

EFFECTIVENESS OF STRUCTURED QUERY LANGUAGE INJECTION  
ATTACKS DETECTION MECHANISMS

NURUL ZAWIYAH BINTI MOHAMAD

A project report submitted in fulfilment of the  
requirements for the award of the degree of  
Master of Science (Computer Science)

Faculty of Computer Science and Information Systems  
Universiti Teknologi Malaysia

OCTOBER 2008

To my beloved father, mother, brother and sister.

## ACKNOWLEDGEMENT

Alhamdulillah, praise to Allah S.W.T for giving me the patience during the course of completing this project. I would like to express my sincere appreciation to my supervisor, Associate Professor Dr. Naomie Salim for having the patience to guide me through this, and for the knowledge that has been shared along the way. Thank you very much to Universiti Malaysia Sarawak (UNIMAS) and Kementerian Pengajian Tinggi for funding my studies. Besides, I would like to thank my wonderful parents, and my siblings for their support and encouragement along the course of my Masters degree.

A bunch of appreciation to my course mates, for sharing information, views and tips and also for showing their support to me, and I wish everyone would graduate with flying colors. Thank you very much.

## ABSTRACT

Database security is one of the most essential factors in keeping stored information safe. These days, web applications are used widely as a meddler between computer users. Web applications are also used mostly by e-commerce companies, and these types of applications need a secured database in order to keep sensitive and confidential information. Since SQL injection attacks occurred as a new way of accessing database through the application rather than directly through the database itself, they have become popular among hackers and malicious users. Many prevention and detection mechanisms are developed to handle this problem but these mechanisms have their limitations. In this study, two mechanisms, AMNESIA and SQL Guard are adopted for a practical evaluation to search for the better technique in detecting SQL injection attacks. These techniques will be called Technique A and Technique B respectively and will be evaluated on their effectiveness and efficiency using precision and recall measure against two web applications, Mekar and myMarket. The study will show that Technique B is a better approach on detecting SQL injection attacks.

## ABSTRAK

Keselamatan pangkalan data adalah salah satu faktor yang amat penting dalam penyimpanan maklumat. Aplikasi yang berasaskan web digunakan oleh kebanyakan syarikat – syarikat e-dagang dan aplikasi seperti ini memerlukan pangkalan data yang selamat kerana melibatkan maklumat peribadi dan juga maklumat sensitif para pengguna lain. Serangan suntikan SQL adalah salah satu cara untuk mencerobohi pangkalan data melalui laman web, dan cara ini telah terkenal di kalangan penggadam untuk mendapatkan maklumat peribadi pengguna lain, dan digunakan untuk kepentingan diri. Pelbagai teknik telah diperkenalkan untuk mengesan serangan suntikan SQL ini, namun teknik-teknik ini masih mempunyai pembatasan tersendiri. Di dalam laporan ini, dua teknik, iaitu AMNESIA dan SQL Guard telah dipilih untuk diujikaji dari segi keberkesanan dan kecekapan dalam mengesan serangan suntikan SQL. Teknik yang terpilih, dinamakan teknik A dan teknik B akan diujikaji ke atas dua aplikasi web, iaitu Mekar dan myMarket. Kajian ini menunjukkan bahawa teknik B adalah teknik yang lebih berkesan dalam mengesan serangan suntikan SQL.

## TABLE OF CONTENTS

CHAPTER	TITLE	PAGE
	<b>DECLARATION</b>	ii
	<b>DEDICATION</b>	iii
	<b>ACKNOWLEDGEMENT</b>	iv
	<b>ABSTRACT</b>	v
	<b>ABSTRAK</b>	vi
	<b>TABLE OF CONTENTS</b>	vii
	<b>LIST OF TABLES</b>	x
	<b>LIST OF FIGURES</b>	xii
	<b>LIST OF SYMBOLS</b>	xiii
	<b>LIST OF ABBREVIATIONS</b>	xiv
	<b>LIST OF APPENDICES</b>	xv
<b>1</b>	<b>PROJECT OVERVIEW</b>	<b>1</b>
	1.1 Introduction	1
	1.2 Problem Background	2
	1.3 Problem Statement	4
	1.4 Project Aim	5
	1.5 Objectives	5
	1.6 Project Scope	6
	1.7 Significance of the project	6

1.8	Report Organization	7
<b>2</b>	<b>LITERATURE REVIEW</b>	<b>8</b>
2.1	Introduction	8
2.2	SQL Injection	8
2.2.1	SQL Injection Vulnerabilities	9
2.2.2	SQL Injection Attacks	9
2.2.2.1	SQL Injection Attacks: Types and Goals	10
2.2.2.2	Examples of SQL Injection Attacks	13
2.3	Prevention and Detection of SQL Injection Attacks	20
2.4	Evaluation using Precision and Recall	27
2.5	Effectiveness and Efficiency	29
2.5	Discussion and Summary	30
<b>3</b>	<b>METHODOLOGY</b>	<b>31</b>
3.1	Introduction	31
3.2	Operational Framework	31
3.2.1	Phase I: Project Planning	33
3.2.2	Phase II: Comparison Experiment	33
3.2.2.1	Web applications as test bed and a set of inputs	34
3.2.2.2	Mechanisms selection and algorithm implementation	35
3.2.2.3	Technique embedment into PHP web applications	43
3.2.3	Phase III: Evaluation and Reporting	44
3.3	Hardware and Software Requirements	45

3.4 Summary	46
<b>4 EXPERIMENTAL RESULTS AND DISCUSSION</b>	<b>48</b>
4.1 Introduction	48
4.2 Phase I: Project Planning	48
4.3 Phase II: Comparison Experiment	49
4.4 Discussion	52
4.5 Summary	54
<b>5 CONCLUSION</b>	<b>55</b>
5.1 Introduction	55
5.2 Findings	55
5.3 Future Works	56
5.4 Summary	57
<b>REFERENCES</b>	<b>58</b>
<b>Appendices A - E</b>	<b>61 - 83</b>

**LIST OF TABLES**

<b>TABLE NO.</b>	<b>TITLE</b>	<b>PAGE</b>
2.1	Types and goals of SQL injection attacks	11
2.2	Related work in detecting and preventing SQLIAs	21
2.3	Selected mechanisms for the purpose of the study	24
2.4	Technique used in selected mechanisms	25
3.1	Several examples on ATTACK and NON ATTACK inputs for testing	34

**LIST OF FIGURES**

<b>FIGURE NO.</b>	<b>TITLE</b>	<b>PAGE</b>
2.1	Example of SQL-query model	27
2.2	Example of SQL Parse Tree with comment token	27
2.2	Precision and Recall for a collection of SQL statements	28
3.1	Operational Framework	32
3.2	Testing and Evaluating developed mechanisms	36
3.3	Identifying a hotspot	43
3.4	Hotspot after instrumentation	44
4.1	Precision and Recall using Technique A	50
4.2	Precision and Recall using Technique B	51

4.3	Comparison on both techniques for Mekar application	52
4.4	Comparison on both techniques for myMarket application	53

**LIST OF SYMBOLS**

L	Set of SQL injection attacks launched
L	Number of SQL injection attacks launched
A	Set of SQL statements detected as attacks
A	Number of SQL statements detected as attacks
La	Number of actual SQL injection attacks detected

## LIST OF ABBREVIATIONS

SQL	Structured Query Language
SQLIAs	Structured Query Language Injection Attacks
DF	Detection-focused
PF	Prevention-focused
FP	False Positives
FN	False Negatives

**LIST OF APPENDICES**

<b>APPENDIX</b>	<b>TITLE</b>	<b>PAGE</b>
<b>A</b>	Project Plan	61
<b>B</b>	Inputs lists	63
<b>C</b>	Technique Algorithms	65
<b>D</b>	Program Code	68
<b>E</b>	Testing data	76

## **CHAPTER 1**

### **PROJECT OVERVIEW**

#### **1.1 Introduction**

Database security is the degree to which all data is fully protected from tampering or unauthorized acts. Security vulnerability, security threat and security risk are the menaces to database. In security threat, individuals are one of its types of threat, where individuals intentionally or unintentionally inflict damage, violation, or destruction to all or any of the database environment components. These individuals include hackers, terrorists, organized criminals, and employees.

In this 21<sup>st</sup> century, web applications are extensively used for generating information, distributing information from an organization to the users over a network, and also as a platform to run e-Commerce websites. These web applications usually have a back-end database to keep the customers' information and all relevant information about the customers, including credit card details, and private data. As these web applications usually accept data from users and bring these data to access the

back-end database, these types of applications carry the possibility of being exposed to the SQL injection attacks.

An SQL injection attack (SQLIA) is a subset of the un-sanitized input vulnerability and occurs when an attacker attempts to change the logic, semantics or syntax of a legitimate SQL statement by inserting new SQL keywords or operators into the statement. (Muthuprasanna *et al.*, 2006). It is also known as a technique that abuses the security vulnerability occurring in the database layer of an application. The attack happens when an attacker is able to insert a series of SQL statements into a query by manipulating input data into the application. Once the attacker successfully injects the attack into the database, the database will be susceptible of being altered, extracted or even dropped. SQL injection happens on web application (ASP, PHP, JSP etc.) itself rather than on the web server or services running in the operating system. According to Muthuprasanna *et al.* (2006), very little emphasis is laid on securing these applications.

## **1.2 Problem Background**

SQL injection attacks have been a serious problem since 2002. Studies by Thomas and William (2007) showed that since 2002, 10% of total cyber vulnerabilities were SQL injection vulnerabilities. The percentage keeps on increasing as developers discover ways to prevent and detect the SQL injection attacks, resulting to attackers developing a wide array of attack techniques that can be used to exploit SQL injection vulnerabilities. The main problem in preventing and detecting any type of SQL injection attacks is the development of a defense mechanism which guarantee no false positives in detecting the attacks (high precision) while ensuring the actual attacks injected are detected (high recall).

There are seven main types of SQL injection attacks (Halfond and Orso, 2008). They are tautologies, union queries, piggybacked queries, malformed queries, inference, alternate encodings and leveraging stored procedures. Each of these types has different techniques to launch an SQL injection attack. They also have their goal; which is the intention of the attacker, whether to add or modify data, or to extract data for personal use and many more.

According to Muthuprasanna *et al.*, (2006) there is no known fool-proof defense against the SQLIAs. The researchers in this field have built several mechanisms, or framework to handle SQLIAs. These mechanisms have been proven effective and efficient when tested, but they do have their own limitations. Some of these mechanisms are for the use during static analysis (McClure and Kruger (2005); Halfond *et al.* (2006); Fu *et al.* (2007)), where the analysis are carried out during coding of the application to identify any vulnerabilities of the application.

Besides static analysis, a number of mechanisms have been developed to be used during both static and dynamic analysis (Halfond and Orso (2005), Muthuprasanna *et al.* (2006), Wei *et al.* (2006) and Kosuga *et al.* (2007)). These mechanisms analyze codes during the static analysis, and validate the coding during dynamic analysis, which is the execution of attacks towards the analyzed codes.

Halfond *et al.* (2006) mentioned that the cause of SQL injection vulnerabilities is relatively simple and well understood; which is insufficient validation of user input. Alfantookh (2004) and Lin and Chen (2006) have developed a mechanism that does the input validation before it is being sent to the web server.

From the works of Halfond *et al.* (2006), an analytical comparison and countermeasures have been carried out to compare existing techniques or mechanisms that exist. The techniques are divided into *detection-focused*, techniques that detect attacks during runtime, and *prevention-focused*, techniques that statically identify vulnerabilities in the code. It is found that a *combination* of static and dynamic analysis *detects* more attack types instead of other techniques. For the purpose of this study, two mechanisms (AMNESIA by Halfond and Orso (2005), and SQLGuard by Buehrer *et al.* (2005)) have been selected to be analyzed and implemented in a new environment. However, Halfond *et al.* (2006) stated that they did not take precision into account in this evaluation. Many of the techniques considered are based on some conservative analysis or assumptions that can result in false positives.

According to Halfond *et al.* (2006), future evaluation work should focus on evaluating the techniques' precision and effectiveness in practice. There has not been an effectiveness and efficiency evaluation carried out to ensure the techniques are fully effective and efficient in practice, although some are claimed to be 100% effective in their experimental result. Empirical evaluations would allow for comparing the performance of the different techniques when they are subjected to real-world attacks and legitimate inputs.

### **1.3 Problem Statement**

The purpose of this thesis is to provide an experimental analysis and comprehensive comparison of defense mechanisms based on the selected mechanisms. The key factors to be evaluated in the comparison are the correctness, effectiveness and the efficiency of the mechanisms. The main problem in preventing and detecting any

type of SQL injection attacks is the development of a defense mechanism which guarantee no false positives in detecting the attacks (high precision) while ensuring the actual attacks injected are detected (high recall). The problem statements emphasizing the goal of this study are: *which SQL injection attack detection approaches is effective and efficient to successfully detect SQL injection attacks?*

#### **1.4 Project Aim**

The aim of this project is to determine the effective and efficient detection technique that is able to detect the SQL injection attacks.

#### **1.5 Objectives**

In order to accomplish the hypothesis of the study, three objectives have been identified as stated below:

1. To investigate and classify SQL injection attacks by types of attacks.
2. To study the techniques of both mechanisms and create the algorithms.
3. To investigate the effectiveness and efficiency of both the SQL injection attacks detection techniques.

## 1.6 Project Scope

The scopes of this project are defined as follows:

1. The techniques are developed in a PHP language.
2. The mechanisms that will be compared in this study are AMNESIA and SQL Guard.
3. The web applications involved for experiment are *mekar* and *myMarket*.
4. The SQL query keyword that will be affected in this study is SELECT statements only.

## 1.7 Significance of the Project

This project will investigate the effectiveness and efficiency of detection approaches in order to spot an incoming of injected attacks through web applications. The analysis will be carried out and to compare which technique works better and more effective. The result of this study can be used to verify the effectiveness and efficiency of the tested approaches and will contribute in future works for future improvements.

## **1.8 Report Organization**

This thesis is divided into five chapters. Chapter 1 will introduce the basic information of SQL injection attacks, problem background, problem statement, objectives and scope. Chapter 2 is a literature review and will explain in-depth on SQL injection attacks and their prevention or detection approaches. Chapter 3 will explain the way this project is conducted, and the methodology that is used to handle the evaluation and the testing. Chapter 4 will show the experiment results from what has been done, and Chapter 5 will summarize and conclude on the thesis.

## **CHAPTER 2**

### **LITERATURE REVIEW**

#### **2.1 Introduction**

In this chapter, a detailed description on SQL injection attacks will be discussed. Discussion will be on the types of the attacks, how risky the attacks are and the impact on web applications if the injections are successful. The same goes with the prevention and detection approaches in handling SQL injection attacks.

#### **2.2 SQL Injection**

Under this subtopic, the two important issues in SQL injection are covered. The first one is SQL injection vulnerabilities and then followed by SQL injection attacks.

### **2.2.1 SQL Injection Vulnerabilities**

SQL injection vulnerabilities have been described as one of the most serious threats for web applications (Halfond et al. 2006; Thomas and Williams, 2007). The two common spots for a vulnerability usually are the user input boxes and the uniform resource locator (URL). These vulnerabilities may allow these attackers to gain complete access to the underlying databases. As most of the database consists of sensitive information and confidential information, these attackers can do harm to the database, resulting security violations, such as identity theft, loss of confidential data and also fraud.

The cause of SQL injection vulnerabilities is relatively simple and well understood, as stated by Halfond et al. (2006); which is insufficient validation of user input. To tackle this problem, developers have proposed a range of coding guidelines that promote defensive coding practices. Unfortunately, in practice, this approach is human-based, thus, it is prone to error. To overcome this problem, developers and researchers have come up with detection and prevention techniques such as AMNESIA (Halfond and Orso (2005)), SQL Guard (Buehrer et al. (2005)), SQL Check (Su and Wasserman (2006)), Java Static Tainting (Livshits and Lam (2005)) and WebSSARI (Huang et al. (2004)) and many others.

### **2.2.2 SQL Injection Attacks (SQLIAs)**

There are plenty of definitions for this term. In general, SQL Injection Attacks (SQLIAs) are a class of code injection attacks that take advantage of the lack of validation of user input. These attacks occur when developers combine hard-coded strings with user provided input to create dynamic queries (Halfond et al. 2008). They occur when an attacker is able to insert a series of SQL statements into

a query by manipulating the input parameters into an application (Alfantookh, 2004). Basically, SQL injection attacks indicate attempts to input known SQL characters and keywords into SQL statements with the intention to maliciously access or modify critical information in the database.

Ulrich and Lam (2008) stated that increasing number of websites create page content on the fly from a SQL database. Page title, tables, checkboxes and basically anything on the page can be generated by using information from the data repository. This idea of creating a page template and generating pages based on data in the database is a dream come true for the development community. On the other side, if the developer of the template is not careful, it might be a door to SQL injection vulnerability, hence, SQL injection can become a serious problem.

Various cheat sheets are also available on the internet for any of the attackers to try on their SQL injection skills. These cheat sheets are considered dangerous means for people to access databases illegally. The attackers are constantly developing new tactics and skills to access back-end databases. But not all injection attacks are performed to maliciously modify the database. Sometimes, a developer of an application will have to test the security of the application by injecting these malicious statements in order to see the point of SQL injection vulnerabilities that exist in the application.

#### **2.2.2.1 SQL Injection Attacks : Types and Goals**

From the papers by Halfond et al. (2006), Buehrer et al. (2005), and Spett (2005), the following table shows the summarized comparison between SQL injection attacks types and goals of each type. There are seven main types of SQLIAs: tautologies, union queries, malformed queries, piggybacked queries,

alternate encodings, inference and leveraging stored procedures. The goals are taken from Halfond et al. (2006), and there are nine different types of goals: identifying injectable parameters, extracting data, determining database schema, adding or modifying data, performing denial of services (DoS), evading detection, bypassing authentication, executing remote commands, and performing privilege escalation.

**Table 2.1:** Types and goals of SQL injection attacks

<b>Types of SQLIAs</b>	<b>Brief Explanation</b>	<b>Goals / intentions of attack</b>
Tautologies	The simplest and the best known type for SQLIA. The general goal is to inject SQL token that will cause the query's conditional statement to always evaluate to true, and if successful, this query will return the whole data of the selected table.	Bypassing authentication, identifying injectable parameters, extracting data.
Union Queries	A more sophisticated type of SQLIA that can be used to achieve additional return data. Database returns a data set that is the union of the results of the original query with the results of the injected query.	Bypassing authentication, extracting data.
Piggybacked Queries	Similar to the union query type, this type appends additional queries to the original query string. The first query is generally the original legitimate query, whereas the subsequent queries are the injected malicious queries. It is more harmful as attackers can append any SQL command.	Extracting data, adding and modifying data, performing denial of services, executing remote commands.
Malformed Queries	This type of SQLIA allows the attackers to take advantage of overly descriptive error	Identifying injectable

	messages that are generated by the database when a malformed query is rejected. The error messages will be manipulated by the attackers to inject more malicious attacks to the database.	parameters, extracting data, determining database schema.
Inference	Similar to malformed queries, this attack let attackers discover information about a database schema. But this type of attack creates queries that can cause application or database to behave differently based on the results of the query. One particular type of inference-based attack is timing attack, which lets attackers gather information from a database by observing timing delays in the database's responses.	Identifying injectable parameters, extracting data, determining database schema.
Alternate Encodings	This type of attack let attackers to modify the injected strings in a way that avoids these typical signature-based and typical-based checks. Encodings such as ASCII, hexadecimal and Unicode can be used in conjunction with other techniques to allow an attack to escape straightforward detection approaches that simply scan for certain known 'bad characters'. This attack can still be successful because it can target different layers in the application.	Evading detection
Leveraging Stored Procedures	The use of stored procedures is one of a strongly advertised solution for SQL injection. Unfortunately, it is a common misconception that the mere use of stored procedures protects an application from SQLIAs; which the safety depends on the way in which they are coded and on the use of	Performing privilege escalation, performing denial of services, executing

	adequate defensive coding practices. Therefore, parametric stored procedures could also be vulnerable to SQLIAs, just like the rest of the code in a web application.	remote commands.
--	---	------------------

### 2.2.2.2 Examples of SQL injection attacks

Each type of SQLIAs has different ways to insert their manipulated SQL statements. In this subtopic, one example in each type will be shown in order to see the way it is inserted into a query. The main query that will be used in the examples is:

```
$query = "SELECT accounts FROM users WHERE login = '" + login + "'
        AND pass = '" + password + "' AND pin = " + pin;
```

Note in the above example, it shows a simple vulnerability that can be fixed directly. This example is used for illustrative purposes because it is general enough to illustrate different types of attacks and easy to understand. For example, if a user submits login, password and pin as "joe", "joe123" and "456", the application dynamically builds and submits the query:

```
SELECT accounts FROM users WHERE login = 'joe' AND pass = 'joe123'
AND pin = 456
```

If the information matched a correct data in the database, the details of Joe's accounts will be displayed, and if not, an error message will appear to the screen.

### i) Tautologies

Usually in this attack, an attacker submits ` or 1=1 -- for the login input field (the input submitted for other fields are irrelevant). The resulting query:

```
SELECT accounts FROM users WHERE login = '' or 1=1 --AND pass = ''
AND pin = 0
```

The code injected in the conditional statement turns the WHERE clause into a tautology (Halfond et al. 2006). This will make the condition to be always true, and will return all rows in the table to the attacker. In this example, it returns all rows of accounts from the `users` table to the attacker.

### ii) Union Queries

Referring to the main example, let the attacker injects the following into the input field:

```
` UNION SELECT cardNo from CreditCards where acctNo=10001 --
```

The resulting query would be:

```
SELECT accounts FROM users WHERE login = '' UNION SELECT cardNo from
CreditCards where acctNo=10001 -- AND pass = '' AND pin = 0
```

This query will first evaluate the login part, and assuming there is no login that equals to `', the result set will return a null set whereas the second query will return a cardNo of a credit card where account number equals to '10001'. The database takes the results of queries, union them, and then returning the result set to the

application. In this example, the attacker will get the card number displayed along with the account information.

### **iii) Piggybacked Queries**

Let the attacker injected this statement into the query:

```
`; DROP table users --
```

This application will generate this query:

```
SELECT accounts FROM users WHERE login = ``; DROP table users --  
AND pass = `` AND pin =0
```

After evaluating the first query, the database will notice the query delimiter (;), and execute the injected query. The second query results in dropping the `users` table, which basically will destroy valuable information about the users in the application. Other types of injected queries may be `INSERT` queries or `UPDATE` queries or execute stored procedures.

### **iv) Malformed Queries**

This type of SQLIA allows the attackers to take advantage of overly descriptive error messages that are generated by the database when a malformed query is rejected. The error messages will be manipulated by the attackers to inject

more malicious attacks to the database. A better solution that does not compromise security would be to display a generic error message that simply states an error has occurred with a unique ID. The unique ID means nothing to the user, but it will be logged along with the actual error diagnostics on the server which the technical support team has access to (Mackay, 2005). It is not wise to have the error messages displaying the query information, the syntax error or anything related to the query, because it would be useful for the attacker to alter the injected code.

#### **v) Inference**

There are two types of the inference technique, which is blind injection and timing attacks.

In blind injection, the information must be inferred from the behavior of the page by asking the server true or false questions. If the injected statements are true, the site continues to run normally; else, the page differs significantly from the normally functioning page although there is no descriptive error message. In timing attacks, although very similar to blind injection, it uses if/else statements as the method of inference. According to Halfond et al. (2006), the if/else statements' branch predicate will correspond to an unknown about the contents of the database. Along one of the branches, the attacker uses `WAITFOR` keyword, which causes the database to delay its response time by specified time. By measuring the increase or decrease in the response time of the database, the attacker can infer which branch was taken in his injection and therefore the answer to the injection question.

Consider two possible injections into the login field. The first being `legalUser' and 1=0 --` and the second being `legalUser' and 1=1 --`.

These injections will result to the following queries:

```
SELECT accounts FROM users WHERE login = 'legalUser' and 1=0 --
AND pass = '' AND pin = 0
```

```
SELECT accounts FROM users WHERE login = 'legalUser' and 1=1 --
AND pass = '' AND pin = 0
```

If we consider two scenarios, one is a secure application, and the other one as an insecure application. In the secure application, both of the queries above will return error messages, because the application validates the input properly. In the other case, if we inject the first query into the application, the query will return error too. It is unknown whether it is because the application validates it, because the query itself is an error. But when we injected the second query into an insecure application, the query will return no error messages, which in turn tells the attacker that the insecure web application is vulnerable to the SQL injection attacks.

The following is the illustration of a way to use timing based inference attack to extract a table name from the database. In this attack, the following is injected in the login input.

```
legalUser' and ASCII (SUBSTRING((select top 1 name from
sysobjects),1,1)) > X WAITFOR 5
```

This produces the following query:

```
SELECT accounts FROM users WHERE login = 'legalUser' and ASCII
(SUBSTRING((select top 1 name from sysobjects),1,1)) > X WAITFOR 5 -
- AND pass = '' AND pin = 0
```

The SUBSTRING function is used to extract the first character of the first table's name. Using a binary search strategy, the attacker can then ask a series of questions

about this character. In this case, the attacker is asking if the ASCII value of the character is greater than or less than or equal to the value of X. If the value is greater, the attacker knows this by observing an additional 5 seconds delay in the response of the database. The attacker can use the binary search by varying the value of X to identify the value of the first character.

#### **vi) Alternate Encodings**

As stated before, alternate encodings such as ASCII, hexadecimal and Unicode are employed to encode the attack strings to evade defense. This will allow the attacks to go undetected. Contributing to the problem is that different layers in an application have different ways of handling alternate encodings. The application layer may scan for certain types of escape characters that represent alternate encodings in its language domain. Another layer, say the database layer, may use different escape characters or even completely different ways of encoding. For example, a database could use the expression `char(120)` to represent the alternately-encoded character 'x', but has no special meaning in the application language context. An effective code-based defense against alternate encodings is difficult to implement because it requires the developers to consider of all the possible encodings that could affect a given query string as it passes through different application layers (Halfond et al., 2006). Therefore, attackers have been very successful in using alternate encodings to conceal their attack strings.

Supposed the following input is injected in the login field:

```
legalUser' ; exec (0x73687574646f7776e) --
```

The resulting query would be:

```
SELECT accounts FROM users WHERE login = 'legalUser';
exec(char(0x73687574646f776e)) -- AND pass = '' AND pin = 0
```

This example makes use of the `char( )` function, and of ASCII hexadecimal encoding. The `char( )` function takes as a parameter an integer or hexadecimal encoding of a character and returns an instance of that character. In this example, the stream of numbers in query is the ASCII hexadecimal encoding of the string `SHUTDOWN`. Therefore, when the query is interpreted by the database, it would result in execution, by the database, of the `SHUTDOWN` command.

#### **vii) Leveraging stored procedure**

Nowadays, most database vendors ship databases with a standard set of stored procedures that extend the functionality of the database and allow for interaction with the operating system (Halfond et al. 2006). Therefore, once an attacker determines which backend database is in use, SQLIAs can be crafted to execute stored procedures provided by that specific database, including procedures that interact with the operating system. According to Howard and LeBlanc (2003) and Mackay (2005), developers are often surprised by the fact that their stored procedures can just be as vulnerable to attacks as their normal applications. Moreover, because stored procedures are written in special scripting languages, they can contain other types of vulnerabilities, such as buffer overflows, that allow attackers to run arbitrary code on the server or escalate their privileges.

This sample of code below shows a stored procedure for checking credentials. This sample is taken from Halfond et al. (2006).

```
1 CREATE PROCEDURE DBO.isAuthenticated
```

```

2     @username varchar2, @pass varchar2, @pin int
3 AS
4     EXEC('SELECT accounts FROM users
5         WHERE login=''' +@userName+ ''' and pass=''' +@pass+ ''' and pin 6
6         = '' +@pin);
7 GO

```

In this example, the assumption on lines 5, 6 and 7 is that they are already replaced by a call to stored procedure defined. The stored procedure returns a true/false value to indicate whether the user's login is authenticated correctly. To launch an SQLIA, the attacker would inject `; SHUTDOWN; --` into either of the username or password fields. This result in the following query:

```

SELECT accounts FROM users WHERE login = ''; SHUTDOWN; --
AND pass = '' AND pin = 0

```

At this point, this attack will work like piggybacked-queries. The first query is executed normally, and then the second malicious query is executed, which cause the database to shut down. This example shows that stored procedures can be vulnerable to the same range of attacks as traditional application code.

### 2.3 Prevention and Detection of SQL injection attacks

There is a wide range of techniques that have been proposed by researchers in order to prevent and detect SQL injection attacks. The following table will show the list of related work of prevention and detection mechanisms for SQL injection attacks in the past years. The table will show the prevention / detection type, year of research, researcher(s) and the findings of each research.

**Table 2.2** Related work in detecting and preventing SQLIAs

<b>Prevention / detection Type</b>	<b>Year</b>	<b>Researcher(s)</b>	<b>Findings</b>
Static Analysis	2007	Fu <i>et al.</i>	SAFELI: A static analysis framework that contains several components, which makes SAFELI generates test cases exploiting SQL injection vulnerability in ASP.NET web applications.
New Query Development Paradigm	2005	McClure and Krüger	SQLDOM: Developing a tool, called sqldomgen, which connects to the database schema, and used while coding, as an IDE, so that the productivity of the developers is not decreasing, ensure safe coding practices (no spelling mistakes) and so much efficient than coding by string to execute, which resulting in preventing SQL injection attacks.
Taint-based Approach	2004	Huang <i>et al.</i>	WebSSARI: Detects input-validation-related errors using information flow analysis. Static analysis is used to check taint flows against preconditions for sensitive functions. The analysis detects the points in which preconditions have not been met and can suggest filters and sanitization functions to satisfy these preconditions.
	2005	Livshits and Lam	Java Static Tainting: Use static analysis to detect vulnerabilities in

			software. The basic approach is to use information flow techniques to detect when tainted input has been used to construct SQL query. These queries will be flagged as SQL injection attacks vulnerability.
Combination of static and dynamic analysis	2005	Halfond <i>et al.</i>	AMNESIA: This mechanism use static analysis during static phase to build models of the different types of queries an application can legally at each point of access to the database. In dynamic phase, it will intercept all queries before they are sent to the database and checks each query against statically build models.
	2005	Buehrer <i>et al.</i>	SQL Guard: Checks queries at runtime to see if they conform to a model of expected queries. The model is expressed as grammar that only accepts legal queries. The model is deduced at runtime by examining the structure of the query before and after the addition of user-input.
	2006	Su and Wasserman	SQL Check: Checks queries at runtime to see if they conform to a model of expected queries. The model is expressed as grammar that only accepts legal queries. The model is specified independently by the developer.
	2006	Muthuprasanna <i>et al.</i>	Transparent Defense Mechanism: This mechanism transforms SQL into a SQL graph during the static analysis.

			During run-time (dynamic), the graph will be validated against all different user inputs.
	2006	Wei et al.	Prevent SQL injection attacks in stored procedures. The concept is the same as the above mechanism, which is transforming SQL into a SQL graph during static analysis and doing the validation during run-time.
	2007	Kosuga et al.	Sania: An approach to evaluate syntactic and semantic analysis for automated testing against SQL injection. Sania uses parse tree to validate the SQL query structure.
Input validation / Anomaly detection	2004	Alfantookh A.	AUSELSQI: Automated universal server level solution. It examines the user input before processing it by the web server and hence before executes any code on the server. This prevents suspicious data from being delivered to the server. Also, it guarantees the prevention of the SQLIAs even if the code was changed seconds before the request is submitted.
	2006	Lin and Chen	An automated revised tool that finds entry pointer in a web that contains HTML form. It gets the variable names, cookies or parameter and the type of the script language. Input validation is done depending on the script language.
	2007	Bertino, Kamra and Early	Use encoding schemas for SQL queries to extract useful information.

			Fingerprinting on database application is done based on SQL queries, in order to detect anomalous behavior.
--	--	--	---

Although there are plenty of detection and prevention mechanisms that have been developed, Halfond and Orso (2006) have selected a number of well known mechanisms, and conducted an analytical evaluation to compare these mechanisms. From this evaluation, these mechanisms were divided into two categories: detection-focused technique (DF) and prevention-focused technique (PF). Two mechanisms have been chosen for the purpose of the study, AMNESIA, which stands for Analyzing and Monitoring for Neutralizing SQL Injection Attacks (Halfond and Orso 2005), and SQL Guard (Buehrer *et al.* 2005). The choices were made according to the ability of each mechanism in detecting and preventing the main variants of SQL injection attacks.

According to the evaluation (Halfond and Orso, 2006), these mechanisms work well against SQL injection attacks. The following table will show the summarized comparison between these two mechanisms. The comparison will cover four aspects; 1) SQLIA types detection, 2) the degree of automation of preventing aspect of the approach, 3) the degree of automation of detecting aspect of the approach and lastly the limitations (Halfond and Orso, 2006) of each mechanism.

**Table 2.3** Selected mechanisms for the purpose of the study

Technique name	SQLIA Types that are detected	Automation		Limitation
		Prevention	Detection	
AMNESIA Halfond and Orso (2005)	All types except stored procedures	Automated	Automated	Its success is dependent on the accuracy of its static analysis for building query

				models.
<b>SQL Guard</b> Buehrer <i>et al.</i> (2005)	All types except stored procedures	Automated	Semi - automated	Requires the developer to either rewrite code to use a special intermediate library or manually insert special markers into the code where user input is added to dynamically generated query.

From the comparison made in the table above, we can see that each mechanism has its own drawbacks even for the automated mechanisms in detecting and preventing SQL injection attacks. A closer look on the techniques used in both mechanisms is showed in the following table.

**Table 2.4** Techniques used in selected mechanisms

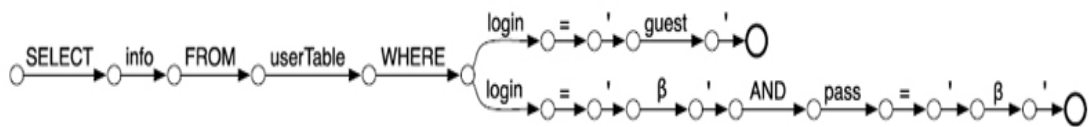
<b>AMNESIA</b> (Halfond and Orso, 2005)	
During Static Analysis (3 steps)	
Identify hotspots	Scan the application code to identify hotspots – points in the application code that issue SQL queries to the underlying database.
Build SQL-query models	For each hotspot, build a model that represents all the possible SQL queries that may be generated at that hotspot.
Instrument application	At each hotspot in the application, add calls to the runtime monitor.

During Runtime (the 4 <sup>th</sup> step)	
Runtime monitoring	At runtime, check the dynamically-generated queries against the SQL-query model and reject and report queries that violate the model.
<b>SQL Guard</b> (Buehrer et al., 2005)	
<p>SQL Parse Tree Validation:</p> <ul style="list-style-type: none"> <li>i) Create parse tree with unpopulated user tokens for user inputs.</li> <li>ii) During runtime, create another parse tree with user inputs and compare for matching structure.</li> </ul>	

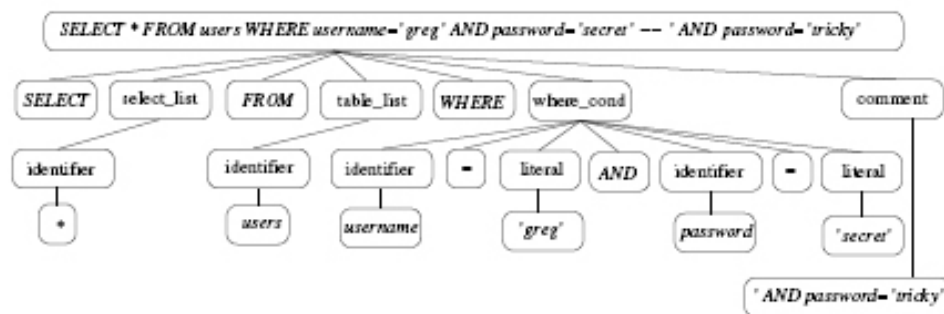
From the table above, the main technique for detecting an SQL injection attack is by parsing them into a parse tree or a SQL query model. In SQL Guard (Buehrer et al., 2005), the SQL parse tree that is used for evaluation is containing an extension node, called ‘comment token’, which is used to store excessive user input, which is probably an attack intended by a user. If the original parse tree did not contain a comment token, then the resulting parse tree with attack will no longer be a match. If the original query somehow has a comment, the parse with attack would still have failed because the value of the two tokens would not be equal.

The SQL-query model built during the static analysis in AMNESIA is a non-deterministic finite-state automaton in which the transition labels consist of SQL tokens (SQL keywords and operators), delimiters, and placeholders for string values. The following process is used to build a SQL-query model for a given hotspot (Halfond and Orso, 2005).

- i) Perform depth traversal of the NDFA for the hotspot.
- ii) Group characters as SQL keywords, operators, or literal values and create a transition in the SQL-query model that is annotated with their literal value.
- iii) Variable strings (i.e. user inputs) will be labeled as  $\beta$ .



**Figure 2.1** Example of SQL-query model (Halfond and Orso, 2005).



**Figure 2.2** Example of SQL Parse Tree with comment token (Buehrer et al., 2005).

These mechanisms have been developed and coded, to produce their algorithms, and is put to test against the SQL injection attacks to confirm the analytical evaluation by Halfond and Orso (2006). The test results will be evaluated using the following evaluation techniques.

## 2.4 Evaluation using Precision and Recall

To illustrate precision and recall measurement applied in this study, consider this example:

In a collection of SQL statements  $S$ , there are two sets; one consisting the number of SQL injection attacks that has been launched,  $|L|$ , and the other set consisting the number of SQL statements detected as ‘attacks’,  $|A|$ . Let  $|L_A|$  be the number of SQL statements in the intersection of the sets  $L$  and  $A$ . Figure 2.1 illustrates these sets. The precision and recall measures are defined as follows:

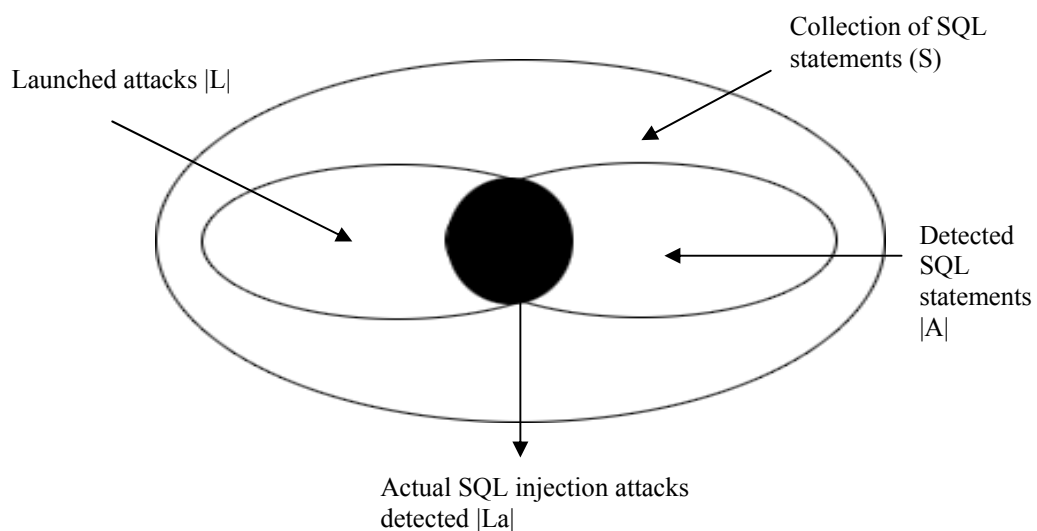
i) Precision is the fraction of the detected SQL statements (set  $A$ ) which are actually SQL injection attacks launched i.e.

$$\text{Precision} = \frac{|L_A|}{|A|}$$

ii) Recall is the fraction of the launched attacks (set  $L$ ) which has been detected i.e.

$$\text{Recall} = \frac{|L_A|}{|L|}$$

Precision and recall, as defined above, assume that all the detected SQL statements in set  $A$  have been examined.



**Figure 2.3** Precision and Recall for a collection of SQL statements

$$\text{Precision} = \frac{\text{Number of actual SQL injection attacks detected}}{\text{Number of SQL statements detected}}$$

$$\text{Recall} = \frac{\text{Number of actual SQL injection attacks detected}}{\text{Number of SQL injection attacks launched}}$$

## 2.5 Effectiveness and Efficiency

The evaluation of these mechanisms using the effectiveness and efficiency measure are generally the same. Valuer et al. (2005) and Livshits and Lam (2005) stated that effectiveness and efficiency can be measured by evaluating the ability to detect novel attacks of SQLIAs, false positives rates, and the overhead of the mechanisms. Su and Wasserman (2005) stated that effectiveness and efficiency can be measured by evaluating the examples of the attacks and the empirical results; which also consist of the combination factors of detection of attacks, false positives and overhead. According to Halfond and Orso (2005), in their experiments for AMNESIA, the effectiveness is measured by the percentage of attacks detected and reported by AMNESIA. For efficiency, they have measured the response time of the application with and without the instrument (i.e. the call for runtime method). For SQL Guard (Buehrer et al., 2005), they only measure the execution time overhead, and compared it between the original application (without SQL Guard) and with guarded application (with SQL Guard).

## 2.6 Discussion and Summary

In the analytical study, Halfond and Orso (2006) found that most of the detection-focused techniques perform fairly uniformly against the variants of SQL injection attacks. From the chosen mechanisms for the purpose of this study (refer Table 2.3), AMNESIA and SQL Guard use the same method to detect SQL injection attacks, which is, by using a parse tree validation. The difference between these mechanisms is that SQL Guard has an added node in the parse tree, which is called the ‘comment token’, and this node will store the attack part of the modified query by the attacker.

In this chapter, several important subtopics regarding the area of interest have been covered. There are many researches that have been done to handle the problems of SQL injection attacks. Although there are plenty mechanisms have been developed, the best approach to prevent and detect any type of SQL injection attack has not yet been decided. This is due to the fact that no practical evaluation has been done to compare how these mechanisms works against real time attacks. Halfond and Orso (2006) have conducted an analytical comparison between several mechanisms. From the results of the comparison, two mechanisms are selected and developed to see the effectiveness in detecting SQL injection attacks.

## **CHAPTER 3**

### **METHODOLOGY**

#### **3.1 Introduction**

In this chapter, the methodology of this study will be discussed. The methodology that will be carried out will aid in finding which detection mechanism is the best in order to detect SQLIAs. This methodology is important in order to achieve the objectives that have been defined in the first chapter.

#### **3.2 Operational Framework**

This study will be conducted according to the workflow process as illustrated in Figure 3.1. The operational framework is divided into four major phases; with each phase containing processes and expected results to accomplish the phase. Each of these phases and processes will be explained thoroughly in the subsequent subsections.

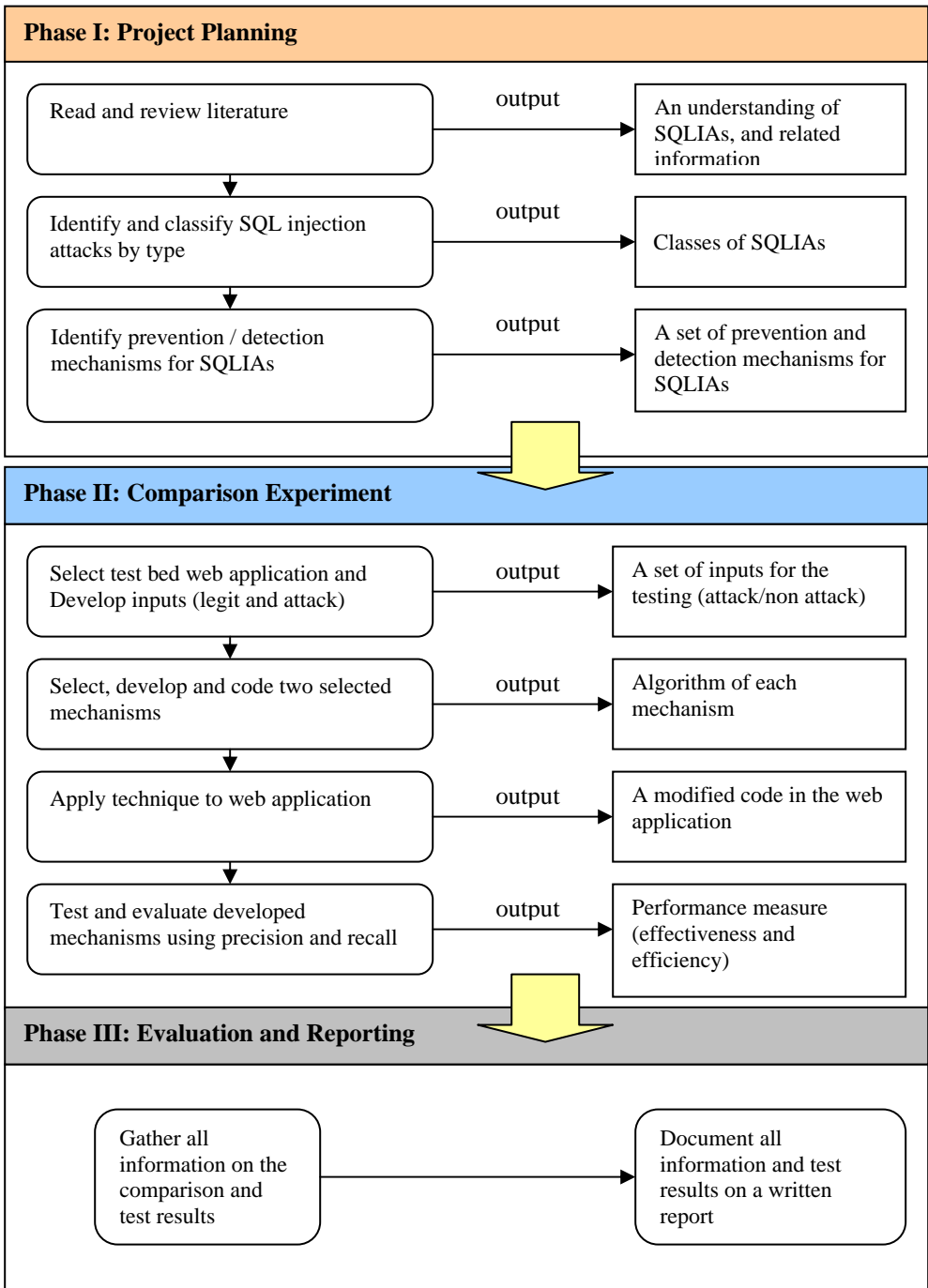


Figure 3.1 Operational Framework

### **3.2.1 Phase I: Project Planning**

This phase contains three processes. The first process is to read and review literature. A number of related literatures have been collected to be reviewed. These literatures covered the basics of SQL injection attacks, and also the prevention and detection mechanisms that have been developed in order to handle SQL injection attacks. After reviewing the literature, an understanding of SQL injection attacks has been established, and the clearer view on how the prevention and detection mechanisms should work has been portrayed. The second process is to identify and classify the SQL injection attacks by type. From the literatures, several examples of the attacks have been showed. After identifying which SQL injection attacks belongs to which class, a few SQL injection attack statements were gathered, and classified into their respective types. The output of this process is the classes of SQL injection attacks. The last process of this phase is to identify the prevention and detection mechanisms for SQL injection attacks. There are plenty of known detection and prevention mechanisms for SQL injection attacks, such as AMNESIA (Halfond and Orso (2005)), SQL Guard (Buehrer et al. (2005)) and SQL Check (Su and Wasserman (2006)). After identifying these mechanisms from the literatures, a set of prevention and detection mechanism for SQL injection attacks have been summarized (please refer to Chapter 2).

### **3.2.2 Phase II: Comparison Experiment**

This phase is the most crucial phase of this study. There will be four processes involved in this phase.

### 3.2.2.1 Web applications as test bed and a set of inputs

The first process is to select test bed web applications and develop a set of inputs. For this study, two web applications has been selected; one web application is from <http://www.sourceforge.net>, an open source website for ready made web applications, and the other web application is built from scratch. The web applications from sourceforge.net, myMarket will be incorporated in the local web server together with the simple built up web application, Mekar Services. The difference between these web applications is that there are possibilities that myMarket have unknown vulnerabilities since this is a ready made applications provided by sourceforge.net, while Mekar is built with known vulnerabilities and this is to ensure that the technique applied to these applications really detect the SQLIAs during the testing phase.

The second part is to develop user inputs and queries, which will be obtained by the application and will be formed into a query before submitting it to the database. These inputs are divided into two smaller sets, called ATTACK and NON ATTACK inputs. These are the inputs that can be used to manipulate or to attack the database. The inputs in the NON ATTACK class are simply various username and password, but not every input will form a correct query. The NON ATTACK inputs are mostly used to detect false positives during testing.

**Table 3.1** Several examples on ATTACK and NON ATTACK inputs for testing

No.	ATTACK inputs
1	1 OR 1=1
2	1' OR '1'='1
3	1'1
4	1 EXEC SP_ (or EXEC XP_)
5	1 AND 1=1

6	1' AND 1=(SELECT COUNT(*) FROM tablenames); --
7	admin' and 1=1 --
8	1\'1
9	' OR uname IS NOT NULL OR uname = '
10	') or ('1'='1
<b>NON ATTACK inputs</b>	
1	user1 (username); user1 (password)
2	brian (username); typ-o (password)
3	user 1 (username); user 1 (password)
4	admin (username); root (password)

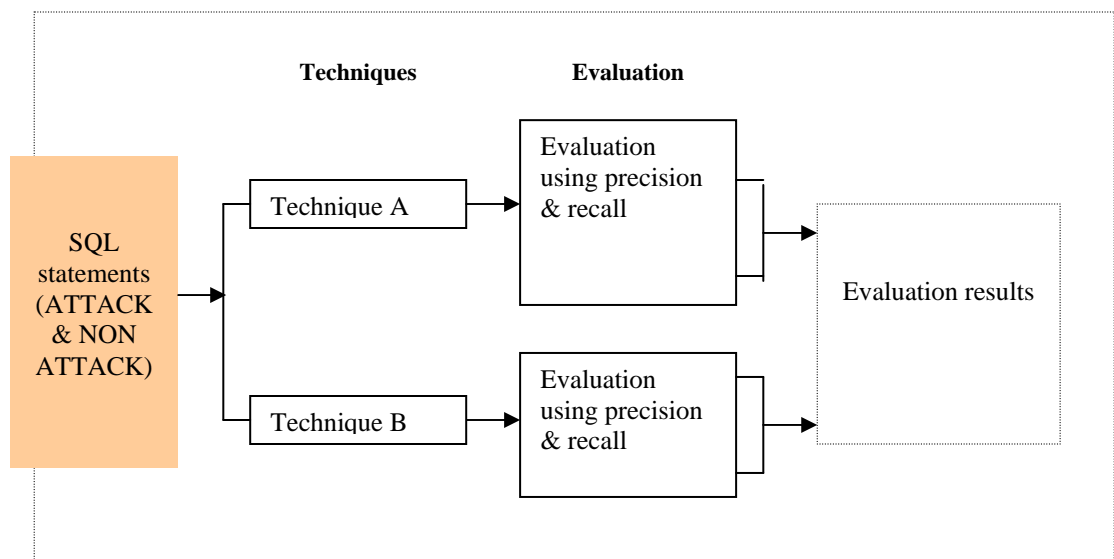
In this case, false positives are the SQL statements that are detected as SQL injection attacks but they are not. False negatives, on the other hand, are the ones that are SQL injection attacks but are not detected. These two situations are also the issues in SQL injection attacks.

### 3.2.2.2 Mechanisms Selection and Algorithm Implementation

After developing the queries for the web application, the next process is to select two mechanisms for the algorithm implementation. Since there is no availability of the original algorithms, the algorithms used in this study are re-implemented using the technique details and explanation as guidelines. Besides that, these algorithms will be coded in PHP language, and to be adopted and tested in PHP web applications. For this study, the AMNESIA and SQL Guard techniques are selected. As the algorithms are not the original algorithms, these techniques will be called Technique A and Technique B,

as these techniques are only adopting the techniques introduced by AMNESIA and SQL Guard, respectively. The selection will be carried on with developing the algorithm and then coding for each technique. After finish implementing the algorithm and coding for each technique, the next step is to test the techniques and to evaluate them using precision-recall measure. This testing process will provide a performance measure from each mechanism, from the view of effectiveness and efficiency.

The two selected mechanisms are: AMNESIA (Halfond and Orso (2005)) and SQL Guard (Buehrer et al. (2005)). Technique A is adopted from AMNESIA and Technique B is adopted from SQL Guard. The testing process can be depicted in the following figure:



**Figure 3.2** Testing and evaluating developed mechanisms

According to Buehrer et al. (2005), the work of Halfond and Orso (2005) is the most similar to theirs. This is because the concept of parse trees and the SQL query model comes from the same grammar; which in this case is the SELECT statement grammar. For both techniques, this will be the syntax for the SELECT statement. This syntax is written in Backus Naur Form (BNF).

### Technique A:

```

<select_query> ::= <select_query> WHERE <where_cond> |
                  SELECT <columns> FROM <table>

<where_cond>    ::= <where_cond><operator><where_cond> |
                  <identifier> = '<literal>'

<columns>      ::= <letters> | "*"

<table>        ::= <letters>

<identifier>   ::= <letters>

<literal>      ::= <letters> | <digits>

<operator>     ::= "AND" | "OR"

<letters>      ::= "a"|"b"|"c"|"d"|"e"|"f"|"g"|"h"|"i"|"j"
                  |"k"|"l"|"m"|"n"|"o"|"p"|"q"|"r"|"s"|"t"
                  |"u"|"v"|"w"|"x"|"y"|"z"

<digits>       ::= "0"|"1"|"2"|"3"|"4"|"5"|"6"|"7"|"8"|"9"

```

For Technique B, as the writers introduced the comment token, the grammar will be modified as below, where a comment token will be inserted at the end of a query:

### Technique B:

```

<select_query> ::= <select_query> WHERE <where_cond>
                  <comment> |
                  SELECT <columns> FROM <table>

<where_cond>    ::= <where_cond><operator><where_cond> |
                  <identifier> = '<literal>'

<columns>       ::= <letters> | "*"

<table>         ::= <letters>

<identifier>    ::= <letters>

<literal>       ::= <letters> | <digits>

<comment>       ::= <letters> | <digits>

<operator>      ::= "AND" | "OR"

<letters>       ::= "a"|"b"|"c"|"d"|"e"|"f"|"g"|"h"|"i"|"j"
                  |"k"|"l"|"m"|"n"|"o"|"p"|"q"|"r"|"s"|"t"
                  |"u"|"v"|"w"|"x"|"y"|"z"

<digits>        ::= "0"|"1"|"2"|"3"|"4"|"5"|"6"|"7"|"8"|"9"

```

These grammars for both techniques will help in the coding to check the user input for any attempted injection attack during runtime. During runtime, the dynamic query is exploded and will be assessed whether it violates the SELECT query model or the SELECT parse tree.

For this study, the following will be the steps involved in order to test the techniques on our web application subjects.

For technique A, adopted from AMNESIA:

- i) Identify hotspots – this step is done manually (in static analysis).
- ii) Build SQL-query model – as this study will focus on SELECT statements only, the SQL-query model that will be used is the same
- iii) Instrument Application – this step is also done manually (after identifying the hotspots)
- iv) Runtime Monitoring – in this step, the dynamic query will be assessed during runtime to see whether it violates the SQL query model.

For technique B, adopted from SQL Guard:

- i) Create parse tree with unpopulated user tokens for user inputs. – in this step, the same model for Technique A is used because this study is focusing on SELECT statements only.
- ii) During runtime, create another parse tree with user inputs and compare for matching structure. – This step is also similar to the fourth step of Technique A. The dynamic query will be assessed during runtime, except in this technique, a comment token is included, and an attempt to inject an attack can be detected if there are malicious inputs contained in the comment token.

The algorithm that is used during runtime is explained below:

#### Technique A

```

1 Get query
2 //for easier assessment because in grammar all lowercase
3 Change query into all lowercase letters
4 Explode query into tokens
5
6 Foreach token {
7     If token value = model value {
8         Display token
9         Set count to 1
10    } Else {
11        display error message
12    }
13 }
14 If exist case {
15     "digit=digit" or
16     "char=char"
17 }
18 Display message:"injection attempt detected at [token_value]"
19
20 Count total of parsed inputs (tot_parsed)
21 Count total of correct inputs (tot_correct)
22
23 //comparing total of parsed and correct inputs
24 if ((tot_parsed) == (tot_correct)) {
25     Display message: "clean user input"
26     set flag = 1
27 } else { set flag = 0 }
28 return

```

From the algorithm above, lines 1 to 3 obtain the dynamic query and then will split the query into tokens. For each of these tokens, assessment is done to see whether it violates the model or the tree assigned value. If it is correct, a counter is increased by 1, to keep track of correct user inputs. In lines 14 to 18, a case of tautologies is detected

(if any). These inputs are usually used in blind injections. If there are any form of “1=1” or “a=a” detected in the dynamic query, the program will display an error message, that an injection attempt is detected at the corresponding token number.

For the instructions in lines 20 to 28, total of tokens, or parsed inputs will be compared to total of correct inputs. This is useful to detect if the user supplied inputs are clean inputs. Clean inputs consists any NON ATTACK inputs, regardless if they will form a correct query, which invokes outputs from the database or not. If the inputs are clean, the flag will be set to 1, and this value is returned to the original page, so that the query can now be executed.

Technique B uses the same algorithm as above, but a little bit of modification is added. It is for the comment token inclusion, which takes the extra inputs at the end of the query that are not assessed because the front inputs are already assessed against the parse tree. The instructions for this portion are added after line 18.

#### Technique B

```

17     }
18     Display message:"injection attempt detected at [token_value]"
19
20     //create a set for comment
21         create empty array (comment array)
22         if token is not empty {
23             if token has not been evaluated
24                 add token value into comment array
25
26         display comment array
27     } else { display message "comment token empty" }
28
29     foreach comment token value{
30         If exist case {

```

```
31         "digit=digit" or
32         "char=char" or
33         "and" or
34         "or"
35     }
36     Display message:"injection attempt detected at [token_value]"
37
38     Count total of parsed inputs (tot_parsed)
39     Count total of correct inputs (tot_correct)
40
41     //comparing total of parsed and correct inputs
42     if ((tot_parsed) == (tot_correct)) {
43         Display message: "clean user input"
44         set flag = 1
45     } else { set flag = 0 }
46     return
```

From this algorithm, lines 21 to 24 will create an empty array for the extra inputs to be stored. For the inputs stored in the comment token, if there are inputs which involves a pattern of 'digit=digit' or 'char=char' or 'AND' and 'OR' operator keyword, then the query will be considered as an injection attempt. This is because, assume the dynamic query has two literal values, then the actual 'AND' or 'OR' operator keyword has already been assessed during parse tree evaluation. Any extra operator keyword will be considered as malicious input.

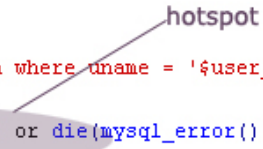
### 3.2.2.3 Technique embedment into PHP web application

This subtopic will explain on how we embed the techniques into the web application. The algorithms from the previous subtopic are coded in an external file, which will be called before a query is sent to the database. In order to place a call to an external file, which is the coding of the techniques, a hotspot needs to be recognized in the web application code page. To find a hotspot, we need to find the `mysql_query()` function. This function sends the query to the currently active database on the server.

```
<?php require_once('Connections/conn.php'); ?>
<? mysql_select_db($database_conn); ?>
<? if(isset($_POST['submit'])) {

    $user_name = $_POST['username'];
    $user_pwd = $_POST['password'];

    $check_login = "SELECT * from tbl_auth where uname = '$user_name' AND pwd = '$user_pwd'";
    $get_result = mysql_query($check_login) or die(mysql_error());
```



**Figure 3.3** Identifying a hotspot

The scope of this study has been defined that we focus on the SELECT statement, and we have defined the SELECT syntax grammar in the previous subtopic. So the SQL query model and parse tree that will be used in this study is by far a static model, a static tree and every dynamic query will be assessed according to this model/tree.

After identifying the hotspot, the next step is instrument application; which is to put a call to the runtime monitor before the dynamic query is sent to the database. This call will be inserted before `mysql_query()` function, so that if the monitor accepts the

dynamic query, which will set the flag to 1, then the query is executed normally. The figure below will show the slight modification in the source code after insertion of the call.

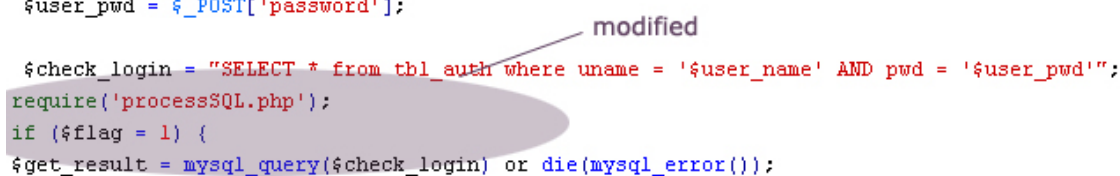
```

<?php require_once('Connections/conn.php'); ?>
<? mysql_select_db($database_conn); ?>
<? if(isset($_POST['submit'])) {

    $user_name = $_POST['username'];
    $user_pwd = $_POST['password'];

    $check_login = "SELECT * from tbl_auth where uname = '$user_name' AND pwd = '$user_pwd'";
    require('processSQL.php');
    if ($flag = 1) {
    $get_result = mysql_query($check_login) or die(mysql_error());

```



**Figure 3.4** Hotspot after instrumentation

The call to Technique A will be in a file named ‘processSQL.php’, while the call to Technique B will be in a file named ‘processSQLc.php’.

### 3.2.3 Phase III: Evaluation and Reporting

In this study, all comparison will be measured using the precision-recall measure. This precision-recall measure is done for determining the correctness of both techniques, and to see if it does what they are intended to do. The procedure for calculating the precision and recall is to have a set of detected SQL statements, and a set of SQL injection attacks launched. The intersection of both sets shows the actual SQL injection attacks that are detected. The precision value is calculated by using equation (1)(a); while the recall value is calculated by using equation (1)(b).

$$\text{Precision} = \frac{\text{Number of actual SQL injection attacks detected}}{\text{Number of SQL statements detected}} \quad (1)(a)$$

$$\text{Recall} = \frac{\text{Number of actual SQL injection attacks detected}}{\text{Number of SQL injection attacks launched}} \quad (1)(b)$$

The interpolated precision-recall curve is also plotted to show the detection effectiveness. The results will be compared between:

- i) Detection of SQLIAs before embedding the techniques to the web application
- ii) Detection of SQLIAs after embedding the techniques to the web application

The information and test results from the comparison are gathered. Report writing will be the last part of this study which will focus on documenting all information and results into the report.

### **3.3 Hardware and Software Requirements**

These are the hardware and software employed in order to complete this study:

#### Hardware

- i) Personal Computer

- a. Intel Pentium Dual Core E2180 @ 2.00 GHz processor
- b. 1GB of RAM
- c. 80GB of hard disk
- d. Monitor
- e. Mouse
- f. Keyboard

#### Software

- i) Microsoft Windows XP Professional with Service Pack 2
- ii) Microsoft Office 2003
- iii) Adobe Acrobat Professional 8.0
- iv) Macromedia Dreamweaver
- v) XAMPP 1.6.3 web server

### **3.4 Summary**

This chapter discussed the methodology which will be used in implementing this project. It has three major phases, starting from the first phase which is project planning, which consists of three processes: read and review literature, identify and classify SQL injection attacks, and identify prevention and detection mechanisms for SQL injection attacks. The second phase is the key phase, named comparison experiment, which consists of four processes: select web applications for test bed, develop queries, select, develop and code two selected mechanisms, AMNESIA and SQL Guard. The last phase is called evaluating and reporting, which consists two processes: gather information and test results, and documenting the information and test

results on a written report. Each phase plays an important role in accomplishing this project.

## **CHAPTER 4**

### **EXPERIMENTAL RESULTS AND DISCUSSION**

#### **4.1 Introduction**

Chapter 3 discussed the steps taken in conducting this study. Thus, this chapter discusses the results of the output shown in Figure 3.2 (refer to Chapter 3).

#### **4.2 Phase I: Project Planning**

In this phase, the results of each step have been documented in previous chapters. Chapter 1 and 2 discussed on the basic knowledge and issues involved in SQL injection attacks. In Chapter 2 (refer to Table 2.1), the types of the SQL injection attacks were documented, along with their brief description and goals of each attack. The prevention and detection mechanisms have also been documented in Chapter 2

(refer to Table 2.2). From this table, two mechanisms were selected (refer to Table 2.3) for the purpose of practical evaluation in this study.

### **4.3 Phase II: Experiment Results and Discussion**

In this phase, a number of steps need to be accomplished before the experiment could be started. As stated in the previous chapter, two web applications have been selected as the test beds for the experiment. A set of ATTACK and NON ATTACK inputs have been developed for the purpose of the experiment. The ATTACK class contains the malicious inputs while the NON ATTACK class contains various legitimate inputs and queries for the database, which are harmless, yet not necessarily correct.

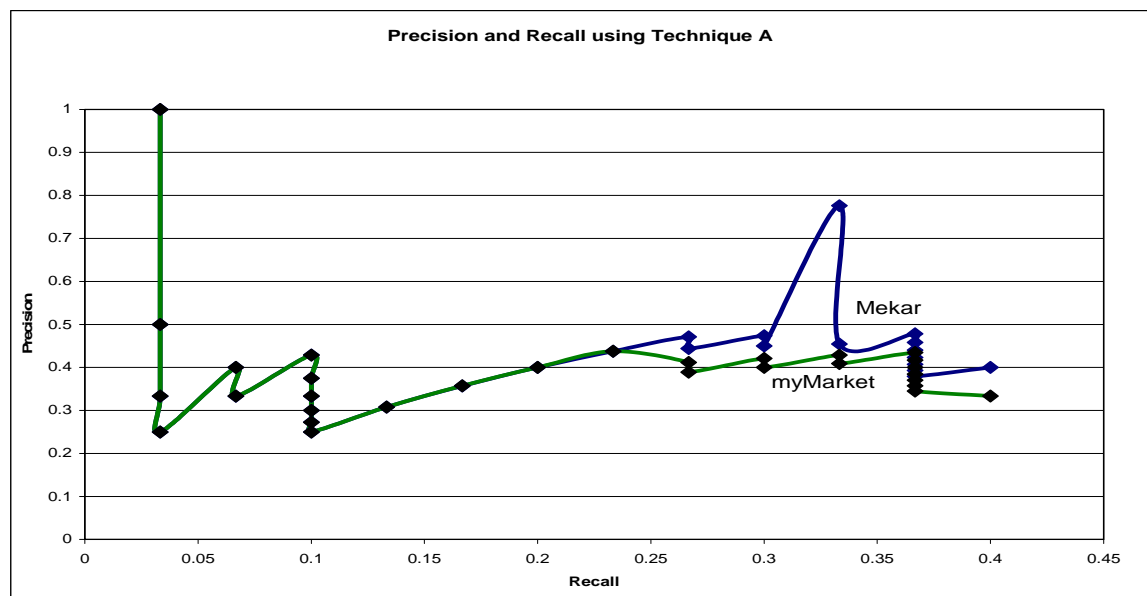
For this study, only SELECT statements are identified as hotspots. There are 30 queries (ATTACK only) and several number of NON ATTACK queries (which are any legitimate queries) that will be used in the experiment. These 30 inputs will be tested on each of the two web applications (please refer Appendix B for inputs list). These queries will be tested at any textbox in the web application; whereby the SQL statement involved is the SELECT statement.

It is important to note that `mysql_query()` function does not support multiple queries. In SQL injection classes, as described in Chapter 2, there are two types that will use multiple queries in attacking applications, which are piggybacked queries and also alternate encodings. As both techniques do not handle stored procedures, that leaves us to four other classes of injection attacks, which are, tautologies, union queries,

malformed queries and blind injections. This is why the testing inputs vary among these four classes only.

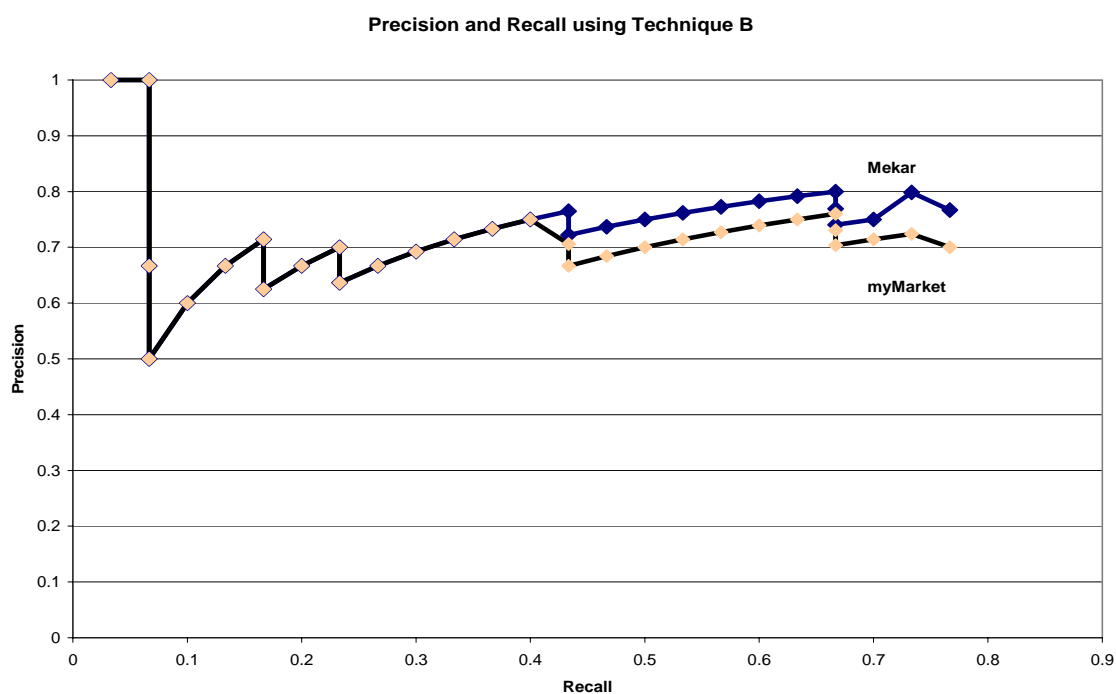
For the first phase, both web applications will be experimented against SQLIAs without the instrument application. In this phase, the total queries are 30++ queries, 30 are malicious queries, and the rest are legitimate queries. This experiment shows zero detection on the malicious queries. All unsuccessful queries are only detected as error in the web application. The attacks are not detected as SQLIAs because there is no validation upon the SQL injection attacks yet.

For the second phase, both web applications are tested against the same 30++ queries, with 30 malicious queries that could perform as SQLIAs, and the rest as legitimate queries using the first technique, adopted from AMNESIA technique. All hotspots (the SELECT statement only) are identified, page by page, and the call to runtime monitoring has been appended. The following figure shows the result of precision and recall for both web applications, using Technique A.



**Figure 4.1** Precision and Recall using Technique A

The third phase is the experiment on the second technique, which is adopted from SQL Guard technique. In this technique, the runtime monitor call is also appended at the same hotspots as the first technique is done. The following figure shows the result of precision and recall for both web applications, using Technique B.

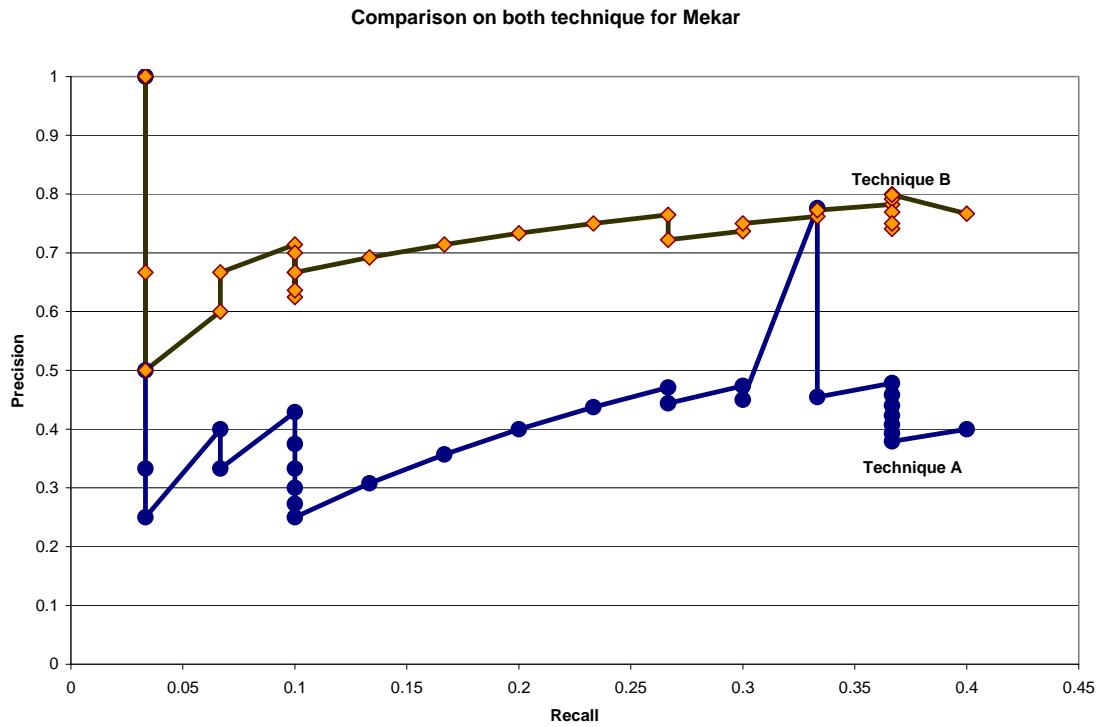


**Figure 4.2** Precision and Recall using Technique B

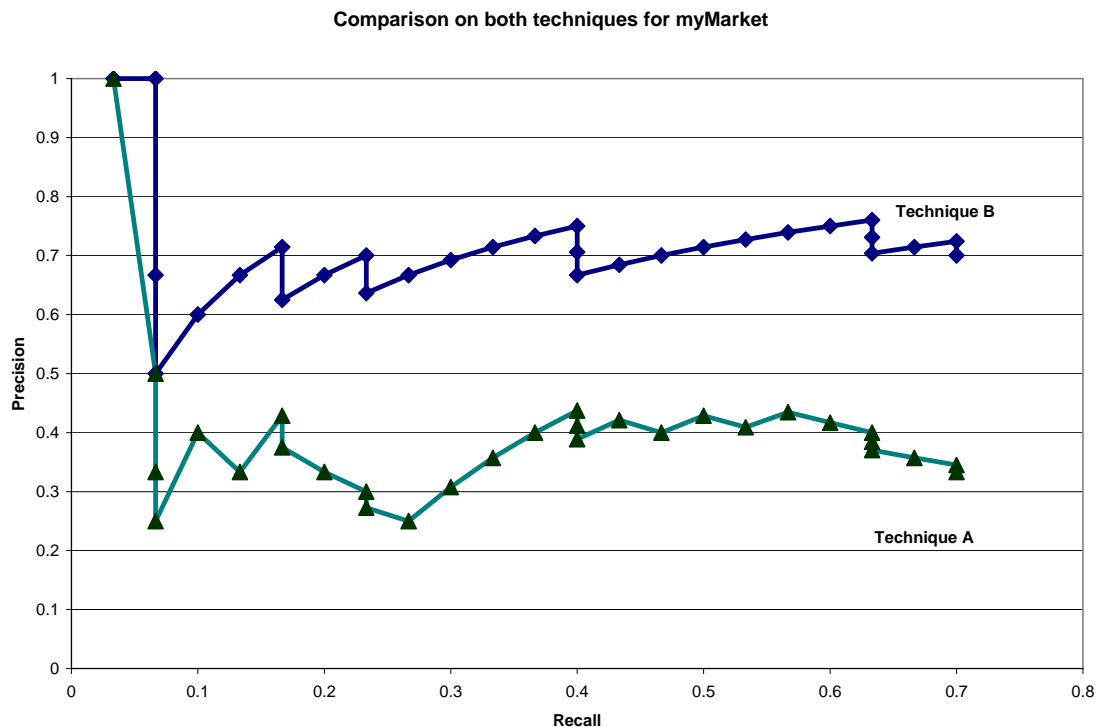
From both Figure 4.1 and 4.2, the difference of accuracy for both techniques in detecting SQLIA is almost insignificant. Application mekar somehow carries higher precision value than myMarket. This is due to the higher vulnerability that mekar has, for example, lack of password hashing. This vulnerability makes an attack easier to detect using the main step in both techniques, which is the comparison between static and dynamic SQL Query Model in Technique A, or static and dynamic parse tree in Technique B. Figure 4.2 shows that Technique B detects more attacks than Technique A. This is because technique B applied a comment token inclusion concept, which detects subtle blind injection attacks that is appended at the end of a dynamic query.

#### 4.4 Discussion

Figure 4.3 and Figure 4.4 shows the comparison on both techniques for both web applications.



**Figure 4.3** Comparison on both techniques for Mekar application



**Figure 4.4** Comparison on both techniques for myMarket application

From the direct comparisons from both of the figures above, Technique B is the better technique ( $\pm 30\%$  better) in detecting SQLIAs. This is due to the fact that Technique B applied the comment token inclusion concept, which helps in catching more SQLIAs. From the experiment, most of the attacks from the tautologies and blind injection types have been successfully detected. Some malformed and UNION queries are not successfully detected as SQLIAs, but both of the techniques report these types as errors. Hence, these queries will not be executed as well. Two other types, piggybacked and alternate encodings are not tested in the experiments, because these types need to have double query in a single SQL statement, and as stated before in this chapter, `mysql_query()` function does not support double queries in one execution.

Both techniques are considered to detect SQLIAs accurately, although not 100% of the injected attacks were detected. Only false negatives were seen in the experiments,

where some of the attacks were not detected as SQLIAs but only reported as application error. The results from both techniques are almost similar, as seen from the pattern of the points in Figure 4.3 and Figure 4.4. This is because they use the same main concept, which is to do a comparison between static and dynamic query parse tree/ SQL model, except that Technique B takes in the possibility of subtle attacks, and applied an addition token to detect these attacks. This helps Technique B to detect more SQLIAs injected in both applications.

In order to measure the efficiency, the overhead between the response time for the first phase testing, compared to the second and third phase testing is negligible. Experimenting attacks through these web applications with instrumentation (phase 2 and phase 3) have almost the same amount of response time with phase 1, which is considered as efficient, for both techniques.

## **4.5 Summary**

This chapter discusses the experimental results for both techniques of SQL injection attacks detection. The results are measured in effectiveness or correctness, by using precision and recall measure and efficiency, according to execution response time. Both of the techniques have a difference of ~30% effectiveness value, but both techniques did detect the actual attacks. The experiment could show better results with more queries included for the penetration testing. Besides, the algorithm should be revised to produce the desired output in the testing results. From the experiments that have been conducted, Technique B, which is derived from SQL Guard technique, is more effective in detecting SQL injection attacks.

## **CHAPTER 5**

### **CONCLUSION**

#### **5.1 Introduction**

This chapter concludes the work that has been done for this study in order to meet the project goal. The aim of this study is to determine the effective and efficient detection approach that is able to detect the SQL injection attacks. There are two techniques that have been selected, and from each of these techniques, the algorithms were implemented, and the codes were developed.

#### **5.2 Findings**

The findings for this project are the outputs of Phase I to Phase III from the operational framework (refer Figure 3.2 in Chapter 3). Experimental results are documented in Chapter 4, where it shows the comparison between the ability of two

techniques in detecting the SQL injection attacks in two different web applications. The results achieved are not as desired, but that does not mean that the techniques are not effective or efficient in detecting the SQLIAs. The implementation on the algorithm should be revised and re-coded to achieve better results in order to improve the effectiveness of both techniques. As in this comparison, Technique B shows a better effectiveness than Technique A. However, the result differences between these two algorithms are not huge; so does the efficiency. Both techniques are efficient enough and did not take so much time to execute.

There are limitations of this project. Due to the implementation of the algorithms and program codes that are translated by analyzing the steps done in both techniques, the program codes used in this study may not be accurate. Hence, the results did not achieve the best value for precision and recall. Besides, hotspot identification, instrumentation application and testing are done manually. This is a tedious work and consumed a lot of development time.

### **5.3 Future works**

In order to achieve more accurate result during the experiment, more injection inputs need to be gathered for the penetration testing. However, the testing phase will be a tedious phase if there is no automatic loader for these inputs. For future works, a complete experiment can be enhanced by creating the automatic loader for the input penetration testing, and a fully automated tool where it helps to scan hotspots and apply the instrumentation automatically. In this study, there were only two subjects that were being tested, and these two were not complicated web applications. Both are fairly simple, and there were not many hotspots. In order to see a bigger picture on the way

these techniques work, more subjects need to be added, with different levels of complexity of the web applications, and also with varying kinds of SQLIAs. Due to the php function that does not support double query in one statement, the detection of piggybacked queries input and alternate encoding queries input remained unknown. This types of attacks can be tested on web applications whereby the database supports multiple queries in one SQL function..

#### **5.4 Summary**

The aim of this project is to determine the effective and efficient detection technique that is able to detect the SQL injection attacks. A comparison experiment has been held, to evaluate two techniques, both tested on two web applications. These techniques consist of Technique A, which is adopted from the concept of AMNESIA, introduced by Halfond and Orso (2005) and Technique B, which is adopted from the concept of SQL Guard, introduced by Buehrer et al. (2005). From the experiments, the results showed that Technique B is more effective in detecting SQLIAs than Technique A. Technique B has a higher precision value, which shows higher correctness value, and detects more attacks than Technique A. Overall, the objectives and aim for this project are achieved, because Technique B, which incorporates static and dynamic parse tree comparison, added with a comment token inclusion proved that this technique can detect SQLIAs accurately.

## REFERENCES

- Alfantooh A. A., (2004). An Automated Universal Server Level Solution for Sql Injection Security Flaw 2004. *IEEE*, pg 131- 135
- Anley C., (2002). Advanced SQL Injection In SQL Server Applications. White paper, Next Generation Security Software Ltd., 2002.
- Bertino E., Kamra A., and Early P. J., (2007). Profiling Database Applications to Detects SQL Injection Attacks. *ISBN 1-4244-1338-6/07.IEEE*
- Buehrer G. T., Weide B. W., and Sivilotti P. A. G., (2005). Using Parse Tree Validation to SQL Injection Attacks. In *International Workshop on Software Engineering and Middleware (SEM), 2005*.
- Fu X., Lu X., Petsverger B., Chen S., Qian K., and Tao L. (2007). A Static Analysis Framework for Detecting SQL Injection Vulnerabilities. *31<sup>st</sup> Annual International Computer Software and Applications Conference (COMPSAC 2007). IEEE*
- Halfond W. G., and Orso A., (2005). AMNESIA: Analysis and Monitoring for NEutralizing SQL-Injection Attacks. In *Proceedings of the IEEE and ACM International Conference on Automated Software Engineering (ASE 2005)*. Long Beach, CA, USA, Nov 2005.
- Halfond W.G., Viegas J., and Orso A., (2006). A Classification of SQL Injection Attacks and Countermeasures. In *Proceedings of IEEE International Symposium Secure Software Engineering*, Mar. 2006.
- Halfond William G. and Orso A., (2008). WASP: Protecting Web Applications Using Positive Tainting and Syntax-Aware Evaluation, *IEEE Transactions On Software Engineering*, Vol. 34, No. 1, January/February ,2008 pg 65-81

- Howard M. and LeBlanc D., (2003). Writing Secure Code. Microsoft Press, Redmond, Washington, second edition, 2003.
- Huang Y., 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. In *Proceedings of the 12th International World Wide Web Conference (WWW 04)*, May 2004.
- Kosuga Y., Kono K., Hanaoka M., Hishiyama M., and Takahama Y., (2007). Sania: Syntactic and Semantic Analysis for Automated Testing against SQL Injection, *23rd Annual Computer Security Applications Conference*, pg 107 - 116
- S. Labs. SQL Injection. White paper, SPI Dynamics, Inc., (2002). <http://www.spidynamics.com/assets/documents/WhitepaperSQLInjection.pdf>.
- Lin J., and Chen J., (2006). An Automatic Revised Tool for Anti-malicious Injection In *Proceedings of The Sixth IEEE International Conference on Computer and Information Technology (CIT'06)*
- Livshits V. B. and Lam M. S., (2005). Finding Security Errors in Java Programs with Static Analysis. In *Proceedings of the 14th Usenix Security Symposium*, pages 271–286, Aug. 2005.
- Mackay C. A., (2005). SQL Injection Attacks and Some Tips on How to Prevent Them. Technical report, The Code Project, January 2005. <http://www.codeproject.com/cs/database/SqlInjectionAttacks.asp>.
- McClure R., and Kruger I., (2005). SQL DOM: Compile Time Checking of Dynamic SQL Statements. In *Proceedings of the 27th International Conference on Software Engineering (ICSE 05)*, pages 88–96, 2005.
- Muthuprasanna M., Wei K., Kothari S. (2006). Eliminating SQL Injection Attacks - A Transparent Defense Mechanism. *Eighth IEEE International Symposium on Web Site Evolution (WSE'06).IEEE*
- Spett K., (2003). Blind sql injection. White paper, SPI Dynamics, Inc., 2003. <http://www.spidynamics.com/whitepapers/BlindSQLInjection.pdf>.
- Su Z., and Wassermann G. (2006). The Essence of Command Injection Attacks in Web Applications. In *The 33rd Annual Symposium on Principles of Programming Languages (POPL 2006)*, Jan. 2006.

- Thomas S., and Williams L. (2007). Using Automated Fix Generation to Secure SQL Statements. *Third International Workshop on Software Engineering for Secure Systems (SESS'07)*
- Ullrich J. B., Lam J., (2008). Defacing websites via SQL injection. *Network Security Magazine*, pg 9-10
- Valeur F., Mutz D., and Vigna G. (2005). A Learning-Based Approach to the Detection of SQL Attacks. In *Proceedings of the Conference on Detection of Intrusions and Malware and Vulnerability Assessment (DIMVA)*, Vienna, Austria, July 2005.
- Wei K., Muthuprasanna M., Kothari S. (2006). Preventing SQL Injection Attacks in Stored Procedures. In *Proceedings of the 2006 Australian Software Engineering Conference (ASWEC'06)*. *IEEE*

## Inputs list

ATTACK inputs	
1	1 OR 1=1
2	1' OR '1'='1
3	1'1
4	1 EXEC SP_ (or EXEC XP_)
5	1 AND 1=1
6	1' AND 1=(SELECT COUNT(*) FROM tablenames); --
7	admin' and 1=1 --
8	1\`1
9	' OR uname IS NOT NULL OR uname = '
10	') or ('1'='1
11	"or "1"="1
12	' or '1'='1
13	Or 1=1--
14	" or 1=1--
15	' or 1=1--
16	' OR 1=1 LIMIT 1
17	admin'--
18	hi" or "a"="a
19	hi" or 1=1 --
20	hi' or 'a'='a
21	' or 0=0 #
22	' UNION SELECT 1, 'Lee', 'lee_pwd', 1--
23	or 0=0 #
24	' union all select * from tbl_auth where id = 1 --
25	1' AND table_test = '1
26	1 AND table_test = 1
27	admin'/*

28	x' OR username LIKE '%Bob%
29	' + (SELECT TOP 1 password FROM users ) + '
30	admin' #
<b>NON ATTACK inputs</b>	
31	user1 (username); user1 (password)
32	user 1 (username); user 1 (password) (with spacing)
33	brian (username); typ-o (password)
<i>or any legitimate inputs can be categorized in NON ATTACK inputs</i>	

**Technique A : adopted from AMNESIA**

```
1 Get query
2 //for easier assessment because in grammar all lowercase
3 Change query into all lowercase letters
4 Explode query into tokens
5
6 Foreach token {
7     If token value = model value {
8         Display token
9         Set count to 1
10        } Else {
11        display error message
12        }
13    }
14    If exist case {
15        "digit=digit" or
16        "char=char"
17    }
18    Display message:"injection attempt detected at [token_value]"
19
20    Count total of parsed inputs (tot_parsed)
21    Count total of correct inputs      (tot_correct)
22
23    //comparing total of parsed and correct inputs
24    if ((tot_parsed) == (tot_correct)) {
25        Display message: "clean user input"
26        set flag = 1
27    } else { set flag = 0 }
28    return
```

**Technique B: adopted from SQL Guard**

```
1 Get query
2 //for easier assessment because in grammar all lowercase
3 Change query into all lowercase letters
4 Explode query into tokens
5
6 Foreach token {
7     If token value = model value {
8         Display token
9         Set count to 1
10        } Else {
11        display error message
12        }
13    }
14    If exist case {
15        "digit=digit" or
16        "char=char"
17    }
18    Display message:"injection attempt detected at [token_value]"
19
20    //create a set for comment
21        create empty array (comment array)
22        if token is not empty {
23            if token has not been evaluated
24                add token value into comment array
25
26        display comment array
27        } else { display message "comment token empty" }
28
29        foreach comment token value{
30            If exist case {
31                "digit=digit" or
32                "char=char" or
33                "and" or
34                "or"
```

```
35     }
36     Display message:"injection attempt detected at [token_value]"
37
38     Count total of parsed inputs (tot_parsed)
39     Count total of correct inputs (tot_correct)
40
41     //comparing total of parsed and correct inputs
42     if ((tot_parsed) == (tot_correct)) {
43         Display message: "clean user input"
44         set flag = 1
45     } else { set flag = 0 }
46     return
```

**Technique A: adopted from AMNESIA**

```

<?
$model = "SELECT table_column FROM table_name WHERE identifierA
= ' literal ' AND identifierB = ' literal '";
$split = explode(" ", $model);
//print_r($split);
$model = str_replace(" ", " -> ", $model);
echo "<b>$model</b>";

$str = $check_login;
echo $check_login;
$str = strtolower($str);
$str = str_replace(" '", "'", $str);
$str = str_replace("'", " ' ", $str);
$str = str_replace(" and", "and", $str);
$str = rtrim($str);

$explode = explode(" ", $str);
echo "<br><br>";

//print_r($explode);
//echo "<br><br>";

foreach($explode as $key => $value) {
if($value == "" || $value == " " || is_null($value)) {
unset($explode[$key]);
}
}

$explode = array_values($explode);
$altered_key = count($explode);

    print_r($explode);
    echo "<br><br>";
    if ($explode[0] === "select") {
    echo "[0]: keyword- ";
    print_r ($explode[0]);
    $count = 1; }
    echo "<br>";

    if (preg_match('/([a-z]|[A-Z])/', $explode[1])) {
    echo "[1]: identifier- ";
    print_r($explode[1]);
    $count += 1;
    } else if($explode[1] === "*") {
    echo "[1]: identifier- ";
    print_r($explode[1]);
    $count += 1;
    } else echo "[1] not an identifier";

```

```

echo "<br>";

if ($explode[2] === "from") {
echo "[2]: keyword- ";
print_r ($explode[2]);
$count += 1; }
echo "<br>";

if (preg_match('/([a-z]|[A-Z])/', $explode[3])) {
echo "[3]: identifier- ";
print_r($explode[3]);
$count += 1;
} else echo "[3] not a column name";
echo "<br>";

if ($explode[4] === "where") {
echo "[4]: keyword- ";
print_r ($explode[4]);
$count += 1; }
echo "<br>";

if (preg_match('/([a-z]|[A-Z])/', $explode[5])) {
echo "[5]: identifier- ";
print_r($explode[5]);
$count += 1;
} else echo "[5] not a column name";
echo "<br>";

if ($explode[6] === "=") {
echo "[6]: operator- ";
print_r ($explode[6]);
$count += 1; }
echo "<br>";

if ($explode[7] === "'") {
echo "[7]: quote- ";
print_r ($explode[7]);
$count += 1; }
else echo "[7] this is not a quote sign";
echo "<br>";

if (preg_match('/([a-z]|[A-Z]|[0-9])/', $explode[8])) {
echo "[8]: literal- ";
print_r($explode[8]);
$count += 1; }
else echo "[8] not a literal value string or digit only";
echo "<br>";

if ($explode[9] === "'") {
echo "[9]: quote- ";
print_r ($explode[9]);
$count += 1; }

```

```

else echo "[9] this is not a quote sign";
echo "<br>";

if ($explode[10] === "and") {
echo "[10]: keyword- ";
print_r ($explode[10]);
$count += 1; }
else echo "[10] wrong operator keyword [AND | OR]";
echo "<br>";
//else print_r($explode[10]);

if (preg_match('/([a-z]|[A-Z])/', $explode[11])) {
echo "[11]: identifier- ";
print_r($explode[11]);
$count += 1; }
else echo "[11] not a column name";
echo "<br>";

if ($explode[12] === "=") {
echo "[12]: operator- ";
print_r ($explode[12]);
$count += 1; }
else echo "[12] not an equal sign";
echo "<br>";

if ($explode[13] === "'") {
echo "[13]: quote- ";
print_r ($explode[13]);
$count += 1; }
else echo "[13] this is not a quote sign";
echo "<br>";

if (preg_match('/([a-z]|[A-Z]|[0-9])/', $explode[14])) {
echo "[14]: literal- ";
print_r($explode[14]);
$count += 1;
}
else echo "[14] not a literal value string or digit only";
echo "<br>";

if(!empty($explode[15])) {
if ($explode[15] === "'") {
echo "[15]: quote- ";
print_r ($explode[15]);
$count += 1; }
else echo "[15] this is not a quote sign";
echo "<br>";
}

foreach($explode as $key => $value) {
if(preg_match('/\d{1}(=\d{1})/', $value)) {
echo "injection attempt detected at $key";
}
}

```

```
$attack = 1;
}
}
echo "<br><br>";

echo "Total parsed input: $altered_key <br>";
echo "Correct user input: $count";
echo "<br><br>";

if ($altered_key == $count) {
echo "Clean user input";
$flag = 1;
} else {
$flag = 0;
}

?>
```

**Technique B: adopted from SQL Guard**

```

<?
$model = "SELECT table_column FROM table_name WHERE identifierA
= ' literal ' AND identifierB = ' literal '";
$split = explode(" ", $model);
//print_r($split);
$model = str_replace(" ", " -> ", $model);
echo "<b>$model</b>";

$str = $check_login;
echo $check_login;
$str = strtolower($str);
$str = str_replace(" '", "'", $str);
$str = str_replace("'", " ' ", $str);
$str = str_replace(" and", "and", $str);
$str = str_replace("and", " and ", $str);
$str = rtrim($str);

$explode = explode(" ", $str);
echo "<br><br>";

//print_r($explode);
//echo "<br><br>";

foreach($explode as $key => $value) {
if($value == "" || $value == " " || is_null($value)) {
unset($explode[$key]);
}
}

$explode = array_values($explode);
$altered_key = count($explode);

    print_r($explode);
    echo "<br><br>";
    if ($explode[0] === "select") {
    echo "[0]: keyword- ";
    print_r ($explode[0]);
    $count = 1; }
    echo "<br>";

    if (preg_match('/([a-z]|[A-Z])/', $explode[1])) {
    echo "[1]: identifier- ";
    print_r($explode[1]);
    $count += 1;
    } else if($explode[1] === "*") {
    echo "[1]: identifier- ";

```

```

print_r($explode[1]);
$count += 1;
} else echo "[1] not an identifier";
echo "<br>";

if ($explode[2] === "from") {
echo "[2]: keyword- ";
print_r ($explode[2]);
$count += 1; }
echo "<br>";

if (preg_match('/([a-z]|[A-Z])/', $explode[3])) {
echo "[3]: identifier- ";
print_r($explode[3]);
$count += 1;
} else echo "[3] not a column name";
echo "<br>";

if ($explode[4] === "where") {
echo "[4]: keyword- ";
print_r ($explode[4]);
$count += 1; }
echo "<br>";

if (preg_match('/([a-z]|[A-Z])/', $explode[5])) {
echo "[5]: identifier- ";
print_r($explode[5]);
$count += 1;
} else echo "[5] not a column name";
echo "<br>";

if ($explode[6] === "=") {
echo "[6]: operator- ";
print_r ($explode[6]);
$count += 1; }
echo "<br>";

if ($explode[7] === "'") {
echo "[7]: quote- ";
print_r ($explode[7]);
$count += 1; }
else echo "[7] this is not a quote sign";
echo "<br>";

if (preg_match('/([a-z]|[A-Z]|[0-9])/', $explode[8])) {
echo "[8]: literal- ";
print_r($explode[8]);
$count += 1; }
else echo "[8] not a literal value string or digit
only";
echo "<br>";

```

```

if ($explode[9] === "'") {
echo "[9]: quote- ";
print_r ($explode[9]);
$count += 1; }
else echo "[9] this is not a quote sign";
echo "<br>";

if ($explode[10] === "and") {
echo "[10]: keyword- ";
print_r ($explode[10]);
$count += 1; }
else echo "[10] wrong operator keyword [AND | OR]";
echo "<br>";
//else print_r($explode[10]);

if (preg_match('/([a-z]|[A-Z])/', $explode[11])) {
echo "[11]: identifier- ";
print_r($explode[11]);
$count += 1; }
else echo "[11] not a column name";
echo "<br>";

if ($explode[12] === "=") {
echo "[12]: operator- ";
print_r ($explode[12]);
$count += 1; }
else echo "[12] not an equal sign";
echo "<br>";

if ($explode[13] === "'") {
echo "[13]: quote- ";
print_r ($explode[13]);
$count += 1; }
else echo "[13] this is not a quote sign";
echo "<br>";

if (preg_match('/([a-z]|[A-Z]|[0-9])/', $explode[14])) {
echo "[14]: literal- ";
print_r($explode[14]);
$count += 1;
}
else echo "[14] not a literal value string or digit
only";
echo "<br>";

if(!empty($explode[15])) {
if ($explode[15] === "'") {
echo "[15]: quote- ";
print_r ($explode[15]);
$count += 1; }
else echo "[15] this is not a quote sign";
echo "<br>";
}

```

```

}

foreach($explode as $key => $value) {
if(preg_match('/\d{1}(=\d{1})/', $value)) {
echo "injection attempt detected at $key";
$attack = 1;
}
}
echo "<br><br>";

$comment = array();

    for ($key = 16; $key < $altered_key; $key++) {
        if(!empty($explode[$key])) {
            $com = array_push($comment, $explode[$key]);
        }
    }

if(!empty($comment)) {
print_r($comment);
} else { echo "comment token empty";
}

echo "<br><br>";

foreach($comment as $key => $value) {
if(preg_match('/\d{1}(=\d{1})/', $value)) {
echo "injection attempt detected at $key";
$attack = 1;
}
}
echo "<br><br>";

foreach($comment as $key => $value) {
if($value == "and" || $value == "or") {
echo "injection attempt detected at $key in Comment
token";
$attack = 1;
}
}
echo "<br><br>";

echo "Total parsed input: $altered_key <br>";
echo "Correct user input: $count";
echo "<br><br>";

if ($altered_key == $count) {
echo "Clean user input";
$flag = 1;
} else {

```

```
$flag = 0;  
}
```

```
?>
```

**Technique A : adopted from AMNESIA****Web #1: mekar**

<b>Input No</b>	<b>Detected as SQLIA?</b>	<b>Error?</b>	<b>ExecTime</b>	<b>Precision</b>	<b>Recall</b>
1	yes	yes	0.0037sec	1	0.0333
2	no	yes	0.0031sec	0.5	0.0333
3	no	yes	0.0030sec	0.333	0.0333
4	no	yes	0.0032sec	0.25	0.0333
5	yes	yes	0.0014sec	0.4	0.0667
6	no	yes	0.0016sec	0.333	0.0667
7	yes	yes	0.0014sec	0.429	0.1
8	no	yes	0.0023sec	0.375	0.1
9	no	yes	0.0017sec	0.333	0.1
10	no	yes	0.0015sec	0.3	0.1
11	no	yes	0.0022sec	0.273	0.1
12	no	yes	0.0035sec	0.25	0.1
13	yes	yes	0.0032sec	0.308	0.1333
14	yes	yes	0.0029sec	0.357	0.1667
15	yes	yes	0.0034sec	0.4	0.2
16	yes	yes	0.0032sec	0.4375	0.2333
17	yes	yes	0.0032sec	0.471	0.2667
18	no	yes	0.0032sec	0.444	0.2667
19	yes	yes	0.0029sec	0.4737	0.3
20	no	yes	0.0031sec	0.45	0.3
21	yes	yes	0.0033sec	0.7762	0.3333
22	no	yes	0.0032sec	0.4545	0.3333
23	yes	yes	0.0013sec	0.4783	0.3667
24	no	yes	0.0016sec	0.4583	0.3667
25	no	yes	0.0015sec	0.44	0.3667

26	no	yes	0.0013sec	0.4231	0.3667
27	no	yes	0.0012sec	0.4074	0.3667
28	no	yes	0.0033sec	0.3929	0.3667
29	no	yes	0.0016sec	0.3793	0.3667
30	yes	yes	0.0014sec	0.4	0.4

**\*Note: in Error Column**

Yes – authentication is not bypassed

**Technique A : adopted from AMNESIA****Web #2: myMarket**

<b>Input No</b>	<b>Detected as SQLIA?</b>	<b>Error?</b>	<b>ExecTime</b>	<b>Precision</b>	<b>Recall</b>
1	yes	yes	0.0024sec	1	0.0333
2	no	yes	0.0030sec	0.5	0.0333
3	no	yes	0.0023sec	0.3333	0.0333
4	no	yes	0.0023sec	0.25	0.0333
5	Yes	yes	0.0023sec	0.4	0.0667
6	no	yes	0.0025sec	0.3333	0.0667
7	Yes	yes	0.0025sec	0.4286	0.1
8	no	yes	0.0023sec	0.375	0.1
9	no	yes	0.0025sec	0.3333	0.1
10	no	yes	0.0024sec	0.3	0.1
11	no	yes	0.0031sec	0.2727	0.1
12	no	yes	0.0025sec	0.25	0.1
13	yes	yes	0.0024sec	0.3077	0.1333
14	yes	yes	0.0025sec	0.3571	0.1667
15	yes	yes	0.0024sec	0.4	0.2
16	yes	yes	0.0024sec	0.4375	0.2333
17	no	yes	0.0025sec	0.4118	0.2333
18	no	yes	0.0023sec	0.3889	0.2333
19	yes	yes	0.0025sec	0.4210	0.2667
20	No	yes	0.0028sec	0.4	0.2667
21	yes	yes	0.0025sec	0.4286	0.3
22	no	yes	0.0025sec	0.4091	0.3
23	yes	Yes	0.0024sec	0.4348	0.3333
24	no	yes	0.0024sec	0.4167	0.3333
25	no	yes	0.0025sec	0.4	0.3333

26	no	yes	0.0025sec	0.3846	0.3333
27	no	yes	0.0025sec	0.3704	0.3333
28	no	yes	0.0029sec	0.3571	0.3333
29	no	yes	0.0026sec	0.3449	0.3333
30	no	yes	0.0026sec	0.3333	0.3333

**\*Note: in Error Column**

Yes – authentication is not bypassed

**Technique B : adopted from SQL Guard****Web #1 : mekar**

<b>Input No</b>	<b>Detected as SQLIA?</b>	<b>Error?</b>	<b>ExecTime</b>	<b>Precision</b>	<b>Recall</b>
1	yes	yes	0.0165sec	1	0.0333
2	Yes	yes	0.0025sec	1	0.0667
3	No	yes	0.0037sec	0.6667	0.0667
4	No	yes	0.0025sec	0.5	0.0667
5	yes	yes	0.0024sec	0.6	0.1
6	yes	yes	0.0027sec	0.6667	0.1333
7	yes	yes	0.0028sec	0.7143	0.1667
8	no	yes	0.0026sec	0.625	0.1667
9	yes	yes	0.0026sec	0.6667	0.2
10	yes	yes	0.0026sec	0.7	0.2333
11	no	yes	0.0024sec	0.6364	0.2333
12	yes	yes	0.0026sec	0.6667	0.2667
13	yes	yes	0.0025sec	0.6923	0.3
14	yes	yes	0.0029sec	0.7143	0.3333
15	yes	yes	0.0025sec	0.7333	0.3667
16	yes	yes	0.0024sec	0.75	0.4
17	yes	yes	0.0025sec	0.7647	0.4333
18	no	yes	0.0025sec	0.7222	0.4333
19	yes	yes	0.0024sec	0.7368	0.4667
20	yes	yes	0.0028sec	0.75	0.5
21	yes	yes	0.0024sec	0.7619	0.5333
22	yes	yes	0.0026sec	0.7727	0.5667
23	yes	yes	0.0027sec	0.7826	0.6
24	yes	yes	0.0024sec	0.7917	0.6333
25	yes	yes	0.0024sec	0.8	0.6667

26	no	yes	0.0024sec	0.7692	0.6667
27	no	yes	0.0024sec	0.7407	0.6667
28	yes	yes	0.0025sec	0.75	0.7
29	yes	yes	0.0026sec	0.7986	0.7333
30	yes	yes	0.0024sec	0.7667	0.7667

**\*Note: in Error Column**

Yes – authentication is not bypassed

**Technique B : adopted from SQL Guard****Web #2: myMarket**

<b>Input No</b>	<b>Detected as SQLIA?</b>	<b>Error?</b>	<b>ExecTime</b>	<b>Precision</b>	<b>Recall</b>
1	yes	yes	0.0012sec	1	0.0333
2	yes	yes	0.0014sec	1	0.0667
3	no	yes	0.0014sec	0.6667	0.0667
4	no	yes	0.0012sec	0.5	0.0667
5	yes	yes	0.0023sec	0.6	0.1
6	yes	yes	0.0013sec	0.6667	0.1333
7	yes	yes	0.0012sec	0.7143	0.1667
8	no	yes	0.0013sec	0.625	0.1667
9	yes	yes	0.0013sec	0.6667	0.2
10	yes	yes	0.0013sec	0.7	0.2333
11	no	yes	0.0012sec	0.6364	0.2333
12	yes	yes	0.0012sec	0.6667	0.2667
13	yes	yes	0.0012sec	0.6923	0.3
14	yes	yes	0.0012sec	0.7143	0.3333
15	yes	yes	0.0012sec	0.7333	0.3667
16	yes	yes	0.0012sec	0.75	0.4
17	no	yes	0.0013sec	0.7059	0.4
18	no	yes	0.0025sec	0.6667	0.4
19	yes	yes	0.0012sec	0.6842	0.4333
20	yes	yes	0.0013sec	0.7	0.4667
21	yes	yes	0.0013sec	0.7143	0.5
22	yes	yes	0.0016sec	0.7272	0.5333
23	yes	yes	0.0018sec	0.7391	0.5667
24	yes	yes	0.0014sec	0.75	0.6
25	yes	yes	0.0014sec	0.76	0.6333

26	no	yes	0.0013sec	0.73088	0.6333
27	no	yes	0.0012sec	0.7037	0.6333
28	yes	yes	0.0014sec	0.7143	0.6667
29	yes	yes	0.0013sec	0.7241	0.7
30	no	yes	0.0034sec	0.7	0.7

**\*Note: in Error Column**

Yes – authentication is not bypassed