 38-45.

SHRIVASTAVA, R. and BHATTACHARYJI, R.S.J., 2012. SQL INJECTION ATTACKS IN DATABASE USING WEB SERVICE: DETECTION AND PREVENTION–REVIEW. *Asian Journal of Computer Science and Information Technology*, **2**(6),.

SIMPSON, M.T., BACKMAN, K. and CORLEY, J., 2010. *Hands-On Ethical Hacking and Network Defense*. Delmar Pub.

SPETT, K., 2003. Blind sql injection. *SPI Dynamics Inc*, .

SPETT, K., 2002. SQL injection: Are your Web applications vulnerable. *SPI Labs White Paper*, .

SQLDICT TOOL, 2008-last update, **SQLdict Tool** [Homepage of VulnerabilityAssessment.co.uk], [Online]. Available: <http://www.vulnerabilityassessment.co.uk/sqldict.htm> [11/13, 2012].

SQLIER, 2006-last update, BCable.net - SQLier Injection Tool. Available: <http://bcable.net/project.php?sqlier> [11/13, 2012].

SQLLIB-TOOL, 2007-last update, Open labs web application security. . Available: <http://www.open-labs.org/sqlibf113b2.tar.gz> [12/10, 2011].

SQLMAP, 2012-last update, sqlmap: automatic SQL injection and database takeover tool. Available: <http://sqlmap.org/> [11/13, 2012].

STUTTARD, D. and PINTO, M., 2011. *The Web Application Hacker's Handbook: Finding and Exploiting Security Flaws*. Wiley.

SULLO, C. and LODGE, D., 16/09/2012, 2012-last update, Nikto2 | CIRT.net. Available: <http://cirt.net/nikto2> [11/18, 2010].

Bibliography

- TAJPOUR, A., MASROM, M., HEYDARI, M. and IBRAHIM, S., 2010. SQL injection detection and prevention tools assessment, *Computer Science and Information Technology (ICCSIT), 2010 3rd IEEE International Conference on 2010*, IEEE, pp. 518-522.
- THIEMANN, P., 2005. Grammar-based analysis of string expressions, *Proceedings of the 2005 ACM SIGPLAN international workshop on Types in languages design and implementation 2005*, ACM, pp. 59-70.
- TROMER, E., 1999-last update, The Java Instrumentation Engine. Available: <http://tau.ac.il/~tromer/jie/> [10/19, 2013].
- VALMARI, A., 1998. The state explosion problem. *Lectures on Petri Nets I: Basic Models*, , pp. 429-528.
- W3C, 2009-last update, Document Object Model (DOM). Available: <http://www.w3.org/DOM/> [04/29, 2012].
- WANG, J., PHAN, R.C.W., WHITLEY, J.N. and PARISH, D.J., 2010. Augmented attack tree modeling of SQL injection attacks, *Information Management and Engineering (ICIME), 2010 The 2nd IEEE International Conference on 2010*, IEEE, pp. 182-186.
- WASSERMANN, G. and SU, Z., 2007. Sound and precise analysis of web applications for injection vulnerabilities, *ACM SIGPLAN Notices 2007*, ACM, pp. 32-41.
- WHALEY, J. and LAM, M.S., 2004. Cloning-based context-sensitive pointer alias analysis using binary decision diagrams. *ACM SIGPLAN Notices*, **39**(6), pp. 131-144.
- WOLPER, P., 1983. Temporal logic can be more expressive. *Information and control*, **56**(1), pp. 72-99.
- WOODGER COMPUTING INC, 2012-last update, Woodger Computing Inc. - General Web Architecture. Available: <http://www.woodger.ca/archweb.htm> [11/18, 2012].
- XIE, Y. and AIKEN, A., 2006. Static detection of security vulnerabilities in scripting languages, *Proceedings of the 15th conference on USENIX Security Symposium 2006*, pp. 179-192.
- YEOLE, A. and MESHARAM, B., 2011. Analysis of different technique for detection of SQL injection, *Proceedings of the International Conference & Workshop on*
-

Bibliography

Emerging Trends in Technology 2011, ACM, pp. 963-966.

ZHOU, S., ZEDAN, H. and CAU, A., 2005. Run-time analysis of time-critical systems. *Journal of Systems Architecture*, **51**(5), pp. 331-345.

Appendix 1: Java Code

Client.Java

```
import java.util.*;
import java.text.*;
import java.rmi.NotBoundException;
import java.rmi.RemoteException;
import java.rmi.registry.LocateRegistry;
import java.rmi.registry.Registry;
public class Client
{
public static String input;
private static Registry r;
private static Server server;
public static String CheckValue(String res) throws Exception
    {
        String recive;
        recive = server.CheckInput(res);
        return recive;
    }
public static void main(String args[]) throws Exception
    { System.out.println("Starting up Client ....");
      r = LocateRegistry.getRegistry("127.0.0.1");
      server= (Server) r.lookup("Server");
      System.out.println("Welcome ");
      Scanner in = new Scanner(System.in);
      do {
          System.out.println("Please Enter the string you like to check:");
          input = in.nextLine();
          }while(input==null);
      System.out.println(CheckValue(input));
    }
}
```


Appendix 1

The client, server and serverimpl files are used to transfer a submitted data from PHP page to Anatempura tool.

The client class is used to test the CheckInput function in the server file as this function communicates with Anatempura using an assertion points.

Server.java

```
import java.rmi.Remote;
public interface Server extends Remote
{
    public String CheckInput(String sentvalue) throws
    java.rmi.RemoteException;
}
```

The server file is used to define the function that can be used by the client to communicate with serverimpl that is used to implement the CheckInput function.

Appendix 1

ServerImpl.java

```
import java.io.*;
import java.util.*;
import java.rmi.RemoteException;
import java.rmi.registry.LocateRegistry;
import java.rmi.registry.Registry;
import java.rmi.server.UnicastRemoteObject;

public class ServerImpl implements Server
{
    public String CheckInput(String sentvalue) throws RemoteException
    {
        try{
            Console c = System.console();
            String in ;
            System.out.println("!PROG: assert Name:"+sentvalue+":9:!\n");
            in = c.readLine("read it \n");
            System.out.println("Tempura Result: "+in);
            return in;
        } catch(Exception e)
        {
            System.out.println("Server data erro:");
            e.printStackTrace();
            return null;
        }
    }
}
```

This file implements the CheckInput function that includes the assertion point. The assertion point will be used to deliver a submitted data to Anatempura and the checking result will be returned using the Console class which is one of the Java system libraries.

Appendix 2: PHP Code

TestDatabaseObserver.php

```
<head>
  <title>PHP Injection test</title>
</head>
<body>
  <form action="TestDatabaseObserver.php" method="post">
  <p>UserName <input type="text" name="textfield" /></p>
  <p>PassWord <input type="text" name="textfield2"/></p>
  <p><input type="submit" name="Submit" value="Submit" /></p>
  <?php
  define ("DB_HOST", "localhost"); // set database host
  define ("DB_USER", "root"); // set database user
  define ("DB_PASS",""); // set database password
  define ("DB_NAME","emad"); // set database name
  $link = mysql_connect(DB_HOST, DB_USER,DB_PASS);
  $db = mysql_select_db(DB_NAME, $link) or die("Couldn't select database");
  if (!$link) { die('Could not connect: ' . mysql_error()); }
  echo 'Connected successfully';
  echo '<p><font size="4">=====</font></p>';
  if (isset($_POST['textfield']))
  { if(isset($_POST['textfield2']))
  {   $user=$_POST["textfield"];
      $pwd=$_POST["textfield2"];
          $ee="select * from test1 where ee='$user' and UPass='$pwd'";
          echo '<p><font size="4">SQL statment:</font></p>';
          echo $ee;
          $result = mysql_query($ee) or die ("<p>Wrong entry try again!");
          echo '<p><font size="4">Number of record:
  </font></p>'.mysql_num_rows($result);} }
  mysql_close($link);
  ?>
  </form> </body> </html>
```

Appendix 2

The TestDatabaseObserver.php file will be used to test the records number condition using *mysql_num_rows(\$result)* that can retrieve the matching record numbers of a select statement.

TestDatabaseObserverInsert.php

```
function begin()
{ mysql_query("BEGIN");}
function commit()
{ @mysql_query("COMMIT");}
function rollback()
{ @mysql_query("ROLLBACK");}
if (isset($_POST['textfield']))
{ if(isset($_POST['textfield2']))
{ $user=$_POST["textfield"];
  $pwd=$_POST["textfield2"];
  begin();
  $query = "INSERT INTO test1(ee,UPass) values('$user','$pwd)";
  echo '<p><font size="4">SQL statment:</font></p>';
  echo $query;
  $result = mysql_query($query) or die ("<p>Wrong entry try again!");
  $RowNo = mysql_affected_rows();
  echo '<p><font size="4">Number of inserted record: </font></p>'.$RowNo;
  if ($RowNo=1)
  { commit();}
  else
  { rollback();
    echo '<p><font size="4">Wrong entry</font></p>';
  }
}}
mysql_close($link);
?>
</form></body></html>
```

Appendix 2

The TestDatabaseObserverInsert.php file has same database connection procedure that is used in a TestDatabaseObserver.php file. This is used to test the record number with an insert statement using *mysql_affected_rows()* that can retrieve the effected record numbers of the insert statement.

TestOutput.php

```
if (isset($_POST['textfield']))
{ if(isset($_POST['textfield2']))
{ $user=$_POST["textfield"];
  $pwd=$_POST["textfield2"];
  $ee="select * from test1 where ee='$user' and UPass='$pwd'";
  echo '<p><font size="4">SQL statment:</font></p>';
  echo $ee;
  $result = mysql_query($ee);
if (mysql_errno())
{ switch (mysql_errno())
{ case 1064:
  echo '<p><font size="4">syntax error try again! </font></p> ';
  break;
case 1061:
  echo '<p><font size="4">There is a duplicated entry </font></p>';
  break;
default:
  echo '<p><font size="4"> Wrong entry try again </font></p> ';}}
  else { echo '<p><font size="4">Number of record:
</font></p>'.mysql_num_rows($result);}
}}
mysql_close($link);
?> </form> </body> </html>
```

The TestOutput.php file is used to test the output checker that checks the message content and block any database information using a switch case command.

Appendix 3: Tempura Code

Anascii Function

```
define codes = " !\"#$%&'()*+,-
./0123456789:;<=>?@ABCDEFGHIJKLMN
NOPQRSTUVWXYZ[\]^_`abcdefghijklmnop
qrstuvwxyz{|}~".

define anascii(X) = {

  if X>=32 and X<=126 then { codes[X-32] } else { "===" }

}.
```

This function is used to return the Ascii code for any character and it.

LowerCase function

```
define LowerCase(X,ReceiveString)= { /* X will be used for return variable*/
exists A2,A1:{stable(ReceiveString) and List(A2,|ReceiveString|) and A2=[] and
  { for i < |ReceiveString|
    do{
      A2:=A2+[ascii(ReceiveString[i])] and skip } };{
      A1="" and stable(A2) and stable(ReceiveString) and
      { for i < |ReceiveString|
        do{
          if (A2[i]>=65 and A2[i]<=90)
            then {A1 := A1 + anascii(A2[i]+32) and skip }
            else {A1:=A1+ReceiveString[i] and skip } } } and fin(X=A1)
          }}
}.
```

This procedure is used to transform all the character in a lowercase.

Appendix 3

DecreaseSpaces function.

```
define DecreaseSpaces(X,S) = {
  exists A1,A2,E : { List(A2,|S|) and stable(S) and A2=[] and {
  for i < |S| do{
  if S[i]~=" "
    then {A2:=A2+[i] and skip }
    else {skip and A2:=A2}}};
  {stable(S) and stable(A2) and A1="" and E=A2[0] and
  for i < |A2|-1 do{
  if (A2[i+1]-A2[i]=1)
  then{ stable(A1) and E:=E and skip }
  else{ A1 := A1+S[E..A2[i+1]]+" " and E:= A2[i+1] and skip}
  } and fin(X=A1+S[E..A2[|A2|-1]+1])
  }
  }
}.
```

This function has been used to reduce the number of sequence spaces to be one space.

StringTokens function

```
define StringTokens(X,S) = {
  exists A1,A2,E : { List(A2,|S|) and stable(S) and A2=[] and {
  for i < |S| do{
  if S[i]~=" " then {A2:=A2+[i] and skip } else {skip and A2:=A2}}};
  {stable(S) and stable(A2) and List(A1,|S|) and A1=[] and E=A2[0] and
  for i < |A2|-1 do{
  if (A2[i+1]-A2[i]=1)
  then { A1:=A1 and E:=E and skip }
  else{ A1:= A1+[S[E..A2[i+1]]] and E:= A2[i+1] and skip}
  } and fin(X=A1+[S[E..A2[|A2|-1]+1]])
  }
  }
}.
```

This function is used to separate the string into tokens.

Appendix 3

SearchGenKeywords function

```
define SearchGenKeywords(X,S) = {
  exists A,I,J,SQLKeys:{{
    StringTokens(X,S) and fin(T=X)};{
    stable(T) and
    SQLKeys=["select","drop","update","delete","alter","create","union","declare",
"bigen","exec","ascii"]
    and stable(SQLKeys) and I=0 and J=0 and A='g' and {
    while I < |SQLKeys|
      do{ while J < |T|
        do{
          if (SQLKeys[I]=T[J])
            then {I:=|SQLKeys| and J:=|T| and skip and A:='n'}
            else {I:=I and skip and J:=J+1 and A:=A }
              };{ I:=I+1 and J:=0 and skip and A:=A }
                } and fin(X=A)
          } } } }.
```

This function has been used to check if the string contains any SQL keywords.

SearchSpecKword function

```
define SearchSpecKword(X,S,D) = {
  exists A,I,Q,G:{stable (D) and stable (S) and {
    {StringTokens(Q,S)};
    {stable(Q) and I=0 and J=0 and A='n' and G=0 and {
      while I < |D|
        do{if (D[I]=Q[0])
          then {I:=|D| and skip and A:='y' and G:=|Q[0]|}
          else { I:=I+1 and skip and A:=A and G:=G }
            }
          }and fin(X=[A,G])    } } } }.
```

This function is used to check if the string contains a specific SQL keyword.

Appendix 3

StringCommentCut function

```
define StringCommentCut(X,S) = {
  exists A1,A2,T,B : { List(A1,|S|) and stable(S) and A1=[] and T=0 and B=0 and {
    for ( i < |S| )
      do{if (S[i..i+2]= "/*" and T=0)
        then {T:=1 and B:=i and A1:=A1}
        else { if (S[i..i+2]= "*/" and T=1 )
          then { A1:=A1+[B]+["s"]+[i+1] and T:=0 and B:=B}
          else { A1:=A1 and T:=T and B:=B} } and skip } };
  { stable(S) and stable(A1) and A2="" and
  if |A1|>0
    then {for (i<|A1|)
      do{if A1[i]="s"
        then { if (|A2|=0)

          then {A2:=S[0..A1[i-1]]+S[A1[i+1]+1..i+1]+S[A1[i+1]+1..A1[i+2]]}

          else { if ((i+2)=|A1|)
            then{ A2:=A2+S[A1[i+1]+1..i+1]+S[A1[i+1]+1..|S|]}
            else{ A2:=A2+S[A1[i+1]+1..i+1]+S[A1[i+1]+1..A1[i+2]]} } }
          else {A2:=A2} and skip} }
          else {A2:= S and skip}
          }and output (S) and fin(X=A2)
          } }.
```

This function can be used to filter a string from the comment that can be used to elude from the detection function.

GoodEntry function

```
define GoodEntry(X,S)={
  exists A,I:{
    stable(S) and A='g' and I=0 and while I < |S| do {
      if (S[0..2]='0x' or S[I..I+3]=" 0x")
        then {I:=|S| and A:='n' and skip}
        else {if (ascii(S[I])>= 97 and ascii(S[I])<= 122)
          then { stable(A) and I:=I+1 and skip}
          else {if (ascii(S[I])>= 48 and ascii(S[I])<=58)
            then {stable(A) and I:=I+1 and skip}
            else {if (ascii(S[I])= 32 or ascii(S[I])= 33 or ascii(S[I]) = 63
or ascii(S[I]) = 95 or ascii(S[I]) = 43 or ascii(S[I]) = 61)
          then {stable(A) and I:=I+1 and skip}
          else {next(A)='n' and I:=|S| and skip } } } } /*return n means not good entry*/
          and fin (X=A)
          } }.
```

Appendix 3

BadEntry function

```
define BadEntry(X,S)={
exists A,D,I,F,G: {
{stable(S) and stable(struct(S)) and A='u' and I=0 and
  while I < |S|
    do{ if ((S[I..I+2]= "--" and (I+2 <= |S|)) or (S[I..I+3]= "- -" and (I+3 <= |S|)))
      then{I:=|S| and A:='b'}
      else{A:=A and I:=I+1} and skip}};
{stable(S) and skip and if (A='b') then{A:=A and I:=I} else{A:=A and I:=0}};
{stable(S) and
while I < |S|
  do{ if (S[I..I+2]= "/"* and (I+2 < |S|)) or (S[I..I+3]= "/"* and (I+3 < |S|))
    then{ D=I+2 and
      while (D < |S|)
        do {if (S[D..D+2]= "*"/* or S[D..D+3]= "*"/*)
          then {I:=|S| and A:='b' and skip and D:=|S|}
          else {I:=|S| and A:='u' and skip and D:=D+1}}}}
    else{ if (S[I..I+2]= "/"* and (I+2 >= |S|)) or (S[I..I+3]= "/"* and (I+3 >= |S|))
      then{I:=|S| and A:='b' and skip and D:=|S|}
      else{A:=A and skip and I:=I+1}}}};

{stable(S) and skip and if (A='b') then{A:=A and I:=I} else{A:=A and I:=0}};
{stable(S) and {
  while I < |S|
    do{
      if (S[I]= ";" and I+1<|S|)
        then{ {stable (A) and stable(I) and
          {SearchSpecKword(N,S[I+1..|S|],["drop","alter","create","declare"]);
          stable N and
          SearchSpecKword(X,S[I+1..|S|],["select","update","bigen","exec","delete","insert"])};
          {if N[0]= 'y'
            then {A:='b' and I:=|S| and stable X}
            else {stable X and stable (A) and stable(I)} and skip};
            {D=I+X[1]+1 and
            if X[0]='y'
            then{ while (D < |S|)
              do {if (S[D]= ";" or S[D]= "#" or S[D]= "--" or S[D]="" or S[D..D+2]= " @" or
                S[D]= "*" or S[D..D+5]="table" or S[D]= "#" or S[D..D+1]= "/"*)
                then {A:='b' and skip and D:=|S| and I:=|S|}
                else {A:='OB' and skip and D:=D+1 and I:=|S|}}}}
            else{A:=A and skip and I:=I+1}}}}
          else{if (S[I]= ";" and (I+1 >= |S|)
            then{I:=|S| and A:='b' and skip}
            else{A:=A and skip and I:=I+1}}}}};
```

Appendix 3

BadEntry part2.

```
{ stable(S) and skip and if (A ='b') then{A:=A and I:=I} else{A:=A and I:=0} };
{ stable(S) and {
  while I < |S|
    do{ if (S[I]= "" and I+1<|S|)
      then{ { stable (A) and stable(I) and
SearchSpecKword(X,S[I+1..|S|],["union","or","and","group","order"]) };
{ D=I+X[1]+1 and if X[0]='y'
  then{ while (D < |S|)
    do { if (S[D]= ";" or S[D]="," or S[D]="=" or S[D]="-" or S[D]="%" or
S[D]="'" or S[D..D+3]="all" or S[D]="#" or S[D..D+1]="/*" or
S[D]=">" or S[D]="<" or S[D..D+6]="select" )
  then {A:='b' and skip and D:=|S| and I:=|S|}
  else {A:='OB' and skip and D:=D+1 and I:=|S|} } }
  else{A:=A and skip and I:=I+1} } }
  else{if (S[I]= "") and (I+1 >= |S|)
    then{I:=|S| and A:='b' and skip}
    else{A:=A and skip and I:=I+1} } } } };
{ stable(S) and skip and if (A ='b') then{A:=A and I:=I} else{A:=A and I:=0} };
{ stable(S) and
while I < |S|
do{ if ((S[L.I+3]= " 0x") and
(((ascii(S[I+3])>= 48 and ascii(S[I+3])<=57) or (ascii(S[I+3])>= 97 and
ascii(S[I+3])<=102)) and
((ascii(S[I+4])>= 48 and ascii(S[I+4])<=57) or (ascii(S[I+4])>= 97 and
ascii(S[I+4])<=102))))
  then{I:=|S| and skip and A:='b'}
  else{A:=A and skip and I:=I+1} } };
{ stable(S) and skip and if (A ='b') then{A:=A and I:=I} else{A:=A and I:=0} };
{ stable(S) and
  while I < |S|
    do{if (S[L.I+6]= " char(" or S[L.I+7]= " char (" or S[L.I+5]= "char(" or
S[L.I+6]= " exec(" or S[L.I+5]= "exec(" or S[L.I+6]= "select(" or S[L.I+6]= " select("
or S[L.I+3]= "%00" or S[L.I+3]= "%2f" or S[L.I+3]= "%2a" or S[L.I+5]= "%252f" or
S[L.I+5]= "%252a" )
      then{I:=|S| and skip and A:='b'}
      else{A:=A and skip and I:=I+1} } };
{ stable(S) and skip and if (A ='b') then{A:=A and I:=I} else{A:=A and I:=0} };
  { stable(S) and if A='u' then A:='g' else A:=A and skip }
  and fin (X=A)
  } }.
```

The GoodEntry function checks if the string contains any SQL keyword or symbols

The BadEntry function checks the string if contains any of the exiting SQL injection attacks.

Appendix 3

extend_list function

```
define extend_list(L,d) = {
  list(next L, |L|+1) and
  forall i<|L|+1: if i<|L| then L[i]:=L[i] else L[i]:=d
}.
```

This function is used to increase the size of the array and add new element to this array.

Filter function

```
define Filter(X,N) = {
  exists A,T:{
  {List(A,|N|) and A=[] and List(T,|N|) and
  while (|N| ~ =0 ) do{
    {A:=A+[N[0]] and N:=N[1..|N|] and skip};
    {stable A and stable N and
    T=[] and for i<|N|
      do { if N[i] ~ =A[|A|-1]
        then {T:=T+[N[i]]}
        else {T:=T} and skip };{N:=T and stable A and skip}
    } } and fin(X=A)    } }.
```

This function is used by a behavioural function to remove a repeated IP address

SearchProcedure

```
define SearchProcedure(X,S,N) = {
  exists A,I,J,DProcedure:{ {
  StringTokens(T,S)};
  {stable(T) and DProcedure=["declare","select","exec"]
  and stable(DProcedure) and I=0 and J=0 and List(A,|T|) and A=[] and {
  while I < |DProcedure|
    do{ while J < |T|
      do{ if (DProcedure[I]=T[J])
        then {if T[J+1][0..1] ="@" then {I:=|DProcedure| and J:=|T| and skip
and A:=A+[N]+[DProcedure[I]]+[T[J+1]]}}
        else {I:=I and skip and J:=J+1 and A:=A}};
      { I:=I+1 and J:=0 and skip and A:=A}
    } and fin(X=A)  } } }.
```

This function is to check if there are frequent uses of a specific procedure.

Appendix 3

ReltedIP function

```
define ReltedIP(H) = {
exists I,X,T,HH,D: {
{ stable(H) and { List(HH,|H|) and HH=[] and List(D,|HH|) and D=[] and
for i<|H| do { if H[i][1] ='b'
then{ HH:=HH+[i] and D:=D+[H[i][0]]
else{ HH:=HH and D:=D } and skip } };
{ stable HH and stable H and Filter(X,D)};
{List(T,|HH|) and T=[] and I=0 and stable HH and stable H and stable X
and while I < |X|
do{ for j<|HH| do { stable I and
if X[I]=H[HH[j]][0]
then {T:=T+[H[HH[j]][3]] }
else {T:=T} and skip };{T:=[] and I:=I+1 and format("%t \n",[[X[I]]+[T]])
and skip } } } } }.
```

This function is used by a behavioural function to find related IP address that is matched with existing attacks.

CheckingModel procedure

```
define CheckingModel(X,S) = {
exists D,R,F,T,M: {
{ stable(S) and LowerCase(T,S)};
{ stable(T) and DecreaseSpaces(R,T)};
{ stable(R) and SearchGenKeywords(D,R)};
{ stable(R) and stable(D) and GoodEntry(F,R)};
{ stable(R) and stable(D) and stable(F) and
{ if (D='g' and F='g')
then{M='g' and empty}
else{BadEntry(M,R)} } } and fin(X=M)
}.
```

This procedure is used to check if the string contains any form of SQL injection attack.

Appendix 3

Behavioural function

```
efine BuildBehaviour(H) = {
exists I,X,T,HH,G,D: {
{ stable(H) and
{ ReltedIP(H) and format("\n*****Related IP with bad input *****\n\n") } ;
{ format("***** Sequence of bad input *****\n\n") and
if |H|>2
then { List(G,|H|) and G=[] and
for i<|H|-2 do { if (H[i][1] = 'b' and H[i+1][1] = 'b' and H[i+2][1] = 'b' and i~=0)
then { if (H[i-1][1] = 'b' and H[i][1] = 'b' and H[i+1][1] = 'b') and i>0
then { G:=G+[[H[i+2][0]]+[H[i+2][3]]] }
else { G:=G+["\n Seq. Attack:
"]+[[H[i][0]]+[H[i][3]]]+[[H[i+1][0]]+[H[i+1][3]]]+[[H[i+2][0]]+[H[i+2][3]]]+["\n"] }
}
else { G:=G } and skip } and fin(output(G) and format ("\n \n ")) }
else { empty } };
{ format("***** Percentage of all input *****\n\n") and
T=0 and I=0 and
for i<|H| do {
if H[i][1] = 'b'
then { T:=T+1 and I:=I }
else { T:=T and if H[i][1] = 'OB' then I:=I+1 else I:=I }
and skip } };
{ stable(T) and stable(I) and
G=itof(T)*$100$/itof(|H|) and D=itof(I)*$100$/itof(|H|) and
format("the percentage of bad inputs is: %F \n",G) and
format("the percentage of unspecified inputs is: %F \n",D) and
format("the percentage of good inputs is: %F \n\n",($100$-(G+D))) and empty };

{ format("***** Related Database Procedure *****\n\n") and
List(HH,|H|) and HH=[] and for i<|H| do {
if H[i][1] = 'b' then { { SearchProcedure(X,H[i][2],i) and stable(HH) };
{ if X~=[] then HH:=HH+[X] else HH:=HH and skip } }
else { skip and HH:=HH } } };
{ stable(HH) and for i<|HH| do { if HH[i][2]=HH[|HH|-1][2] then { format ("
[%t,%t,%t] \n",H[HH[i][0]][0],H[HH[i][0]][2],H[HH[i][0]][3]) and skip } } } } }.
```

This function has been run in several stages to investigate related attacks.

Appendix 3

Testing functions

```
/* run */ define TestCheckingModel() = {
  exists X,S,H:{
    {set outfile="stdout" and set infile="Piggyback" and
    list(H,0) and input X and
    while (X ~= 0)
      do {extend_list(H,X) and skip and next input X }};
  {stable(H) and for i<|H| do {{CheckingModel(S,H[i][1])}};
  {skip and if S = H[i][0]
  then{ format("Number %d done %t : %t \n",i+1,S,H[i][1])}
  else {format("Conflict : Number %d is %t Expecting %t for: %t
\n",i+1,S,H[i][1],H[i][0])} }}}}.
```

This function is used to simulate the checking model with files that consist of various types of attacks. it is also used to check a safe input.

```
/* run */ define TestCheckingRelted () = {
exists X,S,H,BehaviourArray:{
  list(BehaviourArray,0) and
  {set outfile="stdout" and set infile="behaviour" and
  list(H,0) and input X and
  while (X ~= 0) do {extend_list(H,X) and skip and next input X }};
  {stable(H) and BehaviourArray=[] and
  for i<|H| do {{CheckingModel(S,H[i][1]) and stable(BehaviourArray) }};
  {stable(S) and
  if S = H[i][3]
  then{ format("Number %d done %t : %t \n",i+1,S,H[i][1])}
  else {format("Conflict : Number %d is %t Expecting %t for: %t
\n",i+1,S,H[i][1],H[i][3])}
  and extend_list(BehaviourArray,[H[i][0]]+[S]+[H[i][1]]+[H[i][2]])
  and skip }}};
  {stable(H) and format ("\n \n") and stable(BehaviourArray) and
  BuildBehaviour(BehaviourArray)} }}.
```

This function is used to simulate a behavioural function with input samples to find related attacks.