

# Detection and Prevention of SQL Injection Attack in Web Application

*A Thesis submitted in partial fulfillment of the requirements for the degree*

*of*

**Master of Technology**

*in*

**Computer Science & Information Security**

*by*

**Avinash Kumar Singh**

**(Roll No: 955005)**

*under the guidance of*

**Prof. Satarupa Mohanty**

*School of Computer Engineering*



**School of Computer Engineering**

**KIIT University**

**Bhubaneswar-751024, Odisha, India**

**2011**

## CERTIFICATE

This is to certify that the thesis entitled ***Detection and Prevention Of SQL Injection Attack in Web Application***, submitted by **Avinash Kumar Singh**, bearing roll no. **955005** and registration no. **09265813359**, in partial fulfillment of the requirement for the award of the degree of **Master of Technology in Computer Science & Information Security** of KIIT University, Bhubaneswar, Odisha, India, is a bonafide record of research work carried out by him under my/our guidance. Neither this thesis nor any part of it has been submitted for any degree or academic award elsewhere.

(Satarupa Mohanty)

Asst. Professor

School Of Computer Engineering

KIIT University, Bhubaneswar

(Dr. Madhabananda Das)

Dean

School Of Computer Engineering

KIIT University, Bhubaneswar

External Examiner

## Abstract

When Internet was developed, the founding fathers of Internet hardly had any inclination that Internet could transform itself into an all pervading revolution which could be misused. Due to the anonymous nature of the Internet, it is possible to perform various attacks against web applications. SQL Injection is the most threatening attack against web applications, and it covers up to 20% of all cyber attacks, which is maximum among all and a big cause to worry. SQL injection is an attack method used by hackers to retrieve, manipulate, fabricate or delete information in organizations relational databases through web applications. Information in databases usually constitutes an organizations most valuable asset, and attacks on it could threaten the organizations integrity, availability and confidentiality. SQL injection techniques are simple and require no special tools or expertise from the attacker, except for some basic database and server-script language knowledge. Despite being remarkably simple to protect against SQLI, there is an astonishing number of production systems connected to the Internet that are vulnerable to this type of attack.

In this thesis we have proposed two different mechanisms to prevent and detect SQL Injection in an web application. For prevention we have designed *SQL Meta Character Filter* which we have placed in between the user and the application server to intercept all the request coming from the user. The Meta filter parses the input URL to detect attack patterns. If any pattern matches with the stored signature then it redirects the request to the error page. For the detection purpose we have designed *A Network Based Vulnerability Scanner* which scans the whole web application and generates an automatic report, describes which pages are vulnerable and which are not. So developer can fix these problems before launching it on the web. Both of these approaches have been implemented successfully and are fully able to fix SQLI vulnerabilities.

## Dedication

*This Thesis is dedicated to my beloved Grandfather(nanaji)*

## **Acknowledgement**

On the submission of my M.Tech Thesis, I would like to express my deep and sincere gratitude to my supervisor, Prof. Satarupa Mohanty, School Of Computer Engineering, KIIT University, Bhubaneswar for their excellent guidance, invaluable suggestions, and a great support at all the stages of my research work.

My special thanks to Sangita Roy, PhD Scholar, IIT Patna, for her constant inspiration and motivation. Her motivation and confidence in me was the reason for all the success I have made. Her wide knowledge and her logical way of thinking have been of great value for me.

I owe my most sincere gratitude to all the respected teachers of computer science department for the invaluable knowledge they imparted to me and for teaching the principles in an exciting and enjoyable way.

I am grateful to my friend Atul Kumar Uttam and, K.V.V.S.R Chowdary for their assistance, criticisms and useful insights.

I warmly thank Mr. Mritunjay Kumar Ranjan and Mukund Kumar for their valuable advice and friendly help. Their extensive discussions around my work and interesting explorations in coding part have been very helpful for this thesis.

I express my deep sense of reverence and gratitude to my parents specially to my grandfather who creates everything for my life.

**Avinash Kumar Singh**  
**M.Tech(CS&IS)**

# Contents

<b>1</b>	<b>Introduction</b>	<b>1</b>
1.1	Background . . . . .	1
1.1.1	Corporations and Web Applications . . . . .	1
1.1.2	Web Applications and Security . . . . .	2
1.1.3	Web Applications and Vulnerabilities . . . . .	3
1.2	SQL Injection . . . . .	5
1.3	Problem Definition . . . . .	6
1.4	Motivation . . . . .	7
1.5	Organisation of this Thesis . . . . .	8
<b>2</b>	<b>Web Application</b>	<b>9</b>
2.1	Domain . . . . .	9
2.1.1	Web Service . . . . .	10
2.1.2	Business Web Applications . . . . .	10
2.2	Architecture . . . . .	11
2.2.1	Client-Server . . . . .	11
2.2.2	The Client-Server Architecture and Layers . . . . .	12
2.2.3	The Client-Server Architecture and Tiers . . . . .	13
2.3	Software Components . . . . .	14
2.3.1	Components Type . . . . .	15
2.3.2	A General Business Web Application Model . . . . .	15

2.4	Communication . . . . .	18
2.4.1	Information . . . . .	18
2.4.2	Content . . . . .	19
2.4.3	Protocol . . . . .	19
2.4.4	URL Encoding . . . . .	20
<b>3</b>	<b>SQL Injection</b>	<b>22</b>
3.1	Introduction . . . . .	22
3.1.1	Definition . . . . .	22
3.1.2	SQLI Example . . . . .	23
3.1.3	Injection Mechanism . . . . .	25
3.1.4	Attack Procedure . . . . .	27
3.1.5	Attack Intent . . . . .	29
3.2	Types Of SQLI . . . . .	30
3.3	Vulnerabilities . . . . .	39
3.4	Security Layers and SQLI . . . . .	42
<b>4</b>	<b>Proposed Techniques</b>	<b>45</b>
4.1	A Filter Approach to Prevent SQLIA . . . . .	45
4.1.1	Observation . . . . .	45
4.1.2	Related Work . . . . .	46
4.1.3	Proposed Filter Approach . . . . .	51
4.1.4	Implementation Details and Evaluation . . . . .	57
4.2	A Network Based Vulnerability Scanner to Detect SQLIA . . . . .	59
4.2.1	Observation . . . . .	59
4.2.2	Related Work . . . . .	60
4.2.3	Proposed Network Model . . . . .	61
4.2.4	Implementation Details . . . . .	64
4.2.5	Results and Comparison . . . . .	68

<b>5 Conclusion and Future Work</b>	<b>72</b>
<b>References</b>	<b>73</b>
<b>Glossary</b>	<b>78</b>



# List of Tables

4.1	Attack analysis . . . . .	52
4.2	Vulnerability details of different Web Application . . . . .	68
4.3	Results comparison . . . . .	68

# List of Figures

2.1	Components within layers. . . . .	15
3.1	Admin Login . . . . .	24
3.2	Security layers in Web Applications . . . . .	43
4.1	Communication between Filter and Application Server. . . . .	52
4.2	A simple 3-Tier Web Application Architecture with Filter. . . . .	54
4.3	Working of Filter. . . . .	55
4.4	Web Application showing records from the Database. . . . .	57
4.5	An SQLI Attack. . . . .	58
4.6	Error Page serve by the Filter. . . . .	59
4.7	Network Based Vulnerability Scanner . . . . .	62
4.8	Tree structure of a Web Application . . . . .	63
4.9	Vulnerability Detection Graph . . . . .	69
4.10	Vulnerability covered by each tool . . . . .	70
4.11	Time taken by each tool . . . . .	71

# List of Abbreviation & Nomenclature

HTTP	Hyper Text Transfer Protocol
OWASP	Open Web Application Security Project
SQL	Structured Query Language
LDAP	Leightweight Directory Access Protocol
XSS	Cross-Site Scripting
SQLIA	Structured Query Language Injection Attack
RDBMS	Relational Database Management System
VS	Vulnerability Scanner
RMI	Remote Method Invocation
HTML	Hyper Text Markup Language
IDS	Intrusion Detection System
XPATHI	XML PATH Injcetion

# Chapter 1

## Introduction

### 1.1 Background

#### 1.1.1 Corporations and Web Applications

The evolution of the Internet has laid a foundation for the development and usage of new categories of information technology systems operating on the web, these systems are often referred to as web applications and range in complexity from simple implementations, e.g. personal web pages, to advanced business applications: informational web sites, intranets, extranets, e-commerce and business-to-business systems. Web applications provide connectivity, access to information and online services, introducing new opportunities for corporations to realize their ambitions. Therefore, corporations were not late to exploit these possibilities and, in a steadily increasing number, they have begun to take advantage of web applications by connecting their systems to the web.[7][36]

Web applications have introduced a new way of business, where we have the facility to book our railway as well as flight ticket, we can buy any product as well as we can submit our phone bills to, just on a single click without visiting to the concerned office. It saves lots of our time and effort too. Every organization is mapping their

business from a room to the world level with the help of these web applications. Corporations have constantly striven to enhance their communication capabilities, allowing more efficient information exchange within their own organizations as well as between partners in their value chains, i.e. suppliers, distributors and customers. In addition, corporations have sought new alternatives to gain more competitive advantages: managing customer relationships, doing business, creating alliances, moving into new markets, and promoting their products and services. Web applications provide connectivity, access to information and online services, introducing new opportunities for corporations to realize their ambitions.

Web applications are integrated with corporations networking infrastructure and typically encompass the use of commercial components, e.g. web servers and application servers. In addition web applications consist of a back-end database with web pages that contain server-side script written in a programming language that is capable of extracting specific information from a database depending on various dynamic interactions with the user. One of the most common applications for a web application is an e-commerce application, where a variety of information is stored in a database, such as product information, stock levels, prices, postage and packing costs, and so on.

### **1.1.2 Web Applications and Security**

Advances in web technologies coupled with a changing business environment, mean that web applications are becoming more prevalent in corporate, public and Government services today. Although web applications can provide convenience and efficiency, there are also a number of new security threats, which could potentially pose significant risks to an organization's information technology infrastructure if not handled properly [25]. The rapid growth in web application deployment has created more complex, distributed IT infrastructures that are harder to secure. For more than a decade, organizations have been dependent upon security measures at the perimeter

of the network, such as firewalls, in order to protect IT infrastructures. However, now that more and more attacks are targeting security flaws in the design of web applications, such as injection flaws, traditional network security protection may not be sufficient to safeguard applications from such threats. These threats originate from non-trusted client access points, session-less protocols, the general complexity of web technologies, and network-layer insecurity. [7][31][36]

With web applications, client software usually cannot always be controlled by the application owner. Therefore, input from a client running the software cannot be completely trusted and processed directly. An attacker can forge an identity to look like a legitimate client, duplicate a user's identity, or create fraudulent messages and cookies. In addition, HTTP is a session-less protocol, and is therefore susceptible to replay and injection attacks. Hypertext Transport Protocol messages can easily be modified, spoofed and sniffed. As such, organizations must understand and be fully aware of the threats to properly implement appropriate defensive strategies. Additional security controls, both technical and administrative, may be required to reinforce the protection of vital infrastructure in response to the deployment of web applications.

### 1.1.3 Web Applications and Vulnerabilities

In computer security, vulnerability is a weakness which allows an attacker to reduce a system's information assurance. Vulnerability is the intersection of three elements: a system susceptibility or flaw, attacker access to the flaw, and attacker capability to exploit the flaw. To be vulnerable, an attacker must have at least one applicable tool or technique that can connect to a system weakness. In this frame, vulnerability is also known as the attack surface. According to the OWASP (Open Web Application Security Project) most common Web Application Vulnerabilities are.[5][9]

**Injection Flaws:** Injection flaws, such as SQL, OS, and LDAP injection, occur when entrusted data is sent to an interpreter as part of a command or query. The

attacker's hostile data can trick the interpreter into executing unintended commands or accessing unauthorized data.

**Cross-Site Scripting (XSS):** XSS flaws occur whenever an application takes entrusted data and sends it to a web browser without proper validation and escaping. XSS allows attackers to execute scripts in the victim's browser which can hijack user sessions, and redirect the user to malicious sites.

**Broken Authentication and Session Management:** Application functions related to authentication and session management are often not implemented correctly, allowing attackers to compromise passwords, keys, session tokens, or exploit other implementation flaws to assume other users' identities.

**Insecure Direct Object References:** A direct object reference occurs when a developer exposes a reference to an internal implementation object, such as a file, directory, or database key. Without an access control check or other protection, attackers can manipulate these references to access unauthorized data.

**Cross-Site Request Forgery (CSRF):** A CSRF attack forces a logged-on victim's browser to send a forged HTTP request, including the victim's session cookie and any other automatically included authentication information, to a vulnerable web application. This allows the attacker to force the victim's browser to generate requests the vulnerable application thinks are legitimate requests from the victim.

**Security Misconfiguration:** Good security requires a secure configuration defined and deployed for the application, frameworks, application server, web server, database server, and platform. All these settings should be defined, implemented, and maintained as many are not shipped with secure defaults. This includes keeping all software up to date, including all code libraries used by the application.

**Insecure Cryptographic Storage:** Many web applications do not properly protect sensitive data, such as credit cards, SSNs, and authentication credentials, with appropriate encryption or hashing. Attackers may steal or modify such weakly protected data to conduct identity theft, credit card fraud, or other crimes.

**Failure to Restrict URL Access:** Many web applications check URL access rights before rendering protected links and buttons. However, applications need to perform similar access control checks each time these pages are accessed, or attackers will be able to forge URLs to access these hidden pages anyway.

**Insufficient Transport Layer Protection:** Applications frequently fail to authenticate, encrypt, and protect the confidentiality and integrity of sensitive network traffic. When they do, they sometimes support weak algorithms, use expired or invalid certificates, or do not use them correctly.

**Invalidated Redirects and Forwards:** Web applications frequently redirect and forward users to other pages and websites, and use entrusted data to determine the destination pages. Without proper validation, attackers can redirect victims to phishing or malware sites, or use forwards to access unauthorized pages.

## 1.2 SQL Injection

SQL injection is an attack in which SQL code is inserted or appended into application/user input parameters that are later passed to a back-end SQL server for parsing and execution. Any procedure that constructs SQL statements could potentially be vulnerable, as the diverse nature of SQL and the methods available for constructing it provide a wealth of coding options. The primary form of SQL injection consists of direct insertion of code into parameters that are concatenated with SQL commands and executed. A less direct attack injects malicious code into strings that are destined for storage in a table or as metadata. When the stored strings are subsequently concatenated into a dynamic SQL command, the malicious code is executed.[18][27][36] When a Web application fails to properly sanitize the parameters which are passed to dynamically created SQL statements (even when using parameterization techniques) it is possible for an attacker to alter the construction of back-end SQL statements. When an attacker is able to modify an SQL statement, the statement will execute



with the same rights as the application user; when using the SQL server to execute commands that interact with the operating system, the process will run with the same permissions as the component that executed the command (e.g., database server, application server, or Web server), which is often highly privileged. A successful SQL injection exploit can read sensitive data from the database, modify database (Insert/Update/Delete), execute administration operations on the database (such as shutdown the DBMS), recover the content of a given file present on the DBMS file system and in some cases issue commands to the operating system.

## 1.3 Problem Definition

Given the perspective of time passed since web applications entered the commercial market, SQL injection is hardly a new threat. The problem has been described by many security professionals and hackers, and the information is widely spread on the Internet. Many attempts have also been made in order to find countermeasures that can contend with and overcome SQL injection threats. In addition, the countermeasures constitute new solutions regarding application layer vulnerabilities in general and SQL injection threats in particular.

It even seems to be a somewhat common assumption among writers to think that protecting web applications from SQL injection is an easy task, as long as you have an understanding of the SQL injection threat. Nevertheless, corporations, security professionals and hackers continue to announce that SQL injection vulnerabilities are inherent in web applications and reports of compromised applications are frequently published. This clearly indicates that there still seems to exist a lack of awareness, knowledge and respect of SQL injection threats inherent in web applications among security professionals. One reason might be that software development companies and third party vendors do not take a structured security approach when developing web applications. Another reason is that software developers compete in introducing

software to the market.

Several commercial and open source tools were used to detect SQL Injection vulnerabilities in a set of vulnerable web application. The results for penetration testing tools and static code analysis tool were analyzed separately and then compared to better understand the strengths and weaknesses of each approach. A key observation is that different tools implementing the same approach frequently report different vulnerabilities in the same piece of code. Although the results of this study cannot be generalized, they highlight the strengths and limitations of both approaches and suggest that future research on this topic is of utmost important. Finally, results show that web application programmers should be very careful when selecting a vulnerability detection approach and when interpreting the results provided by automated tools.

After the study of existing vulnerability scanning model and tools, we can say that no tool provides 100% accuracy in terms of vulnerability scanning. Besides this it also takes a large span of time to produces reports. We also believe that some of the proposed prevention techniques may contain weaknesses and that they therefore cannot adequately cope with SQL injection.

## 1.4 Motivation

As protection of the web applications from SQL injection attacks are very simple, but still lots of web applications including most of our government and institutions websites designed by software development companies are vulnerable. One of the reasons is the lack of awareness about the security facts, and apart from that most of the solutions proposed to deal with SQLI are tool based. And the tools used to detect SQLI do not provide good results. So these issues inspire us to provide a security solution which do not need any extra overhead, as well as a security mechanism which can be embedded with the web application. For that we explore the language features

and discovered a filter approach for security. By observing different commercial tools we also aim to design a network based tool.

## **1.5 Organisation of this Thesis**

The remaining part of the thesis is organized as follows: In Chapter 2 of this thesis, we discuss common web applications including their domain, architecture and inherent components and present our own model of a general web application. We also describe how communication takes place in web applications and which assets they contain. Chapter 3 gives an introduction to SQL Injection attacks, its types and the different techniques to perform the same, including the working of SQLI. In Chapter 4, we have proposed our filter approach to prevent the SQL injection attack at the run time, including the architecture, and implementations details and in section 4.2 a network based vulnerability scanner have been proposed, we have discussed their model, as well as the implementation details, then we have discussed the comparison details to show the effectiveness of our proposed tool. Chapter 5 concludes the thesis along with the future work, followed by the references and glossary used.

# Chapter 2

## Web Application

In this section we discuss web applications, their architecture and the components they are composed of. We also describe how communication takes place in web applications and identify web application assets. Along the way, we intend to present a general model of an web application which we will refer throughout the rest of this thesis. This enables us to study our problem from a general perspective, without having to consider architectural or implementation specific details, i.e. number of tiers, and the choice of components.

### 2.1 Domain

People browse the web use web applications in one form or another though, as OWASP [8] notes, the everyday web user may not be aware of that fact because of the ubiquity of web applications: "When one visits google.com and the site automatically knows you are a India resident and serves you google.co.in, it is all because of a web application. When we transfer money, search for a flight, check out arrival times or even the latest sports scores online, we are using a web application."

### **2.1.1 Web Service**

Before proceeding, we think it is justified to mention a few words about the concept of web services. Web service is a server oriented system which therefore operates on the server side and perform a task when it is called upon by an application like any service, a web service requires an API to provide an interface which allows it to be called by another application. As can be seen in an operating system of a common personal computer, a service is registered in the system registry which allows application to locate the specific service to process a specific task. In the same way a web service registered in a web service registry, which an application use to call the specific service it requires. Web service is not language and platform independent, it uses XML to communicate with other service or application. However, due to differences in e.g. architecture, languages and protocols between web services and web applications, we prefer to leave web services outside our definition of our general web application.[19][21]

### **2.1.2 Business Web Applications**

OWASP states that any software application built on client-server technology that operates on the web and that interacts with users or other systems using HTTP could be classified as a web application. The client server architecture is discussed in section 2.2 and HTTP is discussed in section 2.4.3. Web applications provide connectivity, access to information and online services for users. Web applications can be implemented in various degrees of complexity, and each implementation has a distinct purpose: like an informational website, an e-commerce website, an extranet, an intranet, an exchange, a search engine, a transaction engine, an e-business. The functions performed can therefore, as OWASP notes, range from relatively simple tasks like searching a local directory for a file, to highly sophisticated applications that perform real-time sales and inventory management across multiple vendors, including both Business to Business and Business to Consumer e-commerce, flow of

work and supply chain management, and legacy applications. Corporations make use of sophisticated web applications, also referred to as business web applications, in order to accomplish more efficient information exchange within their own organizations as well as between partners in their value chains, i.e. suppliers, distributors and customers. In addition, web applications offer corporations management of customer relationships, new ways of doing business and creation of alliances, movement into new markets, and promotion of their products and services. In order to be useful and comply with their purpose, web applications require persistent storage for their data. Usually, a RDBMS is chosen in order to achieve this.[7][32]

## **2.2 Architecture**

From a hacker's perspective, a corporation's web application can be viewed as a horizontal value chain of layers. However, we think that in the context of web application architecture, focusing on where different kinds of tasks are processed is more appropriate. Therefore, an examination of the client-server architecture is motivated.[17][36]

### **2.2.1 Client-Server**

Web itself is comprised of a network of computers, and each computer acts in different roles: as a client, a server or both. In order to accommodate an increasingly decentralized business environment, web applications operating on the web use the client-server architecture. The term client-server, refers to the processes with which software components interact to form a system, i.e. client processes require resources provided by server processes. Combinations of the client-server architecture, or topology, include: (a) single client, single server; (b) multiple clients, single server; (c) multiple clients, multiple servers. The client in a web application is usually represented by a web browser like Internet Explorer or Netscape Navigator. Servers typically

include web servers, e.g. Microsoft Internet Information Server, Apache and Tomcat. In client-server architecture, applications can be modeled as consisting of different logical strata. One problem is that there are different opinions regarding about the meaning of logical strata, i.e whether to view them as layers or tiers. We prefer the approach of dividing strata into a software view and a hardware view. In the software view, the strata consist of layers and in the hardware view, tiers represent the strata.

### 2.2.2 The Client-Server Architecture and Layers

Before proceeding, we stress that the layers we will refer to throughout this thesis concern responsibilities and task processing in web applications, and not how communication in networks are organized into abstraction levels. Therefore, we do not consider the layered approach taken in models such as the Open Systems Interconnection (OSI) reference model to be relevant in our discussions. In client-server architecture, applications can be modeled as consisting of logical layers. While there exist different conventions for naming those layers, we conclude that the following three different layers are included

- **Presentation:** The layer where information is being presented to users and which constitutes the interaction point between users and the application. This layer is actually constituted of two parts, where one part is dedicated to the client-side and the other part concerns the server side. While this layer generates and decodes web pages, it can also be responsible for presentation logic, meaning that components of this layer can reside both on the client-side and server-side. Distributed logic needed to connect to a proxy layer on the server-side along with a proxy tier in order to make use of middle-ware, e.g. CORBA and RMI, could also reside here.
- **Application logic:** The layer where application logic and business logic and

rules are implemented. This layer processes user input, makes decisions, performs data manipulation and translation into information, including calculations and validations, manages work flow, e.g. keeping track of session data, and handles data access for the presentation layer.

- **Data management:** The layer responsible for managing both temporary and permanent data storage, including database operations.

### 2.2.3 The Client-Server Architecture and Tiers

We have found several different models which describe how web applications are composed using the logical layers mentioned in section 2.2.2. One main characteristic shared by those models constitutes the combination of logical layers into a 2, 3 or n-tier architecture in order to provide for a separation of tasks, where a tier is defined as one of two or more rows, levels and ranks arranged one above another.[41]

- **Two-Tier Architecture:** In the two-tier client-server architecture, clients constitute the first tier and servers the second tier. A client is primarily responsible for presentation services, including handling user interface actions, performing application logic and presentation of data to the user and performing the main business application logic. The server is primarily concerned with supplying data services to the client. Data services provide limited business application logic, typically validation of the client and access control to data. Typically, the client would run on end-user desktops and interact with a centralized DBMS over a network.
- **Three-Tier Architecture:** In three-tier architecture, the first tier still constitutes the client which is now considered a thin client, i.e. is only responsible for the application's user interface and possibly simple logic processing, such as input validation. The core business logic of the application now resides in its



own tier, the middle tier that runs on a server and is often called the application server. The third tier constitutes an RDBMS, which stores the data required by the middle tier, and may run on a separate server called the database server.

- **N-Tier Architecture:** This type of architecture simply implies any number of tiers. One example of this is when the web server and database server reside in separate computers. Another example is when several database servers are used and one computer is dedicated responsible of managing access to each database server, running on separate computers

## 2.3 Software Components

In order to be able to define our own model of a general web application and its inherent components, we have studied a number of existing models. OWASP points out that the discussion of what components a web application actually consists of is rife with confusion. We can also conclude that the terms found in different models have no clear definition, are used differently by different authors, and even overlap each other. Development languages, web servers, application servers, middleware and databases can all be used in secure and insecure manners. Hence, vulnerabilities inherent in web applications do not solely rely on the type of components chosen.[6][27] For these reasons, we selected a collection of components from the available material that we consider representative in order to compose the general web application we will refer to throughout this thesis. These components were then grouped according to their logical layer membership defined in section 2.2.2 and furthermore divided into different types. Note that the presentation layer is divided into two layers, where one layer is dedicated to the client-side and the other layer resides on the server-side. Moreover, we consider some components as occurring both at the client-side and the server-side. This stems from the fact that some components, e.g. web pages, are stored or generated at the server-side, but they are sent to the client side.

### 2.3.1 Components Type

We have identified three distinct types of components that we consider meaningful in order to avoid overwhelming this thesis with complexity and to allow a discussion of SQL injection in respect of where different tasks are processed within a web application:

**Software components** if they are a part of software implementation.

**Data-processing components** if they operate on data.

**Data components** if they are stored or processed data.

### 2.3.2 A General Business Web Application Model

In this section, we have described a general model of an web application [27] with its components mapped to layers. The components chosen and described are not meant to be exhaustive. Rather, our intention is to present a foundation for our discussions in chapter 4. This model is also illustrated in figure 2.1.

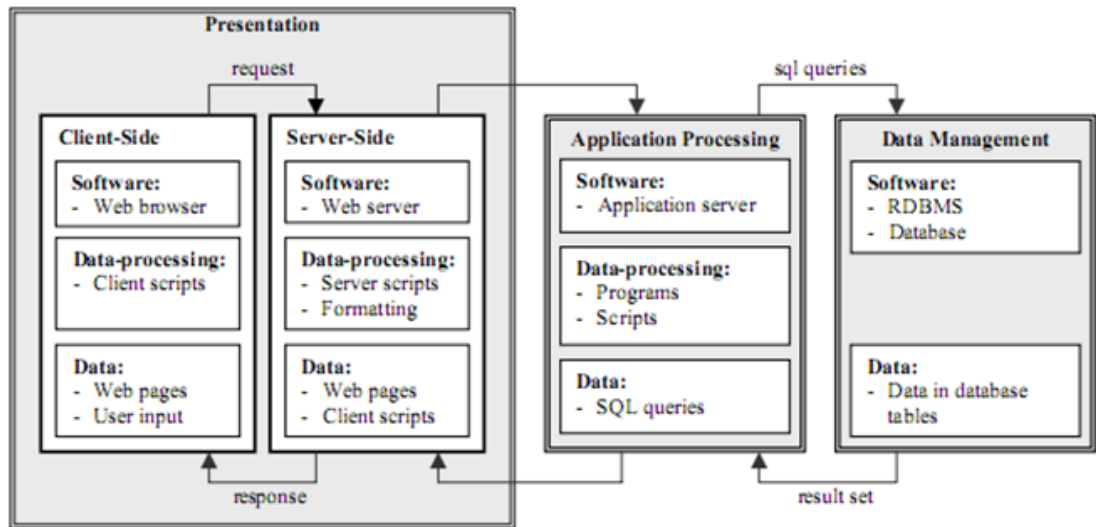


Figure 2.1: Components within layers.

## Presentation Layer

- *Software components*
  - A web browser, e.g. Microsoft Explorer, Netscape Navigator, Mozilla and Opera, which operates at the client-side. A browser loads static and dynamic web pages along with client-scripts and stylesheets from a web server, and presents them in a graphical user interface. This component may also constitute the point where users interact with the application.
  - A web server, e.g. iPlanet, Apache, Zeus, Microsoft IIS and Netscape Enterprise, which operates at the server-side, receives requests from the client-side and processes them. The server returns responses containing data components such as static or dynamic web pages. The dynamic web pages may be generated by the server itself or by data-processing components.
- Data-processing components
  - Web browser scripts written in script languages such as JavaScript or VBScript that constitute presentation logic on the client-side. They extend user interactivity and perform tasks, e.g. input validation.
  - Server scripts such as Active Server Pages (ASP) or Java Server Pages (JSP) that perform presentation logic on the server-side and generate dynamic web pages.
  - Formatting components written in stylesheet languages, including Cascading Style Sheets (CSS) and The Extensible Stylesheet Language Transformations (XSLT). XSLT is designed for use as part of XSL, a stylesheet language for XML, and transforms XML documents into other XML documents.
- Data components

- Web pages written in e.g. HTML and the Wireless Markup Language (WML), that are stored on the server-side, loaded into the client-side and presented in the web browser. They can be static, i.e. hard coded, or dynamic, i.e. generated by server-side scripts.
- User input data that is entered in forms of web pages or the URL header in the web browser.
- Client scripts sent to the client-side.
- Request objects sent from the client-side to the server-side using HTTP, HTTPS or XML.
- Data sent between the server-side of the presentation layer and application processing layer.

## **Application Processing Layer**

- *Software components*

- An application server that serves as a framework or development environment for technologies that implement application or business logic in stand-alone programs. Examples of application servers include IBM Websphere, BEA Weblogic, JBoss and iPlanet, Zone and Zend. Technologies include Servlets and Enterprise Java Beans written in Java and the Common Gateway Interface (CGI) that takes advantage of scripts written in various languages, including C, Perl and Python. Other examples of technologies include PHP, Visual Basic (VB) and .NET services.

- *Data-processing components*

- Stand-alone programs, e.g. Servlets or Enterprise Java Beans.
- CGI scripts written in languages such as C, C++, Java or Perl.

- *Data components*

- Data sent between the application processing layer and the server-side presentation layer.

### **Data management layer**

- *Software components*

- A RDBMS that inserts, retrieves and manipulates data in a database through SQL queries, as well as controls access to the database through access control mechanisms.
- A relational database that contains valuable data.

- *Data components*

- Data in database tables.
- Data and information sent between the application processing layer and data management layer, e.g. SQL queries and result sets containing data from the database.

## **2.4 Communication**

A typical communication exchange in a business web application is initiated by users that request information. The client takes a users request, checks the syntax and generates database requests in e.g. SQL. Then, the client transmits the message to the server, waits for a response, and formats the response for the end-user. The server accepts and processes the database requests, then transmits the results back to the end-user.[3][27]

### **2.4.1 Information**

Information on the web is stored in documents and the formatting language, or system, most commonly used is the HTML. Using HTML, documents are marked up, or

tagged, to allow for publishing on the web in a platform independent manner. HTML documents are displayed in web browsers, that understand and interpret HTML.

### 2.4.2 Content

HTML documents stored in files constitute static content, i.e. the content of the document does not change unless the file itself is changed. However, documents resulting from requests such as queries to databases need to be generated by the web servers. These documents are dynamic content and as databases are dynamic, changing as users create, insert, update, and delete data, the generation of dynamic web pages is a more appropriate approach than static content, particularly in web applications.

### 2.4.3 Protocol

The exchange of information in web applications is mainly governed by protocols such as HTTP or HTTPS, which define how clients, i.e. web browsers, and servers, i.e. web servers, communicate. HTTP relies on a request-response paradigm and a transaction consists of the following stages.[3]

**Connection** The client establishes a connection with the web server.

**Request** The client sends a request message to the web server.

**Response** The web server sends a response, i.e. a HTML document, back to the client.

**Close** The connection is closed by the web server.

Basically, a request in a HTTP connection constitutes an object containing, e.g. a requested resource. Consequently, a response is the result to be presented in the web browser. When a user visits a page, web pages, client-side scripts and formatting components are sent back to the client for rendering and presentation. In the case a user requests data contained in a relational database, user input parameters are typically embedded in the request. Those parameters can be included as arguments to

methods in application processing components that dynamically build SQL queries. They may also indicate which SQL query is to be executed, in case that static SQL is used, e.g. stored procedures or prepared statements). The response object will contain data for presentation in the web browser. That data may have been parsed and prepared by either application processing components, server-side scripts or both for rendering purposes: either for tailoring the graphical design or ease the rendering process in the web browser.

HTTP brings up several security weaknesses. A HTTP request is composed of different parts and attackers can manipulate those parts in order to try to bypass security mechanisms. The web server listens on an open port for incoming requests from clients. For general web traffic, i.e. HTTP, port 80 is often used as the default port and for encrypted traffic, i.e. HTTPS, port 443 is normally chosen. However, each web server requires a unique port to listen to and since corporations can have several web servers, the port of each server has to be configured. Moreover, application servers require open ports as well. While this means that an attacker can not always assume that the web server of choice listens to port 80, the important issue is that there exists an open port through security mechanisms such as firewalls into a corporations web server.

#### **2.4.4 URL Encoding**

According to OWASP [8], a server can receive input from a client in two basic ways: either data is passed in HTTP headers or it can be included in the query portion of the requested Uniform Resource Locator (URL), which uniquely defines where resources can be found on the Internet. Both methods correspond to two methods used when including input in client requests: GET and POST. Manipulation of a URL or a form is simply two sides of the same issue. [31]

However, when data is included in a URL, it must be specially encoded to conform to proper URL syntax. Unfortunately, as OWASP notes, the URL encoding mechanism

allows virtually any data to be passed from a client to the server. Proper precautions must be taken by the application logic, when accepting URL encoded data since this mechanism can be used for disguising malicious code.



# Chapter 3

## SQL Injection

This section constitutes the first part of our attempt to contribute to the SQL injection problem. The area of SQL injection, including definitions, context of operations, conditions, vulnerabilities, attack methods, and existing detection and prevention techniques will be explained and classified.

### 3.1 Introduction

#### 3.1.1 Definition

SQL injection belongs to a class of code-injection attacks where the data provided by the user is included in a SQL query, such that part of the user's input is treated as SQL code [23]. SQLIAs are a type of vulnerability that is caused by insufficient input validation - they occur when data provided by the user is not properly validated and is included directly in a SQL. By leveraging these vulnerabilities, an attacker can submit SQL commands directly to the database. This kind of vulnerability represents a serious threat to any web application that reads input from the users (e.g., through web forms) and uses it to build SQL queries to an underlying database. Most web applications used on the Internet or within enterprises work this way and could therefore be vulnerable to SQL injection.[18]

SQL injection has probably existed since SQL databases were first connected to Web applications. However, Rain Forest Puppy is widely credited with its discovery or at least for bringing it to the public's attention. On Christmas Day 1998, Rain Forest Puppy wrote an article titled "NT Web Technology Vulnerabilities" for Phrack. SQLI attacks result in severe information breach which can lead to exposing confidential information. Like In July 2008, Kaspersky's Malaysian site was hacked by Turkish hacker, who claimed to have used SQL injection, On August 17, 2009, the United States Justice Department charged an American citizen Albert Gonzalez and two unnamed Russians with the theft of 130 million credit card numbers using an SQL injection attack. In reportedly "the biggest case of identity theft in American history", the man stole cards from a number of corporate victims after researching their payment processing systems, On 8th November 2010 the British Royal Navy website was compromised by TinKode using SQL injection, On March 27th 2011 mysql.com, the official homepage for MySQL, was compromised by TinKode using SQL blind injection etc.[1]

### 3.1.2 SQLI Example

Attacks on the database can be made in order to gain access to the web application, and thus threatening the application's authentication security. Suppose that an attacker would like to try and log in into a system that uses a web interface which has not been protected from SQL injection attacks. This might be the first step an attacker takes in order to be able to commit further attacks on an application. In order to do this, an attacker might try to analyze how a login page, like the one in figure 3.1 below might be exploited. [5] [18]

In our example we have taken a simple web application login which checks the userid and the password supplied by the client. The application is designed to check that only the legitimate users who have the valid userid and password will proceed otherwise they receive a error message like invalid userid or password. We are assuming

Admin Control System	
User :	<input type="text"/>
password :	<input type="password"/>
<input type="button" value="login"/>	

Figure 3.1: Admin Login

that the login button will submit the userid and password to the corresponding server side script written in php, and the code is not prevented against the SQLI, means not sanitized, like below.

```
// username and password sent from Client
myusername = $_POST['myusername'];
$mypassword=$_POST['mypassword'];
$sql="SELECT * FROM admin WHERE username='$myusername'
and password='$mypassword'";
$result=mysql_query($sql);
// Mysql_num_row is counting table row
$count=mysql_num_rows($result);
// If result matched $myusername and $mypassword, table row must be
1 row
if($count==1)
//Redirect to file "login_success.php"
header("location:login_success.php");
else
echo "Wrong Username or Password";
```

So if the attacker will pass ' OR '1'='1 in uerid and password then the query will become like `$sql="SELECT * FROM admin WHERE username='' OR '1'='1'`

and password=''' **OR '1'='1'**'''. Now if look this query this always evaluates to true because the half part of this will always be true, and then it will return the first record from the admin table and count will be 1, hence attacker bypass the authentication. So even the attacker don't know the userid and password he can log in as administrator. This is only the one form of SQLI, an attacker can inject other commands too, to tamper the database. Like if attacker will submit '**having 1=1**-' in the user field and password is anything, then it will return an error like.

*Column 'users.id' is invalid in the select list because it is not contained in an aggregate function and there is no GROUP BY clause.*

This error message contains information about the SQL query: it concerns a table named users with a column named id.

The attacker continues investigating the query by writing:

Username: ' group by users.id having 1=1-

Which results in the error message:

*Column 'users.username' is invalid in the select list because it is not contained in either an aggregate function or the GROUP BY clause.*

So now the attacker knows that the table also contains the column name username.

The attacker may continue to acquire information about the query until he recovers the entire syntax used by the script to build the dynamic SQL query.

### 3.1.3 Injection Mechanism

Malicious SQL statements can be introduced into a vulnerable application using many different input mechanisms. There are various ways to perform the SQLI listed below. We here explain the most common mechanisms. [29]

#### Injection through user input

In this case, attackers inject SQL commands by providing suitably crafted user input. A Web application can read user input in several ways based on the environment in which the application is deployed. In most SQLIAs that target Web applications, user

input typically comes from form submissions that are sent to the Web application via HTTP GET or POST requests. Web applications are generally able to access the user input contained in these requests as they would access any other variable in the environment.

### **Injection through cookies**

Cookies are files that contain state information generated by Web applications and stored on the client machine. When a client returns to a Web application, cookies can be used to restore the clients state information. Since the client has control over the storage of the cookie, a malicious client could tamper with the cookies contents. If a Web application uses the cookies contents to build SQL queries, an attacker could easily submit an attack by embedding it in the cookie.

### **Injection through server variables**

Server variables are a collection of variables that contain HTTP, network headers, and environmental variables. Web applications use these server variables in a variety of ways, such as logging usage statistics and identifying browsing trends. If these variables are logged to a database without sanitization, this could create an SQL injection vulnerability. Because attackers can forge the values that are placed in HTTP and network headers, they can exploit this vulnerability by placing an SQLIA directly into the headers. When the query to log the server variable is issued to the database, the attack in the forged header is then triggered.

### **Second-order injection**

In second-order injections, attackers seed malicious inputs into a system or database to indirectly trigger an SQLIA when that input is used at a later time. The objective of this kind of attack differs significantly from a regular (i.e., firstorder) injection attack. Second-order injections are not trying to cause the attack to occur when the malicious input initially reaches the database. Instead, attackers rely on knowledge of where the input will be subsequently used and craft their attack so that it occurs during that usage.

To clarify, we present a classic example of a second order injection attack. In the example, a user registers on a website using a seeded user name, such as admin – . The application properly escapes the single quote in the input before storing it in the database, preventing its potentially malicious effect. At this point, the user modifies his or her password, an operation that typically involves (1) checking that the user knows the current password and (2) changing the password if the check is successful. To do this, the Web application might construct an SQL command as follows:

```
queryString="UPDATE users SET password=" + newPassword + " WHERE user-  
Name=" + userName + " AND password=" + oldPassword + ""
```

*newPassword* and *oldPassword* are the new and old passwords, respectively, and *userName* is the name of the user currently logged-in (i.e., admin–). Therefore, the query string that is sent to the database is (assume that *newPassword* and *oldPassword* are *newpwd* and *oldpwd*):

```
UPDATE users SET password=newpwd WHERE userName= admin–  
AND password=oldpwd
```

Because – is the SQL comment operator, everything after it is ignored by the database. Therefore, the result of this query is that the database changes the password of the administrator (admin) to an attacker-specified value.

Second-order injections can be especially difficult to detect and prevent because the point of injection is different from the point where the attack actually manifests itself. A developer may properly escape, type-check, and filter input that comes from the user and assume it is safe. Later on, when that data is used in a different context, or to build a different type of query, the previously sanitized input may result in an injection attack.

### **3.1.4 Attack Procedure**

We have found that attackers in general follow a procedure consisting of a series of steps. Our compiled procedure represents the set of all steps identified with respect

to all available attack methods. Attackers may combine methods in the attack in order to fulfill their objectives, but the process is executed for each attack method in an iterative manner. Depending on the attack method used, some steps may be ignored.[27]

### **Setting the objective**

Whether explicit or arbitrary, attackers have one or more objectives for conducting SQL injection attacks. These objectives relate to security services. A concrete example might be that an attacker wants to access the web application in order to obtain information about a corporation's customers. This is an attack on the security service confidentiality.

### **Choosing the method**

In some cases, the attacker is only interested in gaining access to the web application and therefore tries to bypass authentication. In other cases, bypassing authentication is only one step before he can try to reach his objectives. Hence, several methods can be chosen.

### **Examining prerequisites**

In order to determine if the objectives can be reached, the attacker systematically checks which prerequisites is supported. Prerequisites may be necessary conditions for a given attack method, or make the attack easier to conduct. Testing for vulnerabilities: The attacker begins testing for vulnerabilities to exploit, e.g. experimenting with input validation by entering single quotes, enumerating privileges or evaluating returned information.

### **Choosing means**

Depending on supported prerequisites and found vulnerabilities, the attacker chooses his means for the attack.

### **Designing the query**

The query designed by the attacker needs to follow the proper structure of an SQL query expected by the RDBMS. If not, syntax errors are generated and displayed

in error messages. One example of syntax errors relates to quotation marks, i.e. if SQL injection is possible without escaping them. Another example is if parentheses are used in the underlying query. Depending on the objective, other syntax errors that concern information retrieval of database structure may have to be overridden. Examples of such errors include table names, column names, number of columns and data types.

### **3.1.5 Attack Intent**

Attacks can also be characterized based on the goal, or intent, of the attacker [23]. Therefore, each of the attack type definitions that we provide in Section 3.2 includes a list of one or more of the attack intents defined in this section.

#### **Identifying injectable parameters**

The attacker wants to probe a Web application to discover which parameters and user-input fields are vulnerable to SQLIA.

#### **Performing database finger-printing**

The attacker wants to discover the type and version of database that a Web application is using. Certain types of databases respond differently to different queries and attacks, and this information can be used to fingerprint the database. Knowing the type and version of the database used by a Web application allows an attacker to craft database specific attacks.

#### **Determining database schema**

To correctly extract data from a database, the attacker often needs to know database schema information, such as table names, column names, and column data types. Attacks with this intent are created to collect or infer this kind of information.

#### **Extracting data**

These types of attacks employ techniques that will extract data values from the database. Depending on the type of the Web application, this information could be sensitive and highly desirable to the attacker. Attacks with this intent are the most



common type of SQLIA.

#### **Adding or modifying data**

The goal of these attacks is to add or change information in a database.

#### **Performing denial of service**

These attacks are performed to shut down the database of a Web application, thus denying service to other users. Attacks involving locking or dropping database tables also fall under this category.

#### **Evading detection**

This category refers to certain attack techniques that are employed to avoid auditing and detection by system protection mechanisms.

#### **Bypassing authentication**

The goal of these types of attacks is to allow the attacker to bypass database and application authentication mechanisms. Bypassing such mechanisms could allow the attacker to assume the rights and privileges associated with another application user.

#### **Executing remote commands**

These types of attacks attempt to execute arbitrary commands on the database. These commands can be stored procedures or functions available to database users.

#### **Performing privilege escalation**

These attacks take advantage of implementation errors or logical flaws in the database in order to escalate the privileges of the attacker. As opposed to bypassing authentication attacks, these attacks focus on exploiting the database user privileges.

## **3.2 Types Of SQLI**

In this section, we present and discuss the different kinds of SQLIAs known to date. For each attack type, we provide a descriptive name, one or more attack intents, a description of the attack, an attack example, and a set of references to publications and Web sites that discuss the attack technique and its variations in greater detail.

The different types of attacks are generally not performed in isolation; many of them are used together or sequentially, depending on the specific goals of the attacker. Note also that there are countless variations of each attack type. For space reasons, we do not present all of the possible attack variations but instead present a single representative example.[23]

- **Piggy-backed Queries**

Piggy backed queries are the type of SQL Injection queries formed when the hackers insert additional queries to be executed by the database, along with the intended query. Hence this type effectively piggybacks the attack on the top of a legitimate request. This class of attacks will be successful only if the server configurations allow several different queries within a single line of code.

For example, let us consider the query string for login:

*QueryString = "SELECT info FROM userTable WHERE" + login= + login + AND pin = + pin;*

User Input for PIN: **0; DROP database webApp**

The above user input forms two queries:

*QueryString = SELECT info FROM userTable WHERE login='admin'*  
*AND pin=0; DROP database webApp*

Which will execute both the queries instead of the single select statement.

- **Tautologies**

Tautologies are the simplest form of SQL injection queries which always evaluate to true for entries in the database. Hence they always return results upon the evaluation of the WHERE clause.

The following are a sample list of inputs that form tautologies:

' or 1=1–

" or 1=1–

or 1=1–

' or 'a'='a

" or "a"="a

) or ('a'='a

These inputs can also be given in their hex encoded equivalent also. For example

' or 1=1- can be given as 0x27206F7220313D312D2D

The following simple login query example illustrates tautology:

```
QueryString = "SELECT info FROM userTable WHERE login='" + login + "'  
AND pin =" + pin;
```

User Input: **user or 1=1** -

The above user input forms the following queries:

```
QueryString = SELECT info FROM userTable WHERE login='user' or 1=1 -  
AND pin=;
```

Which will evaluate to true regardless of the login name input by the attacker, and will be able to bypass the authentication mechanisms in place.

- **Illegal/Logically Incorrect Queries**

This technique is usually used during the information gathering stage of the attack. Through injecting illegal/logically incorrect requests, an attacker may gain knowledge that aids the attack, such as finding out the inject-able parameters, data types of columns within the tables, names of tables, etc. This is usually done using the HAVING and GROUP BY clause. The following example illustrates the same: [18][30]

First, the attacker wants to establish the names of the tables that the query operates on, and the names of the fields. To do this, the attacker uses the 'having' clause of the 'select' statement:

```
QueryString = "SELECT info FROM userTable WHERE login='" + login + "'  
AND pin =" + pin;
```

User Input Login: **having 1=1** -

Forms the following query:

*QueryString = "SELECT info FROM userTable WHERE login=" having 1=1  
- AND pin =";*

This provokes the following error on SQL Server:

Microsoft OLE DB Provider for ODBC Drivers error '80040e14'

*[Microsoft] [ODBCSQLServerDriver] [SQLServer] Column útersidís invalid  
in the select list because it is not contained in an aggregate function and there  
is no GROUP BY clause.*

So the attacker now knows the table name and column name of the first column in the query. They can continue through the columns by introducing each field into a 'group by' clause, as follows:

User Input Login: ***group by users.id having 1=1-***

Forms the following query

*QueryString = "SELECT info FROM userTable WHERE login=" group by  
users.id having 1=1- AND pin =";*

Which produces the error on SQL Server... Microsoft OLE DB Provider for ODBC Drivers error '80040e14'

*[Microsoft] [ODBCSQLServerDriver] [SQLServer] Column 'users.username'  
is invalid in the select list because it is not contained in either an aggregate func-  
tion or the GROUP BY clause.*

From this the attacker can find out the next field as users.username. Now this field can be included in the group by clause along with the previous field and this process is repeated until no error is reported by the server, after which the attacker would be able to obtain all the fields of the table.

- **Union Query**

This technique is used to deceive the servers to return data that were not intended to be returned by the developers. The statement UNION SELECT can be added along with a query which will return the union of the intended dataset with the target dataset. [28][32]

The attacker could determine the types of each column retrieved using the 'Having' and 'Group by' clause using a 'type conversion' error message along with the union query, like:

Username: ' *union select sum (username) from users*—

This takes advantage of the fact that SQL server attempts to apply the 'sum' clause before determining whether the number of fields in the two rowsets is equal. Attempting to calculate the 'sum' of a textual field results in this message:

Microsoft OLE DB Provider for ODBC Drivers error '80040e07'

[Microsoft] [ODBCSQLServerDriver] [SQLServer] The sum or average aggregate operation cannot take a varchar data type as an argument.

Which tells us that the username field has type varchar. We can use this technique to approximately determine the type of any column of any table in the database.

- **Stored Procedures**

Database server often contains system stored procedures that programmers may use when developing applications [16]. If the attacker has knowledge about the backend database server used, then these stored procedures can be exploited to perpetrate their attacks. Stored procedures may allow the attackers to interact with the targets beyond the database, like the underlying Operating System, for example. `xp_cmdshell` is a built-in extended stored procedure that allows the execution of arbitrary command lines. For example:

**`exec master..xp_cmdshell 'net1 user'`**

will provide a list of all users on the machine. Since SQL server is normally running as either the local 'system' account, or a 'domain user' account, an attacker can do a great deal of harm.

The following are the list of extended Stored Procedures that can be used:

- **xp\_cmdshell** - Used to run commands as the SQL server user, on the database server.  
Eg: `exec master xp_cmdshell 'dir'` - obtain a directory listing of the current working directory of the SQL Server process
- **xp\_regXXX** - Used to manipulate on registry keys. The following are the list of the xp\_regXXX
  - xp\_regaddmultistring
  - xp\_regdeletekey
  - xp\_regdeletevalue
  - xp\_regenumkeys
  - xp\_regenumvalues
  - xp\_regread
  - xp\_regremovemultistring
  - xp\_regwrite Eg: `exec xp_regread HKEY_LOCAL_MACHINE, 'SYSTEM\CurrentControlSet\Services\lanmanserver\parameters', 'nullsessionshares'`  
- determines what null-session shares are available on the server
- **xp\_servicecontrol** Used to start, stop, pause and continue services.  
Eg: `exec master..xp_servicecontrol 'start', 'schedule'`  
`exec master..xp_servicecontrol 'start', 'server'`
- **xp\_availablemedia** - Reveals the available drives on the machine.
- **xp\_dirtree** Allows a directory tree to be obtained
- **xp\_enumdsn** - Enumerates ODBC data sources on the server
- **xp\_loginconfig** - Reveals information about the security mode of the server.

- **xp\_makecab** - Allows the user to create a compressed archive of files on the server (or any files the server can access)
  - **xp\_ntsec\_enumdomains** - Enumerates domains that the server can access
  - **xp\_terminate\_process** - Terminates a process, given its PID
  - **xp\_OACreate, sp\_OAMethod and sp\_OAGetProperty** - These system stored procedures can be used to create Ole Automation (ActiveX) applications that can do everything an ASP script can do.
- Apart from these stored procedures, the following commands can also be used:
- **bulk insert** - Used to read any file on the server.
  - **bcp** - Used to create arbitrary text files on the server

- **End of Line Comment**

After injecting code into a particular field, legitimate code that follows are nullified through usage of end of line comments. An example would be to add - - or " # " after inputs so that remaining queries are not treated as executable code, but comments. For example,

*QueryString = SELECT info FROM userTable WHERE login='admin' or 1=1-  
AND pin=;*

will comment out the part of the query AND pin=, hence logging in as admin.

- **Blind Injection**

When the systems are more secure, the attackers may probe for vulnerable parameters or extract data by using this technique. Blind injection allows the attackers to infer the database construct through evaluating expressions that are coupled with statements that always evaluate to true or false. For example, the attacker can add and 1=0 – for one attempt, while and 1=1 – is used for another attempt, both added onto the same query. Through examining the

behavior of the server, the attacker may then deduce whether the particular parameter is vulnerable or not: When the two attempts result in the same behavior, the parameter is secure, while different behavior resulting from the two statements suggest that the parameter is vulnerable.[34]

- **OPENROWSET Result Retrieval**

When trying to exploit SQL injection in an application, an attacker needs a method of retrieving the results. The OPENROWSET function allows a user in SQL Server to open remote data sources. The function OPENROWSET is most commonly used to pull data into SQL Servers to be manipulated. They can however also be used to push data to a remote SQL Server. Below is an example of pushing data to an external data source:

```
insert into OPENROWSET (SQLoledb, server=servername; uid=sa;  
pwd=HACKER, select * from table1) select * from table2
```

In the example above, all rows in table2 on the local SQL Server will be appended to table1 in the remote data source.

- **Timing Attack**

A timing attack allows an attacker to gain information from a database by observing timing delays in the response of the database. This attack is very similar to blind injection, but uses a different method of inference. To perform a timing attack, attackers structure their injected query in the form of an if/then statement, whose branch predicate corresponds to an unknown about the contents of the database. Along one of the branches, the attacker uses a SQL construct that takes a known amount of time to execute, (e.g. the WAITFOR keyword, which causes the database to delay its response by a specified time). By measuring the increase or decrease in response time of the database, the attacker can infer which branch was taken in his injection and therefore the answer to the injected question.



Example: Using the code from our running example, we illustrate two ways in which Inference based attacks can be used. The first of these is identifying injectable parameters using blind injection. Consider two possible injections into the login field. The first being `legalUser and 1=0` - - and the second, `legalUser and 1=1` - -. These injections result in the following two queries:

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

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

Now, let us consider two scenarios. In the first scenario, we have a secure application, and the input for login is validated correctly. In this case, both injections would return login error messages, and the attacker would know that the login parameter is not vulnerable. In the second scenario, we have an insecure application and the login parameter is vulnerable to injection. The attacker submits the first injection and, because it always evaluates to false, the application returns a login error message. At this point however, the attacker does not know if this is because the application validated the input correctly and blocked the attack attempt or because the attack itself caused the login error. The attacker then submits the second query, which always evaluates to true. If in this case there is no login error message, then the attacker knows that the attack went through and that the login parameter is vulnerable to injection. The second way inference based attacks can be used is to perform data extraction. Here we illustrate how to use a Timing based inference attack to extract a table name from the database. In this attack, the following is injected into the login parameter:

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

This produces the following query:

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

In this attack the SUBSTRING function is used to extract the first character of the first tables name. Using a binary search strategy, the attacker can then ask a series of questions about this character. In this case, the attacker is asking if the ASCII value of the character is greater-than or less-than-or-equal-to the value of X. If the value is greater, the attacker knows this by observing an additional 5 second delay in the response of the database. The attacker can then use a binary search by varying the value of X to identify the value of the first character.

### 3.3 Vulnerabilities

In this section, we present vulnerabilities that might be inherent in web applications and that can be exploited by SQL injection attacks.[36]

- **Invalidated input**

Unchecked parameters to SQL queries that are dynamically built can be used in SQL injection attacks. These parameters may contain SQL keywords, e.g. INSERT or SQL control characters such as quotation marks and semicolons

- **Error message feedback**

Error messages that are generated by the RDBMS, as defined in section 3.1.2, or other server-side programs may be returned to the client-side and printed in the web browser. While these messages can be useful during development for debugging purposes, they can also constitute risks to the application. Attackers can analyze these messages to obtain information about database or script structure in order to construct their attack

- **Uncontrolled variable size**

Variables that allow storage of data that is larger than expected may allow attackers to enter modified or fabricated SQL statements. Scripts that do not control variable length may even open for other attacks, such as buffer overrun

- **Variable morphism**

If a variable can contain any data, it is possible for an attacker to store other data than expected. Such variables are either of weak type, e.g. variables in PHP, or are automatically converted from one type to another by the RDBMS, e.g. numeric values converted into a string type. For example, SQL keywords can be stored in a variable that should contain numeric

- **Generous privileges**

Privileges defined in databases are rules that state which database objects an account has access to and what functions the user(s) associated with that account are allowed to perform on the objects. Typical privileges include allowing execution of actions, e.g. SELECT, INSERT, UPDATE, DELETE, DROP, on certain objects. Web applications open database connections using a specific account for accessing the database. An attacker who bypasses authentication gains privileges equal to the accounts. The number of available attack methods and affected objects increases when more privileges are given to the account. The worst case is if an account is associated with the system administrator, which normally has all privileges.

- **Dynamic SQL**

Dynamic SQL refers to SQL queries dynamically built by scripts or programs into a query string. Typically, one or more scripts and programs contribute and successively build the query using user input such as names and passwords as values in e.g. WHERE clauses. The problem with this approach is that query building components can also receive SQL keywords and control characters,

creating a completely different query than the intended

- **Stored procedures**

As discussed in section 3.2, stored procedures are statements stored in RDBMSs. The main problem using these procedures is that an attacker may be able to execute them, causing damage to the RDBMS as well as the operating system and even other network components. Another risk is that stored procedures may be subject to buffer overrun attacks. System stored procedures that comes with different RDBMS are well-known by attackers and fairly easy to execute

- **Client-side-only control**

When code that performs input validation is implemented in client-side scripts only, the security functions of those scripts can be overridden using cross-site scripting. This opens for attackers to bypass input validation and send invalidated input to the server-side.

- **INTO OUTFILE**

Support If the RDBMS supports the INTO OUTFILE clause, an attacker can manipulate SQL queries so that they produce a text file containing query results. If attackers can later gain access to this file, they can use information in it in order to e.g. bypass authentication.

- **Sub-Selects**

If the RDBMS supports sub-selects, the variations of attack methods used by an SQL injection attacker increases. For example, additional SELECT clauses can be inserted in WHERE clauses of the original SELECT clause.

- **JOIN/UNION**

If the RDBMS supports JOIN or UNION, the variations of attack methods used by an SQL injection attacker increases. For example, an original SELECT class can be modified with a JOIN SELECT or UNION SELECT clause.

- **Multiple statements**

If the RDBMS supports JOIN or UNION, the variations of attack methods used by an SQL injection attacker increases. For example, an additional INSERT statement could be added after a SELECT statement, causing two different queries to be executed. If this is performed in a login form, the attacker may add himself to the table of users.

### 3.4 Security Layers and SQLI

A web application can, from a hacker's perspective, be viewed as consisting of the following layers: desktop layer, transport layer, access layer, network layer and application layer. At the desktop layer, computers with web browsers acting as clients are used for accessing a system. The transport layer represents the web, and the access layer constitutes the entrance point into a corporation's internal system from the web[27]. The network layer consists of the corporation's internal network infrastructure and finally, the application layer includes web servers, application servers, and application logic and data storage. Every layer may have its own implemented countermeasures in order to detect, prevent and recover from attacks, as shown in figure 3.2.

Unfortunately, SQL injection attacks can only be prevented in application logic components such as scripts and programs in the application layer. No matter how many resources and how much effort a corporation spends in the other layers, if application security has not been properly applied in the application layer, their web applications may contain vulnerabilities that SQL injection attackers can exploit. We do not say that countermeasures like encryption, firewalls, intrusion detection and database security are not important. They are effective when dealing with other types of attacks. However, they have been shown insufficient and ineffective regarding SQL injection and therefore we will not consider them. Encryption for example, only protects stored

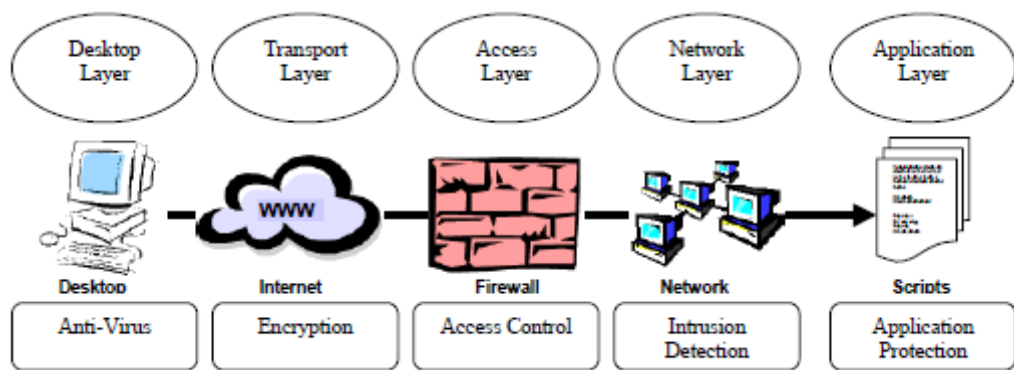


Figure 3.2: Security layers in Web Applications

data or data during transport in and between lower layers. In the context of web applications, user input may be encrypted between the client-side and server-side. Furthermore, SQL queries may be encrypted during transport between components such as scripts and programs on the server-side. But in order for the server-side to construct SQL queries and for RDBMS to execute them, they must first be decrypted. The data may still be encrypted but the SQL queries could have been manipulated through SQL injection. Basically, most web servers are protected by firewalls. However, from a security perspective, web applications offer users legitimate channels through firewalls into corporations systems. The reason for this is that when clients request services from servers on the web, the underlying communication takes place through HTTP, and web applications are no exceptions. HTTP is firewall-friendly, i.e. it is one of the few protocols most firewalls allow through. This stems from the fact that HTTP requests are considered legal, since traffic between clients and servers must be allowed in order for the web applications to be of any use. SQL injection takes advantage of this property by embedding attacks in HTTP requests. These attacks are therefore carried out behind firewalls through the application layer. Therefore, SQL injection requires neither specialized tools nor extensive experience and knowledge. A web browser is sufficient in order to perform SQL injection at-

tacks against web applications, as long as the attacker has basic knowledge of HTTP, relational databases and SQL . Even if the RDBMS is secured through proper configuration, the database can still be vulnerable for SQL injection attacks. In RDBMS, SQL queries are executed as long as they are valid and well-formed and users have the required privileges. Therefore, while security flaws are often to be found in the RDBMS due to improper security configuration.

# Chapter 4

## Proposed Techniques

### 4.1 A Filter Approach to Prevent SQLIA

#### 4.1.1 Observation

After closely observing the behavior of SQL Injection we come to know that in SQLI attack, attackers basically concatenating their special crafted SQL queries with the predefined SQL codes. So by doing this, they actually change the behavior of predefined SQL queries and, the database hence executed the attacker intended queries. The key observation is that, they pass the SQL Meta Characters (SQL expression) through the user input fields like userid, password or by supply it within the url, which is never offered by any web developer. When web developers develop the web application, they created predefined server code for each task; hence they never send any SQL command through user input. So if in the request, which is coming from the user, any SQL command is present then it will always so the intention of SQLI. So if we block the SQL command within the request we can prevent SQLI.



### 4.1.2 Related Work

In this section we have studied and discussed all the existing techniques which have been proposed in this area.

#### 1. Defensive Coding Practice

The root cause of SQL injection vulnerabilities is insufficient input validation. Therefore, the straightforward solution for eliminating these vulnerabilities is to apply suitable defensive coding practices. Here, we summarize some of the best practices proposed in the literature for preventing SQL injection vulnerabilities.[8]

- **Input type checking**

SQLIAs can be performed by injecting commands into either a string or numeric parameter. Even a simple check of such inputs can prevent many attacks. For example, in the case of numeric inputs, the developer can simply reject any input that contains characters other than digits. Many developers omit this kind of check by accident because user input is almost always represented in the form of a string, regardless of its content or intended use. [23]

- **Encoding of inputs**

Injection into a string parameter is often accomplished through the use of meta-characters that trick the SQL parser into interpreting user input as SQL tokens [26]. While it is possible to prohibit any usage of these meta-characters, doing so would restrict a non-malicious user's ability to specify legal inputs that contain such characters. A better solution is to use functions that encode a string in such a way that all meta-characters are specially encoded and interpreted by the database as normal characters.

- **Positive pattern matching**

Developers should establish input validation routines that identify good

input as opposed to bad input. This approach is generally called positive validation, as opposed to negative validation, which searches input for forbidden patterns or SQL tokens. Because developers might not be able to envision every type of attack that could be launched against their application, but should be able to specify all the forms of legal input, positive validation is a safer way to check inputs.

- **Parameterized statements**

With most development platforms, parameterized statements can be used that work with parameters (sometimes called placeholders or bind variables) instead of embedding user input in the statement [1][13]. In many cases, the SQL statement is fixed, and each parameter is a scalar, not a table. The user input is then assigned (bound) to a parameter. This is an example using Java and the JDBC API:

```
PreparedStatement prep = conn.prepareStatement("SELECT * FROM 'users'
WHERE USERNAME=? AND PASSWORD=?");
prep.setString(1, username);
prep.setString(2, password);
prep.executeQuery();
```

Unfortunately prepared statements can also be vulnerable to SQLIAs unless developers rigorously apply defensive coding guidelines.

- **Identification of all input sources**

Developers must check all input to their application. As we outlined in Section 4.1.2, there are many possible sources of input to an application. If used to construct a query, these input sources can be a way for an attacker to introduce an SQLIA. Simply put, all input sources must be checked. Although defensive coding practices remain the best way to prevent SQL injection vulnerabilities, their application is problematic in practice. Defensive coding is prone to human error and is not as rigorously and com-

pletely applied as automated techniques. While most developers do make an effort to code safely, it is extremely difficult to apply defensive coding practices rigorously and correctly to all sources of input. In fact, many of the SQL injection vulnerabilities discovered in real applications are due to human errors: developers forgot to add checks or did not perform adequate input validation. In other words, in these applications, developers were making an effort to detect and prevent SQLIAs, but failed to do so adequately and in every needed location

## **2. Static Code Checkers**

JDBC-Checker is a technique for statically checking the type correctness of dynamically-generated SQL queries. This technique was not developed with the intent of detecting and preventing general SQLIAs, but can nevertheless be used to prevent attacks that take advantage of type mismatches in a dynamically-generated query string. JDBC-Checker is able to detect one of the root causes of SQLIA vulnerabilities in code- improper type checking of input. However, this technique would not catch more general forms of SQLIAs because most of these attacks consist of syntactically and type correct queries. Wassermann and Su propose an approach that uses static analysis combined with automated reasoning to verify that the SQL queries generated in the application layer cannot contain a tautology. The primary drawback of this technique is that its scope is limited to detecting and preventing tautologies and cannot detect other types of attacks. [37]

## **3. Intrusion Detection Systems**

Valeur and colleagues [39] propose the use of an Intrusion Detection System(IDS) to detect SQLIAs. Their IDS system is based on a machine learning technique that is trained using a set of typical application queries. The technique builds models of the typical queries and then monitors the application at runtime to

identify queries that do not match the model. In their evaluation, Valeur and colleagues have shown that their system is able to detect attacks with a high rate of success. However, the fundamental limitation of learning based techniques is that they can provide no guarantees about their detection abilities because their success is dependent on the quality of the training set used. A poor training set would cause the learning technique to generate a large number of false positives and negatives.

#### **4. Instruction Set Randomization**

SQLrand[35] is an approach based on instruction-set randomization. SQLrand provides a framework that allows developers to create queries using randomized instructions instead of normal SQL keywords. A proxy filter intercepts queries to the database and de-randomizes the keywords. SQL code injected by an attacker would not have been constructed using the randomized instruction set. Therefore, injected commands would result in a syntactically incorrect query. While this technique can be very effective, it has several practical drawbacks. First, since it uses a secret key to modify instructions, security of the approach is dependent on attackers not being able to discover the key. Second, the approach imposes a significant infrastructure overhead because it requires the integration of a proxy for the database in the system.

#### **5. Combined Static and Dynamic Analysis**

AMNESIA is a model-based technique that combines static analysis and runtime monitoring[13][40]. In its static phase, AMNESIA uses static analysis to build models of the different types of queries an application can legally generate at each point of access to the database. In its dynamic phase, AMNESIA intercepts all queries before they are sent to the database and checks each query against the statically built models. Queries that violate the model are identified as SQLIAs and prevented from executing on the database. In their evaluation,

the authors have shown that this technique performs well against SQLIAs. The primary limitation of this technique is that its success is dependent on the accuracy of its static analysis for building query models. Certain types of code obfuscation or query development techniques could make this step less precise and result in both false positives and false negatives.

Similarly, SQLGuard [20] also checks queries at runtime to see if they conform to a model of expected queries. In these approaches, the model is expressed as a grammar that only accepts legal queries. In SQLGuard, the model is deduced at runtime by examining the structure of the query before and after the addition of user-input. Additionally, the use of SQLGuard requires the developer to either rewrite code to use a special intermediate library or manually insert special markers into the code where user input is added to a dynamically generated query.

## 6. Taint Based Approaches

WebSSARI detects input-validation related errors using information flow analysis [42]. In this approach, static analysis is used to check taint flows against preconditions for sensitive functions. The analysis detects the points in which preconditions have not been met and can suggest filters and sanitization functions that can be automatically added to the application to satisfy these preconditions. The WebSSARI system works by considering as sanitized input that has passed through a predefined set of filters. In their evaluation, the authors were able to detect security vulnerabilities in a range of existing applications. The primary drawbacks of this technique are that it assumes that adequate preconditions for sensitive functions can be accurately expressed using their typing system and that having input passing through certain types of filters is sufficient to consider it not tainted. For many types of functions and applications, this assumption is too strong.

Livshits and Lam [38] use static analysis techniques to detect vulnerabilities in

software. The basic approach is to use information flow techniques to detect when tainted input has been used to construct an SQL query. These queries are then flagged as SQLIA vulnerabilities. The authors demonstrate the viability of their technique by using this approach to find security vulnerabilities in a benchmark suite. The primary limitation of this approach is that it can detect only known patterns of SQLIAs and, because it uses a conservative analysis and has limited support for untainting operations, can generate a relatively high amount of false positives.

### 4.1.3 Proposed Filter Approach

#### 1. Pattern Identification

A primary technique of SQLIA is to inject SQL meta characters into the input string parameter, so as to trick the SQL parser interpreting the user input as a SQL query. A straightforward solution is to prohibit the usage of SQL Meta characters but doing so would prevent legal inputs that contain such characters. So for each such attack we identify a pattern of the attack. A pattern is a sequence of characters that will always appear in the Request for that particular attack type. We call such pattern the signature of the attack. Our aim is to extract a signature for the known SQLI attacks and then use these signatures to prevent such attacks shown in table 4.1.

#### 2. Proposed Filter Model

Filter is a program that runs on the server before the requested (url pattern) page with which it is associated [4]. A single filter can be attached to one or more pages and can examine the request information going into these resources shown in figure 4.1. After doing so, it can choose among the following options.

- *Authentication*-Blocking requests based on user identity.

Table 4.1: Attack analysis

Type of attack	Signature
Tautologies	(') followed by OR operator
Illegal/logically incorrect queries	('), AND operator, ORDER BY, HAVING clause
Union Query	UNION
Piggy-Backed Query	(;) SQL command terminator
Stored Procedure	(;)SQL command terminator, EXEC
Blind SQLI	AND operator and Conditional operator
Timing Attack	WAITFOR,IF,ELSE
Alternate encoding	Char(),ASCII(),HEX(),UNHEX()

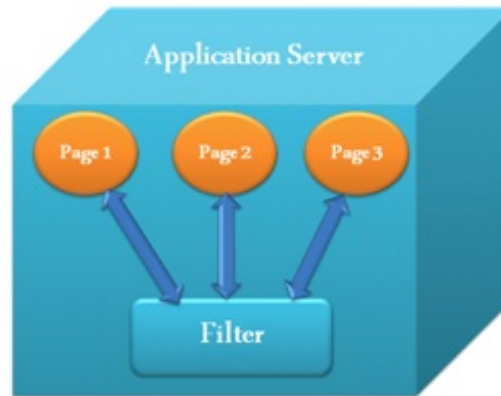


Figure 4.1: Communication between Filter and Application Server.

- *Logging and auditing*-Tracking users of a web application.
- *Image conversion*-Scaling maps, and so on.
- *Data compression*-Making downloads smaller.
- *Localization*-Targeting the request and response to a particular locale.
- *XSL/T transformations of XML content*-Targeting Web application responses to more than one type of client.

These are just a few of the applications of filters. There are many more, such as encryption, tokenizing, triggering resource access events, mime-type chaining, and caching[2]. The great advantage of using filter is that we can make a single filter for many pages. So it enhances the reusability and as well as scalability, the main concern of filter designing is to provide security against the SQLI, generally attackers launch their attacks with the help of URL modification, because of the non sanitized URL the request directly goes to the database server and the database server will act accordingly, so by doing a little modification in the URL, an attacker can take control all over the application. By placing filter between the database server and the request we can actually secure the web application.

The above figure 4.1 clearly shows that if any request will come for the page1, page2 or any page in the application server then it first goes to the filter. After that filter checks the request, if this is a valid request then it returns back to the same page for that request, otherwise it redirects the request to the default error page. So, any changes in the URL will not be considered as the legitimate request and greet with the error page or any message.

### 3. Proposed architecture

This architecture consists of three major building blocks: the clients who send the request, the application server where the logic of the program will be decided and the database server which can store the clients' data for the future



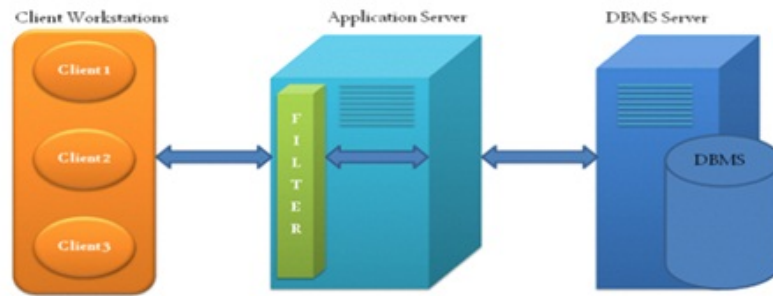


Figure 4.2: A simple 3-Tier Web Application Architecture with Filter.

use. In normal 3-tier web application architecture there is no filter deployed in application server.

In server-side architecture, a user invokes the services provided by the application server using a browser. The services provided by the application server are presented to the user as a GUI. The input provided by the user is usually sent to the application server in the form of a parameter string. The application server uses this input to generate a SQL query to retrieve information from the database or update it. As shown in figure 4.2, our proposed Meta filter is positioned between the user and application server. The filter intercepts the input from the user, parses it into SQL Meta character tokens. In the previous we have listed the various input patterns that can appear in a SQLI. If the input from the user contains any attack pattern (some combination of SQL Meta characters) then the injected input is treated as an attack and an error page is displayed, otherwise the input is processed by the application server normally. As can be seen from figure 4.3, the SQL query and as such the SQL Meta characters are generated by the application server. Our proposed Meta filter checks the presence of SQL Meta characters before the input is processed by the application server. Therefore, our proposed solution will not block legal inputs. Moreover, the attack patterns have been so designed so that it is robust to SQL Meta characters that can accidentally occurs in a legal input.

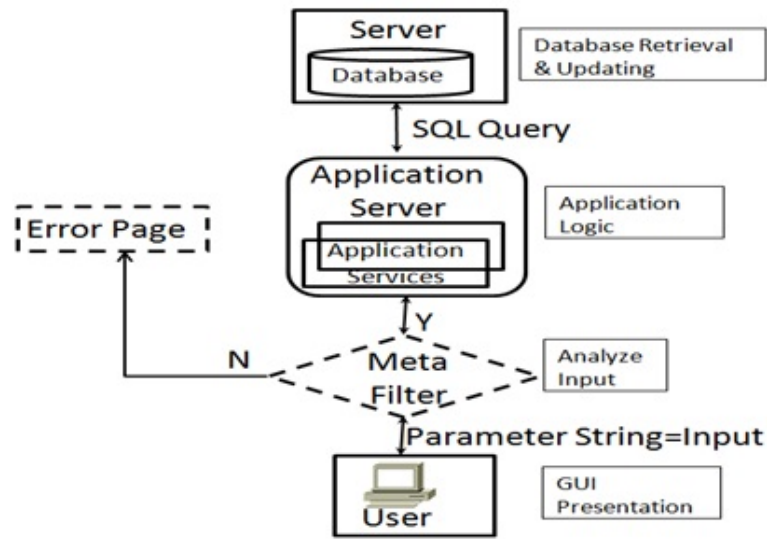


Figure 4.3: Working of Filter.

#### 4. Filter Design

In this section we show how to design a Meta filter using Java code. The input to the filter is the request coming from the user. The detailed steps involved in designing the filter are outlined below: [33]

**Step1: Get url request from User**

```
HttpServletRequest req1 = (HttpServletRequest)req;
```

**Step2: Parse the Input**

```
String qry = req1.getQueryString();
```

**Step3: Check whether attack pattern is present in input**

```
String[] AttackPattern=""," CREATE"," create"," ALTER"," alter"," DROP"," drop",
" RENAME"," rename"," SELECT"," select"," INSERT"," insert","
UPDATE"," update"," DELETE"," delete"," GRANT"," grant"," RE-
VOKE"," revoke"," char"," int"," @@version","@@VERSION"," exec","
EXEC"," union"," UNION"," WAITFOR"," waitfor"," ORDER BY"," or-
der by",";";
int i=0,fnd;boolean chk=true;
```

```

for(i=0;i<AttackPattern.length;i++)
{
fnd=query.indexOf(AttackPattern [i]);
if (fnd!=-1)
{
chk=false; //Attack Pattern Present
break;
}
}
if (chk==true)
{chain.doFilter (req, res); //Process the query
else
{res1.sendRedirect("error.html"); } //Redirect to error page
}

```

## 5. Filter Mapping

We have to little modification in web.xml file which are as following.

```

< filter >
< filter - name > fil < /filter - name >
< filter - class > filter < /filter - class >
< /filter >
< filter - mapping >
< filter - name > fil < /filter - name >
< url - pattern > /* < /url - pattern > //this will map all url in the web
application
< /filter - mapping >

```

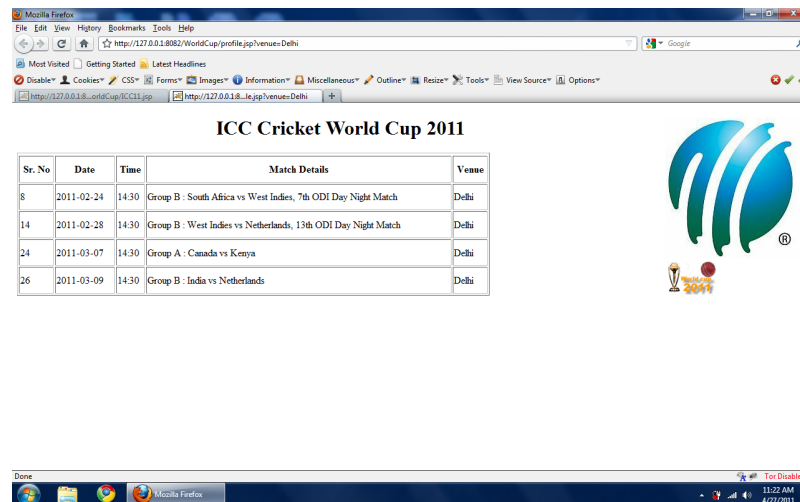
We have designed the filter using java servlet [24]. In step 3 we checked all the SQL meta characters which can appear in the url for SQLIA. In section 3.2 we

have analyzed all the different type of SQLIA and found out the different SQL Meta character required for that attack. In this section we have shown how we are analyzing all the required SQL Meta character to prevent SQLIA. If our filter gets any SQL Meta character in the url attack pattern it will forward the request to the error page otherwise it will greet us with the required page.

#### 4.1.4 Implementation Details and Evaluation

We have implemented the above code in our localhost. For the original web page we have the URL i.e. [http://127.0.0.1:8082/avinash/profile.jsp?venue=Delhi], here venue=Delhi is a query string of the url and venue is the input for this, which means that this profile.jsp page will display the records from the database which have the values Delhi shown in figure 4.4. The query will be represented like this:

*SELECT \* FROM schedule WHERE venue= 'Delhi';* For example, When the attacker



Sr. No	Date	Time	Match Details	Venue
8	2011-02-24	14:30	Group B : South Africa vs West Indies, 7th ODI Day Night Match	Delhi
14	2011-02-28	14:30	Group B : West Indies vs Netherlands, 13th ODI Day Night Match	Delhi
24	2011-03-07	14:30	Group A : Canada vs Kenya	Delhi
26	2011-03-09	14:30	Group B : India vs Netherlands	Delhi

Figure 4.4: Web Application showing records from the Database.

will launch any SQLI attack like we have taken here, the URL will be modified as `http://127.0.0.1:8082/avinash/profile.jsp?venue=Delhi' union all select`

*null, null, column\_name, null, null from information\_schema.columns where table\_name = "user" - "*.

The corresponding SQL query will be

*SELECT \* FROM schedule WHERE venue= 'Delhi' UNION ALL SELECT null, null, column\_name, null, null from information\_schema.columns where table\_name="user" - "*;

If the corresponding server page is not filtering the input values and directly using this to constructing then this query will extract all the column name from the table user, which is the by default table of mysql shown in figure 4.5. If we uses the filter

**ICC Cricket World Cup 2011**

Sr. No	Date	Time	Match Details	Venue
8	2011-02-24	14:30	Group B : South Africa vs West Indies, 7th ODI Day Night Match	Delhi
14	2011-02-28	14:30	Group B : West Indies vs Netherlands, 13th ODI Day Night Match	Delhi
24	2011-03-07	14:30	Group A : Canada vs Kenya	Delhi
26	2011-03-09	14:30	Group B : India vs Netherlands	Delhi

0	null	null	Host	null
0	null	null	User	null
0	null	null	Password	null
0	null	null	Select_priv	null
0	null	null	Insert_priv	null
0	null	null	Update_priv	null
0	null	null	Delete_priv	null
0	null	null	Create_priv	null
0	null	null	Drop_priv	null
0	null	null	Reload_priv	null
0	null	null	Shutdown_priv	null
0	null	null	Process_priv	null
0	null	null	File_priv	null
0	null	null	Grant_priv	null

Figure 4.5: An SQLI Attack.

approach then our filter will intercept the all the request coming from the client and execute itself and perform their task, like here it will prevent the SQLI attack and divert the request to the by default error page shown in figure 4.6. We have implemented a filter which can successfully work with any version of servlet 2.3. For the deployment we have used localhost of apache server. Similarly we can map it globally. The current filter design is fully and perfectly able to block SQL injection

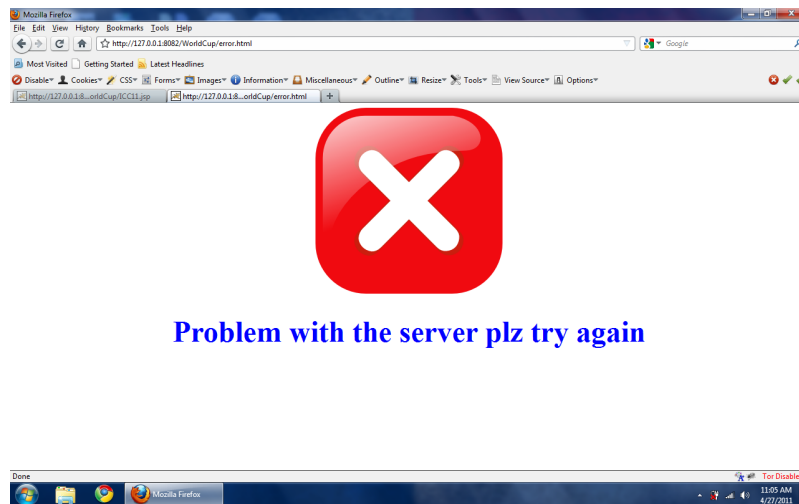


Figure 4.6: Error Page serve by the Filter.

attack without any complexity.

We have implemented it only on jsp and servlet. When the web page is designed using jsp our approach is fully able to block the SQLI attack. With the current approach on defensive coding practice where a developer has to check the all input points to validate it, our approach takes less time for validation.

## 4.2 A Network Based Vulnerability Scanner to Detect SQLIA

### 4.2.1 Observation

After analyzing the trend of SQL injection we found that if before launching the web applications to the world wide, if we come to know that which page is vulnerable and which is not then, we can easily fix it. Web vulnerability scanners do the same task, they provide a automatic report, which tells that which page is vulnerable. So we here proposed a network based vulnerability scanner which will generate report much

faster than existing scanners and also provide a good accuracy ratio almost without any false positive.

### 4.2.2 Related Work

Web Application Vulnerability Scanners are designed to penetrate the web applications against the security issues. They are the automated tools designed in such a way that they will perform the same attack as we do manually, the advantage of using scanners is that they generate the automated report which shows that what are the input points and which are vulnerable.

**Static code analyzers**[14] analyze the code without actually executing it. The analysis performed by existing tools varies depending on their sophistication, ranging from tools that consider only individual statements and declarations to others that consider the complete code. Among other usage (e.g., model checking and data flow analysis), these tools provide an automatic way for highlighting possible coding errors. Research on static code analysis is nowadays very intense, including on the identification of the code patterns that are prone to vulnerabilities like SQL and XPath Injection.

**The FindBugs**[23] tool has been used, as it is widely used in practice, is able to perform a thorough code analysis, is very easy to use, and has a specific module for SQL Injection vulnerabilities. However, results were quite disappointing. In fact, FindBugs was able to mark only 2 in a universe of 22 vulnerabilities that were previously identified by a team of security experts by executing code inspection and penetration tests.

In our literature survey we have gone through various commercial vulnerability scanners, which are being used widely now days, for checking the effectiveness of these tools we have created 5 web applications in localhost and took 3 public web application, and check with these tools. None of the tools provide completeness as well as

they took very large amount of time depending on the size of the web application. The tools we surveyed are as follows:

**Acunetix:** Acunetix WVS automatically checks web applications for vulnerabilities such as SQL Injections, cross site scripting, arbitrary file creation/deletion and weak password strength on authentication pages. AcuSensor technology detects vulnerabilities which typical black box scanners miss. Acunetix WVS boasts a comfortable GUI, an ability to create professional security audit and compliance reports, and tools for advanced manual webapp testing.[10]

**Netsparker:** It can find and report security issues such as SQL Injection and Cross-site Scripting (XSS) in all web applications regardless of the platform and the technology they are built on. Netsparker's unique detection and exploitation techniques allow it to be dead accurate in reporting.[11]

**WebCruiser:** this is an effective and powerful web penetration testing tool, it has a Vulnerability Scanner and a series of security tools. It can support scanning website as well as POC (Proof of concept) for web vulnerabilities: SQL Injection, Cross Site Scripting, XPath Injection etc. So, WebCruiser is also an automatic SQL injection tool, an XPath injection tool, and a Cross Site Scripting tool!.[12]

### 4.2.3 Proposed Network Model

All the vulnerability scanners as we surveyed are based on the standalone system, so they took very much time as well as these also affect the accuracy, because their attack rule library is not very much efficient; due to this we get false positive. So we here tried to make different library for different databases. As this is the network approach shown in figure 4.7, its efficiency will depend on how many system we are using and also how many connection MySQL supports, because for storing our data we are using MySQL in the backend.

In our approach we moved one step forward. We are checking what are the SQL commands actually an attacker can run in the background and as well as the privileges





Figure 4.7: Network Based Vulnerability Scanner

of different commands against different tables and users, so at the same time we are providing a report which is effective or and also we discover problems which comes in coding and as well as database privileges problem.

This model consists of three main parts: Crawler (Scanning), Attack Simulation and the Network Setup.

### 1. Scanning The Whole web Application

To scan the whole web application the basic structure will be represented in the form of a tree [22]. Like as below figure : In the figure 4.8 a.php is represented as the home page of the web application and all the child nodes b.php, c.php, etc are the respective pages of the web application. The Scanning will perform in the following way.

**Step1:** Create a FIFO queue with two fields URL (Primary Key), STATUS.

**Step2:** Insert the target URL and set its STATUS=0.

**Step3:** Update STATUS=1

Send Request for the URL

Analyze the response, Extract its entire links.

Insert these links in FIFO Queue and set STATUS=0.

**Step4:** Go To Step3 While STATUS=0 ELSE Go To Step 5.

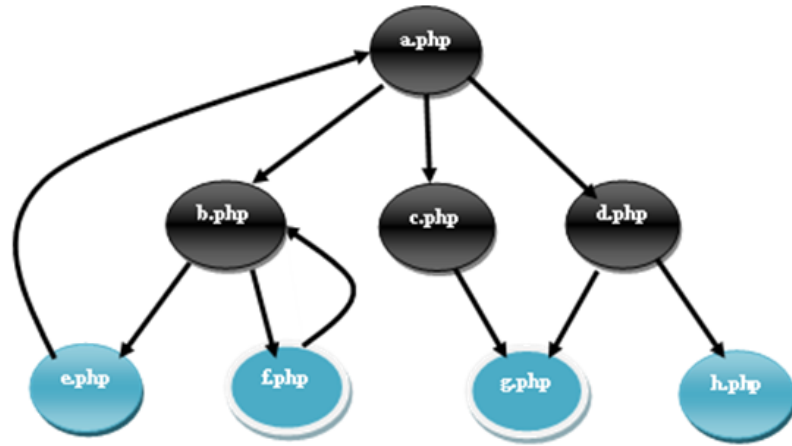


Figure 4.8: Tree structure of a Web Application

**Step5:** Finish.

## 2. Attack Simulation

Attack Simulation consist of three parts: (1) Payload Setup (2) Generating Attack and Response Analysis and (3) Report Generation.

- **Payload Setup** In this phase the attack payload will be created based on the previous existed SQLI attacks, for generating the payload we have created a list in which we have grouped all the common SQLI which an attacker used to reveal the database, for each type of database we have specially modified the attack.
- **Generating Attack and Response Analysis** In this phase we generate the attack by concatenating the attack payload with the original query URL of the web application, and make request of this specially crafted attack URL. Afterward analyze the response and find out the patterns of SQLI inside the responses.
- **Report Generation** On behalf of the response, if any SQLI pattern found

inside this, then the corresponding URL will be added in the report. At the completion of the attack simulation and scanning phase we have automated report, which consist of all the pages list which are vulnerable.

### 3. Network Setup

For connecting different system within a network we can created an ad-hoc network, and connected each system with this, and on each system we have created a RMI server whcih will listen the request comming form the crawler system. By using the RMI(Remote Method Invocation) we will send different urls to the different attack system, on behalf of these urls each system will put attack and check for the vulnerability if any vulnerability will find then it will updates its report.

#### 4.2.4 Implementation Details

In this phase we have shown the practical approach of our implementation, for implementing to this approach we have used JAVA as a programming language and MYSQL as database for storing the data. This consists of three things, (i) Scanner Implementation and (ii) Attack Implementation (iii) Network Implementation.

- **Scanner Implementation**

For scanning the whole web application we have created two function first is SeedUrl which provide the URLs to the Crawling function which have the STATUS=0, Crawling function takes the value from SeedUrl and send request for it after that write the response in file and extract all its link and insert it into the database and set status=0. The overall process will repeat till any URL has STATUS=0. The JAVA code has been given below.[15][33]

```
//SeedUrl Starts here
```

```
Void SeedUrl(){
```

```
Boolea flag=true;
```

```

ResultSet rs=stmt.executeQuery("select distinct status,url from '"+TableName+"'
where status=0");
While(rs.next){
String str=rs.getString("url");//get the url from the database
Crawling(str);//send str to the main crawler
Flag=false;}
If (flag==false)
SeedUrl()
Else
System.out.print("Crawling Complete");
};//SeedUrl() ends here
Crawling(String str){
Update status=1 where url=str;
URL u = new URL(str);
URLConnection uc = u.openConnection(); //Send request for the str to
the application server
FileWriter fw=new FileWriter("URL.txt",false);
//Create a file
while ((ct = r.read( )) != -1)
fw.write((char) ct);//Write the response into a file
String regex="href";
Pattern pattern = Pattern.compile(regex,Pattern.CASE_INSENSITIVE
—Pattern.MULTILINE);
//Finding and extracting all the href attributes
qry="insert into url values(0,"+"'" +href_url+"'"+"")";
stmt.executeUpdate(qry);
//Insert them into the database and set status=0
};//crawling() ends here

```

- **Attack Implementation**

In this phase we have drawn attacks from attack database and then add with the URL got from the Scanner, after that we will make a modified attack URL and will make request for it, the response will write in a file and analysis for SQLI pattern. We have defined various different database errors in the database and will search those errors in the response. On behalf of the SQLI pattern we will summarize it in the form of the report, which will mention that which page, and which parameter is vulnerable and for which reason.

```
// drawn attack from attack library
ResultSet rs=stmt.executeQuery("SELECT * FROM attack");

//Setting up the SQLI attack
while(rs.next())
{String attack_url=url+rs.getString(lib);
URL u = new URL(attack_url); //sending request
URLConnection uc = u.openConnection( );//open connection
FileWriter fw=new FileWriter("Attack.txt",false);//define attack file
while ((ct = r.read( )) != -1)
fw.write((char) ct);//write response in file

//Select Database errors
ResultSet rs1=stmt1.executeQuery("select error from db_error");
while(rs1.next())
{String db_error=rs1.getString("error");
Pattern pt = Pattern.compile
(db_error,Pattern.CASE_INSENSITIVE— Pattern.MULTILINE);
Matcher mat = pt.matcher(stt);//define pattern for each attack
while (mat.find())
{flag=true;
error=db_error;}
```

```

If(flag==true)
{stmt1.executeUpdate("insert into "+table+" values('"+urlss[0]+"',
"+prm1.substring(1)+"', '"+error+"', 'yes')");}
else {
stmt1.executeUpdate("insert into "+table+" values('"+urlss[0]+"',
"+prm1.substring(1)+"', '"+error+"', 'no')");}
} //db_error recordset closed
} //attack recordset closed

```

- **Network Implementation**

### **Interface Setup**

*//Setting up the interface between clint and server*

```

public interface Calculate extends Remote
{ public String GetUrl(String url,String table) throws RemoteException;
public String CreateTable(String tab) throws RemoteException;}

```

### **Server Setup**

*//Listen the request*

```

CalculateEngine ce = new CalculateEngine( );
Naming.rebind("Calc", ce);
System.out.println("Attack Engine ready.");

```

### **Client Bindup**

*//Sending request from client*

```

Object o = Naming.lookup("rmi://10.0.33.35/Calc");
Calculate c = (Calculate) o;
c.GetUrl(str,TableName);//sending url and table info

```

### 4.2.5 Results and Comparison

For showing the effectiveness of our penetration testing tool we have designed 5 different web applications in the local host, and took 3 public web applications. In the manual testing we have tested all these web applications and find different vulnerability, which we have shown here in table 4.2.

Table 4.2: Vulnerability details of different Web Application

<b>Web Application</b>	<b>Domain</b>	<b>Vulnerable Pages</b>
Karnel Travel Guide	LocalHost	No
On line Real State	LocalHost	SQLI(1)
ICC World Cup	LocalHost	SQLI(1)
On line Tutorial	LocalHost	SQLI(1)
Acquiex	LocalHost	SQLI(1)
Inside-india	Public	SQLI(1)
GGI	Public	SQLI(4)
GLA	Public	SQLI(12)
	<b>Total Vulnerability=</b>	<b>21</b>

When we applied the three commercial scanners as we discussed in literature survey in section 4.2.2, and our designed network vulnerability scanner NVS, we got different results as follows. The results shows that none of the tools provides full coverage ratio, and they took much time to generate the report shown in table 4.3. Figure 4.9

Table 4.3: Results comparison

	<b>Acunetix</b>	<b>Netsparker</b>	<b>WebCruiser</b>	<b>NVS</b>
<b>Vulnerability Detected</b>	18/21	16/21	15/21	21/21
<b>Average Time(hr)</b>	1.84	1.20	0.15	0.01
<b>False Positive</b>	12	3	1	0

shows the results for the execution of penetration testing tools and for the NVS tool. As we can see, the different tools reported different numbers of vulnerabilities. An important observation is that the NVS tool was able to identify a much higher number of vulnerabilities than the remaining tools. In fact, all the penetration-testing tools detected less than 85% of the vulnerabilities, while our tool detected all vulnerabilities. Considering only the penetration testing tools, Acunetix identified the higher number of vulnerabilities (85% of the total vulnerabilities). However, it was also the scanner with the higher number of false positives (it detected 12 vulnerabilities that, in fact, do not exist). The very low number of vulnerabilities detected by Webcruiser with the false positive only 1. As different tools detect different sets of vulnerabilities an interesting analysis is how these sets intersect each other.

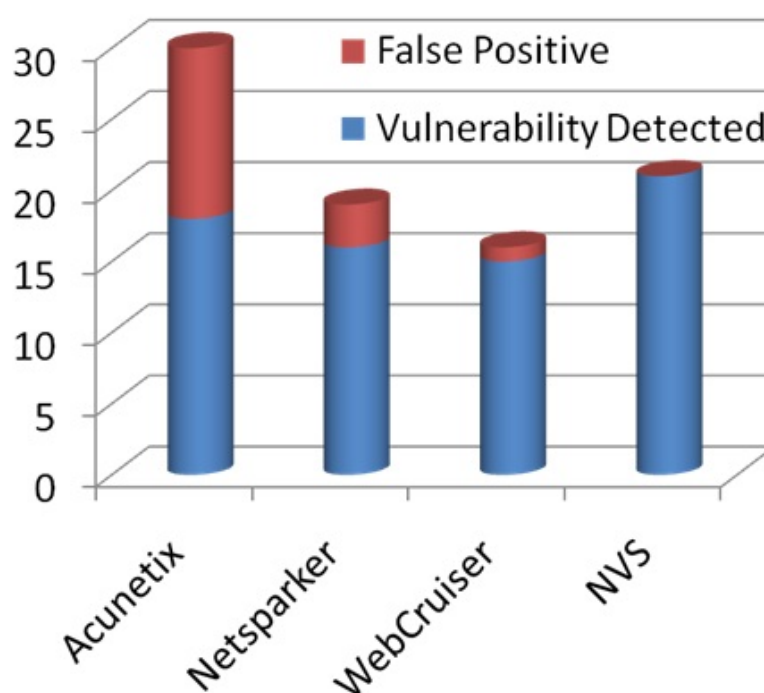


Figure 4.9: Vulnerability Detection Graph

In Figure 4.10, each circle represents the vulnerabilities detected by a tool and each



intersection area represent vulnerabilities found by more than one tool. The circles area is roughly proportional to the represented number, but the same does not happen with the intersection areas, as it would be impossible to represent it graphically. As we can see, there are 21 vulnerabilities that are detected only by NVS. Considering only the remaining 20 vulnerabilities, we observe that Acunetix misses only three. This is interesting, considering that Netsparker detected it, although they presented lower coverage than Acunetix. Additionally, all 15 vulnerabilities were detected by all the testing tools. In Figure 4.11 we have shown the time taken by each graph, it

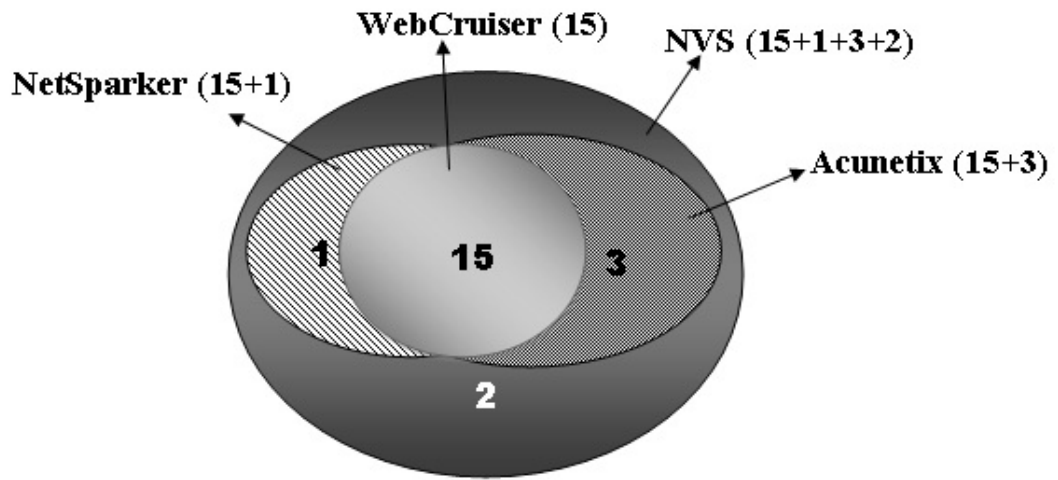


Figure 4.10: Vulnerability covered by each tool

shows that NVS takes very less time around 6 minute, which is magnificent among all tools. At the same time NVS provide a good coverage without any false positive, we have tested it in the network of two system, but its efficiency can be improved if more system will connect in the network.

***NVS(Efficiency)  $\propto$  No of System in the Network***

To conclude, our approach has three key advantages when compared to penetration testing:

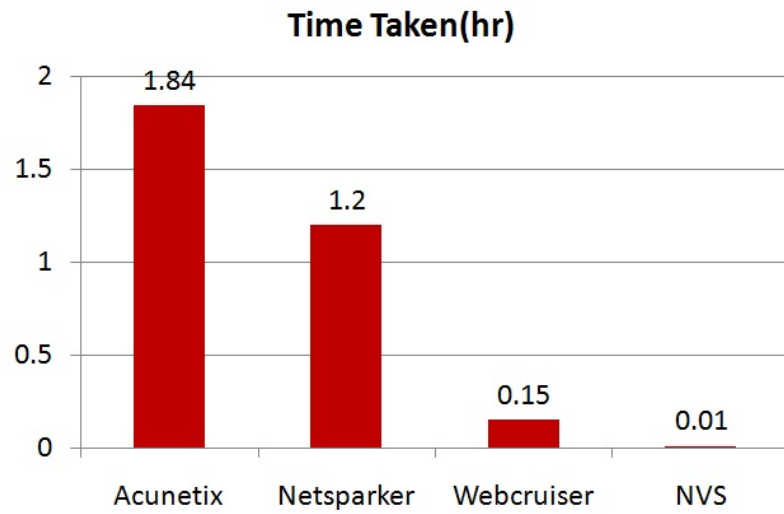


Figure 4.11: Time taken by each tool

- Comparing to vulnerability scanners, we generate a more complete workload to exercise the Web Application and learn the SQL commands issued in the presence of valid inputs.
- The attacks are a compilation of the attacks performed by a large set of commercial and open source web vulnerability scanners plus many methods found in the literature.
- We identify vulnerabilities based on the analysis of the SQL commands actually (to detect the injection of code) and not on the Web Application response, as is the case of the vulnerability scanners.

## Chapter 5

# Conclusion and Future Work

In our thesis we have proposed two different methods to handle the SQLI in web application, in our first approach we have designed and implemented a filter, which prevent the SQLI at the running time, it intercepts all the request coming from the client side and find the SQLI pattern inside that. If any pattern matches it redirects the request to the by default error page. So an attacker never be able to reach to the back end database. We have implemented it in JAVA, and it is has blocked all types of SQLI attacks. In our second approach we have designed a network based vulnerability scanner which shows all the pages which are vulnerable to the SQLI. It saves the web developer's time, because a developer need to check only the vulnerable pages to fix those. Our proposed tool shows a good detection ratio with no false positive. Its report generation time is very less in comparison to other vulnerability scanners. The efficiency of the scanner is dependent to the number of systems attached to the network, if more systems are connected it will generate report more faster.

In our proposed Network Vulnerability Scanner(NVS), we have made attack part distributed which executed on different systems connected within the network, so in future we will try to make whole system distributed including the crawling part. So that it will enhance the quality of the NVS. In future we wish to make our NVS as web service so anybody can use it.

# Bibliography

- [1] Technical report, Wikipedia. [http://en.wikipedia.org/wiki/SQL\\_injection](http://en.wikipedia.org/wiki/SQL_injection).
- [2] The essentials of filters. Online document, Oracle. <http://www.oracle.com/technetwork/java/filters-137243.html>.
- [3] Hyper text markup language(html) home page. Online documentation, W3C. <http://www.w3.org/MarkUp/>.
- [4] Servlet and jsp filters. Online document, Sun Microsystems and Prentice Hall. <http://www.moreservlets.com/>.
- [5] Sql injection are your web applications vulnerable? Online document, HP. [https://products.spidynamics.com/asclabs/sql\\_injection.pdf](https://products.spidynamics.com/asclabs/sql_injection.pdf).
- [6] Writing secure web applications. Online documentation, Advosys Consulting Inc., june 2002. <http://advosys.ca/papers/web/61-web-security.html>.
- [7] The critical need to secure the web in your company. An osterman research white paper, Webroot, Inc., Jan 2010. [http://www.ostermanresearch.com/whitepapers/or\\_or0210a.pdf](http://www.ostermanresearch.com/whitepapers/or_or0210a.pdf).
- [8] Owasp secure coding practices quick reference guide. Online document, Open Web Application Security Project, 2010. [https://www.owasp.org/index.php/Category:OWASP\\_Code\\_Review\\_Project](https://www.owasp.org/index.php/Category:OWASP_Code_Review_Project).

- [9] The ten most critical web application security risk. On-line document, Open Web Application Security Project, 2010. [https://www.owasp.org/index.php/Category:OWASP\\_Top\\_Ten\\_Project](https://www.owasp.org/index.php/Category:OWASP_Top_Ten_Project).
- [10] Acunetix web vulnerability scanner. Technical report, Acunetix, 2011. <http://www.acunetix.com/vulnerability-scanner/>.
- [11] Netsparker web vulnerability scanner. Technical report, Netsparker of Mavituna Security Ltd., 2011. <http://www.mavitunasecurity.com/netsparker/>.
- [12] Webcruiser web vulnerability scanner. Technical report, WebCruiser of Janus Security, 2011. <http://sec4app.com/WebCruiser-Web-Vulnerability-Scanner.htm>.
- [13] Kasra Amirtahmasebi, Seyed Reza Jalalinia, and Saghar Khadem. A survey of sql injection defense mechanisms. *The 4th International Conference for Internet Technology and Secured Transactions (ICITST-2009)*, pages 1–8, Nov 2009.
- [14] Nuno Antunes and Marco Vieira. Comparing the effectiveness of penetration testing and static code analysis on the detection of sql injection vulnerabilities in web services. *15th IEEE Pacific Rim International Symposium on Dependable Computing*, pages 301–306, Oct 2009.
- [15] Ivan Bayross. *Teach Yourself SQL & PL/SQL Using ORACLE 8I & 9I with SQLJ*. BPB, first edition, 2003.
- [16] Cesar Cerrudo. Manipulating microsoft sql server using sql injection. Online document, APPLICATION SECURITY, INC. [http://www.appsecinc.com/presentations/Manipulating\\_SQL\\_Server\\_Using\\_SQL\\_Injection.pdf](http://www.appsecinc.com/presentations/Manipulating_SQL_Server_Using_SQL_Injection.pdf).
- [17] Robert Chartier. Application architecture: An n-tier approach - part1. Online documentation, 2001. <http://www.15seconds.com/issue/011023.htm>.
- [18] Justin Clarke. *SQL Injection Attacks and Defences*. Syngress Publishing, Inc., 2009.

- [19] Sacha Faust. Soap web service attacks. White paper, SPI Dynamics, 2003.
- [20] Buehrer Gregory T., Weide Bruce W., and Sivilotti Paolo A., G. Using parse tree validation to prevent sql injection attacks. *International Workshop on Software Engineering and Middleware (SEM)*, 2005.
- [21] Jacques Guillaumier. Web services - the technology and its security concerns. White paper, Acunetix, May 2007. [http://www.ostermanresearch.com/whitepapers/or\\_or0210a.pdf](http://www.ostermanresearch.com/whitepapers/or_or0210a.pdf).
- [22] Yang Haixia and Nan Zhihong. A database security testing scheme of web application. *IEEE 4th International Conference on Computer Science and Education*, pages 953–955, July 2009.
- [23] William G.J. Halfond, Jeremy Viegas, and Alessandro Orso. A classification of sql injection attacks and countermeasures. *14th ACM SIGSOFT international symposium on Foundations of software engineering*, March 2006.
- [24] Jason Hunter and William Crawford. *Java Servlet Programming*. O’Reilly & Associates, Inc, first edition, 1998.
- [25] Maguire John R. and Gilbert Miller H. Web application security from reactive to proactive. *IEEE Computer Society*, 2010.
- [26] NTAGW ABIRA Lambert and KANG Song Lin. Use of query tokenization to detect and prevent sql injection attacks. *The 5th International Conference for Internet Technology and Secured Transactions (ICITST-2010)*, 2:438–440, July 2010.
- [27] Uzi Ben-Artzi Landsmann and Donald Stromberg. Web application security: A survey of prevention techniques against sql injection. Master’s thesis, Department of Computer and Systems Sciences Stockholm University / Royal Institute of Technology, June 2003.

- [28] David Litchfield. Data-mining with sql injection and inference. Online document, Next Generation Security Software Ltd, 30 September 2005. [www.databasesecurity.com/webapps/sqlinference.pdf](http://www.databasesecurity.com/webapps/sqlinference.pdf).
- [29] Joe McCray. Advanced sql injection. Technical report, Learn Security Online.
- [30] Stefano Di Paola. And more advanced sql injection. Online document, Wise SECurity, 20 September 2004. [http://www.wisec.it/en/Docs/and\\_more\\_sql\\_injection.pdf](http://www.wisec.it/en/Docs/and_more_sql_injection.pdf).
- [31] Kapil Raina. Trends in web application security. Technical report, Symantec, 26 Oct 2004. <http://www.symantec.com/connect/articles/trends-web-application-security>.
- [32] Raghu Ramakrishnan and Johannes Gehrke. *Database Management Systems*. McGraw-Hill Higher Education, second edition.
- [33] Herbert Schildt. *Java2 The Complete Reference*. McGraw-Hill Higher Education, fifth edition, 2002.
- [34] Kevin Spett. Blind sql injection. Online document, SPI DYNAMICS, 2005. [www.securitydocs.com/pdf/2651.PDF](http://www.securitydocs.com/pdf/2651.PDF).
- [35] Boyd Stephen W. and Keromytis Angelos D. Sqlrand: Preventing sql injection attacks. *The 2nd Applied Cryptography and Network Security (ACNS) Conference*, pages 292–302, June 2004.
- [36] Dafydd Stuttard and Marcus Pinto. *The Web Application Hackers Handbook: Discovering and Exploiting Security Flaws*. Wiley Publishing, Inc., 2007.
- [37] Atefeh Tajpour, Maslin Massrum, and Mohammad Zaman Heydary. Comparison of sql injection detection and prevention techniques. *2nd International Conference on Education Technology and Computer (ICETC 10)*, 5:174–179, june 2010.

- [38] Livshits V., B. and Lam M., S. Finding security vulnerabilities in java applications with static analysis. *The 14th Usenix Security Symposium*,, pages 271–286, August 2005.
- [39] Fredrik Valeur, Darren Mutz, and Giovanni Vigna. A learning-based approach to the detection of sql attacks. *The 33rd Annual Symposium on Principles of Programming Languages (POPL 2006)*, Jan 2006.
- [40] Halfond William G., J. and Alessandro Orso. Amnesia: Analysis and monitoring for neutralizing sqlinjection attacks. *The IEEE and ACM International Conference on Automated Software Engineering (ASE 2005)*, Nov 2005.
- [41] James Woodger. General web architecture. Web advisory, Woodger Computing Inc., Jan 2002. <http://www.woodger.ca/archweb.htm>.
- [42] Huang Y., Yu F., Hang C., Tsai C., H., Lee D., T., and Kuo S., Y. Securing web application code by static analysis and runtime protection. *The 12th International World Wide Web Conference (WWW 04)*, May 2004.



# Glossary

We here presenting some definitions which we have used throughout this thesis.

**Access Control:** The ability to limit and control the access to host systems and applications via communications links. To achieve this control, each entity trying to gain access must first be identified, or authenticated, so that access rights can be tailored to the individual.

**Authentication:** A mechanism that determines whether a user is, who he or she claims to be.

**Authorization:** The granting of a right or a privilege, which enables a subject to have legitimate access to a systems object.

**Availability:** The property that a procedures services are accessible when needed and without undue delay.

**Buffer Overflow:** A buffer overflow is when a program allocates a block of memory of a certain length and then tries to stuff too much data into the buffer, with the extra overflowing and overwriting possibly critical information crucial to the normal execution of the program.

**Confidentiality:** Prevention of unauthorized disclosure of information.

**Cross-site scripting** malicious script that runs with the privileges of a legitimate script originating from the legitimate web server.

**Database Server:** A server on which a DBMS runs.

**Decryption:** The process of decoding encrypted data.

**DBMS:** A software system that enables users to define, create and maintain the database and provides controlled access to this database.

**Encryption** The encoding of the data by a special algorithm that renders the data unreadable by any program without the decryption key.

**Extranet:** An intranet that is partially accessible to authorized outsiders.

**Firewall:** Any security system protecting the boundary of an internal network.

**HTML:** HTML is the lingua franca for publishing hypertext on the World Wide Web. It is a non-proprietary format based upon SGML.

**HTTP:** The Hypertext Transfer Protocol (HTTP) is an application-level protocol with the lightness and speed necessary for distributed, collaborative, hypermedia information systems. HTTP has been in use by the World-Wide Web global information initiative since 1990.

**HTTPS:** The secure hypertext transfer protocol (HTTPS) is a communications protocol designed to transfer encrypted information between computers over the World Wide Web. HTTPS is http using a Secure Socket Layer (SSL). A secure socket layer is an encryption protocol invoked on a Web server that uses HTTPS.

**Integrity:** Prevention of unauthorised modification of information.

**Intranet:** A Web site or group of sites belonging to an organization, accessible only by the members of the organization.

**OWASP:** The Open Web Application Security Project (or OWASPpronounced OH WASP) was started in September of 2001. OWASP is an open source reference point for system architects, developers, vendors, consumers and security professionals involved in Designing, Developing, Deploying and Testing the security of web applications and Web Services. In short, the Open Web Application Security Project aims to help everyone and anyone build more secure web applications and Web Services.

**Prepared Statement:** While prepared statements can be implemented in various ways, we prefer the concept as it is defined and used by Sun Microsystems. Using Java technologies, a prepared statement is an object that is given an SQL statement when it is created. In most cases, this SQL statement will be sent to the RDBMS right away, where it will be compiled. As a result, the prepared statement object contains not just an SQL statement, but an SQL statement that has been precompiled. This means that when the prepared statement is executed, the RDBMS can just run the prepared statements SQL statement without having to compile it first. According to Sun Microsystems, the main advantages relate to performance factors.

However, we have found that for prepared statements that require arguments, the parameters sent will be considered illegal if they contain SQL keywords. Therefore, prepared statements are considered more secure than dynamic SQL.

**RDBMS:** An RDBMS which supports the relational model.

**SQL:** SQL (Structured Query Language) is a standard interactive and programming language for getting information from and updating a database. Although SQL is both an ANSI and an ISO standard, many database products support SQL with proprietary extensions to the standard language. Queries take the form of a command language that lets you select, insert, update, find out the location of data, and so forth.

**SQL Injection:** (also named Direct SQL Command Injection, Query Poisoning, Malformed SQL, SQL Tampering, SQL Insertion and One-Way SQL Hacking) A technique that enables malicious users to exploit web applications and bypass control mechanisms in order to gain access to and manipulate information assets outside their privileges.

**Threat:** Any situation or event, whether intentional or unintentional, that will adversely affect a system and consequently an organization.

**URL:** A string of alphanumeric characters that represents the location or address of a resource on the Internet and how that resource should be accessed.

**Vulnerability:** A weakness in computer-based system that may be exploited to cause loss or harm.

**Web Application:** Web-based business applications that use a relational database for persistent storage of data.

**Web Services:** A collection of functions that are packaged as a single entity and published to the network for use by other programs. Web services are building blocks for creating open distributed systems, and allow companies and individuals to quickly and cheaply make their digital assets available worldwide.