CHAPTER-1
INTRODUCTION

1.1 Background

1.1.1 Corporations and 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[6].

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 mentioned above. 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[6,9].

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, application logic components process input, perform calculations and generate output based on received and stored data[6,7].

1.1.2 Security

Historically, information security was primarily concerned with administrative and physical means, e.g. filing cabinets where documents could be locked in. With the widespread usage of data processing equipment, an evident need for protecting files and other information stored on computers began to concern information security. This resulted in the generic name computer security, which encompasses the collection of tools designed to protect data and defend systems against threats.

Computer security is the prevention and detection of unauthorized actions performed by users of a computer system[3]. In general, three key objectives or security services are pointed out: confidentiality, integrity and availability, in order to protect assets in computer systems with respect to security services, several mechanisms have been invented, including encryption, authentication and access control. When implementing computer security in systems one must consider important design parameters, including within which layers security mechanisms should be implemented.

The field of computer security has evolved over time. When the only means of compromising data was by infecting a personal computer with a virus contained on a floppy disk, desktop security was applied in parallel with the expansion of the Internet corporations developed internal and external networks, resulting in a need for network security. As corporations intensified the offering of services through applications, computer security reached its current age, also concerning the application layer of systems: application security[6].

Security professionals and corporations have traditionally spent a major part of their security efforts and resources on operating system and network security. Assessment services heavily relied on automated tools to find holes in those layers. Conventional security measures typically included network monitoring and logging, authentication protocols, firewalls, intrusion detection systems and encryption techniques. Furthermore, special security measures, e.g. access control mechanisms, have been integrated in Database Management Systems (DBMS), to ensure database security. As a result, network inherent components such as routers and web servers as well as operating systems and DBMSs are in general easy to protect. Attack methods aimed at these components have been known for some time and standardized countermeasures have been developed and implemented to prevent and detect such threats effectively[6,7,9].

1.1.3 Web Applications and Security

The increased accessibility to corporations' systems through web applications has also had an impact on the ever increasing need for computer security. Web applications pose unique security challenges to businesses and security professionals in that they expose the integrity of their data to the public[9].

Web application security is a concept that originates from application security and mainly comprises of threats that exploit web application vulnerabilities. Among the new threats that have emerged, many concern the application layer of web applications, including illegal access to information, data manipulation and theft. Several attempts have been made to identify and classify the threats[1]. Among others, the threats are divided into groups aiming at different levels of abstraction within the attacked system. Moreover, instead of focusing on abstraction levels, \ the vulnerabilities can be categorized into a list of broad types. Regardless of the approach chosen, the classified threats all concern the application layer.

Basically, most web servers are separated from clients by firewalls. However, from a security perspective, web applications offer users legitimate channels through firewalls into corporations' systems. When launched from within the application logic, it is harder to detect attacks and protect the application form these attacks[9].

The traditional approach has therefore been proven insufficient for systems that offer services through web applications. While conventional security measures were sufficient for older systems and applications, they seem to be both outdated and ineffective in compensating for vulnerabilities in web applications. During the last years, increases of vulnerabilities inherent in web applications have been noted and reports of attacked systems have frequently published.

A large community of attacker's uses their knowledge and experience frequently to find application layer vulnerabilities in order to commit attacks on organizations' systems through that layer[3].

1.1.4 Web Applications and Data Storage

The most serious threats that web applications may be exposed to, concern attacks aimed directly at data storage. Web servers communicate with back-end running systems such as Relational Database Management Systems (RDBMS), that offer persistent storage of data in relational databases. Relational databases are crucial components in web applications since the most valuable information assets, corporate and customer data are stored there. Successful attacks can cause database content and structure to be exposed, manipulated and compromised [8].

Even though extensive database security mechanisms have been implemented, every web application must act upon the channels mentioned above and make sure

that only legitimate usage is allowed. Unfortunately, lack of security awareness while implementing database connections can lead to open holes in the system. [6,7].

While security flaws in the RDBMS are most certainly often to be found due to improper security configuration, a common mistake is to think of the RDBMS as being improperly secured. Unfortunately, a RDBMS accepts any valid and well-formed query built using a Structured Query Language (SQL), and as long as users have the required privileges, queries are executed on a relational database that contains the data and the result is returned. This property is one factor contributing to a new menace against relational databases connected to the web.

A chain is never stronger than its weakest link. The RDBMS, web server and application logic components are all parts of a larger system, the web application. A strong security policy for the database therefore cannot compensate for poor security in the application logic, since the overall security in the system will be equal to the component with the weakest security.

1.1.5 SQL Injection

SQL injection is a particularly dangerous threat that exploits application layer vulnerabilities inherent in web applications. Instead of attacking instances such as web servers or operating systems, the purpose of SQL injection is to attack RDBMSs, running as back-end systems to web servers, through web applications.

More specifically, attackers can bypass existing security mechanisms implemented to enforce security services, and may therefore gain access to and manipulate information assets outside their privileges. This is accomplished by modifying input parameters expected in fields of forms embedded in web pages, in order to change the underlying queries built with SQL and passed to the database through the web server. Another method is to insert arbitrary SQL directly in the query string portion of an URL in the address field of web browsers.

SQL injection requires neither specialized tools nor extensive experience and knowledge. A web browser is sufficient in order to perform SQL injection attacks against web applications, as long as the attacker has basic knowledge of HTTP, relational databases and SQL. SQL queries are executed in RDBMSs 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, Andrew and Peikari and Fogie and Liu conclude that one must instead

consider that database security as a single measure is not sufficient to guarantee protection of data in web applications.

Every web application, using a relational database, can theoretically be a subject for SQL injection attacks. Those databases usually contain corporations' most valuable information assets: corporate and customer data. Those data are vital for the functions of a corporation's web applications, but often even more crucial and valuable for the corporation itself: user credentials, sensitive financial information, preferences, invoices, payments, inventory data etc. If successful, SQL injection attacks may therefore result in exposure of and serious impact on the corporations most valuable information assets. These attacks may in the worst case result in a completely destroyed database schema, which in turn may a.ect a corporation's ability to perform business[8].

1.2 Problem Statement

Most SQL injection attacks are executed through an application that takes user-supplied input for query parameters. The attacker supplies a carefully crafted string to form a new query with results very different from what the application developer intended. For example, consider a script on a web site that takes a search parameter like Zip code to return selected results from a database. A very simple attack may be possible by simply providing something, like "1OR 1=1" in the text field, which causes the SQL server to return all records from a particular table. An attacker can often gain access to anything available with the script's privileges, which is often full access to one or more databases.

While SQL injection attacks could be executed against any application, web applications are the most commonly vulnerable. The attacker can easily explore a site for vulnerabilities without being caught or having to work through sophisticated network intrusion techniques as most prospective targets leave their web site applications wide open. Firewalls and traditional network intrusion detection systems are useless against SQL injection since it is an application exploit that in most cases is indistinguishable from expected use.

SQL injection affects every database on every platform. Attacks can be used to gain information disclosure, to bypass authentication mechanisms, to modify the database, and, in some cases, to execute arbitrary code on the database server itself! This

research work will examine ways to build an intrusion detection system specifically designed to be situated at the database server to detect SQL injection attack.

1.3 Research Objectives

Objectives of this research work are:

- To show how SQL statement can be injected and attacked
- To detect the query injection.
- To prevent the injected query executing on server

1.4 Literature Review

Use of web application is increasing rapidly everywhere. Numerous research works have been conducted and conclusions are published in papers or internet. We have already discussed that where web applications are applicable, in background. Companies and organizations use web applications to provide a broad range of service to users. Database of web application contains the important information of that organization like customer and financial record; these applications are frequent target for attacks. There are numerous attacking technique, one of them is SQL Injection, can give attackers a way to leak, modify and some time delete information that is stored in the database. In other words SQL Injection can occur when a web application receive user input and use it to build a database query without validating it. Consequently, SQLIAs could be prevented by a more rigorous application of defensive coding techniques.

In our research work we mainly concentrate on the technique to remove the necessity of the source code modification as well as to minimize the runtime response time for both plain query as well as embedded query.

CHAPTER-2
WEB APPLICATIONS

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 give our own model of a general web application which we will refer to 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.

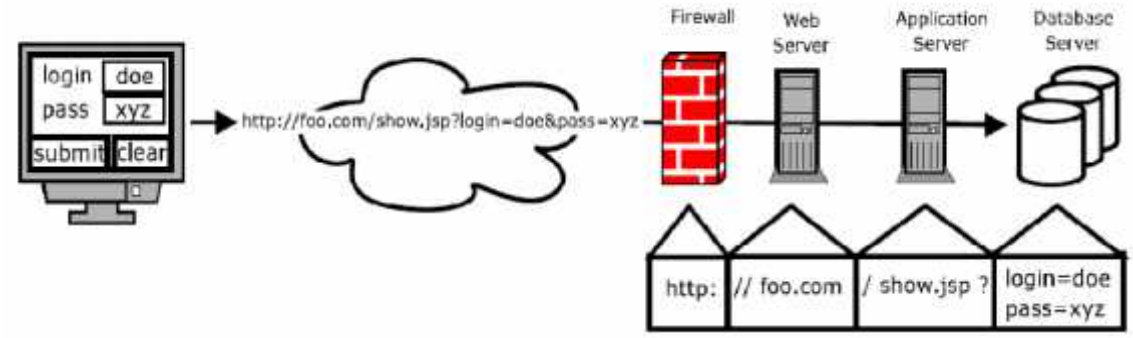

Figure 2.1: Example of interaction between a user and a typical web application.[9]

People that browse the web use web applications in one form or another though the everyday web user may not be aware of that fact because of the ubiquity of web applications:

When one visits cnn.com and the site automatically knows you are a US resident and serves you US news and local weather, it is all because of a web application. When you transfer money, search for a flight, check out arrival times or even the latest sports scores online, you are using a web application[9].

2.1 Web Services

Before proceeding, we think it is justified to mention a few words about the concept of web services. Web services or the similar term inter-web applications, is subject for

an ongoing discussion that treats web services as either the largest technology breakthrough since the web itself or simply further evolved web applications. The standpoint taken in this matter will not have an impact in this thesis but the differences and similarities between the concepts of web services and web applications will. The two concepts both ultimately face the same security issues and taking this under consideration, SQL injection, and therefore our results, is of equal relevance to web services. 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.

2.1.1 Business Web Applications

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.

Web applications provide connectivity, access to information and online services for users. Web applications can today be implemented in various degrees of complexity, and each implementation has a distinct purpose: "... 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, 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[1,6,9].

2.2 Architecture

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.

2.2.1 Client-Server

The 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, as mentioned by Connolly et al., 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[10].

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.

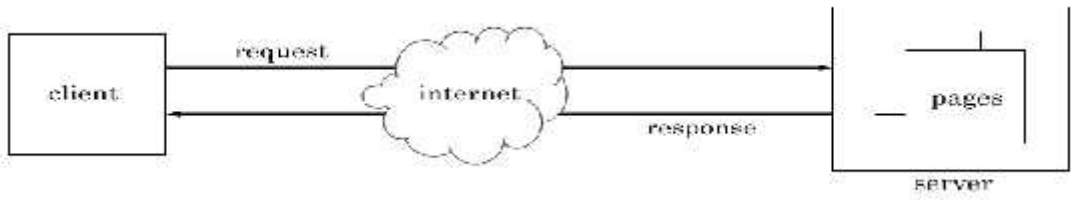

Figure 2.2: Static Web Site[1]

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.

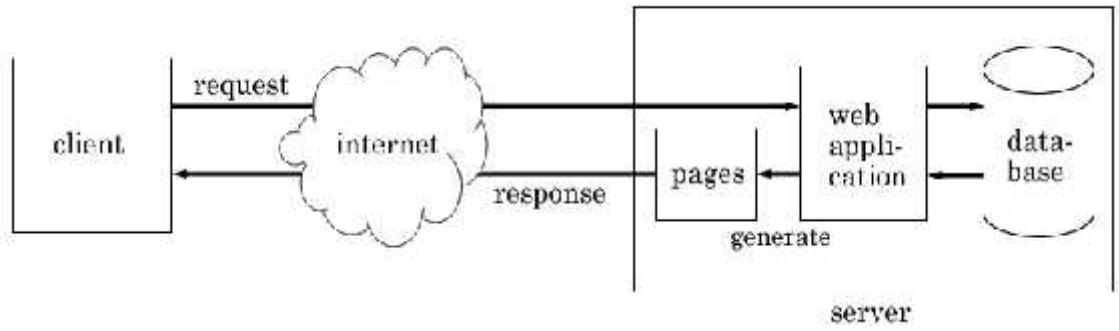

Figure 2.3: Dynamic Web Applications[3]

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.

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. While the different tiered approaches turn out to be irrelevant in our general model, as explained in section 2.2.4, we think that they are worth mentioning for the sake of clarity.

Two-Tier Architecture

In the two-tier client-server architecture, as the basic model for separating tasks, 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[10].

Three-Tier Architecture

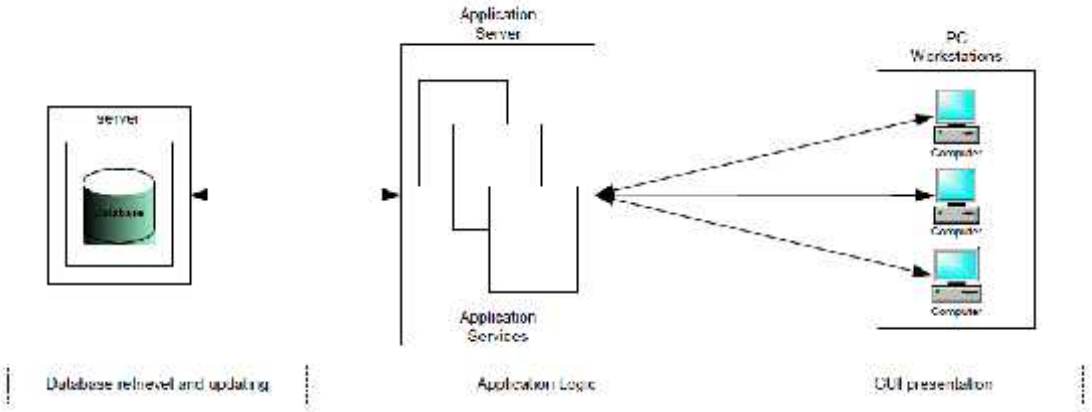

Figure 2.4: 3-tier web application model[4]

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.

2.2.4 General Web Application Architecture

Web applications all contain the three logical layers. Since we in this thesis intend to center our discussions on one consistent model, we will discard arguments for different tiered models, i.e. how layers are composed into a number of tiers, and instead concentrate on the processes inside and between the different layers. We have also found several reasons that support our intention.

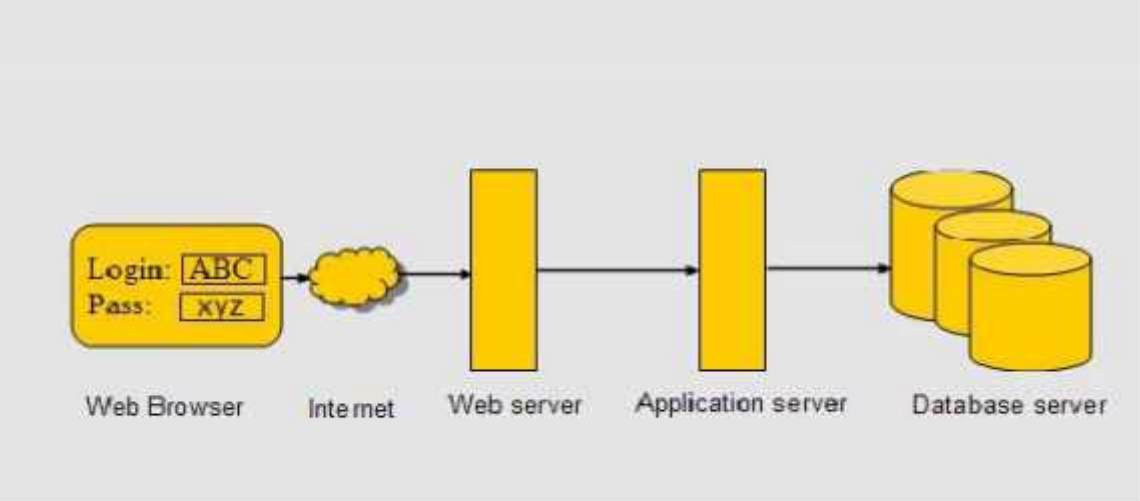

Figure 2.5: Architecture of a typical Web based system[4]

A web system consists of a web browser at the user end. The user is connected to the web application through the internet. A firewall protects the web system from intrusion and allows traffic at port 80 only. The web server receives request from the browser, processes them and passes the dynamic part to the application server, which

processes server side code like JSP. All requests for database access are passed to the database server. The results are then moved back to the web browser as HTML web pages.

2.3 Web Application and SQL Injection

Most of the current web applications use RDBMS. Sensitive information like credit card num, social security num are stored in these databases. Web application allows users to view, edit, or store information in RDBMS through programs written by web application programmers, who may be unaware of technique for writing secure code. They may focus on implementing desired functionalities and focus less on security aspects. This will results in vulnerabilities in web application, Attackers would send SQL to interact with RDBMS servers or modify existing SQL to retrieve unauthorized information without any authentication. If the application is open source and the attacker is able to gain source code through other means, he can analyze the code to find out vulnerabilities to make attacks and hence can modify the database.

CHAPTER-3
SQL INJECTIONS

3.1 Introduction

3.1.1 Scope

SQL injection is a technique used for manipulating server-side scripts that send SQL queries to an RDBMS. This is done by manipulating client-side data, including changing SQL values and concatenations of SQL statements, which are sent to a web server embedded in HTTP requests. Once the web server receives a request, it forwards the information in it to a script which uses that information to build SQL queries. The goal of the attacker who uses SQL injection is to manipulate with the SQL query used by the script so that it would yield unwanted results, such as fetching, inserting, manipulating or deleting protected rows or tables in the database.

Attack methods of SQL injection have by some authors been classified into direct and indirect attacks.

Using direct attacks, an attacker tries to take control of an RDBMS. The purpose of such attacks is to further take control of other host computers and compromise a network. First, attackers scan for open ports that database servers are listening to. If such ports are found, they continue with executing system commands through a command console, communicating with the RDBMS directly.

Indirect attacks, on the other hand, are performed through web applications. True, it is possible to execute commands by embedding calls to stored procedures in dynamic SQL and that may cause devastating results if successful. However, the main purpose is to directly attack the RDBMS in general and its stored data in particular.

A majority of the authors do not mention or discuss direct attacks. This may stem from the fact that they either are not aware of such flaws or that they do not consider them as falling into the scope of SQL injection. Regardless of the reason, direct attacks are conducted through the RDBMS and aims at the network infrastructure. When discussing direct attacks, authors refer to direct communication with the RDBMS and not attacks on the RDBMS itself. It seems to be true that attackers can take advantage of some aspects of SQL injection when performing such

attacks. However, we consider the concept of direct attacks to be somewhat misleading since it does not relate to web applications. Furthermore, from a security perspective, we think that direct attacks relate to network security rather than application security. Flaws like open ports that allow attackers to communicate with the RDBMS using arbitrary protocols from command consoles can be prevented by existing network security countermeasures as well as database security configuration, e.g securing the system administrator account. Therefore, we consider direct attacks to be outside the scope of this thesis.

3.1.2 Basics

web application can, from a attacker'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. The network layer consists of the corporation's internal network infrastructure and finally, the application layer includes web servers, application servers, 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[6].

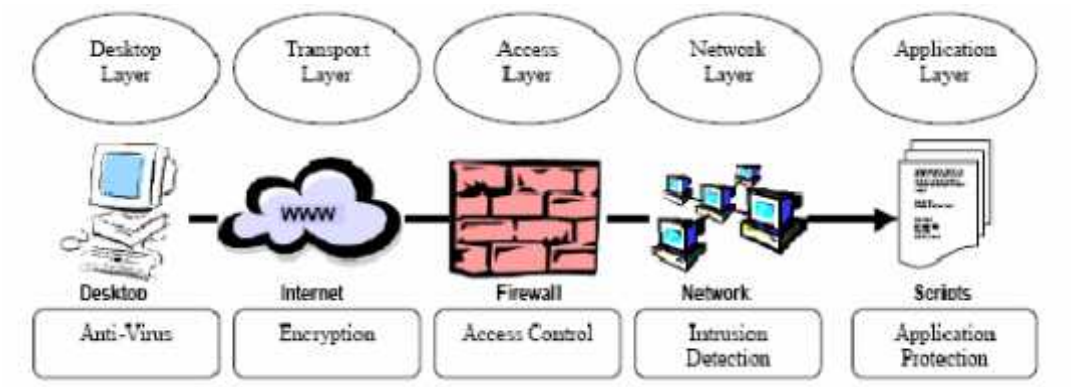

Figure 3.1: Security layers in web applications[6]

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 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 attacks 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, one must instead consider that database security as a single measure is not sufficient to guarantee protection of data in web applications.

3.1.3 Attack Procedure

We have found that attackers, in general, follow a procedure [HA, et al 06] 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. [AR, *et al* 03]

Setting the objective: Whether explicit or arbitrary, attackers have one or more objectives for conducting SQL injection attacks. 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.

3.2 Nomenclature

3.2.1 Security Services

The objectives of SQL injection attacks can be expressed in terms of compromising security services. We consider the security services below relevant to maintain asset security in respect to SQL injection. We have taken the liberty of altering their definitions slightly:

s1. Access control: Access control involves ensuring that users can only access and manipulate data according to their privileges.

s2. Availability: The services offered by a web application must be available to users when they request them.

s3. Authenticity: Ensuring that users who log in to a web application are who they claim to be.

s4. Confidentiality: Ensuring that information is kept secretly. This security service can be divided into privacy and secrecy.

s4.1. Privacy: Personal information, concerning employees and customers must be kept secret.

s4.2. Secrecy: Sensitive business-related information must be kept secret.

s5. Integrity: Information consistency must be maintained.

3.2.2 Means

Attackers can use different means [AR, *et al* 03] [DI, 09] to perform an attack:

m1. Web page form manipulation: An attacker can use forms to enter parts of SQL statements such as SQL keywords, control characters or data in order to manipulate underlying application server-side scripts or programs.

m2. URL header manipulation: In a similar way as m1, parts of SQL queries can be entered into a page's URL, sending manipulated arguments to the server-side.

m3. Cross-site scripting: By viewing the source code of web pages and examining existing client-side scripts, an attacker can write a fabricated script and use it to send information to the underlying server side scripts and programs instead of using the original script.

m4. Error message interpretation: By examining error messages generated by either the RDBMS or server-side scripts, an attacker can retrieve information about the database structure: table and column names, number of columns and column data types. Furthermore, error messages can contain information about SQL query syntax and how scripts are formed.

3.2.3 Attack Methods

There are numerous ways to conduct SQL injection attacks, and the chosen methods depend on what the attacker will accomplish, i.e. which security services to endanger, and what vulnerabilities the web application contains. These methods constitute threats and they can be grouped into two main categories:

a1. Data manipulation: Using data manipulation, an attacker can bypass authentication as well as retrieve, change, fabricate or delete data in a database.

Category a1 can further be roughly divided into distinct methods:

a1.1. Authentication bypass: An attacker may use this method to pretend to be a legitimate user .

a1.2. Information retrieval: Attackers can try to manipulate or execute SELECT statements in order to get access to information beyond their privileges. This could be achieved by e.g. manipulating the WHERE clause. One example of this is that more rows than intended can be retrieved from the table specified in the original query. Another example is by using UNION, causing rows from more tables to be returned than specified in the original query.

a1.3. Information manipulation: Attackers can try to manipulate or execute UPDATE statements in order to alter information beyond their privileges.

a1.4. Information fabrication: Attackers can try to manipulate or execute INSERT statements in order to alter information beyond their privileges.

a1.5. Information deletion: Attackers can try to manipulate or execute DELETE or DROP statements in order to alter information beyond their privileges.

3.2.4 Prerequisites (That make attacks easier)

We have found that different SQL injection attack methods need different prerequisites in order to be carried out. These prerequisites are related to both query execution properties and other features that RDBMS support as well as properties of programming languages used for implementing scripts and programs. What is important here is not which properties are offered by which RDBMS and programming languages. Rather, the question concerns which properties are supported by the RDBMS and programming languages chosen in a given web application. These prerequisites are not necessary to conduct SQL injection attacks. Rather, they should be viewed as components that make attacks easier to conduct. For example, prerequisite p4 may enable an attacker to add an INSERT query after an intended SELECT query. However, the attacker could also try to find a field in a form where an INSERT query is expected.

p1. Sub-selects: Sub-selects are multiple SELECT statements used together. A top-level SELECT statement is using other lower-level statements to retrieve values to be used in a WHERE clause.

p2. JOIN clause: JOIN clause can be used when multiple SELECT queries are combined in the same query.

p3. UNION clause: UNION clause can be used when multiple SELECT queries are combined in the same query.

p4. Multiple statements: Refers to the ability to allow execution of multiple SQL statements, where each statement is separated by a delimiter, e.g. a semicolon.

p5. End-of-line comments: the ability to comment out parts of a SQL statement, meaning that the RDBMS will not take notice of the SQL syntax followed by a comment symbol. For example, some RDBMSs uses '–' as a comment symbol.

p6. Privileged accounts: Accounts defined in the database are used by database connections to access the database. An attacker could only use attack methods that execute SQL statements associated with defined privileges in the account used by the web application. For example, if the account does not specify DELETE as a privilege, the attacker cannot use m4 as attack method.

p7. Error messages: Errors that occur in the RDBMS or in any server side script or program can produce an error message that can be sent to the client and printed in the web browser.

p8. Weak data types: Several script and programming languages used in web application development support variables of weak type, i.e. variables that can store data of arbitrary type.

p9. Data type conversion: Several RDBMSs support variable type conversion, e.g. allowing numeric values to be converted automatically into a string type.

p10. Stored procedures: Such procedures, supported by some RDBMSs, allow execution of system or database commands and SQL sub-routines in the RDBMS.

p11. Dynamic SQL: In order to embed user input into SQL queries, server-side scripts and programs can use dynamically built SQL queries where SQL statements are combined with user input and then sent into the RDBMS for execution. Another approach is to let dynamically built SQL queries call stored procedures.

p12. INTO OUTFILE support: If INTO OUTFILE is supported by the RDBMS, users may print query results into a text file on the host computer.

3.2.5 Vulnerabilities

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

v1. 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.

v2. Error message feedback: Error messages that are generated by the RDBMS 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.

v3. 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 bu.er overrun

v4. 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 values.

v5. 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.

v6. 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.

v7. Stored procedures: 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.

v8. 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.

v9. 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.

v10. 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.

v11. 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.

v12. 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.

3.2.6 Countermeasures

In this section, we present technically oriented countermeasures found during the part of our survey that concerns prevention techniques against SQL injection attacks :

c1. Different accounts: Default accounts that come with some RDBMS, such as the account used by the system administrator, should never be used for web application access. Instead, different accounts should be created and used for different client profiles. Moreover, in case that the RDBMS chosen has a default system administrator password, change it immediately after installation.

c2. Limited privileges: Permissions granted to database accounts, used by web applications, should be given according to the principle of least privilege. The actions made by web applications users on stored procedures should be limited too, i.e. if the intention is to not use or have unused stored procedures they should be removed or moved to an isolated server. This minimizes the damage an attacker can cause in case of authentication bypass. Normally, SELECT is a privilege that in almost any case will be given to an account. This is used for logging in to the system and retrieves information. But when logging in, a user should not have any other privileges such as INSERT or DELETE.

c3. Static SQL: Static SQL refers to SQL statements that cannot be altered by inserting SQL keywords where values are expected. Examples of static SQL include taking advantage of prepared statements and the use of stored procedures. This countermeasure implies that static SQL should be used in favor of dynamic SQL, which should be avoided.

c4. Error handling: Error messages generated by RDBMSs or web servers should never be passed back to the client-side since they may contain information about database and script structure. Instead, handle error messages at the server-side and send back messages to the client-side that do not contain information that could be used for SQL injection attacks.

c5. Input validation: All data sent by the client-side should be considered potentially harmful. While input validation could be implemented in client-side scripts for performance factors, one should never rely on client-side scripts for security. Therefore, every parameter sent from the client-side should always be examined and validated by server side scripts and programs. This could be done by for example using comparison and replacing functions if the development platform enables it. Developers can otherwise write such functions themselves. One can also take advantage of regular expressions. If concatenation is necessary, then use numeric values for the concatenation part or check the input for malicious character strings and sequences, e.g. SQL keywords, such as UNION, or meta characters and SQL control characters such as single and double quotation marks and semicolons.

c6. Character escaping: In case characters such as quotation marks or semicolons must be allowed, for example if arbitrary text should be accepted, those characters should be escaped using a scheme, e.g. ASCII code or using bind variables.

C7. Variable size: Control variable length and size. Even the length limitation is long enough to fit a few additional queries, the inability to input an infinitely long string disables the attacker from employing evasion techniques such as encoding.

C8. Strong typing: try to avoid weak variables that have either no clear type, e.g. variables in PHP, or are automatically converted from one type to another by components such as the RDBMS. Instead, make sure that variables are explicitly typed. In case variables of weak type need to be used, check their content.

3.3 How it happens?

SQL is the standard language for accessing Microsoft SQL Server, Oracle Server and other database servers. Most web applications need to interact with a database. SQL injection vulnerabilities most commonly occur when the web application developer does not ensure that values received from a web forms, input parameter etc. are validated before passing to SQL queries that will be executed on a database server. If an attacker can control the input that is sent to an SQL query and manipulate that input so that the data is interpreted as code instead of as data, the attacker may be able to execute code on the back-end database.

A lot of web sites that offer tutorials and code examples to help application developers solve common coding problems often teach insecure coding practices and has resulted the web application vulnerable.

3.3.1 Dynamic String Building

Dynamic string building is a programming technique that enables developers to build SQL statements dynamically at run time. A dynamic SQL statement is constructed as execution time, for which different conditions generate different SQL statements. It can be useful when developers need to decide at runtime what fields to bring back from (say SELECT statements) to acquire data from different tables on different conditions.

3.3.2 Insecure Database Configuration

Besides securing the application code, we should look at the database itself. Databases come with a lot of default users preinstalled. For example,””sa” for Mocrosoft SQL server, ”root” for MySQL and “sys” for Oracle are created respectively at the time when database is created and are of course preconfigured with default well-known passwords.

Application developers often code their applications to connect to a database using one of the in-built privileged accounts instead of creating specific user accounts. These powerful accounts can do anything to the data stored. When an attacker exploits an SQL injection vulnerability in an application that connects to the database with previleges of that account, he can execute code on the database with that privilege and thus can manipulate the data as he wants to.

3.4 SQL Injection Attack Examples

We have studied various attack examples as well as tested them in our system. Our intention is not to describe every type of attack method and their variations, since they can be studied in our referenced research material. However, for issues of clarity, we present examples that both give information on how SQL injection attacks could be performed and how we processed attack examples.

Attacks on the database can be made in order to gain access to the web application, and thus threatening the application's authentication security service. 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 might be exploited.

In order for that to work, a few prerequisites are needed, and some related vulnerabilities should be left non-handled. To begin with, the application must support dynamic SQL queries (p11 and v6), and the application should not validate the user's input (v1) or control the input's type and size (p8, v3 and v4). Furthermore, if it also displays server-side-generated error messages in the user's web browser (p7 and v2), it would give helpful feedback. Finally, it would be useful to the attacker if end-of-line comments are supported (p5). The attacker may begin the attack by entering some chosen symbols and SQL keywords in the login page fields and analyzing the error messages. Suppose that the attacker prints the following in the login field:

Username:’ having 1=1--

The error message returned might be:

Microsoft OLE DB Provider for ODBC Drivers error ’840e14’
[Microsoft][ODBC SQL Server Driver][SQL Server]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--

This results in the error message:

Microsoft OLE DB Provider for ODBC Drivers error ’840e14’
[Microsoft][ODBC SQL Server Driver][SQL Server]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:

"SELECT * FROM users WHERE username = ’" + username + "’AND password = ’" + password + "’"

The attacker could now bypass the authentication control by entering the following fragment into the username field:

Username:’ OR 1=1--

Upon concatenating that input into the SQL query, the following query will be send to the database:

SELECT * FROM users WHERE username = ’’ OR 1=1--

(the rest of the statement is ignored)

The query is sent to a SQL Server database, which returns the rows that match the query. The script checks whether any rows were returned. If so, it returns true, indicating that the user is identified. Otherwise, the script returns false meaning no

such user is registered in the database. The system considers the user authenticated or not based on the return value received from the script or program that requested the execution of the query, and allows the user to use it if confirmed.

Since the conditions in the WHERE clause will be evaluated to true, the query will return a result containing all users in the database. As a consequence, the script will also return true, and the query yields a valid user whose identity the attacker would assume while using the system. The attacker will have the privileges specified in the account used for accessing the RDBMS. Different accounts may have been defined in the RDBMS and application logic may choose different accounts for different users. In case all rows are returned, they might have been sorted by the RDBMS according to id, name or some other criteria. The application logic may determine which user is logging in due to the result set by looking at the first row. The system administrator may be the first user defined in the table of users, hence having the lowest id. Therefore, if the application logic chooses the first row, it is likely that the attacker will log in as the system administrator and normally, that account will be given all privileges.

Suppose that the attacker would now want to continue the attack, but instead of bypassing authentication control, he would attempt to insert a fabricated user identity into the database table, which he could later use for further attacks. The attacks would thus attempt to fabricate information (a1.4) and would endanger the application's access control security service as well as its integrity. The attacker would now need a way to manipulate the application into entering the user identity into the users table in the database. This could be done in a few ways: the attacker could try to call a stored procedure (if such exists in the RDBMS used by the application) that would enter a row into the users table (a2). A second way would be to try and find a field in the application that is used for entering information into the database, and tamper with it (a1.4). A third way would be to manipulate the above mentioned login field query and add to it an INSERT statement (a1.4).

All the attacker needs to do is to find out exactly which fields the users table contains and which types each column requires and then enters the manipulating data into the login field. That data could look like the example below:

```
'; INSERT INTO users (username, password)
VALUES ('hacker', '666');
```

This would yield the following query:

```
SELECT * FROM users WHERE username = ’’;
INSERT INTO users (username, password)
VALUES (’hacker’, ’666’);
```

Once the manipulated query executes in the database, it would result in a new row in the users table containing the fabricated identity.

CHAPTER-4

A MODEL FOR PREVENTING SQLIAS

4.1 Observations

When we closely analyzed the code structure of all SQLIA then we found that hacker inject the user string in a way that could alter the structure of the original query and every injection is done either at the middle of the query or at the end; means hacker usually append the query. So if we store all the information regarding to the structure of the valid query and cross check it with the entire dynamically generated query then we can determine that the dynamically generated query is a SQLIA or a valid query.

4.2 Proposed Methodology

As in the previous section in combined method we see that to reduce false positive and false negative it maintains a database for storing valid query structure. In runtime validation it checked the dynamically generated query with the previously stored query structure to determine the possible SQLIA. Data structure of the valid query structure is made in static analysis phase. We are also using the same method to storing the valid query structure. We are storing all the valid query structure by linked list representation where each individual singly link list represents a valid query structure and to store the starting address of all these singly link list we use a doubly link list called as main link list whose each node store the starting address on a singly link list. So when we found a new query is arrived to the database server we start searching the structure of the query in our linked representation. If it is a successful search then the query is a valid query otherwise it is an SQLIA.

In this scheme we stored the structure of the query by preserving the order of the sequence of token generated by the query; means we are checking the sequence of token generated by the arrived query is in the same order as we stored in our valid query database. If the sequence of the arrived query is in same order as the query stored in our database then the arrived query is a valid query, otherwise if we do not find any ordered sequence like the arrived query in our entire database then it is possible structure of the valid SQL query in our database in static analysis phases.

As this technique we do not take any support from the application program so when a query comes from the application program for validation it does not know that

which hotspot, a hotspot is defined as a point in the application code that issues SQL queries to the underlying database, of the application program generates this query, so we have to match with all the possible structure similar to the incoming query. In the most of the cases as the incoming query is not an SQLIA so in most of the cases it's a successful search. So now the problem becomes a searching problem as to increase performance gain we have to search fast because to validate each and every query coming to the database it's an expensive task.

As there are many possible links in the main link list which stored the starting node of the each link list storing the query structure and there is a huge request to the database server we use the searching technique in a hash table. To store a valid individual query structure, we preserve the sequence of token generated by the query using a singly link list where each node store a single token of a query having an additional field to mark that the node is a position of user input or it store a token of the static part of the query. We use a doubly linked list called as a main link list whose each node store the staring address of the a singly link list. To find an ordered sequence of token in that singly link list for an incoming query to the database we first separate the tokens from the query by a SQL parser of the specific DBMS we are trying to save from a possible SQLIA. After token separation we get the ordered sequence of the tokens of the query then we start searching the singly link list. While searching the single link list if position of the token from the incoming query matches the token of the same position in the singly link list and if it is not a user input then we move to the next token as well as to the next node of the list until any mismatch found or the end of the list. In this way if we reach at the end of the single link list and there is no more token left in the incoming query then it's a successful search and the incoming query is valid query.

From the above description of matching technique it is clear that for a successful search, number of token in the incoming query is same as the length of the link list stored its structure. Figure given below shows a sequence of tokens extracted from an incoming query.

For runtime token matching if we used literal wise matching like others then it will be a huge computational overhead. For example if there are 'n' literals in incoming query string and there are 'q' query structures available in the data structure of the same length of incoming query. In worst case if we assume the mismatch occurs at the last position for each and every query structure in database and if it is an

unsuccessful search then we have to check ‘n’ number of literals for each ‘q’ no. of query, so the complexity will be O(n*q). So to avoid this huge computational over head we use a different technique instead of using literal wise string matching algorithms we simple mapped each token into an integer value.

select productid from product

Figure 4.1 A structure of an incoming query

Select	Productid	from	Product

Figure 4.2 Sequence of tokens after separation

We also store these integer values in our database instead of storing the tokens in a string format as a query figure print. It also takes very less space.

In run time validation, we can determine the type of query (like either it is SELECT or DELETE or INSERT etc), by comparing the integer value of corresponding token from hash table in constant time, i.e. O (1). Now it searches the number of valid tokens from the list pointed by the resulted token. Suppose that if there are number of blocks is “n”, then it takes O (n). Now the total time complexity is O (1) + O (n) = O (n).

The formula we are using to transform a token into its corresponding integer value is to multiply each ASCII decimal value of a literal by its position number occurring in token and summed it up. For example consider the keyword ‘SINSERT’ the corresponding ASCII decimal values for the literals is I=73, N=78, S=83, E=69, R=82, T=84, now the position of each literal is I=1, N=2, S=3, E=4, R=5, T=6, so after multiplying the ASCII value of each literals with its position we have to sum up. So, now the value of the token becomes 73*1+78*2+83*3+69*4+82*5+84*6=1668. So the corresponding integer value of ‘INSERT’ is 1668. Figure given below shows a valid query translation to its corresponding integer sequence.

Let us take the above example of figure 4.1

2236	4809	1099	3069

Figure 4.3 Sequence of integers after tokenization[11]

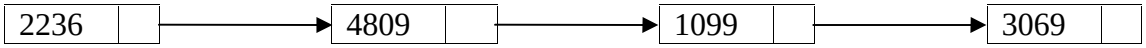

Figure 4.4 Linked list representations after tokenization[11]

It is already known that for a valid incoming query the number of tokens is same as the number of tokens in its corresponding query structure in the data base. So to reduce the search space we group together all the query structure having same token number. It means if a query having 10 tokens belongs in a separate group than a query having 12 tokens. So before searching the similar structure for an incoming query we first calculate the number of tokens it have, then we start searching in the group having all the structure of valid query having the same number of tokens. To group together all the singly link list having same number of tokens we use a doubly link list usually referred as main link list whose each node holds the starting address of a singly link list among all the singly link list having same number of tokens. That means if we have ‘n’ groups of singly link list then we have ‘n’ no. of doubly link list and for an incoming query we only search a single doubly link list among the ‘n’ no. of list. Figure 4.5 shows the group representation of queries having 4 tokens.

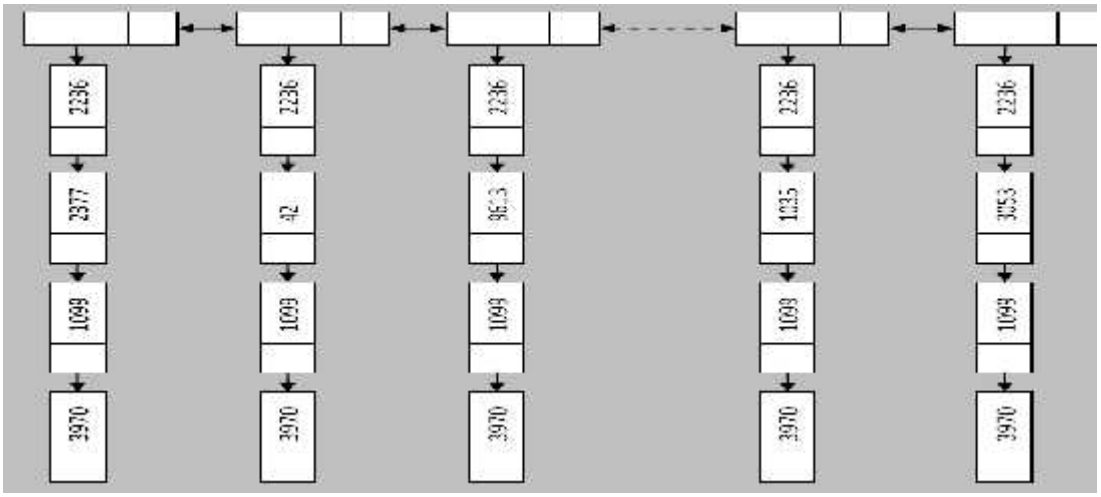

Figure 4.5: Grouping of queries having 4 tokens[11]

To store the starting address of the doubly link list called as the starting address of the group we use an array where each cell of the array stores the starting address of a group. We store the starting address of a group in a way that the index of an array cell should represent the group number, the number of token each singly link list possesses in that group. For example if a group having all the singly link list having ‘n’ no. of tokens then the starting address of the group be assigned to nth cell

in the array. Though in this representation there are many cells in the array may not be used but by using some extra storage we have a great advantage that we don't need to search the starting address of a specific group because after calculating the number of tokens in the incoming query the number itself represents the cell number of the array holding the starting address of the group that the incoming query may belong. Figure4.6 [DI, 09] shows the complete structure of store all valid queries.

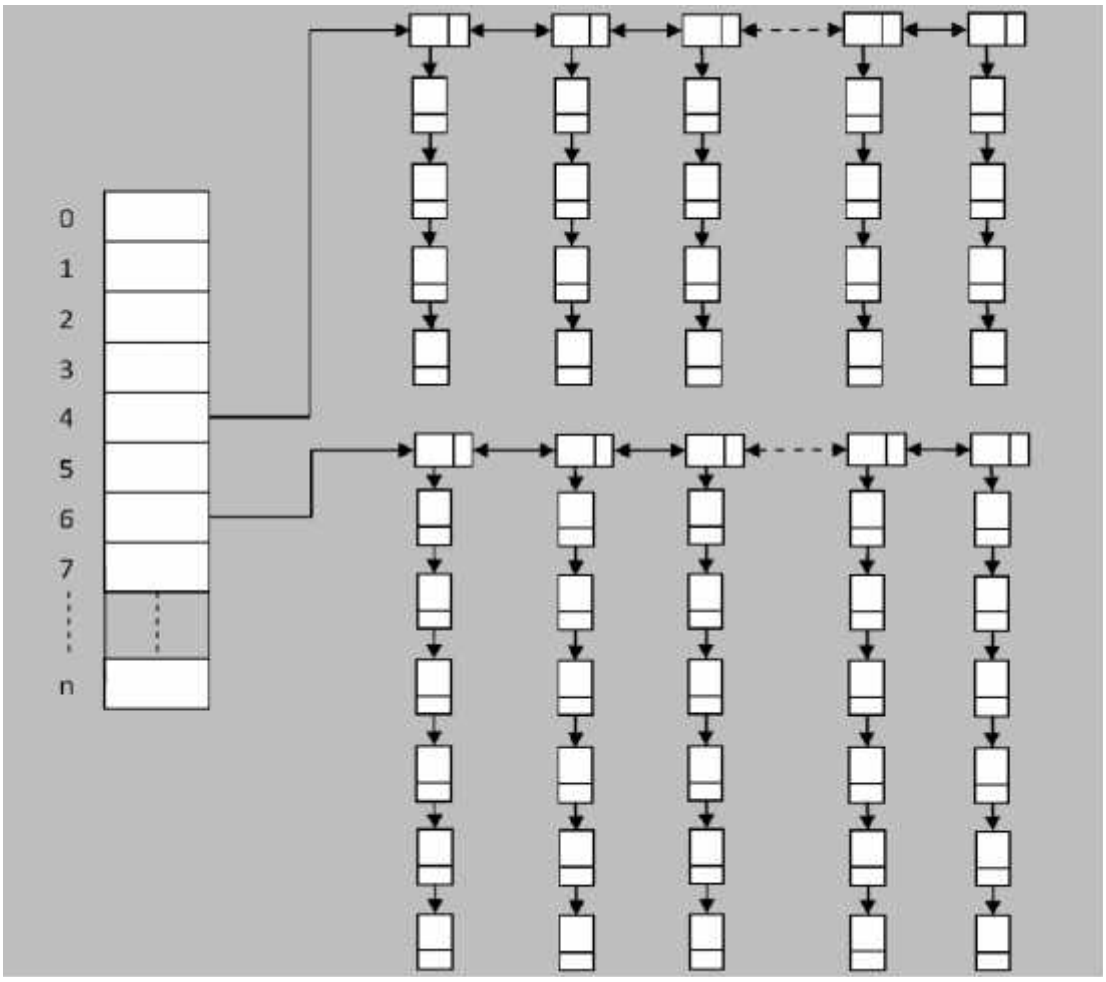

Figure 4.6: Complete structure to store all valid queries[11]

4.3 Proposed architecture:

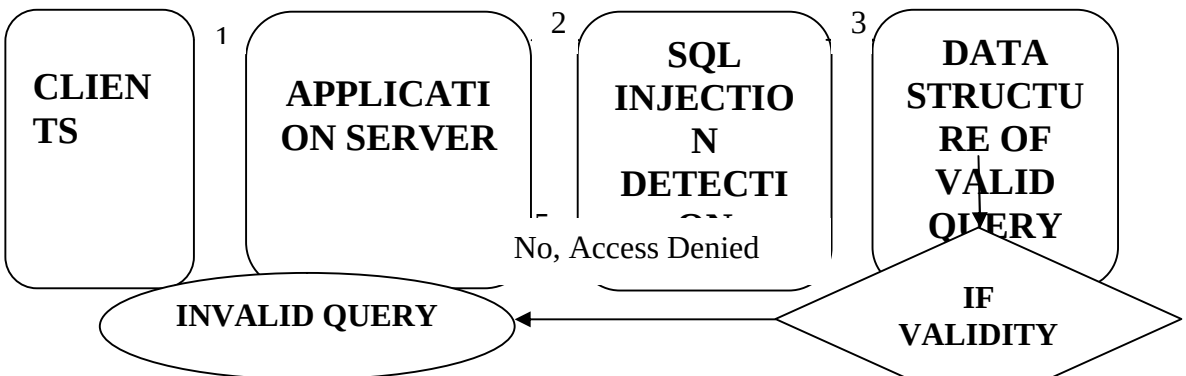

Figure 4.7: Proposed Architecture

This scheme is implemented as a different layer in between application program and the data base. These layer will perform as a virtual database to the application programs as it takes the query from the application programs, analyze it, if it is not an SQLIA then it sends the query to the data base, get the results from the database and send the result to the application programs. As this scheme is totally depends on the token generation means directly depends on the parsing technique of a specific DBMS, so the implementation will be different for different DBMS, as different DBMS has different key word set, different function names, as well as supports different syntax.

4.4 Algorithm

1. Input the query string sql
2. for each embedded sql query

```
Tokenize the query into n tokens
Convert each token into corresponding integer
Search the hash table to access the hash value (hashval) for query
For ( i=0; i < size of (hashval); i++)
{
        If (n==indexof (i))
                the query is valid
                return
}
```

the query is invalid

return

3. End

CHAPTER-5
TESTING AND ANALYSIS

5.1 Testing and Analysis

The list of valid queries in the implementation is as follows:

1. insert into product values('name','description',12)
2. insert into product(name, description, unitprice) values('name','description',1)
3. select productid from product
4. select distinct name from product
5. select all name from product
6. select productid, name, unitprice*100 from product
7. select name from product where unitprice = ? and description like %
8. select name from product where unitprice between ? and ?
9. select category.name, product.name from category, product where category.categoryid = product.productid
10. select category.name, product.name from category, product where category.categoryid =? or category.name = ?
11. select category.categoryid, product.name from category inner join product where product.categoryid = category.productid
12. select * from product where name like '%a%'
13. select distinct name from product where unitprice < ? order by name

14. select * from product order by name desc
15. select * from category
16. select name from category union select name from product
17. select name from category union all select name from product
18. select name from category intersect select name from product
19. select name from category intersect all select name from product
20. select name from category except select name from product
21. select name from category except all select name from product
22. select avg(unitprice) from product
23. select sum(unitprice) from product where name = ?
24. select name, min(unitprice) from product group by categoryid
25. select name,count(distinct description) from category,product where category.categoryid = product.category group by category.categoryid
26. select max(unitprice),name from product group by name having max(unitprice) > 10
27. select avg(unitprice) from product
28. select count(*) from product
29. select category.* from category where unitprice is null
30. select * from product where unitprice it not null
31. select distinct name from product where categoryid not in (select categoryid from category)
32. select name from category where > all (select unitprice from product where description = ?)
33. select * from product where name not in (select name from product where unitprice < 1000)
34. select productid, name from product
35. update product set name=?,description=?,unitprice=? where productid=?
36. delete from product where productid=?
37. delete from proudct
38. select top 1 from product
39. update product set name = b.a from (select max(unitprice) as a from product) b
40. select sum(unitprice) from product where name in (select product from product.name where name = 'west')

Some examples of SQL attack are as follows:

1. select name from product where name = '' or 1=1
2. select name from product where name = 'admin'--'and unitprice > 1000
3. select * from product;delete from category--
4. select name from product where name = 'ram' union select name from product--
5. insert into product values('food','good food',2);delete from product--)

5.2 Performance analysis

No of SQL statement	Total no of tokens	Average no of tokens per query	No of valid query	No of SQLIA	Total time required (Millisecond)	Average time per query (nanosecond)
45	1232	27	40	5	6	133

Table 5.1 Performance analysis

No of query	Total false positive	Total false negative
40(valid)+5 (SQL Injection)	0	0

Table 5.2 Accuracy Result

System Configuration: Intel® core™ 2 Duo, 2 GB RAM

5.3 Precision, Recall and F-Measure

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

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

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

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

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

Now, using the observations in our implementation part, we came to the following result.

Precision= |La| / |A| = 35/40 = 0.88

Recall= = |La| / |L| = 35/45 = 0.78

$$F\ \text{Z}measure\ \text{X}\frac{2 \mid recall \mid precision}{recall\ \Gamma\ precsion} = \frac{2 \mid 0.78 \mid 0.88}{0.78\,\Gamma\,0.88} = \frac{1.37}{1.66}\ \text{X}0.83$$

5.4 System Validation

As the research work is carried out as an extension of the research work carried out by Dibyendu Aich [DI, 2009] to perform the task left for future work in [DI, 2009]. Here, Precision, Recall and F-Measure of the proposed module have been compared with that of [DI, 2009].

Precision = 0.87

Recall = 0.77

F- Measure = 0.82

Precision		**Recall**		**F- Measure**	
Previous Model(PS)	Proposed Model(PD)	Previous Model	Proposed Model	Previous Model	Proposed Model
0.87	0.88	0.77	0.78	0.82	0.83

Table 5.3: Comparison between different effectiveness & efficiency measures

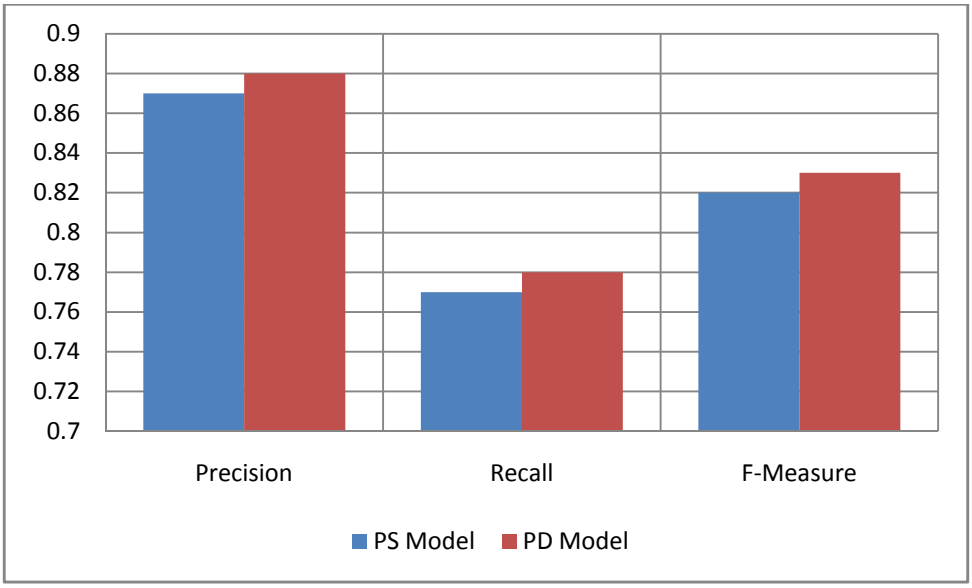

Figure 5.1: Comparison between different effectiveness & efficiency measures

While talking about complexity, we can determine the type of query (like either it is SELECT or DELETE or INSERT etc) by comparing the integer value of corresponding token from hash table in constant time, i.e. O (1). Now it searches the number of valid tokens from the list pointed by the resulted token. Suppose that if there are number of blocks is "n", then it takes O (n). Now the total time complexity is O (1) + O (n) = O (n).

In comparing time complexity with previous model [DI, 2009], it has been concluded that the time complexity of the algorithm implemented in previous model [DI, 2009] is O (m × n), where the query needs m times to find the type of query and n times to find the structure of query. But we have reduced this complexity to O (n).

CHAPTER-6
CONCLUSION AND FUTURE WORK

SQL Injection is a technique that most of the attackers use to modify the underlying database in several web applications in present day. These attackers can modify the SQL query and change the behavior of the program for the benefit of the hacker. In this research, we have studied several SQLIA prevention techniques for the detection and prevention for these attacks. We have proposed a technique to counter SQL injection which combines conservative static analysis and runtime monitoring to detect and stop illegal queries before they are executed on the database. In the static part, the technique builds a conservative model of the data structure of the legitimate queries that could be generated by the application. In its dynamic part, the dynamically generated queries by the application are inspected for resemblance or compliance with statically-build model. Even for fast searching we use the concept of linked list representation and doubly linked list. The proposed model can catch different types and modes of execution of SQLIAs and thus protect the back-end database of a web application.

6.2 Limitations and Future work

The data structures of the valid queries may not be sufficient to test validity of all incoming query since only a SQL expert can define all the structures of a valid query. The proposed model cannot work in PL/SQL blocks since it will not handle Pl/SQL code blocks. So, in future we may expand our solution to handle PL/SQL code block.

REFERENCES:

[1] Andrew Jaquith.. Technical report, @stake, feb 2002. *The security of applications: Not all are created equal* http://www.atstake.com/research/reports/acrobat/atstake_app_unequal.pdf.

[2] Buehrer, G.T, Weide, B.W. and Sivilotti, P.A.G(2005): "*Using parse Tree Validation to Prevent SQL Injection Attacks*". 5th International Workshop on Software Engeneering and Middleware(SEM)

[3] Dieter Gollmann. Computer Security(John Wiley & Sons, 2001).

[4] Halfond, William G.J., and Orso (2005): "*Combining Static Analysis and Runtime Monitoring To Counter SQL Injection Attacks*", in proceeding of the third international ICSE workshop on Dynamic Analysis(W

[5] Hung, Y-W., Hang C., Tsai, C-H, Lee, D and Yu, F, *Securing Web Application Code by Static Analysis and Runtime Protection.* In processing of the 12th International World Wide Web Conference(2004).

[6] Kevin Spett. Security *at the next level-Are your web applications vulnerable?* Technical report, SPI Dynamics, 2000. http://www.spidynamics.com/whitepapers/webappwhitepaper.pdf.

[7] Matthew Levine. *The importance of application security.* Technical report, @stake, Jan 2003. http://www.atstake.com/research/reports/acrobat/atstake_application_security.pdf.

[8] Mitchell Harper. *SQL Injection Attacks - are you safe?* Technical report, DevArticles, may 2002. http://www.devarticles.com/content.php? articleId=138&page=2.

[9] The Open Web Application Security Project. A guide to building secure web applications, Version 1.1.1. Online Documentation, Sep 2002. http://www.owasp.org/.

[10] Thomas Connolly, Carolyn Begg, and Ann Strachan. Database Systems - A Practical Approach to Design, Implementation, and Management. (Addison - Wesley, 1999).

[11] [DI,09] Dibyendu Aich, *Secure Query Processing By Blocking SQL Injection*, research work,may 2009.