

Spring 2017

Securing Database Users from the Threat of SQL Injection Attacks

Nisharg Shah
npshah@smu.edu

Follow this and additional works at: https://scholar.smu.edu/engineering_electrical_etds



Part of the [VLSI and Circuits, Embedded and Hardware Systems Commons](#)

Recommended Citation

Shah, Nisharg, "Securing Database Users from the Threat of SQL Injection Attacks" (2017). *Electrical Engineering Theses and Dissertations*. 1.
https://scholar.smu.edu/engineering_electrical_etds/1

This Thesis is brought to you for free and open access by the Electrical Engineering at SMU Scholar. It has been accepted for inclusion in Electrical Engineering Theses and Dissertations by an authorized administrator of SMU Scholar. For more information, please visit <http://digitalrepository.smu.edu>.

SECURING DATABASE USERS FROM THE THREAT
OF SQL INJECTION ATTACKS

Approved by:

Dr. Jennifer Dworak
(Associate Professor)

Dr. Ping Gui
(Professor)

Dr. Frank Coyle
(Senior Lecturer)

Dr. Freeman Moore
(Adjunct Faculty)

Dr. Theodore Manikas
(Clinical Professor)

SECURING DATABASE USERS FROM THE THREAT
OF SQL INJECTION ATTACKS

A Thesis Presented to the Graduate Faculty of

Bobby B. Lyle School of Engineering

Southern Methodist University

in

Partial Fulfilment of Requirements

for the degree of

Master of Science in Electrical Engineering

by

Nisharg Pankaj Shah

(Bachelor of Engineering, Mumbai University, INDIA, 2014)

December 16, 2017

Shah Nisharg

B.Eng., Mumbai University, 2014

Advisor: Dr. Jennifer Dworak

Securing Database Users from the Threat of SQL Injection Attacks

Masters of Science conferred December 16, 2017

Thesis completed November 20, 2017

In the 21st century, there has been a significant rise in dependency on the Internet for daily activities. Web applications such as online banking, web-based emails, social networking, and many more services have become an instant means of communication. These web applications and the data to which they have access are often targeted using malicious attacks, including SQL (Structured Query Language) Injection Attacks (from now on referenced as SQLIAs), which may cause serious damage. In particular, attackers use SQLIAs to target interactive web applications that incorporate database services. In a SQLIA, an attacker can insert malicious SQL code as an input to perform unauthorized database operations, which could potentially jeopardize the privacy, integrity and security of the users.

This thesis proposes an unconventional hardware-based approach for solving SQL vulnerabilities and thwarting SQLIAs. Specifically, we are approaching the problem of SQL Injection by using an FPGA to search for action-based binding keywords which join multiple queries. These keywords form an integral part of any attack query. We search for these keywords in a user's input space and replace them with a null string. By doing so, the binding query is nullified, and the attacking query is rendered harmless.

Table of Contents

List of Tables	vi
List of Figures	vii
ACKNOWLEDGEMENT	ix
CHAPTER 1. INTRODUCTION	1
1.1 Input Vulnerability-Based Exploits	2
1.2 Motivation of the Thesis	4
1.3 Thesis Organization	5
CHAPTER 2 BACKGROUND	6
2.1 Definitions for SQL and related terms.....	6
2.4 SQL injection attack shopping cart example	13
2.4 Mechanisms used for causing SQLIA	15
CHAPTER 3 LITERATURE SURVEY	25
CHAPTER 4 INTRODUCTION TO THE PROPOSED HARDWARE-BASED SQLIA PREVENTION APPROACH	31
4.2 Substring Match and Replace	35
CHAPTER 5 SQL INJECTION ATTACKS DETECTION AND.....	37
PREVENTION IMPLEMENTATION	37
5.1 Key word selection process	37
5.2 Keyword searching circuit design.....	39
5.3 Implementation evaluation.....	43

4.4 SQL injection defense analysis in Software.	47
CHAPTER 6 CONCLUSIONS & FUTURE WORK	51
5.1 Conclusions.....	51
5.2 Future Work	52
Appendix A.....	54
Appendix B Software Implementation	57
REFERENCES	59

List of Tables

Table 1	Table Relational Match Operator	7
Table 2	Tautology attack using input box	17
Table 3	Tautology attack using URL	17
Table 4	Error based illogical Query attack.	18
Table 5	Example of a piggy backed SQL injection attack.....	21
Table 6	SQLIA by using alternate encoding.....	22
Table 7	Reserved keywords and function	38
Table 8	Comparison of search algorithm performance with other methods implemented previously	46

List of Figures

Figure 2.1 A canonical web architecture .	8
Figure 2.2 SQL injection Bobby drop tables	10
Figure 2.3 SQLIA shopping cart example .	14
Figure 2.4 DBMS Function invoking attack	19
Figure 2.5 Union attack	20
Figure 2.6 A second order SQLIA	24
Figure 4.1 The sequence of work behind the web page form	32
Figure 4.2 Parse trees for malicious vs legitimate queries.	33
Figure 4.3 System architecture for modified SQL processing system modified by adding the string searching hardware.	36
Figure 5.1 SQL reserved keywords testing	37
Figure 5.2 Block diagram for SQL string search.	41
Figure 5.3 Technology mapping for fixed keyword matcher	41
Figure 5.4 Design for one-character comparison	42
Figure 5.5 Design for one-word comparison	43
Figure 5.6 Timing simulation on cyclone IV device for keyword matching	43
Figure 5.7 Testing the results of string search outputs on SQL practicing website.	45
Figure 5.8 Flow chart for software implementation of string searching algorithm	47
Figure 5.9 Comparison of the designed approach in software and hardware	49

ACKNOWLEDGEMENT

First of all, I am deeply grateful to my parents for making me realize my potential and putting me through such good education system. I am thankful for all the sacrifices that they have made.

I am deeply grateful, and I wish to express my deepest gratitude to

Dr. Jennifer Dworak

Associate Professor at the Lyle School of Engineering for suggesting the point of this thesis, supervision of work and invaluable guidelines, the longtime and tremendous efforts to offer every possible help to finish this thesis. It was great honor to finish this work under her supervision.

I am genuinely appreciative of

Dr. Frank Coyle

Senior Lecturer at Lyle School of Engineering for his valuable scientific assistance in understanding the scale of my thesis and his undoubted efforts in making my life easy. I am also thankful to Dr Coyle for seeding the idea of implementing a solution for SQLIA on FPGA.

I am honored to work with

Dr. Freeman Moore

Adjunct Faculty at the Lyle School of Engineering for helping me in understanding the analytical aspects of my thesis and for his wonderful insights on SQL processing.

I would also like to thank **Dr. Ping Gui** and **Dr. Theodore Manikas** for their unconditional support and for being on my examination committee.

CHAPTER 1.

INTRODUCTION

Global development relies on insights and advancements in technology. As “The Stone Age did not end for lack of stones, and Oil Age will end long before the world runs out of oil.” (said by Sheikh Zaki Yamani in the Economist edition of October 2003) [1], [53]. Today, we live in what many call the Information Age, and we are absolutely in no danger of running out of information. There is a general perception that we are overwhelmed with data, making the ability to store, process, analyze, consume, secure and act upon data as a primary concern. For large scale, multi-national organizations, such as the finance industry, or the health industry, the situation has become very complex and challenging. The question then becomes, how do we process and store this large amount of data while maintaining its secrecy, reliability, and availability.

In many cases, big data is stored for efficient access by creating relations in data and then storing them into database, which is then called relational database. These databases are accessed for data via websites and APIs (Application Program Interface) through a language called Structured Query Language (SQL). Many of these websites are vulnerable to hackers seeking to perform attacks because backdoors are left open by poor coding practices that do not adhere to good security standards. Hackers unleash hordes of bots to look for sites that have these coding weaknesses[42]. In some cases, specific websites may be targeted due to the value of the data a hacker hopes to access. In both cases, trusting designers to simply write good code has proven to be an inadequate

response, and no perfect protection system exists. It has become crucial to address this issue and to counteract the attacker's creative steps for hacking into systems.

1.1 Input Vulnerability-Based Exploits

Many of the vulnerabilities in web applications take advantage of the ability of the hacker to access that application as a user or potential user. Thus, the data entered by the user becomes a vector into the system that can be used to carry out a desired exploit. The number, diversity, and complexity of these web based exploits have been gradually increasing ever since they came in to existence. Here we present some examples of such exploits:

- **SQL Injection:** SQL injection is a subset of code injection attack [54]. The attacker embeds a piece of code in a computer program via user input/data entry. Execution of the infected program provides the attacker with improper access to the computer program or application. Although web-based applications are developed to achieve desirable access to the database by authorized users, the attacker uses them to his benefit to gain access to confidential information stored in the database. In this thesis, SQL Injection Attacks will be abbreviated as SQLIA.
- **Cross Site Scripting:** Cross Site scripting (XSS) is a code injection attack where malicious code is injected in the website and is executed in a browser. The attacker inserts a script in the victim's browser, and when the compromised website is accessed, the script then executes on the victim's computer. Through XSS, the attacker can control browser of the attacked entity remotely. It is a way of bypassing the Standard Operating Procedure (SOP) concept. For example, whenever HTML code is generated dynamically, and the

user input is not sanitized and is reflected on the page, an attacker could insert his own HTML code in that page [45].

- Cross Site Request Forgery: This is an attack in which the attacker deceives the user into executing actions that are helpful to the attacker while accessing a web application. A CSRF vulnerability allows an attacker to force the user to perform activities that the attacker wants to carry out, when the user is logged into a website. For example, a perpetrator may forge a request for transferring fund to a website.

The above-mentioned vulnerabilities are all instances of input-based vulnerabilities. There are many other Web-attack vulnerabilities such as buffer errors, path traversal, code injection etc., but they don't necessarily depend on user's input for the victim's exploitation.

The relative prevalence of some of these attacks is shown in the figure below. This data was obtained from HACKING & TRICKS, a blog about Hacking & Computer Security by Nirav Desai, in an entry from January 8th, 2013 [2]. Clearly, SQL Injection comprises a significant fraction (approximately 25%) of all web-based attacks.

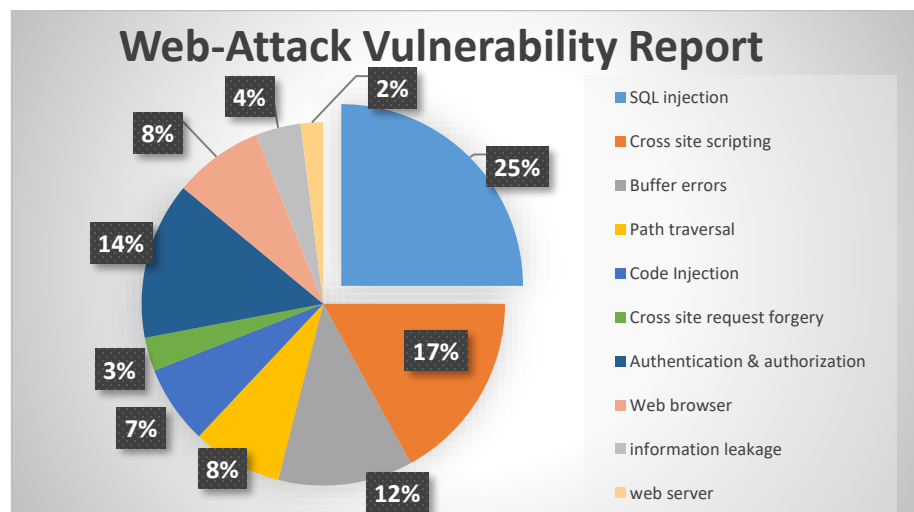


Figure 1.1 Web-attack Vulnerability Report [2].

1.2 Motivation of the Thesis

The prevalence of SQLIAs and the potential damage that they can cause, including identity theft and the extraction of financial data, requires good solutions to address these exploits. The applications affected by SQLIAs is constantly increasing, and history has shown that depending on website and application developers to write perfect code that adequately checks user input is an inadequate solution. Alternative approaches are needed to help protect these databases.

The basic intention of this thesis is to design a solution for preventing SQLIA where the solution does not depend on good coding practices. In addition, the method should provide good performance. Although checking of user input for correctness and safety has previously been proposed, such approaches may take a significant number of clock cycles and require the programmer to implement them correctly. In this research, we explore the potential implementation of alternative checker for user input in dedicated hardware instead of software. Specifically, we implemented a string search and replace method in hardware with the goal of removing one of the primary vectors of SQLIA attacks. The strings to be matched in SQL statements are the action-based binding keywords necessary for joining attacking parameters to a query in a SQL statement. The use of an FPGA (Field-Programmable Gate Array) for this checking allows multiple characters and keywords to be checked simultaneously in parallel. When a keyword is found, it is removed from the user's input and replaced by null characters. Thus, only sanitized user input may be seen by the system. By replacing these keywords with nulls, we achieve a very good secured database system, which is immune to SQLIAs that depend on those keywords.

1.3 Thesis Organization

The rest of the thesis is organized as follows. In Chapter 2, we review the basic concepts of SQL and investigate various factors leading to SQLIAs. In particular, we try to answer why SQLIAs occur and describe the exploits in a web application that an attacker uses to cause a SQLIA. In Chapter 3, we describe work done previously on the prevention of SQLIAs. In Chapter 4, we describe, our proposed method for preventing SQLIAs. In Chapter 5, we describe implementation of our design and evaluate the performance of our design when implemented in software and hardware. Furthermore, we compare the performance of our design in hardware to other previously applied solutions in hardware. Finally, we conclude our thesis in Chapter 6, and lay our plans for future work on this project.

CHAPTER 2

BACKGROUND

2.1 Definitions for SQL and related terms

SQL (pronounced as “sequel” and an abbreviation for “Structured Query Language”) is a high-level language [55], the basis of which is heavily dependent upon relational algebra and relational Calculus. SQL is made up of declarative elements such as queries, expressions, clauses, statements, etc. It is widely known for being a powerful query language. The basic difference between a query language (QL) and a programming language (PL) is that QLs are not expected to be Turing complete, they are not expected to be used for complex calculations, and they have a very good efficiency for handling large data sets [46].

Implementation of a query language is based on the relational algebra (operational part) and the relational calculus (declarative part) [3]. Relational Algebra refers to a specification of a sequence of operations for performing a particular request, whereas relational calculus refers to the specification of a required output without any information about the sequence of operation required to process a request [47].

Table 1 lists a few examples of Relational Math Operators that are used in the development of a query statement.

Table 1 Table Relational Match Operator [3].

(σ) Selection	Selects a subset of rows from a table.
(π) Projection	Deletes attributes from tables which is not present in relation list.
(X) Cross product	Allows combinations of relational search.
(-) Set difference	Defines mutually exclusive relations.
(U) Union	Defines mutual dependability of relations.

Additional operators, such as intersection, join, division, and renaming are useful as well.

To summarize SQL, it's a relational model that is defined rigorously with simplicity and the power of operational algebra to efficiently carry out all the database related tasks with the best possible optimization. As a result, SQL is the lingua franca for accessing database systems.

After looking at SQL definitions and related terms, we consider the underlying method for processing on the web and discuss the architecture required for processing SQL. We describe the processing of a web application with the following sequence:

1. When a client (typical web user's internet connected device) enters a web address in a web browser application, the sever (computer that stores webpages, sites, or apps) sends a copy of a form to the client.
2. Users of the web browser enter data in this form and submit it to the server.
3. The server runs the script for the form and when it determines it is appropriate gives the client access to the underlying application or database.

The architecture required for processing a web application is as shown in Figure 2.1.

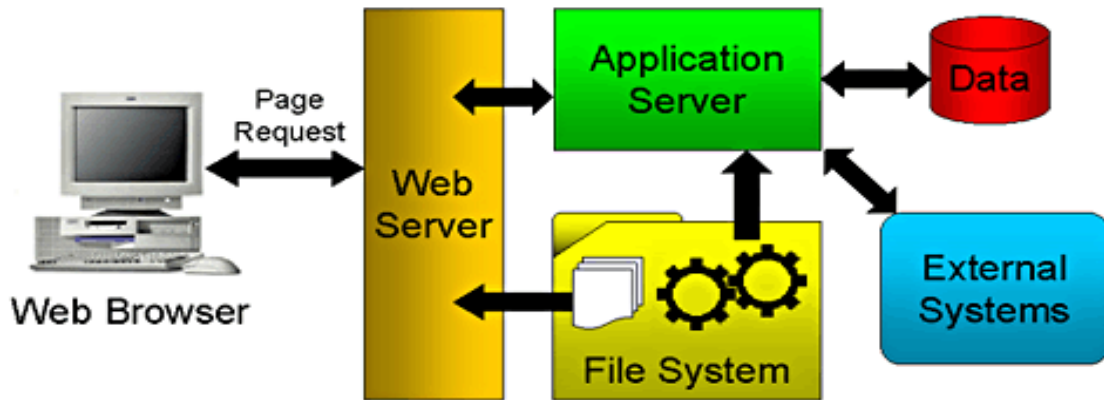


Figure 2.1 A canonical web architecture [4].

The web architecture is broadly classified into three tiers of processing. The web browser in Figure 2.1 represents the Presentation Tier. The Web Server, Application Server, and File System make up the Server Tier. Finally, the Data represents the Database Tier of a web application system. Functions of each of these tiers are described in [48] and summarized below.

1. Presentation Tier: holds the conversation with the users for the system. It deals with the user's input, the response from the system, and is the face of the web service for the user.
2. Server Tier: is the backbone of the entire database systems as it encapsulates the logic required for the database to work.
3. Database Tier: is the storage point where all the data are stored.

Security measures could be implemented at the Presentation Tier (the application), or at the Server Tier (server space), or at the Database Tier (physical database location). Protective measures which could be applied to web applications could include software patches on data acquisition processes at the Presentation Tier, data encryption at the Database Tier. However, in many cases security measures at the Server Tier of a web application system are especially appropriate for preventing various types of attacks.

2.2 SQL Injection Attack (SQLIA)

A SQL injection attack is a type of code injection attack. This exploit is carried out by adding SQL code in the user's input to gain access to unauthorized resources. SQLIAs may occur when the query is built by concatenating the user's input, such as data entered into a web form, with unintended data, including URL (Uniform Resource Locator) data, data obtained from cookies etc., without proper validation. The contributors to Rain Forest Puppy, a black-hat community website, were the first ones to ever publish information about SQLIAs in their paper "NT Web Technology Vulnerabilities" [5]. SQL injection is one of the favorite attacks for many cybercriminals because it can be executed remotely, and the attack surface is obvious. Commercially available vulnerability detection tools are accessible to attackers as well, and with help of these resources, the attacker could find loopholes in a security system and web vulnerabilities in a mere fraction of a second. SQL is a very flexible language, and these attacks can be extremely stealthy and could pass through firewalls and intrusion prevention systems very with little effort [59].

Figure 2.2 shows a well-known cartoon illustrating of the effects of SQLIAs. As we can, see when Robert'); DROP TABLE students; __ was inserted in the school's database, all the entries of the students for that year were lost. This is a humorous example that shows how SQLIAs can be caused by inserting particular phrases in a SQL query. (In this example, the child just happens to be named with a SQL expression that caused the deletion of the database.) In general, such entries would not be normal, but would be entered maliciously by an attacker to perform a similarly problematic function. The possibilities of exploitation using SQLIAs are great and a significant number of security

enterprises and websites report on these exploits to create awareness among people about threats due to SQL injection attacks.

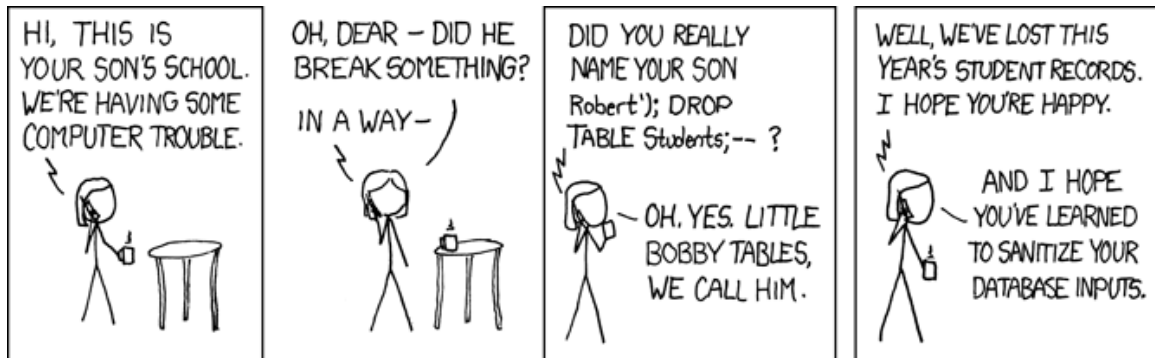


Figure 2.2 SQL injection Bobby drop tables [6]

According to the 2016 Vulnerability Statistics Report [43], by edgescan, 82% of vulnerability factors arise from the application layer of the network, which is a high-risk target. With respect to databases up to 95% of the critical risk factors were found in the web application layer. Examples of SQLIAs consist of the 2011 hack of the security firm HB Gray Federal, which allowed attackers to steal passwords for company's corporate emails and more than 60,000 emails were exposed online [43]. A SQL Injection attack vector was used in hacking the Chinese toy company VTE, in which the information of around 4.9 million people was stolen [7]. The famous Albert Gonzalez attack, including the hacking of 7-Eleven, Hannaford Brothers, and Heartland payment systems included SQL injection as the active attack vector [7]. The ease with which a SQL attack can be conducted and the extent of damage that it causes makes SQL Injection a preferable choice for hackers. Defense against SQLIAs requires knowing how these vulnerabilities prevail in the system and how they are taken advantage of by hackers to gain profit.

For a SQLIA to affect a system, the attacker needs to know about the injectable parameters of the system (i.e. he needs to reconnoiter the vulnerabilities present in the

web application and look for the injectable location for forming an attack which would yield the most promise). The attacker must then formulate an attack that is stealthy enough to avoid detection from firewalls and protection systems. He must then garner as much data as he can from the affected system so that he can make the most of victim's data. SQLIAs are caused mainly due to the lack syntax constraints of some programming languages and poor programming practices. Four different categories of SQLIA vectors are described in [8] and summarized below:

1. SQL manipulation: In this attack, the clause following the “where” clause is manipulated to produce behavior not expected by the database programmer (e.g. supplying the where clause with a union statement can provide access to data for which the user should not have access.)
2. Code injection: In this attack, a new SQL statement is concatenated to the previously present SQL statement (e.g. appending an execute statement at the end of a general statement.) A limitation of this kind of SQLIA is that the database must support multiple SQL statements per query.
3. Function call injection: This is the secondary injection attack wherein the attacker uses the inbuilt functions of the database to cause an SQLIA that manipulates the data according to the needs of the attacker.
4. Buffer overflow attack: In this case, the data entered as input would heavily exceed the memory bounds of the planned storage space. It would overwrite the data pointers and could also be used to point towards an executable causing the system to execute any file of the attacker's intent.

In this thesis, we attempt to prevent SQL injection attacks related to the first three categories. We do not address buffer overflow.

2.3 SQL Processing Overview

SQL processing happens in three main phases [9]:

1. Parse
2. Execute
3. Fetch

Parsing phase: Parsing, is the first stage in processing a query which checks for syntax and semantic validity of the query statement by tokenizing the query statements. For parsing a sentence, to make it a meaningful query, the sentence needs to be analyzed lexically, syntactically, and semantically.

Execution phase: The query execution phase starts with query optimization. Optimization consists of generating an appropriate sequence in which the logical process of query execution proceeds and is mapped to physical resources. Specifically, optimization generates an execution plan that is selected to increase performance and reduce the use of server resources [60]. The cost factor for the execution plan depends upon physical resources such as I/O, CPU, memory, the number of records the engine would need to process, etc. The actual execution starts with the optimized plan and executes the mapped physical instructions;

Fetch phase: This is the phase where the rows of the results to the query are obtained and made accessible to the end users. The execution phase determines what data must be extracted for the query and the fetch phase retrieves that data. The data can then be operated

on by the execution engine before being sent back to the client. Now let us consider an example of a SQL injection attack and see how SQLIAs can occur with small changes in user input data.

2.4 SQL injection attack shopping cart example

A typical web application consists of a client-side codebase and a server-side codebase executing on the client's web browser and the web server respectively. The client-side code interacts with the users and supplies user inputs to the server-side code. The server-side code computes the operations based upon these inputs to provide appropriate services. Figure 2.3 shows a shopping cart example where the user has already filled the cart with the products that he or she wishes to purchase. The checkout form asks for the payment information and the delivery information from the user.

Considering that the backend database is parameterized, this example would construct an insert query, for appending the user's input to the 'orders' table. Consider the user giving following inputs:

Qty₁=1, Qty₂=1, Qty₃=1, card = "xxxx-xxxx-xxxx-8739", name = "Nancy",
address = "ABC street, #123, Texas-72771, USA" operation= "purchase"

The screenshot shows the HAUTELOOK checkout page. At the top, there's a navigation bar with 'Nordstrom Rack' and 'HauteLook' logos, and links for 'Hello, (Not You?)', 'Account', 'Help', and 'Invite a Friend, Get \$10'. Below this is a dark blue header with 'HAUTELOOK' in large white letters, and 'FREE SHIPPING ON ORDERS OVER \$100 | RETURNS TO NORDSTROM RACK' on the right. Navigation links for 'ALL EVENTS', 'WOMEN', 'MEN', 'KIDS', 'HOME', and 'BEAUTY' are present, along with a 'NORDSTROM rack for HauteLook' logo.

The main content area is titled 'Checkout' and is divided into three sections:

- 1. SHIPPING:** Includes a dropdown menu for 'Home', a 'Add a New Shipping Address' link, and a '2. PAYMENT METHOD' section with a 'Payment Type' dropdown (set to 'Credit Card'), a 'Visa ****' card, and an 'Add a New Payment Method' link.
- 3. CART:** Displays two items:
 - Patty Set (Baby Girls):** Color: Blue / Size: 24M, Returnable, Extended Delivery Dates: Tue 06/24/14 to Thu 07/03/14, Price: \$13.97.
 - 7 For All Mankind Bootcut Jean:** Color: TOLUCA BRIGHT BLUE / Size: 25, Returnable, Estimated Delivery Dates: Thu 06/12/14 to Mon 06/16/14, Price: \$99.97.

Buttons for 'HAVE QUESTIONS?', 'PLACE ORDER', 'Remove', and 'Qty' are visible throughout the cart section.

Figure 2.3 SQLIA shopping cart example [10].

Let us assume that the total purchase is of \$130. The query generated for this set of inputs will be

```
INSERT INTO ORDERS ('ps', 'bj', 'hs')
VALUES ('Nancy', '130', 'xxxx-xxxx-xxxx-8739', 'ABC street, #123, Texas-72771,
USA')
```

When the above query is executed, a record of purchase is updated to the Orders table.

Now consider, a malicious attacker who wants to supply as input the *address* = “ABC street, #123, Texas-72771, USA”; *DROP TABLE Orders*; --”.

The query generated for this set of input will be


```
INSERT INTO ORDERS ('ps', 'bj', 'hs') ,  
VALUES ('Nancy', '130', 'xxxx-xxxx-xxxx-8739', 'ABC street, #123, Texas-72771  
USA');  
DROP TABLE Orders; --)
```

As we can see, the address parameter is followed by the delimiter placed after the original address input. In addition, the delimiter is followed by an attack sequence, which drops the table. Another thing to notice here is that the attacker would not be charged any money because of the following hyphen, which is another delimiter used for commenting out the string present after it. Hence, the system doesn't get any purchase confirmation, but the result of this query is that the Orders table is deleted from HAUTELOOK's database. To further analyze SQL injection attacks, we need to explore the techniques that attackers use to cause SQL injection.

2.4 Mechanisms used for causing SQLIA

As previously noted, we consider SQLIAs that are caused by code injection, SQL manipulation, and function call injection in this thesis. There are two ways of injecting malicious data in a database

1. Data manipulation through the user's input
2. Data injection by seeding an indirect trigger

Data manipulation using user's input: In this attack, the attacking parameters are inserted in the input that is taken from the user (i.e. the attacker injects malicious code by masquerading as the user's input deep inside the SQL query); this makes it difficult for the SQL Data Base Management System (DBMS) to prevent illegal access to the database.

The authors of [44] classify SQL injection attacks implemented through user input data as belonging to one of five categories: tautologies, logically incorrect queries, union based queries, piggy backed queries, and alternate encoding. The details of these types of attacks and the ways in which can be executed are discussed below:

i) Tautologies: The attack location for a tautology is in an input field or URL that is vulnerable and openly exploitable. This type of attack requires a conditional statement to execute successfully. There are many models of SQL injection attacks based on the “=” operator, but many relational operators such as “<”, “>”, “<=”, “>=” also relate to a result in a true or a false fashion. Possible goals of an attacker in a tautology attack include identifying injectable parameters, bypassing authentication, and extracting data. For example, tautologies can be used for causing more advanced attacks, such as changing the admin passwords without checking for the old password [44], as shown in the following query:

```
UPDATE users SET password= 'I got u' WHERE Username= "admin" ' _ _ '
AND password = '1=1'
```

In this example, the tautology arises from the entry of ‘1=1’ in the password field. Because this statement is always interpreted as true by SQL, the system will execute the command to change the password without the attacker needing to know the previous password.

Tables 2 and 3 provide two more examples of tautology attacks: one for a user input box and one for a URL. In both cases, a legitimate query, SQL injection affected modified query, and the result obtained due to the execution of modified query are listed.

Table 2 Tautology attack using input box [41]

Example	1>0 is always true
Point of insertion	Input box
Normal Query	SELECT * FROM Customers WHERE Country='Germany' AND City='Berlin';
Modified query	SELECT * FROM Customers WHERE Country='Germany' AND 1>0;
Result comparison	Normal query displays just one element of the table Customers (the one who lives in Berlin) but the modified query gives data for 91 customers—all the customers in Germany.

Table 3 Tautology attack using URL [41]

Example	https://www.xyz.org/index.aspx?id==3343' or $\bar{\omega}_l = \bar{\omega}_l$;
Point of insertion	URL
Normal Query	SELECT *from users WHERE ccnum=xxxxxxxxxxxx9875
Modified query	SELECT * from users WHERE ccnum=xxxxxxxxxxxx9875 or $\bar{\omega}_l = \bar{\omega}_l$;
Result comparison	Normal query displays just one element of the table users, but the modified query results in all entries of data from the table users

ii) Logically incorrect queries: These attacks are a type of brute-force technique used for obtaining information about the backend system. They form a major part of a standard reconnaissance effort. The performance of this type of attack depends upon the number of requests required to perform the data extraction process and the time that is required to

gather enough data to enable the attacker to cause a successful attack [51]. Commonly known injection vectors have a history of depending upon the Database structure and the signature of the SQL Injection attacks to which the database is vulnerable. A few ways to investigate the backend database of a system to fingerprint the vulnerability list include:

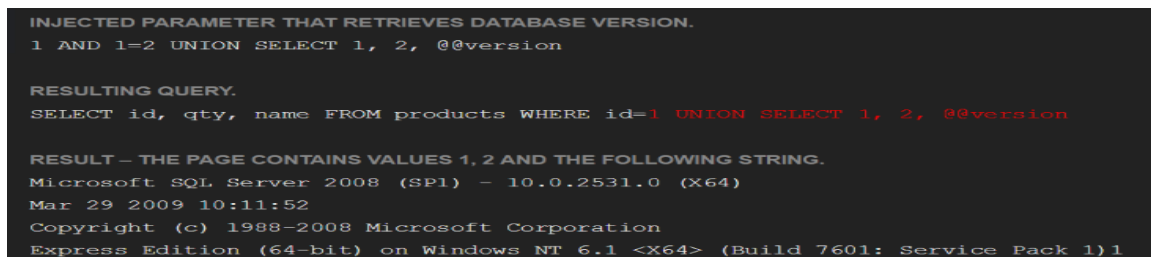
a) Unique error based vulnerability detection: This type of attack occurs when an attacker deliberately writes an illogical query and waits for the system to generate an error message. Table 4 lists an example of input data provided by an attacker in the URL of website where he tries to use the keyword *near* with the wrong syntax. A legitimate query, the SQL injection affected modified query, and the result obtained due to execution of the modified query are shown below. The response to the modified query provides information regarding the version of the backend SQL database. Such information is useful to an attacker in carrying out subsequent attacks.

Table 4 Error based illogical Query attack.

Example	http://example/index.php?near id=1'
Point of insertion	URL
Normal Query	SELECT * FROM users WHERE id = '1'
Modified query	SELECT * FROM users WHERE id = '1''
Result	You have an invalid query, please check the manual for mssql version 5.2

The success of this kind of attack depends upon the error messages displayed by the database.

b) DBMS Function Invoking: In this attack, the inbuilt debugging functions are used by the attacker to obtain information regarding the database's backend. For example, the easiest and most accurate way to identify which database is used is to ask the database to identify itself. This would require the attacker to possess good union attack skills. The DBMS querying functions such as @@version or version () are functions which would provide information about the underlying software used for the system configuration. Because the configuration details of SQL are well documented and standardized, such functions will be known to an attacker *a priori*. Figure 2.5 displays use of the inbuilt DBMS function @@version for inferring the backend database.

The image is a screenshot of a terminal window with a dark background and light-colored text. It shows the output of a SQL injection attack. The first line is a header: "INJECTED PARAMETER THAT RETRIEVES DATABASE VERSION." followed by the injected payload: "1 AND 1=2 UNION SELECT 1, 2, @@version". The next line is "RESULTING QUERY." followed by the full SQL query: "SELECT id, qty, name FROM products WHERE id=1 UNION SELECT 1, 2, @@version". The final section is "RESULT - THE PAGE CONTAINS VALUES 1, 2 AND THE FOLLOWING STRING." followed by the database's response: "Microsoft SQL Server 2008 (SP1) - 10.0.2531.0 (X64)", "Mar 29 2009 10:11:52", "Copyright (c) 1988-2008 Microsoft Corporation", and "Express Edition (64-bit) on Windows NT 6.1 <X64> (Build 7601: Service Pack 1)1".

```
INJECTED PARAMETER THAT RETRIEVES DATABASE VERSION.  
1 AND 1=2 UNION SELECT 1, 2, @@version  
  
RESULTING QUERY.  
SELECT id, qty, name FROM products WHERE id=1 UNION SELECT 1, 2, @@version  
  
RESULT - THE PAGE CONTAINS VALUES 1, 2 AND THE FOLLOWING STRING.  
Microsoft SQL Server 2008 (SP1) - 10.0.2531.0 (X64)  
Mar 29 2009 10:11:52  
Copyright (c) 1988-2008 Microsoft Corporation  
Express Edition (64-bit) on Windows NT 6.1 <X64> (Build 7601: Service Pack 1)1
```

Figure 2.4 DBMS Function invoking attack [11]

c) Fingerprinting using Inference Attack: Here the attacker submits a SQL statement that is only valid for one DBMS. If the injected statement is correctly executed, the attacker can conclude that he has discovered which database is being used. There are also many other ways of using logically incorrect queries to help cause a SQL injection that will provide confidential information about the underlying database system.

iii) Union query: Union is a reserved keyword used in all database development languages, including SQL, to query the database for the union of two data entities (i.e. it is a relation that includes all the tuples that are either in the first or in the second tuple or in both the tuples, without including the duplicates in the tuples, where the tuple is a set of

data.) For the union attack to work, both the tuples must be compatible (i.e. both must have the same number of tuples and the domain of each attribute in column order should be same in both the tuples). Generally, a union attack is caused by specifying a union of the original query and a second query with the same structure. For example, an original query for usernames could be “unioned” with an additional query for credit card information. This type of query attack requires a good amount of reconnaissance. Figure 2.5 is a snapshot of a terminal window where a union query attack is carried out. The original query was intended to provide price data related to category 1. However, the prices of all entities in categories A and B are displayed as well by using the UNION SELECT statement in the SQL query.

```
USER INPUT.  
1 UNION SELECT 'A', 'B', 3 FROM all_tables  
  
QUERY GENERATED.  
SELECT name, description, price FROM products WHERE category=1 UNION SELECT 'A', 'B', 3  
FROM all_tables  
  
RESULT.  
No error message is returned and data is listed.
```

Figure 2.5 Union attack [11]

In general, union attacks happen in two steps. The first step is to gather information about the database by crafting a valid SELECT statement. The second step is to retrieve any information available, taking care that the injected query follows the structure established during reconnaissance. If the security of the system doesn't allow for error reporting, then causing a Union query based SQL injection would become difficult but still be possible.

iv) Piggy backed query: Piggybacking is a method of collecting statistics by requesting some additional retrievals during the processing of a user query. Piggy backed attacks are used by attackers to extract data, modify data, perform denial of service,

and/or execute remote commands [44], by adding extra statements to an existing statement. The attacker retains the original query but includes new queries that piggy-back on the original query [63]. As a result, the DBMS receives multiple SQL queries. The first query executes normally whereas the queries following the original query carry out the attacker's chosen logic in the database. This kind of attack requires the use of delimiters to join multiple queries together. For example, the delimiter ";" could be used for causing a piggy backed SQLIA. Scanning for SQL piggybacking in a network using an Intrusion Detection System (IDS) is difficult because such delimiters can be used in valid code traveling through the network. This type of attack has a dependency on the database configuration because it cannot be executed if the configuration of the DBMS doesn't allow multiple SQL statements in a single string. Table 5 lists an example of such an attack where the ";drop table users - -" is the piggy backed query.

Table 5 Example of a piggy backed SQL injection attack

Example	' ; drop table users - -
Point of insertion	Users input field
Normal Query	SELECT * FROM users WHERE username= 'strange' AND passwd= '12345'
Modified query	SELECT * FROM users WHERE username= 'strange' AND passwd= '12345'; drop table users _'
Result	Successful execution of dropping of table users.

Alternate encoding: This is an attack that tries to bypass protection systems by representing the user data in a different language that the attacker hopes will be ultimately

decoded by the DBMS. For example, SQL servers currently support six alternate encoding schemes such as char, ASCII, Unicode, nchar, convert and cast. The effectiveness of SQLIA using alternate encoding depends upon the efficiency of the attacker in masking the attack sequence to make it look like a simple SQL query.

Table 6 SQLIA by using alternate encoding

Example	Hex (6e 69 73 68 61 72 67 20 6f 72 20 31 3d 31 2d 2d)
Point of insertion	URL
Normal Query	SELECT * FROM users WHERE username = 'John' and pass= '*****';
Modified query	SELECT * FROM users WHERE username = nisharg or 1=1 _ _ and pass= ';
Result	The user gets data for all the users in the database

Table 6 shows an example of the user's input being provided in URL of a website where the attacker tries to evade detection by encoding the input in hexadecimal. It shows the mapping of the user input to a legitimate query and the SQL injection affected modified query along with the result obtained due to execution of the modified query. In this case, the result displays data for all the users in the database.

In addition to the user input data attacks shown above, indirect attacks are also possible:

Data injection by seeding an indirect trigger: In this attack, the malicious code is the SQL statements injected into persistent storage (such as a table record) [56]. The table record is considered a trusted source, but it could indirectly trigger an attack contents maliciously inserted in the table are used later [53]. To make use of this kind of

vulnerability, it is necessary to submit suitable data in one location, and then use some other application's function to process the data in the attacker's intended way.

A basic scenario for an attacker to cause a second order SQLIA would be to ascertain a web application that stores usernames alongside other session information. He would use a cookie to trace a username from the system as reconnaissance for an attacking parameter to be used later. Then he would use a Union statement and an update patch for updating the users profile. Here the attacker can plant any primary form of query casting bug and then leave the system as bait for vulnerable users to trigger the input function. When another user triggers the attack embedded within the system, the attacker would be informed and provided information from the backend database. For example, the attacker can create a certain function in the definition of the system such that this vulnerability, when used by any user, would cause a SQLIA.

Figure 2.6 shows the SQL injection process for causing an indirect trigger to the seeded data. The attacker is seeding the session id data to update the user profile and after doing so, he uses an 'or' condition to get the social security number for Jane.

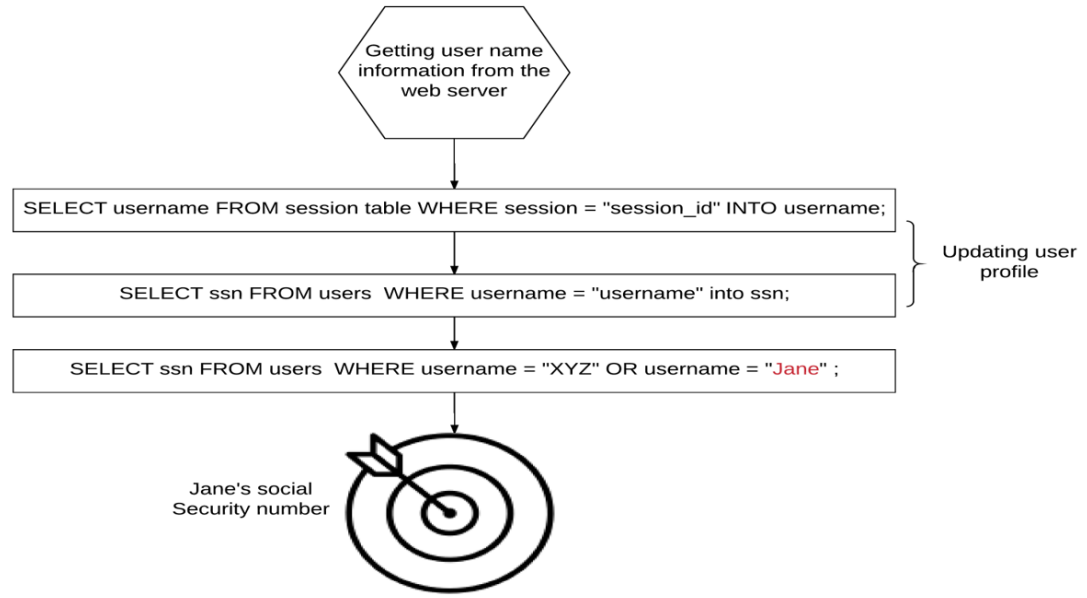


Figure 2.6 A second order SQLIA

Secondary injection attacks are difficult and time consuming to use for exploiting a system because they require significant reconnaissance and must be crafted carefully. However, the outcome of a secondary injection attack, if carried out successfully, is often far more fruitful for the attacker than first order attacks.

CHAPTER 3

LITERATURE SURVEY

Since SQL works across multiple database platforms, and is designed to allow people to access information, it is inherently vulnerable. Due to lack of sufficient resources, and knowledge about attacker's intentions, some companies incorporate security in the source code of web applications. Such implementation of security in the system itself is called defensive coding. One way of achieving protection against SQLIA is by not allowing risky searching variables in search box, but providing users with predefined search inputs for querying database, but this limits the right of users to use application.

We think that data is the primary target for SQLIAs, and protecting data is important. We start by a simple thought process of obfuscating any means of accessing data illegally. To support our thought process, for preventing SQLIAs, [12] & [13] proposes a runtime approach for preventing SQLIA by comparing the cryptographic hash of user input submitted dynamically with hash value stored in the database at the time of account creation. If an attacker tries to insert malicious data in user input field, the hash function won't match. A similar approach of hashing user's credentials is proposed in [14] & [15], but the authors here, go a little bit further and encrypt the data provided by the users, the hash function works on encrypted data. Since hash methods are difficult to reverse, it successfully prevents some form of SQL injection attacks, but the overhead for calculating hash values for inputs is high, and the users must wait for hash process to complete to access the database.

To overcome these limitations of SQLIA prevention methods, authors of [16] came up with a novel approach of SQL instruction set randomization. They inserted random variations in the SQL instruction set. When the user input triggers generation of a SQL statements, these random SQL instructions replace normal query instructions and a query is generated. At the time of execution, these random instructions are derandomized for processing by database engine. Any malicious data, will not get parsed by the system and will make no sense in the query statement, due to randomization. Cryptographic hash functions and randomization process creates a huge overhead on the system. Using encryption to achieve SQLIA prevention, consumes much more time and resources in general because it costs database engine an average of 2.5 time's normal processing time [17]. Due to the above-mentioned limitations, we thought of exploring techniques which prevent SQLIAs by taking previous attack scenarios into consideration and develop secured system for processing SQL.

An alternative approach for preventing SQLIAs using signature or pattern of attack was explored. In [18], the author describes a method in which three special files are created, these files contain definition and rule sets for matching user generated queries for detecting and preventing SQLIAs. The files in this approach consist of definition of all legitimate queries, definition of illegitimate queries and formats and syntax of all possible dynamic queries respectively. When a query is generated using user's input, this query is intercepted by the script and it converts query to an Extensible Markup Language (XML). This XML data is then compared against the special files. If the XML data matches the legitimate query file, then it is send for execution to the database engine, otherwise the file is checked for syntactic match, if it exceeds the threshold limit set by the authors, then

it is considered legitimate query otherwise it is discarded. On similar concepts, authors of [19] & [20] propose a pattern matching algorithm where the service provider creates a database of known attacks and compares the user generated queries with it to generate an anomaly. Rejection of query execution depends upon the match ratio of the user generated query, If the query match totally, its execution is rejected. If the anomaly matches partially, then that user query is sent to administrator for SQLIA inspection. Drawback for this method is that to avoid false positives, partially matched queries must be inspected manually, and anomaly detection library has to be entered manually. Another approach described building static models of legitimate queries using parse trees. These parse trees are stored in memory as tokens of data, when a user generated query is provided to the design, the parse trees are compared to match for structure, if all the tokens match, the query is allowed to execute on the database engine, otherwise it is rejected ([21]- [23]). A similar approach was used in [24] & [29] where the query model is created with grammatical syntax of legitimate queries and attribute values of grammatical construct of a legitimate SQL statement respectively. These authors also accounted for nested query checking by checking for each condition on the stacked query, so that it matched with grammatical syntax of each separately defined query in the statically defined model. They labelled any query which did not abide by the syntactic rules in their library model to be an injection attack. This approach, left a loop hole in security aspect, which was that the attacker could fingerprint the database. ([25]- [27]) overcome the limitations of [24] by using stored procedure mechanism, where strict functional definitions of queries are made, and the users are expected to use just the legitimate functions defined in the library of stored functions to query the database.

Authors in [28], create a model by using lexical analysis of the queries generated. The user generated queries are tokenized based on lexical analysis and total number of tokens in the generated query are calculated and compared with a threshold for stored values of token values. If the token values exceed a threshold, the statement is considered malicious. This method assumes that an attacking query has an extra logic in the SQL statement.

Authors of [29] proposed a mutation based SQL injection vulnerability detection approach to prevent SQLIAs. They named a tool based on mutation based testing as MUSIC, and created a set of 9 rules which they called as mutants. When an application was tested, one mutant was inserted in the SQL query for testing vulnerabilities in the application. These mutants are like test vectors, If the test vector executes successfully, that means mutant has been killed and signifies an open vulnerability in the system. Mutation based testing is a static approach, which is used for testing vulnerabilities in the applications.

Since attackers are becoming smart every day, security measures have to be updated and made smart enough to accommodate all possibilities of SQLIAs. In [30] & [31], authors proposed to use Bayesian algorithm to detect and prevent SQL injection attack. A web monitor intercepts SQL query, inserted by users and breaks it into number of keywords to calculate the length of dynamic SQL query. It also calculates the number of keywords present in that query after doing so, it makes use of machine learning for classifying the obtained data. The classifier, calculates the probability of SQL injection in the query, and then compares the probability of SQL injection calculated, with one defined by user threshold as training data. If probability of user generated SQL query calculated by

classifier matches the probability of legitimate query compute in training dataset, the query is allowed; otherwise it is blocked. The training data set for machine can also accommodate Blind SQLIAs. Based on same concept, authors in [32], create a training dataset by analyzing the source code of the application to calculate entropy of static SQL query. Entropy calculation were made to get better result in comparison with probability based systems such as [30] & [31]. Entropy is calculated for user input data and that is then compared with the entropy of stored query data, a match factor decides whether the query would get executed by the database engine or get rejected. A very good example of machine learning detection for preventing SQLIAs is given in [33], where the authors use Term Frequency–Inverse Document Frequency (TF-IDF) method for calculating weights of generated queries. They use Naïve-Bayes, k-Nearest Neighbor Algorithm and Support Vector Learning (SVM) Algorithm to evaluate attack sequence. Their approach indicates a high precision of about 99.16% detection rate for SQLIAs.

Some authors used tools such as Static Analysis Frame work for detecting SQL injection Vulnerabilities (SAFELI), String constraint solver (SUSHI), Dynamic Candidate Evaluations for Automatic Prevention of SQL Injection attacks (CANDID) in ([30]-[36]). All of the above-mentioned methods were implemented in software and gave good performance, we also want to explore hardware solutions for preventing SQL Injection Attacks. One possible solution which we felt was to create a different memory space for user input data and software defined code i.e. to make a machine based on Harvard architecture and process SQL on it. In [37], authors show that any kind of code injection attack can be prevented by separating code and data memories. This method gives a primary protection from any kind of code injection attacks since the architecture would

not be able to support code as data or data as code, and hence the attacker cannot not inject any data into the instruction memory and at the same time, he cannot execute anything content in the data memory. To scale this concept to real-world example, the authors in the paper [37] used inbuilt TLBs and page tables to achieve this goal of creating a Harvard architecture on an x86 system. The memory accessed by CPU is defined in pages and each page is cloned and then marked for CPU access and Data access where only the code or the instructions gets fetched by the instruction pointer from the page dedicated as the instruction page and similarly only the data processing such as read and write operations happen in the cloned page; thus, defining a boundary between data and code. A huge hurdle in applying this method to SQLIA prevention is that to apply this method to SQL processing, entire SQL must be changed, and the functioning of OS should be changed according to the defined process.

To get enhanced performance and security for securing database against SQLIAs, we will explore an approach in hardware which would be implemented on FPGA. Decision for using FPGA's to implement a solution for preventing SQLIAs was made to take advantage of reusability and extreme parallelism available in a Field Programmable Gate Array. We discuss our approach for preventing SQL injection attacks in chapter 4.

CHAPTER 4

INTRODUCTION TO THE PROPOSED HARDWARE-BASED SQLIA PREVENTION APPROACH

Throughout this thesis, we have been trying to answer questions that would help us understand the scenarios under which SQL injection attacks occur and that would help us create a good hardware-based solution for preventing SQLIAs. In particular, this thesis has been focusing on answering two questions:

Research Question 1: What are some of the currently available options for securing a database against SQLIAs?

Research Question 2: Can we take advantage of hardware solution for securing database applications?

We answered first Research Question in the previous chapter, and are trying to begin answering the second one in this chapter.

4.1 Binding Keywords Used for SQLIAs

To understand the definition of our system, we consider a form for generating a SQL query as shown in Figure 4.1. The code shown in this figure is the backend programming required for generating a web form and getting user input to form a query.

For example, if we provide “alice@bank.com” as the user name and the password as “alice123”, Then the generated query would be

```
SELECT from users WHERE “username” =  
“alice@bank.com” and “password” = “alice123”
```

Next, let's consider that the system is under attack and the attacker replaces the username with "aha!! or 1>0" and the password with "alice OR 1=1". According to [38], now the generated query would look like:

SELECT from users WHERE "username" = "aha!! OR 1>0"

and "password" "alice OR 1=1".

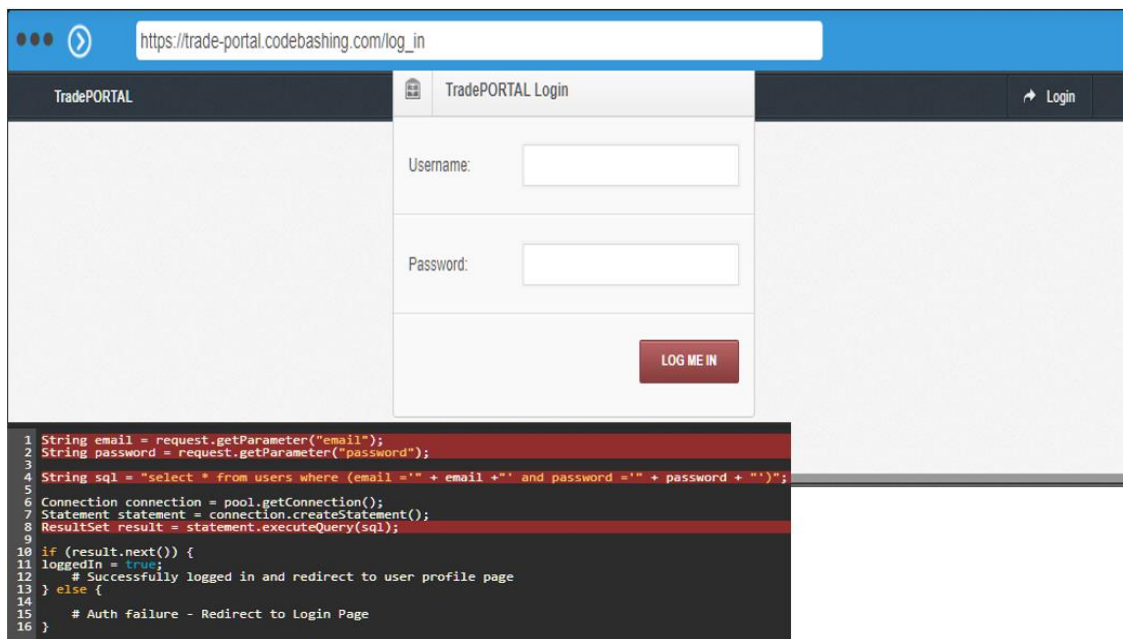
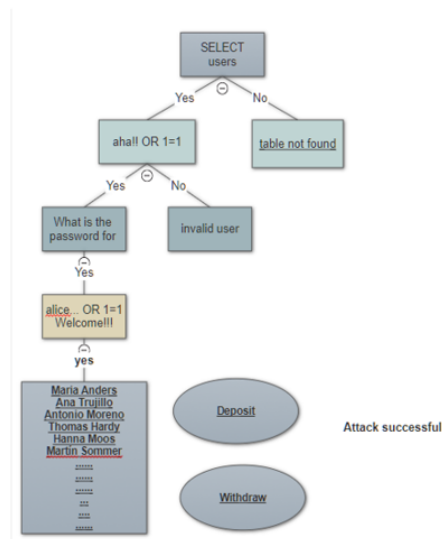


Figure 4.1 The sequence of work behind the web page form [38]

Note: The website [38] may be compromised. Use at your own risk.

A SQL parser would create a parse tree for the generated SQL statements. We have used Lucid chart [57] to create a block diagram representation of the parse tree generated by the SQLIA sequence generated for this query by [38]. The chart is shown in Figure 4.2.

Select from 'Users' WHERE "Username"= "aha!!
or 1=1" and "password" = "alice or 1=1"



Select from 'Users' WHERE "Username"= "alice@bank.com" and "password" = "alice123"

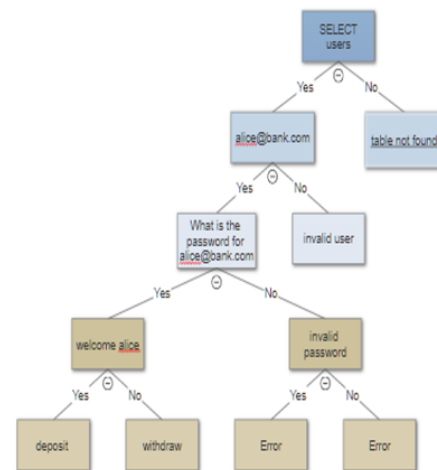


Figure 4.2 Parse trees for malicious vs legitimate queries

Figure 4.2 shows parse tree comparison of a non-legitimate (left side) and a legitimate (right side) query. For the legitimate query, the system checks for the username entered by the user in its database, if it matches any information in the database, it asks for the password for that entity in the database, if both the username and password checks out with a data entry in the database, the user is given access to his bank account. In case of the non-legitimate query, when the user gives "aha !! or 1=1" as input, the SQL parser considers the word "or" as a SQL function 'or' and tokenizes aha!! and 1=1 as two separate inputs for the where clause of username. Note that if the attacker doesn't have any information about a username in the database, and uses aha!!, this would likely not match with any real user in the database, but 1=1 would generate a condition which would match all the entries in the database. Thus, by doing this, the attacker has made a query to all the entries in the database for the table 'username.' In the same way, the attacker passes the authentication phase for password by using input "alice or 1=1" to

cause a tautology attack. The final outcome for this attack is that the attacker gets access to all the users of the bank and can make transactions from any one's account, whose information is stored in table 'username.' Taking this example, we decided to eliminate the keywords that bind harmful data to a legitimate query entered by users, before it is passed to the SQL database engine so that the malicious execution will not occur. We aim to look for substrings that are known to be used to bind the normal query to injection parameters/attack parameters. We will then constrain the entry of the user's input to the SQL engine by blocking the query structures that would allow the input substring to change the syntactic structure of the query. We believe that by eliminating these binders from the user's input before the user data is seen by the database that we can prevent the use of attacking parameters embedded into a SQL query for a SQLIA.

As an initial proof-of-concept of the keyword elimination approach, we carried out SQL injection attacks on freely available SQL coding practice sites and experimented with our concept of eliminating binders from user's input. We observed that since the attacking parameters cannot be embedded into the SQL query (because we find and remove them), the parser does not recognize the following part of the query as a different input. Therefore, it is unable to create different data entities and relate the attacking parameter in relation to a good query.

For example, when an attacker puts "xyz or a=a" as an input, the parser checks authentication for both xyz and a=a when the 'or' is allowed to remain in the statement., if any one of these two inputs is true, the statement gets authorized. However, if we find and remove 'or' from the user's input, then there is no way for joining the tautology factor which in this case is "a=a" to xyz. So, the parser will look for an input "xyz 1=1" in

the database to find a user with this name. As long as no such user exists, the attacker will not obtain access to the database. By removing the binding keywords, the attacker's ability to compromise the system is drastically reduced. Although detecting and eliminating such keywords may be done in software, in this thesis we aim to explore a method of doing so in hardware so that we can take advantage of the inherent speed and parallelism of such an approach. In the next section, we provide a high-level overview of the proposed hardware design.

4.2 Substring Match and Replace

We implement behavioral coding in Verilog for designing a string-matching machine that will be simulated using the Intel Altera Quartus 13.1 web edition. A high-level description of the designed circuit is as follows:

- The string matching machine loads a word of user input into one register, which is to be checked for keywords.
- The user data register is then compared against each keyword to check for a match.
- The result of these comparison is used as a selection criterion for allowing the user's word to remain in the query (i.e. if it is not a prohibited keyword) or for replacing a prohibited keyword that has been found in the user's data with Nulls.

This design, when implemented on an FPGA, will have to be attached to front-end servers, where the data from client end, first enters the server space, as shown in Figure 4.3. By doing so, we restrict the entry of malicious data to the back-end servers and database.

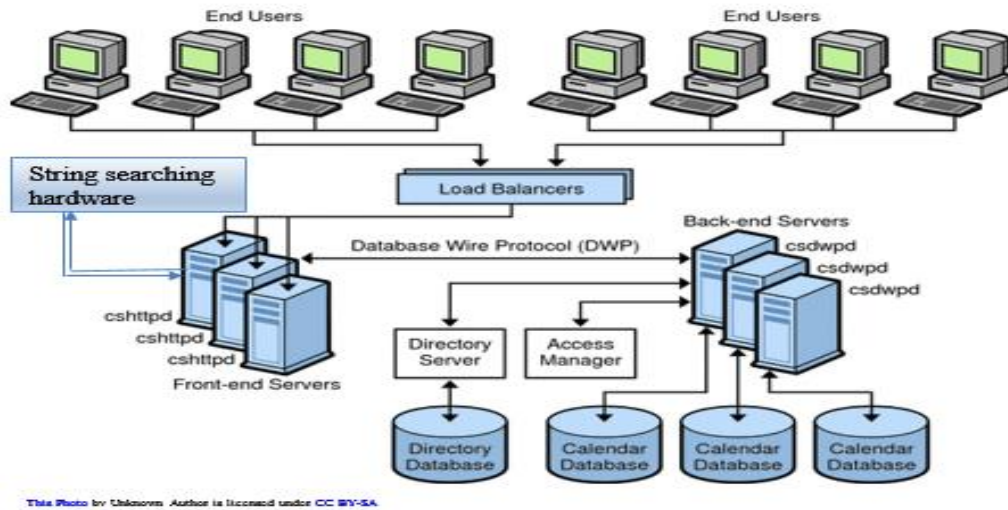


Figure 4.3 System architecture for modified SQL processing system [39] modified by adding the string searching hardware.

Implementation and detailed explanation of circuit design for the above-mentioned approach is shown in chapter 5.

CHAPTER 5

SQL INJECTION ATTACKS DETECTION AND PREVENTION IMPLEMENTATION

5.1 Key word selection process

We practiced SQL injection attacks on code Hackthissite, bWAPP and code bashing.com/sql-demo, which could be accessed online [40], to explore the effect of binding keywords present in attacking queries (Note, these websites may be compromised. Use at your own risk.) In a manner similar, to the example shown in Figure 5.1, each of the potential keywords was tried and tested for to see if it was capable of causing SQLIAs.



The screenshot shows a web interface for testing SQL queries. At the top, it says "SQL Statement:" followed by a text area containing the query: `SELECT * FROM Customers WHERE City='Berlin' or '1=1';`. Below the text area, it says "Edit the SQL Statement, and click 'Run SQL' to see the result." There is a green button labeled "Run SQL >". Below the button, it says "Result:" followed by a table showing the results of the query. The table has 7 columns: CustomerID, CustomerName, ContactName, Address, City, PostalCode, and Country. The table shows 91 records in total, with the first two records displayed.

CustomerID	CustomerName	ContactName	Address	City	PostalCode	Country
1	Alfreds Futterkiste	Maria Anders	Obere Str. 57	Berlin	12209	Germany
2	Ana Trujillo Emparedados y helados	Ana Trujillo	Avda. de la Constitución 2222	México D.F.	05021	Mexico

Figure 5.1 SQL reserved keywords testing [41].

More specifically, the database used in Figure 5.1 for investigating SQL injection attacks had a total of 91 entries for customers, but only 1 entry for a customer belonging to city of 'Berlin'. However, due to the SQL Injection Attack performed by querying "Berlin or

1=1” in a field requesting “city,” we were able to extract data of *all* the customers in the database. By experimenting with many such SQL queries on the above mentioned SQLIA practicing website, we created a set of reserved keywords, that should be sanitized from the user’s input. In Table 7, we describe the keywords that we found to cause SQL Injection Attacks and specify the reason that these keywords create an opportunity for attackers. Specifically, this table was generated by taking the list of 250 keywords found in Reserved Keywords (Transact- SQL) documentation on Microsoft.com [62]. Each keyword was simulated using SQLIA analysis websites, like the one shown above, for its ability to perform an attack when present. (Note that even if such keywords were inserted by a user accidentally instead of maliciously, they could cause serious problems. Thus, removing them from user input is often appropriate even if no attack is intended. Also note that cases where combinations of keywords must be present to perform an attack were not included in this list.)

Table 7 Reserved keywords and function

and	and operator displays a record if all the conditions separated by and is true.
or	or operator displays a record if any of the conditions separated by or is true
between	between operator selects values within a given range.
not	not operator displays a record if the condition(s) is not true
insert	insert statement is used to insert new records in a table.
set	set operations allow the results of multiple queries to be combined into a single result set
delete	delete statement is used to delete existing records in a table.
like	like operator is used in a where clause to search for a specified pattern in a column.
in	in operator allows you to specify multiple values in a where clause,the in operator is a shorthand for multiple or conditions.

join	a join clause is used to combine rows from two or more tables
union	union operator is used to combine the result-set of two or more select statements.
into	the into creates a new table in the default filegroup and inserts the resulting rows from the query into it
--	-- is used to comment the preceding statements in a query
create	create is used to create new instances of data in a database
drop	drop statement is used to drop an existing sql database.
alter	alter statement is used to add, delete, or modify columns in an existing table.
add	add statement is used for entering new enteries into existing table.
;"	;" is used as a statement terminator
all	all operator returns true if all of the subquery values meet the condition.
""	"" is a character delimiter in sql statement
any	any operator returns true if any of the subquery values meet the condition.
exists	exists operator is used to test for the existence of any record in a subquery.
some	compares a scalar value with a single-column set of values
as	it creates copies of the table present in database table.
kill	kill is used to kill a process

5.2 Keyword searching circuit design

To search for the keywords, in the query given by the user, each word is compared with the stored keywords specified in Table 7. For this design, we assume that the user data has been ‘pre-parsed’ so that only individual words are given to the FPGA. Furthermore, we assume that none of the words given to the FPGA would exceed a length of 16 bytes (16 characters) in length. Words that are less than 16 bytes in length are packed with 0s to make a 16-byte word that will be compared to each of the stored keywords. In the experiments performed in this paper, the data fed into the machine is manually formatted to achieve these desired characteristics. Future work will explore whether and how the

parsing and padding with zeroes can be efficiently incorporated into the FPGA design itself.

Figure 5.2 shows a high-level block diagram of the components of the “string search and replace” design. The operations which happen in the string search and replace circuitry, shown in figure 5.2 are as follows:

1. The machine takes in the user input data in increments of one word of 16 Bytes at a time.
2. The comparator block checks the data against 25 keywords for a match. The input is checks each character with the corresponding keyword character simultaneously and compares the input data to all keywords in parallel. An output “match signal” is produced after user data is compared with all the 25 reserved keywords.
3. This match signal is used as a trigger for the replacement circuit, which is a multiplexer or a switch that switches to the secondary input on a match, and provides with a null string to the output of the multiplexer instead of the original “forbidden” keyword if a keyword match was found. This will have the effect of removing the keyword from the user’s input. Otherwise, the original user’s input is simply passed to the output.
4. The final product of this design is filtered data, which is free of any harmful keywords.

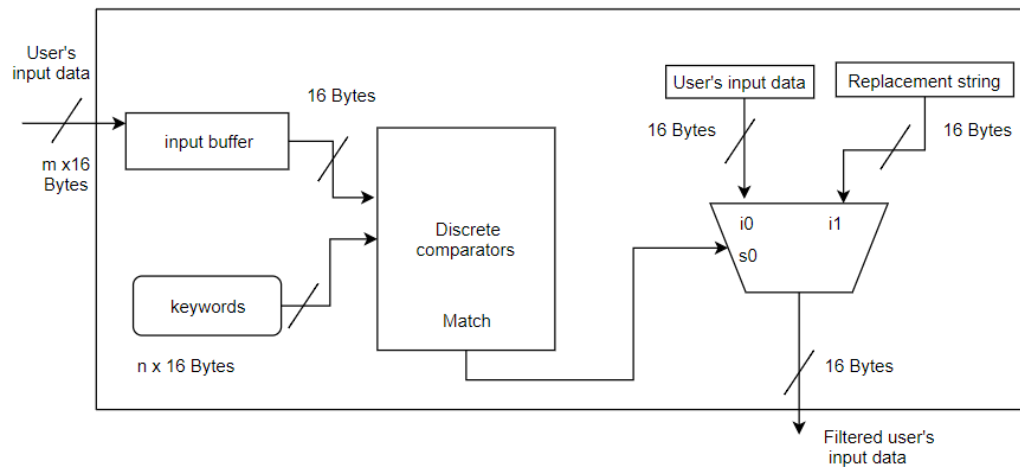


Figure 5.2 Block diagram for SQL string search.

Note that the keywords register is hardcoded with the set of 25 key words, and the user input data inputs get new pre-parsed data from the user input space every clock cycle. Figure 5.3 shows a technology mapping of the string search and replace circuitry created using the Altera Quartus 13.1 Web Edition. A more detailed description of each of the components follows.

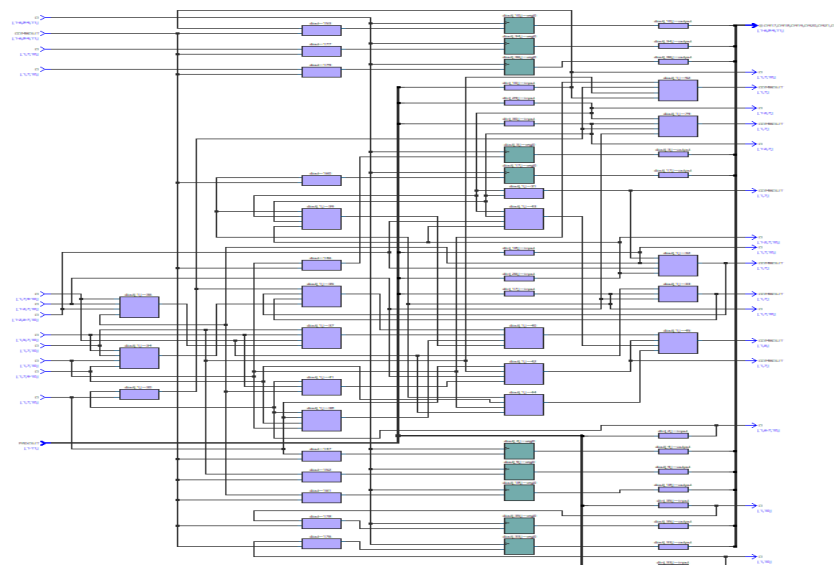


Figure 5.3 Technology mapping for fixed keyword matcher

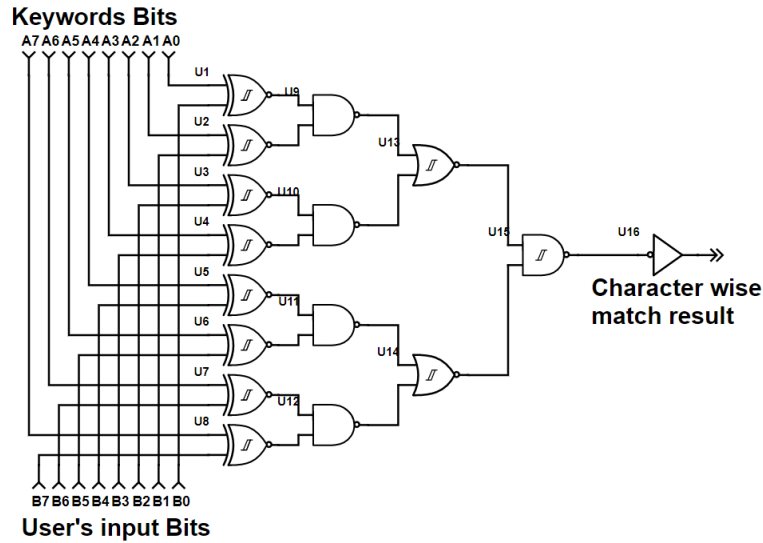


Figure 5.4 Design for one-character comparison [61].

Figure 5.4 shows the design of discrete comparator for comparing one character. In this figure, one character from the user's input and one character from a keyword are compared bitwise, and the resulting output we get is the answer for the match condition of one byte of data. When 16 such comparators give the result for a match of each character, we then 'AND' the results of all the outputs of the 1-character comparators, to get a final answer for comparison of one word of 16 bytes. This is shown in Figure 5.5. To compare all 25 keywords simultaneously, the circuitry in Figure 5.6 is replicated 25 times, and the results of these 25 keyword comparators are OR'ed together to get the final match signal used to replace a problematic keyword with NULL characters.

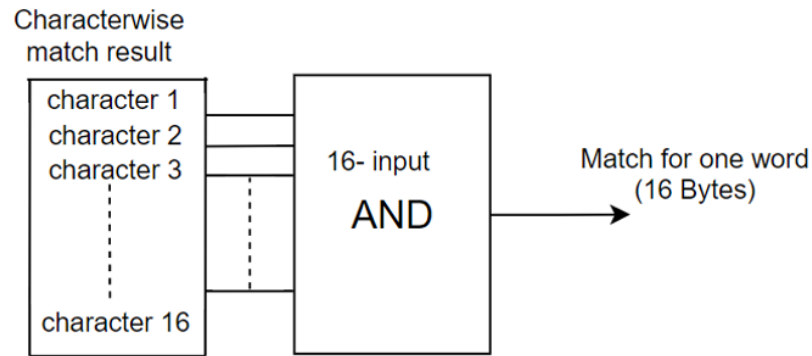


Figure 5.5 Design for one-word comparison

5.3 Implementation evaluation

The Verilog specified design was synthesized using the Altera Quartus II Design Suite for programmable logic. We used synthesizable behavioral coding for designing the string matching circuitry where the keyword comparison is carried out by the circuit.

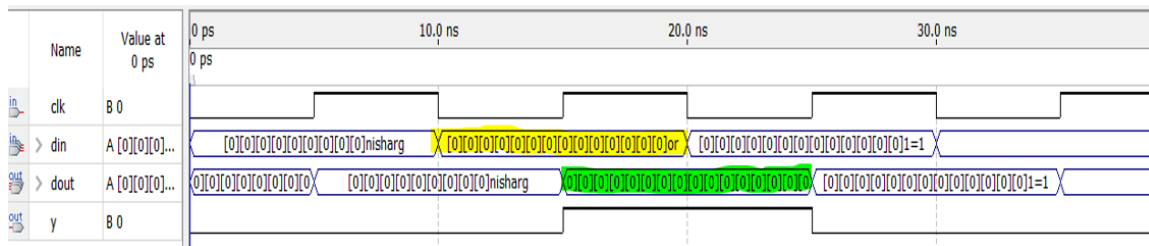


Figure 5.6 Timing simulation on cyclone IV device for keyword matching

Figure 5.6 shows a subset of a set of waveforms for one verification test of the proposed circuitry. The top waveform corresponds to the clock. The second waveform corresponds to the user's input. The third waveform corresponds to the output generated after replacing a reserved keyword with nulls or the original input if no reserved keyword was

found. The fourth waveform corresponds to the match signal, *y*, which is asserted if a match with a reserved keyword has been detected.

When performing the simulation shown in Figure 5.6, the user input given was “nisharg or 1=1”. We assume that this input string was pre-parsed into three distinct words: 1) nisharg, 2) or, and 3) 1=1. In our test, a new “word” is placed on the “din” inputs on the falling edge of each clock cycle. The match signal attains the correct value for that input data and is used to select the correct filtered output. Both *y* and the filtered output are registered outputs and are shown changing on the rising edge of the clock.

As we can see from Figure 5.7, the user input at 15ns contains the keyword ‘or’. This ‘or’ is found by the machine as is indicated by the raised output shown in *y*. Also note that the “or” was replaced by null characters in the corresponding output string shown at *dout*. Thus, the circuit was able to successfully filter out these keywords.

To verify the effects of removing the reserved keywords from user input on a SQL query, we next consider a similar example input on the SQL practice site.

First, assume that the input entered by the attacker for the username is:

users = “Berlin or 1=1”

The generated query would be:

SELECT from users WHERE “users” = “Berlin or 1=1”

and the SQLIA would have been successful, but when the inputs are sanitized by the machine, the input provided by attacker will be filtered and will become

Users = “Berlin 1=1”

and the corresponding query would be

SELECT from users WHERE Users = "Berlin 1=1"

A screenshot from this experiment is shown in Figure 5.7

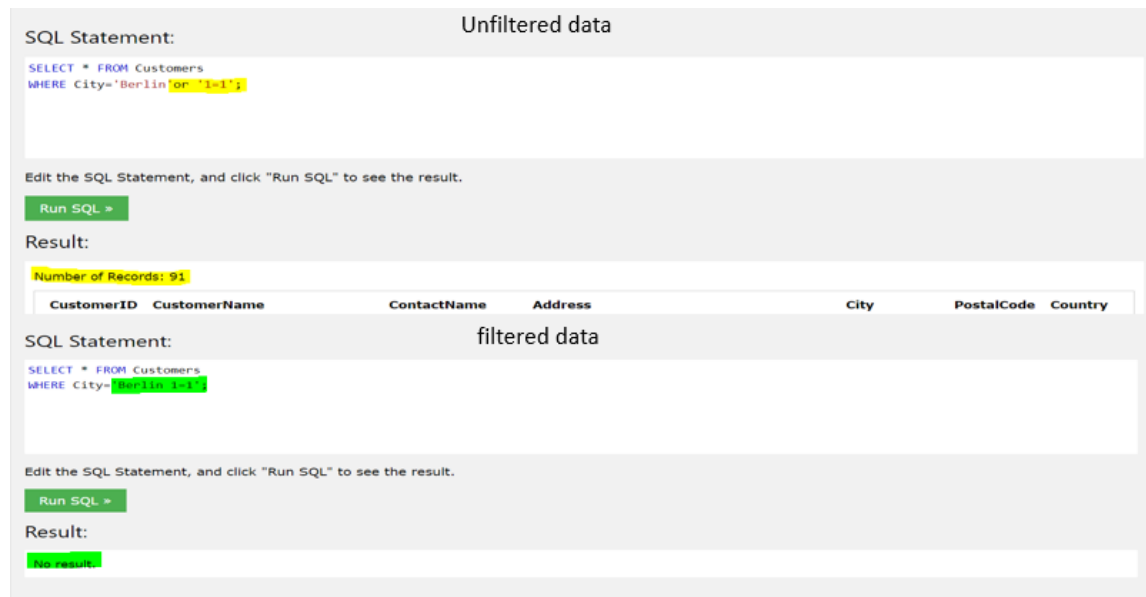


Figure 5.7 Testing the results of string search outputs on SQL practicing website.

Figure 5.7 shows that we have successfully evaded the SQLIA by removing SQL reserved keywords from the user input. The top SQL statement, where the output has not been filtered, provides access to all 91 records. This is highlighted in yellow. In contrast, the same statement with the “or” keyword removed generates “no result” and does not provide access to any record. This is highlighted in green.

The functionality of solution required for SQL injection prevention has been verified with the above example and similar examples, and now we look at the performance aspect of the design. For testing performance, we compare the throughput and the input data accepted by the machine per clock cycle with previously applied methods in hardware.

The performance comparison for our method with other previously applied methods is shown in Table 8. The first entry in the table is of our approach for preventing SQLIAs and the other entries are of previously applied string search circuit designs for intrusion detection systems. Our approach provides a throughput of up to 12.8Gbps at a 100MHz clock frequency. (Note that this throughput is calculated based upon an input of 16 characters, or 128 bits per clock. In many cases, the words being input are less than 16 characters long. Thus, an alternative metric could be 100 million words per second.)

This security system appears to be more efficient than some of the previous methods because the search and replace method applied is simpler and more targeted and does not have an exponentially increasing set of rules to check when evaluating the user inputs. Furthermore, these other approaches evaluate all of the data coming into the network instead of being targeted only for user data whose destination is the SQL server. We do not include the extraction of this user data from the network as part of our overhead.

Table 8 Comparison of search algorithm performance with other methods implemented previously

Description	Input Bits/clock	device	frequency	Throughput
Discrete comparators	128	Cyclone IV	100 MHz	12.8Gbps
Gokhale et al. [23] disc comparator	32	VirtexE-1000	68 MHz	2.1Gbps
Cho et al. [14] disc comparators	32	Altera EP20K	90 MHz	2.8Gbps
Baker et al [27] KMP	8	Vitex 2 pro-4	221 MHz	1.8Gbps
Franklin et al [28]	8	Vitex -1000	31 MHz	0.24Gbps

4.4 SQL injection defense analysis in Software.

To compare the performance of the hardware string search method with a software based approach, I replicated the functionality of the designed hardware in the high level language C, by comparing the keyword pattern and the text, one character at a time using the template provided in [59].

The algorithm implemented in C works as follows:

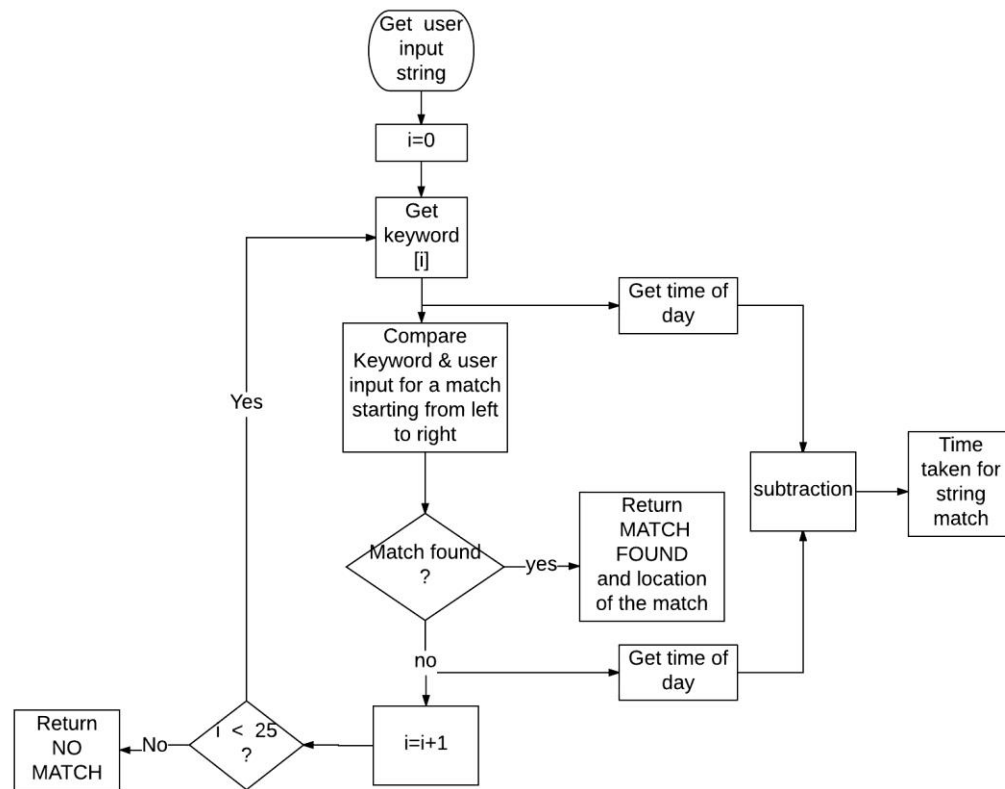


Figure 5.8 Flow chart for software implementation of string searching algorithm

Figure 5.8 shows a flow chart of the string match implementation in software. When a user provides an input, it is compared with the keywords. Because we have 25 keywords

to compare, we iterate the comparison process 25 times. We get the time of the day when matching starts and when the match output is obtained. By subtracting the start time from the end time, we obtain the total time required to search for the keywords in the user input. For example, when 'or 1=1' was used as the data input, the time taken by the computer to process this string and find the leading 'or' was 742.3727 μ s.

To account for the variance in time due to the OS' Priority Scheduling policy on a modern processor, we ran each experiment 10,000 times and averaged the result to obtain a realistic estimate of the expected time. This experiment was performed on an Intel i5-6500HQ running at 2.3GHz. We show the result of our comparison for several different user data entry strings in Figure 5.9. This figure depicts the result of experiment conducted on software and hardware implementation of the string search algorithm. The y-axis is plotted with a logarithmic scale to fit the performance measure of both the software and hardware implementation on one graph.

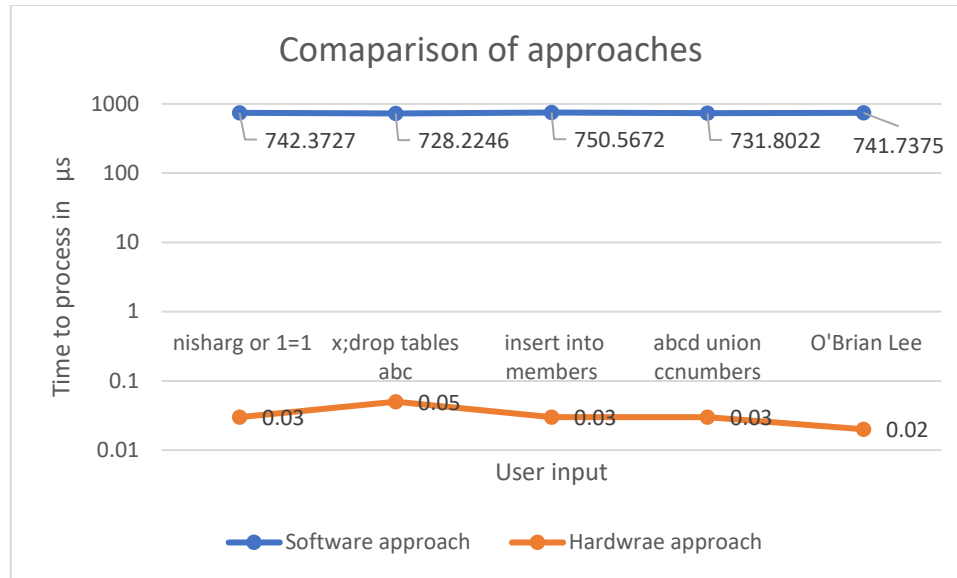


Figure 5.9 Comparison of the designed approach in software and hardware

When the same input strings are applied to both software application and hardware application of the proposed string search method, the time taken by the software approach exceeds that of hardware implementation approximately by a factor of 10^4 . Note that the time allotted to the hardware approach corresponds to one clock cycle per word. The main reason for the performance gain in the hardware approach over software is due to massive parallel searching of input data for a match against the key word rules. As an additional advantage, the resources of the server would be free for utilization elsewhere. (Note that we also implemented C code that used the strcmp function. The results for finding a match with a keyword were generally greater than or equal to the data shown in Figure 5.9.)

The FPGA utilization of the current design is very small. Only approximately 2% of the system resources on the Cyclone IV device were needed. This is encouraging because it indicates that a significant amount of additional functionality may be included in the

FPGA, allowing it to be used for meeting additional security, repair, test, or performance enhancement goals.

4.5 Limitations of SQLIA prevention using the string search and replace method

As seen in this chapter, we implemented a string search and replace machine in hardware for detecting and preventing SQLIAs. Although operation of this method is quite fast, it has some limitations. Most importantly, because we are removing the SQL reserved keywords from the user's input, legitimate users cannot take advantage of the functions of the reserved SQL keywords for querying database. This could potentially be handled through the use of pull-down menus or other data entry options that would allow such keywords only in restricted circumstances. In addition, database inference attacks by using logically incorrect queries are only partially prevented. Because error messages are used for debugging the database, they are not prevented from occurring, but any information gained about the backend database, will be less useful to an attacker because binding factors are sanitized. The current version of the proposed solution also does not implement the parsing of user data into words. Ideally, this portion of the work would be capable of being done quickly in the FPGA as well. Finally, this solution is customized for a specific database system, i.e. MySQL, but it can be applied to other database systems as well by tweaking the keywords for a particular database system.

CHAPTER 6

CONCLUSIONS & FUTURE WORK

5.1 Conclusions

In this thesis, we implemented a hardware design mechanism for SQL injection detection and prevention. We began by investigating the characteristics of SQL, SQL injection, and how they relate to the underlying structure of web-based systems. We also discussed SQL processing and identified one of the most vulnerable parts of SQL processing to be the “parsing process” where maliciously inserted keywords may be used to change user data input from standalone data to executable code by binding malicious queries to legitimate queries. This technique can be harnessed to provide unauthorized access and control to a database. Thus, we identified the removal of such keywords from the user input stream as a possible means of preventing such attacks.

Although checking of the user data for appropriate form and content has previously been proposed and implemented in software, such approaches have proven to be inadequate. Human error and sometimes a lack of safe coding knowledge still leave databases susceptible to SQL injection attacks.

Thus, we designed a solution for preventing SQLIA without depending on the good coding practices of the database software engineer. Instead, we propose implementing the search and replace functionality in hardware on an FPGA. The proposed solution is capable of performing comparisons of multiple binding keywords to the user’s input data in parallel, significantly increasing performance. Furthermore, the FPGA-based filter should be placed in the system such that user data cannot even reach the servers

performing the processing until it has been checked and sanitized. The FPGA resources required for mapping this solution to hardware included 271 out of 109424 Logical Elements on a Cyclone IV GX: EP4CGX110DF3117 device. Our evaluation showed that when we clock the hardware to a speed of 100 MHz, the machine can evaluate and filter up to 12.8Gb of data per second with a single instance of the design.

We also verified that the proposed removal of chosen keywords and replacement of those keywords with spaces was able to defeat traditional SQL injection attacks. Unfortunately, while there are multiple advantages to implementing a solution in hardware, such an implementation also comes with increased costs in the system and the need to change some of the underlying structure of the system itself.

5.2 Future Work

Future work will investigate the use of alternative designs and implementations for preventing SQLIAs without depending on the software designer to maintain good coding practices. One could also implement various software searching algorithms in hardware to see if there is a certain algorithm that gives a better performance and throughput in comparison with the current implementation. One could also explore applying intelligent context language semantics and machine learning techniques, to make the thwarting process smart, so that the design knows or can predict the context in which the data is used.

Other future work will explore adding functionality to the current hardware design. For example, the current design depends on the input data to be presented to the FPGA in a

word-by-word fashion. In the future, one could automate this process by designing a circuit, to match for end-of-statement character in the user memory, to check for different user inputs, and tokenize them in words.

Other future work will further investigate how the FPGA should be inserted efficiently into a database system and how that database system will most efficiently communicate with the FPGA. Finally, other software based coding approaches are possible, and we will compare our method with other software coding algorithms. Those that are particularly efficient may become even more efficient when implemented well in hardware.

Appendix A

CONCEPT OF MEMORY OPERATION FOR STRING MATCHING

```
always @(posedge clk)

begin

for (innerloop_count=0;
innerloop_count<key_depth;innerloop_count=innerloop_count+1)

begin

for
(outerloop_count=0;outerloop_count<memory_depth;outerloop_count=outerloop_count+
1)

begin

addr = (innerloop_count*memory_depth)+outerloop_count;

match[addr] = memory_0[innerloop_count]== memory_1[outerloop_count];// match is
given a result of matching both the memory locations exhaustively

if(memory_0[innerloop_count]== memory_1[outerloop_count])

begin

newmem[addr] = 128'b0;

end

else

begin

newmem[addr] = memory_1[outerloop_count];

end

//memory_2[outerloop_count] = memory_2[outerloop_count] &
newmem[(outerloop_count*memory_depth)+innerloop_count];
```


Figure A.1.1 snippet of formatted input data to the comparison machine

[illegible]

Figure A.1.2 Example data of keywords file

[illegible]

Figure A1.2 Example data of filtered output

Appendix B

Software Implementation

Code for implementing software String searching in C

```
#include <sys/time.h>
#include <time.h>
#include <stdio.h>
#include <string.h>

void search (char* pattern, char* text)
{
    int M = strlen(pattern);
    int N = strlen(text);

    for (int i = 0; i <= N - M; i++) {
        int j;

        for (j = 0; j < M; j++)
            if (text[i + j] != pattern[j])
                break;

        if (j == M)// if pattern[0...M-1] = text[i, i+1, ...i+M-1]
        {
            printf("Keyword \"%s\" found at index %d\n\n",pat, i);
        }
    }
}

int main()
{
    struct timeval start,end;
    int i;

    //printf("input the username/password:");
    //scanf("%s",&txt);
    char text[] = "O'Brian";
    int pattern_length = 25;
    double sum=0;
    double temp=0;
```

```

char pattern[25][10] =
{"and","or","between","not","insert","set","delete","like","in","join","union","into","--",
"create","drop","alter","add",";","all","","any","exists","some","as","kill"};

int count=1000;

for(int j=0;j<count;)
{
    gettimeofday(&start,NULL);

    for(int i=0;i<pat_length;i++)
    {
        search(pat[i], txt);
    }
    gettimeofday(&end,NULL);

    temp = (end.tv_sec * 1000000 + end.tv_usec) - (start.tv_sec * 1000000 +
start.tv_usec);
    sum+=temp;
    if(temp!=0)
    {
        j++;
        printf("Iteration: %d, Time: %f\n",j,temp);
    }
}
sum=sum/count;
printf("Average: %f",sum);

return 0;

```

REFERENCES

- [1] Sheikh Zaki Yamani, <http://www.economist.com/node/2155717>
- [2] “HACKING&TRICKS” a Blog about Hacking & Computer Security by Nirav Desai
<https://tipstrickshack.blogspot.com/2013/01/list-of-vulnerability-its-tutorial.html>
- [3] https://en.wikipedia.org/wiki/Relational_algebra
- [4] SENG 130 lecture-2 on web server in seng130.wordpress.com/lectures-2/web-servers/
a blog for WWW and Mobile Applications.
- [5] Rain forest Puppy “NT Web Technology Vulnerabilities” in Phrack Magazine
Volume 8, Issue 54 Dec 25th, 1998
- [6] xkcd a Web Comic of Romance, Sarcasm, Math and language.
https://imgs.xkcd.com/comics/exploits_of_a_mom.png
- [7] “SQLI HALL-OF-SHAME” by the Code Curmudgeon in a blog on Ranting about
Software, Security and Tech.
- [8] Stephen Kost (Integrity Corporation) in white paper “AN INTRODUCTION TO SQL
INJECTION ATTACKS FOR ORACLE DEVELOPERS” March 2007.
- [9] Oracle Database Tuning Guide.
https://docs.oracle.com/cd/E11882_01/server.112/e41573/toc.htm
- [10] Amy Schade in “Decision Making in the Ecommerce Shopping Cart: 4 Tips for
Supporting Users” article <https://www.nngroup.com/articles/shopping-cart/>
- [11] Justin Clarke “SQL INJECTION ATTACKS AND DEFENCE Second Edition”
2012.

- [12] Mihir Gandhi, JwalantBaria 2013. SQL INJECTION Attacks in Web Application International Journal of Soft Computing and Engineering (IJSCE) ISSN: 2231-2307, Volume-2, Issue-6.
- [13] Raghav Kukreja , Nitin Garg 2014. OVERVIEW OF SQL INJECTION ATTACK international journal of innovative research in technology Volume 1 Issue 5.
- [14] Ms. Mira K. Sadar et al 2014. Securing Web Application against SQL Injection Attack: a Review International Journal on Recent and Innovation Trends in Computing and Communication ISSN: 2321- 8169 Volume: 2 Issue: 3
- [15] Neha Mishra, Sunita Gond 2013. Defenses to Protect Against SQL Injection Attacks International Journal of Advanced Research in Computer and Communication Engineering Vol. 2, Issue 10
- [16] Boyd, S.W. & Keromytis, A.D. (2004) SQLrand: Preventing DQL injection attacks. In *Applied Cryptography and Network Security* Springer Berlin Heidelberg.
- [17] Muhammad Saidu Aliero, Abdulhamid Aliyu Ardo, Imran Ghani, Mustapha Atiku In IOSR Journal of Engineering (IOSRJEN) ISSN (e): 2250-3021, ISSN (p):2278-8719 Vol. 06, Issue 02 (February 2016).
- [18] D. Das, U. Sharma, D.K. Bhattacharyya "An approach to detection of SQL injection attack based on dynamic query matching" *International Journal of Computer Applications* February 25, 2010.
- [19] Prabakar, M. Amutha, M. Karthikeyan, and K. Marimuthu 2013. "An efficient technique for preventing SQL injection attack using pattern matching algorithm."

Emerging Trends in Computing, Communication and Nanotechnology (ICE-CCN), 2013
International Conference on. IEEE

[20] Shi, Cong-cong, et al. (2012) "A New Approach for SQL-Injection
Detection." *Instrumentation, Measurement, Circuits and Systems*. Springer Berlin
Heidelberg, 245-254.

[21] Halfond, W.G., Orsr,A. (2006) Preventing SQL injection attacks using AMNESIA.
In *International Conference on Software Engineering*.

[22] Buehrer, G., Weide, B. W., and Sivilotti, P. A. (2005). Using parse tree validation to
prevent SQL injection attacks. In *Proceedings of the 5th international Workshop on
Software Engineering and Middleware* (Lisbon, Portugal, September 05 - 06, 2005). SEM
'05. ACM, New York, Pages 106-113.

[23] Su, Z. and Wassermann, G. (2006) The essence of command injection attacks in web
applications SIGPLAN Not. 41, 1 (Jan. 2006), 372-382.

[24] Narayanan, Sandeep Nair, AlwynRoshanPais, and Radhesh Mohandas
2011. "Detection and Prevention of SQL Injection Attacks Using Semantic Equivalence."
Computer Networks and Intelligent Computing. Springer Berlin Heidelberg, 2011. 103-
112

[25] Neha Mishra, Sunita Gond 2013. Defenses to Protect Against SQL Injection Attacks
*International Journal of Advanced Research in Computer and Communication
Engineering* Vol. 2, Issue 10

[26] Boyd, S.W. & Keromytis, A.D. (2004) SQLrand: Preventing DQL injection attacks.
In *Applied Cryptography and Network Security* Springer Berlin Heidelberg.

- [27] Muhammad Saidu Aliero, Abdulhamid Aliyu Ardo, Imran Ghani, Mustapha Atiku In IOSR Journal of Engineering (IOSRJEN) ISSN (e): 2250-3021, ISSN (p):2278-8719 Vol. 06, Issue 02 (February 2016).
- [28] Bangre, Shruti, and Alka Jaiswal (2012) "SQL Injection Detection and Prevention Using Input Filter Technique." International Journal of Recent Technology and Engineering (IJRTE)145-149.
- [29] H. Shahriar, M. Zulkernine "Music: Mutation-based sql injection vulnerability checking" In *the Eighth International Conference on Quality Software* IEEE Computer, 2008.
- [30] Cheon, Eun Hong, Zhongyue Huang, and Yon Sik Lee (2013). "Preventing SQL Injection Attack Based on Machine Learning." International Journal of Advancements in Computing Technology 5.9
- [31] Joshi, Anamika, and V. Geetha. (2014) "SQL Injection detection using machine learning." Control, Instrumentation, Communication and Computational Technologies (ICCICCT), International Conference on. IEEE, 2014.
- [32] Shahriar, Hossain, and Mohammad Zulkernine. "Information-theoretic detection of sql injection attacks." High-Assurance Systems Engineering (HASE), 2012 IEEE 14th International Symposium on. IEEE, 2012.
- [33] Ryohei Komiya, Incheon Paik, Masayuki Hisada "Classification of Malicious Code by Machine Learning" published in Awareness Science and Technology (iCAST), 2011 3rd International Conference on 27-30 Sept. 2011 Print ISSN: 2325-5986
Electronic ISSN: 2325-5994.

- [34] Xiang Fu, Kai Qian “SAFELI – SQL Injection Scanner Using Symbolic Execution” *Workshop on Testing, Analysis and Verification of Web Software*. July 21, 2008.
- [35] X. Fu and C.-C. Li, “A String Constraint Solver for Detecting Web Application Vulnerability,” Proc. 22nd Int’l Conf. Software Eng. and Knowledge Eng. (SEKE 10), Knowledge Systems Institute Graduate School, 2010.
- [36] P. Bisht, P. Madhusudan, and V.N. Venkatakrishnan, “CANDID: Dynamic Candidate Evaluations for Automatic Prevention of SQL Injection Attacks,” ACM Trans. Information and System Security, Feb. 2010; www.cs.illinois.edu/~madhu/tissec09
- [37] Ryan Riley, Xuxian Jiang and Dongyan Xu in “An Architectural Approach to Preventing Code Injection Attacks” publishes in IEEE Transactions on Dependable and Secure Computing (Volume: 7, Issue: 4, Oct-Dec. 2010) Pages 351-365 Print ISSN: 1545-5971.
- [38] https://www.codebashing.com/sql_demo
- [39] Sun Java Communications Suite 5 Deployment Planning Guide by ORACLE <https://docs.oracle.com/cd/E19566-01/820-0086/index.html>
- [40] <https://www.checkmarx.com/2015/04/16/15-vulnerable-sites-to-legally-practice-your-hacking-skills/>
- [41] SQL practicing website <https://www.w3schools.com/>
- [42] DANIEL CID in his blog titled as Google Bots Doing SQL Injection Attacks on SUCURI Blog found at <https://blog.sucuri.net/2013/11/google-bots-doing-sql-injection-attacks.html>
- [43] 2016 Vulnerability Statistics Report by edgescan <https://www.edgescan.com/assets/docs/reports/2016-edgescan-stats-report.pdf>

[44] Pankajdeep Kaur, Kanwal Preet Kour “SQL Injection: Study and Augmentation” in *International Conference on Signal Processing, Computing and Control* 2015.(ISPPCC’2015).

[45] WEB SECURITY READINGS blog at Netsparker Ltd.

<https://www.netsparker.com/blog/web-security/cross-site-scripting-xss/>

[46] Bernard Marr in blog titled as Ten top languages for crunching Big Data on Data Science Central. The online resource for Big Data Practitioners.

<https://www.datasciencecentral.com/profiles/blogs/ten-top-languages-for-crunching-big-data>

[47] Dinesh Thakur in his website named ECOMPUTER NOTES in chapter titled as “What are Relational Algebra and Relational Calculus”.

<http://ecomputernotes.com/database-system/rdbms/relational-algebra-and-relational-calculus>

[48] Multitier architecture found at https://en.wikipedia.org/wiki/Multitier_architecture.

[49] SQL injection (SQLi) Web Security website by acunetix

<https://www.acunetix.com/websitesecurity/sql-injection/>

[50] Metaprogramming in SQL by Keith McDonald published on June 24th, 2010

<http://www.thekguy.com/metaprogramming-in-sql-part-1.html>

[51] R. B. Buitendijk “Logical errors in database SQL retrieval queries” in *Computer Science in Economics and Management*. January 1988, Volume 1 Issue 2, pp 79-96.

[52] SQL injection (second order) on PORTSWIGGERWEB SECURITY website.

https://portswigger.net/kb/issues/00100210_sqlinjectionsecondorder

[53] CLEANFLEET REPORT <http://www.cleanfleetreport.com/>

[54] Top 8 website vulnerabilities a hacker can exploit a Blog on TemplateToaster let's enjoy Web Designing Blog. Published on February 15, 2016

<https://blog.templatetoaster.com/>

[55] Bharti Nagpal, Naresh Chauhan, Nanhay Singh "A Survey on the Detection of SQL Injection Attacks and Their Countermeasures" J Inf Process Syst, Vol.13, No 4, pp. 689-702, August 2017. ISSN 1976-913 (Print) ISSN 2092-805(Electronic).

[56] SQL Injection description at <http://hwang.cisdept.cpp.edu/swanew/Text/SQL-Injection.htm>

[57] Flow chart maker called Lucid Chart <https://www.lucidchart.com/>

[58] Memory descriptions for Verilog coding found at

<http://www.verilog.renerta.com/source/vrg00024.htm>

[59] <http://www.geeksforgeeks.org/searching-for-patterns-set-1-naive-pattern-searching/>

[60] Query Optimization on GeeksforGeeks A computer science portal for geeks.

<http://www.geeksforgeeks.org/query-optimization/>

[61] SchemeIt a Free Online Schematic Drawing Tool, by Digi key corporation USA

<https://www.digikey.com/schemeit/project/>

[62] Reserved Keywords (Transact-SQL) by Rick Bayham, Alma Jenks, Bruce Hamilton, Solomon Rutzky and Craig Guyer on March 14th 2017

<https://docs.microsoft.com/en-us/sql/t-sql/language-elements/reserved-keywords-transact-sql>

[63] Secure Web Development Course by Dr.Drew Huang for Computer Information Sysytems, Calpoly Ponomo. 2013

<http://hwang.cisdept.cpp.edu/swanew/Code.aspx?m=SQL-Injection>

