Print Form

Secure Query Processing By Blocking SQL injection

Thesis submitted in partial fulfillment

Of the requirements for the degree of

Master of Technology

in

Computer Science and Engineering

by

Dibyendu Aich

(Roll: 207CS103)

Department of Computer Science and Engineering
National Institute of Technology Rourkela
Rourkela-769008, Orissa, India

May 2009

Secure Query Processing By Blocking SQL Injection

Thesis submitted in partial fulfillment
Of the requirements for the degree of

Master of Technology

in

Computer Science and Engineering

by

Dibyendu Aich
(Roll: 207CS103)

under the guidance of

Prof S. K. Jena

Department of Computer Science and Engineering
National Institute of Technology Rourkela
Rourkela-769008, Orissa, India
May 2009

National Institute of Technology Rourkela
Rourkela-769008 (Orissa)

Certificate

This is to certify that the work in this Thesis Report entitled "***Secure Query Processing By SQL Injection***" by **Dibyendu Aich** has been carried out under my supervision in partial fulfillment of the requirements for the degree of ***Master of Technology*** in Information Security during session 2007-2009 in the Department of Computer Science and Engineering, National Institute of Technology Rourkela, and this work has not been submitted elsewhere for a degree.

Place: Rourkela
Date: May 11, 2009

(Dr.S.K.Jena)
Professor
Department of CSE

Acknowledgement

Many people have contributed, directly or indirectly, to the successful completion of this thesis. They will all be remembered in my heart. First, I would like to thank my advisor, **Prof.S.K.Jena** for his guidance from conducting the research to writing the thesis. I really appreciate the patience and respect that he has given me.

I am extremely grateful to **Prof.B.Majhi** for being a great source of advice. I would like to say "Thank you!" to my classmates, for their constant help and encouragement during my thesis work, and my family, for their comforting and encouraging me during my stressful time.

Dibyendu Aich
207CS103,
M.Tech.(CSE)
2007-2009

Abstract

With the rise of the Internet, web applications, such as online banking and web-based email the web services as an instant means of information dissemination and various other transactions has essentially made them a key component of today's Internet infrastructure. Web-based systems consist of both infrastructure components and of application specific code. But there are many reports on intrusion from external hacker which compromised the back end database system, so we introduce briefly the key concepts and problems of information security and we present the major role that SQL Injection is playing in this scenario. SQL-Injection Attacks are a class of attacks that many of these systems are highly vulnerable to, and there is no known fool-proof defense against such attacks. Based on the above analysis and on today's computer security state-of-the-art, we focus our research specifically on the SQLIAs, which are still one of the most exploited and dangerous intrusion techniques used to access web applications.

In this thesis, we propose a technique, which uses runtime validation to detect the occurrence of such attacks, which evaluation methodology is general and adaptable to any existing system. The most available solution of that problem either requires source code modification, which is an overhead to an existing system as well as which can increase the possibilities of new injection points, or required a computational overhead at run time which increase the minimum response time, as well as most of them are not taking the advantage of the modern age processor architecture. To overcome these problems of existing solutions we use link representations which store the valid query structures in terms of an orders sequence of tokens. To perform fast searching among these various lists we start searching in a multithreaded way. To avoid the huge computation over head of string matching algorithm to match two tokens we convert each token into an integer value and store that integer value instead of that token in our database and while searching we simple match these integer values. For finding the correct group of list we use an array representation which eliminates the need of searching the specific group. Even for minimizing the response time we use a hit count method to predict the possible list for searching the incoming query structure. So in a brief this technique eliminates the need of source code modification along with an improved overall efficiency.

Contents

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Motivations: Importance of the Problem

Information is the most important business asset today and achieving an appropriate level of “Information Security" can be viewed as essential in order to maintain a competitive edge. SQL Injection Attacks (SQLIAs) are one of the topmost threats for web application security, and SQL injections are one of the most serious vulnerability types. Recently the incident of SQLIA is so high that in year 2008 it increases by 134% [12] and become a predominant type of web vulnerabilities. They are easy to detect and exploit; that is why SQLIAs are frequently employed by malicious users for different reasons, e.g. financial fraud, theft confidential data, deface website, sabotage, espionage, cyber terrorism, or simply for fun. Furthermore, SQL Injection attack techniques have become more common, more ambitious, and increasingly sophisticated, so there is a deep need to find an effective and feasible solution for this problem in the computer security community. Detection or prevention of SQLIAs is a topic of active research in the industry and academia. To achieve those purposes, automatic tools and security systems have been implemented, but none of them are complete or accurate enough to guarantee an absolute level of security on web applications. One of the important reasons of this shortcoming is that there is a lack of common and complete methodology for the evaluation either in terms of performance or needed source code modification which in terms is an over head for an existing system. So we feel that there should be such type of mechanism which will be easily deployable, does not need source code modification as well as provide a good performance and to achieve this, our research work is driven to the way of developing a new modified SQL injection detection technique.

1.2 Research Focus and Original Contributions

Our research work focused on the analyses and resolution of the problem of SQL Injection attacks, in order to protect and make reliable any vulnerable web applications. Firstly, we deeply analyze the different technique to perform a successful SQLIA then we try to provide a technique to prevent SQLIAs without any source code modification and without huge performance degradation. To achieve our goal we propose a general and complete evaluation methodology which can be easily deployable to an existing system to preserve the security of the system against SQLIAs. Our key original contributions can be identified as follows:

- We propose a complete evaluation methodology supported by abstract and detailed diagrams and step-by-step procedure for implementing the technique to prevent SQLIAs.

- We provide a technique for effective dynamic detection and prevention of SQLIAs without access to the application source code.

- We implement a heuristic approach to searching a valid query structure in our database to minimize the response time.

- We used the modern age processor architecture by making our search in a multi threaded way to minimize the response time.

- We avoid the computational overhead of string matching algorithm for matching tokens by converting the token to some integer values and checking those integer values instead of the tokens.

1.3 Structure of the Work

The rest of this work is organized as follows. In Chapter 2 we briefly provide general background knowledge on the key terminologies, concepts and problems of information security focusing on the critical role of web applications. Moreover, we show the state of the art of computer security, describing the recent input validation based vulnerability in web applications, scenario of mostly SQL injection attacked organizations, the current situations of these last years, highlighting the important position of SQL Injection and the vulnerability resolved last year. Chapter 3 explains both theoretically and with practical examples, how SQL Injection attacks work, and its consequences. It also furnishes classifications of SQLIA techniques, a methodology for a successful attack and the typical countermeasures adopted against them. In Chapter 4 we review existing work and the draw backs of them. Here we also characterize in details, respectively, our proposed evaluation technique, its various optimization technique to minimize response time, step-by-step procedure for the technique, analysis of the proposed technique its implementation, the results obtain as well as we discuss about its advantages and limitations. Finally, in section 5, we draw our conclusions, outlining the future directions of this work.

Chapter 2

State Of The Art

In this chapter we will provide the reader with a brief overview of the general concepts of information and computer security. We will introduce important factors such as web applications and vulnerabilities. In addition, we will present the security problems that currently affect our society.

2.1 Computer and Information Security: an Overview

2.1.1 Terminologies and Formal Definitions

Computer security is a branch of technology known as information security, applied to computers. Information security is based on the general concept of the protection of data against unauthorized access. The objective of computer security varies and can include protection of information from theft or corruption, or the preservation of availability, as defined in the security policy. Computer security is the process of preventing and detecting unauthorized use of your computer. Prevention measures help you prevent unauthorized users, also known as intruders", from accessing any part of your computer system. Detection helps you to determine whether or not someone attempted to break into your system, whether or not the breach was successful, and the extent of the damage that may have been done. This makes computer security particularly challenging because it is difficult enough just to ensure that computer programs do everything they are designed to do correctly. Nowadays most information in the world is processed through computer systems, so it is common to use the term information security to also denote computer security. This is quite a common mistake: in fact, academically, the definition of information security includes all the processes of handling and storing information. Information can be printed on paper, stored electronically, transmitted by post or by using electronic means, shown on films, or spoken in conversation.

The U.S. National Information Systems Security Glossary [1] [2] [3] defines Information systems security (INFOSEC) as:

> *"the protection of information systems against unauthorized access to or modification of information, whether in storage, processing or transit, and against the denial of service to authorized users or the provision of service to unauthorized users, including those measures necessary to detect, document, and counter such threats."*

It defines computer security[3] as:

> *"Measures and controls that ensure confidentiality, integrity,*
>
> *and availability of the information processed and stored by a computer"*

This observation on information pervasiveness is especially important in today's increasingly interconnected business environment. As a result of it, information is exposed to a growing number and a wider variety of threats and vulnerabilities, which often have nothing to do with computer systems at all. In this work, however, we will deal mostly with computer security and not information systems in general.

2.1.2 The C.I.A. Paradigm

Information security has held that confidentiality, integrity and availability, known as the C.I.A. paradigm [1][2], are the core principles of information security.

Confidentiality is the ability of a system to make its resources accessible only to the parties authorized to access them. Confidentiality is the property of preventing disclosure of information to unauthorized individuals or systems. For example, a credit card transaction on the Internet requires the credit card number to be transmitted from the buyer to the merchant and from the merchant to a transaction processing network. The system attempts to enforce confidentiality by encrypting the card number during transmission, by limiting the places where it might appear (in databases, log files, backups, printed receipts, and so on), and by restricting access to the places where it is stored. If an unauthorized party obtains the card number in any way, a breach of confidentiality has occurred. Confidentiality is necessary, but not sufficient for maintaining the privacy of the people whose personal information a system holds.

Integrity is the ability of a system to allow only authorized parties to modify its resources and data, and only in authorized methods which are consistent with the functions performed by the system. Integrity means that data cannot be modified without authorization. Integrity is violated, for example, when someone accidentally or with malicious intent deletes important data _les, when a computer virus infects a computer, when an employee is able to modify his own salary on a payroll database, when an unauthorized user vandalizes a web site, when someone is able to cast a very large number of votes in an online poll, and so on.

Availability is the important property that a rightful request to access information must never be denied, and must be satisfied in a timely manner. In other words, for any information system to serve its purpose, the information must be available when it is needed. Ensuring availability also involves preventing denial-of-service attacks.

Sometimes other goals have been added to the C.I.A. paradigm, such as authenticity, accountability, non-repudiation, safety and reliability. However, the general consensus is that these are either a consequence of the three core concepts defined above, or a means to attain them.

2.1.3 The A.A.A. Architecture

In software engineering terms, we could say that the C.I.A. paradigm belongs to the world of requirements, stating the high-level goals related with security of information. The A.A.A. architecture and components are specifications of a software and hardware system architecture which strives to implement those requirements. Then, of course, security systems are the real world implementations of these specifications. In computer security A.A.A. stands for Authentication, Authorization and Accounting [4]. These are the three basic issues that are encountered frequently in many network services where their functionality is frequently needed. Examples of these services are dial-in access to the Internet, electronic commerce, Internet printing, and Mobile IP. Typically, authentication, authorization, and accounting are more or less dependent on each other. However, separate protocols are used to achieve the A.A.A. functionality.

Authentication: refers to the process of establishing the digital identity of one entity to another entity. Commonly one entity is a client and the other entity is a server. Authentication is accomplished via the presentation of an identity and its corresponding credentials. Examples of types of credentials are passwords, one-time tokens and digital certificates. So authentication is a security measure designed to establish the validity of a transmission, message, or originator, or a means of verifying an individual's eligibility to receive specific categories of information.

Authorization: access rights granted to a user, program, or process. It refers to the granting of specific types of privileges (or not privilege) to an entity or a user, based on their authentication, what privileges they are requesting, and the current system state. Authorization may be based on restrictions, for example time-of-day restrictions or physical location restrictions. Most of the time the granting of a privilege constitutes the ability to use a certain type of service. Examples of types of service include, but are not limited to: IP address filtering, address assignment, route assignment and encryption.

Accounting: refers to the tracking of the consumption of network resources by users. This information may be used for management, planning, billing, or other purposes. Real-time accounting refers to accounting information that is delivered concurrently with the consumption of the resources. Batch accounting refers to accounting information that is saved until it is delivered at a later time. Typical information that is gathered in accounting is the identity of the user, the nature of the service delivered, when the service began, and when it ended.

2.2 Vulnerabilities, Risks and Threats

There is the need of some other formal definitions and practical observations related to the world of computer security. This will outline better concepts and main actors that play an important role in computer security and differentiate one from the other.

Risk: combination of the likelihood of an event and its impact.

Threat: a series of events through which a natural or intelligent adversary (or set of adversaries) could use the system in an unauthorized way to cause harm, such as compromising confidentiality, integrity, or availability of the systems information.

Vulnerability: if computer security is applied to a weakness in a system which allows an attacker to violate the integrity of that system. This weakness of an asset or group of assets can be exploited by one or more threats. Vulnerabilities may result from different reasons such as weak passwords, software bugs, a computer virus, other malware, script code injection or a SQL injection.

Information security tasks are all related to managing and reducing the risks related to information usage in an organization, usually, but not always, by reducing or handling vulnerabilities or threats. Thus, it is wrong to think of security in terms of vulnerability reduction. Security is a component of the organizational risk management process; a set of coordinated activities to direct and control an organization with regard to risk. In other words, information security is the protection of information from a wide range of threats in order to ensure continuity, minimize risk, and maximize return on investments and business opportunities. The main phases of a proper security risk recovery are:

Risk analysis/assessment: process of analyzing threats and vulnerabilities of an information system, and the potential impact that the loss of information or capabilities of a system would have on national security and using the analysis as a basis for identifying appropriate and cost-effective measures. It is the systematic use of information to identify risk sources and to estimate the risk;

Risk evaluation: the process of comparing the estimated risk against given risk criteria to determine the significance of the risk;

Risk management: Process concerned with the identification, measurement, control, and minimization of security risks in information systems.

2.3 Web Applications

Web application, or webapp, is the general term that is normally used to refer to all distributed web-based applications. According to the more technical software engineering definition, a web application is described as an application accessible by the web through a network. Many companies are converting their computer programs into web-based applications. Web Applications are similar to computer-based programs but differ only in that they are accessible through the web, allowing the creation of dynamic websites and providing complete interaction with the end-user. Web Applications are placed on the Internet and all processing is done on the server, the computer which hosts the application [5].

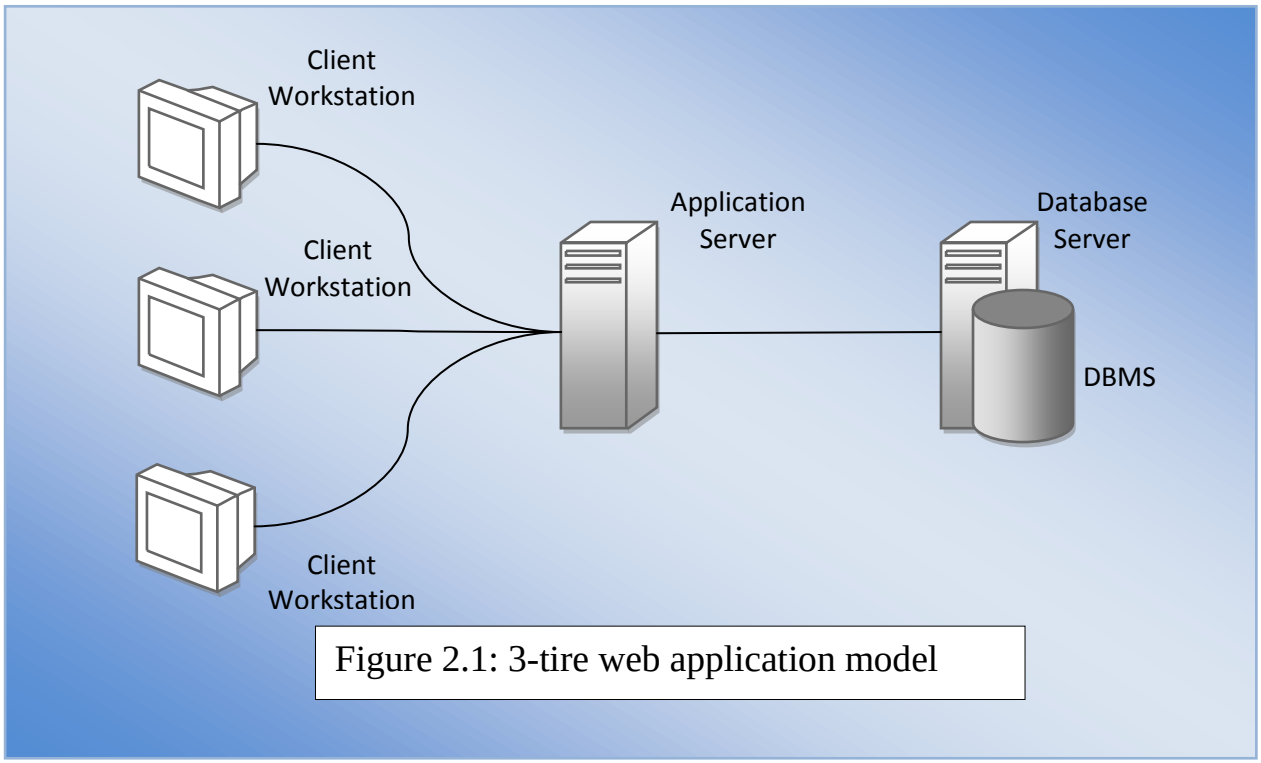


Figure 2.1: 3-tire web application model

Web applications are sets of web pages, files and programs that reside on a company's web server, which any authorized user can access over a network such as the World Wide Web or a local intranet. A web application is usually a three-tiered model. Normally, the first tier is a Web browser on the client side, the second is the real engine on the server-side where the applications core runs, and the third layer is a database as shown in Figure-2.1. A server processes all user transactions and usually the end-user simply accesses the web application by a Web browser, interacting with it. Since web applications reside on a server, they are easy to manage. In fact, they can be updated and modified at any time by the

web applications owner with minimal effort and without any distribution or installation of software on the clients' machines. This is the main reason for the widespread adoption of Web applications in today's organizations [5].

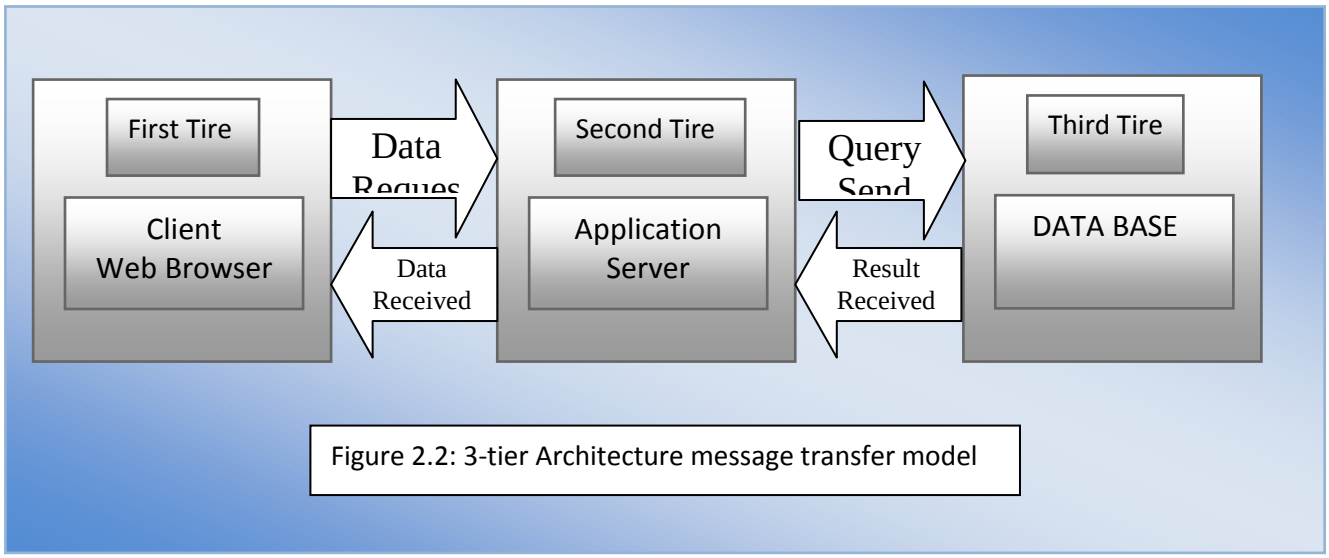


Figure 2.2: 3-tier Architecture message transfer model

Figure 2.2 shows typical system architecture for web applications. This three-tiered architecture consists of a web browser, which functions as the user interface; a web application server, which manages the business logic; and a database server, which manages the persistent data. The web application server receives input in the form of strings from both other tiers: user input from the browser and result sets from the database server. It typically incorporates some of this input into the output that it provides to the other tiers, again in the form of strings: queries to the database server, and HTML documents to the browser, both of which get executed by their respective tiers. The web application server constructs code dynamically, so the code for the entire web application does not exist in any one place at any one time for any one entity to regulate. The flow of data among tiers gives rise to the input validation problem for the web application server; it must check and/or modify incoming strings before processing them further or incorporating them into output that it passes to other tiers to execute. Failure to check or sanitize input appropriately can compromise the web application's security.

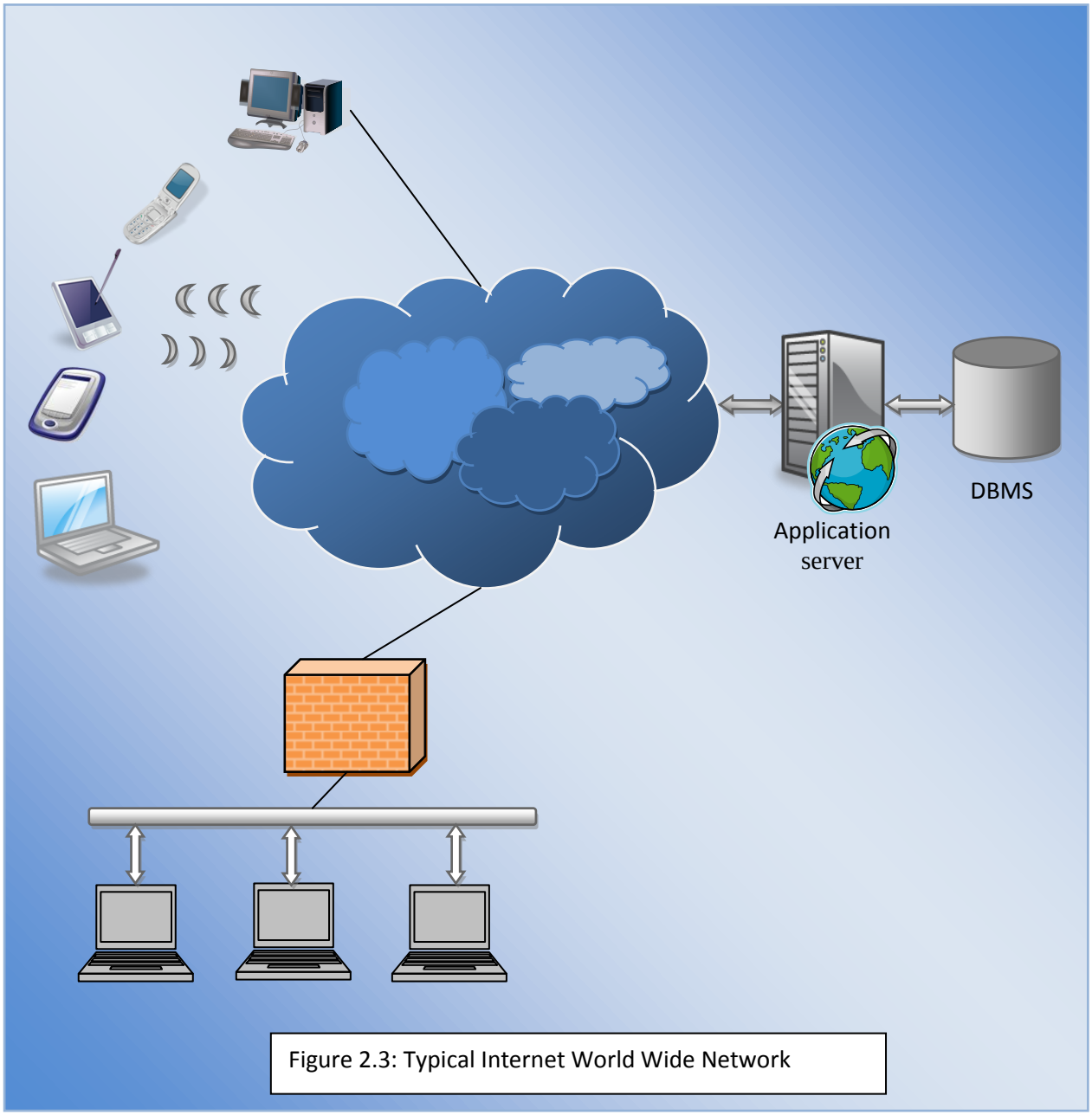


Figure 2.3: Typical Internet World Wide Network

Nowadays, web applications are becoming increasingly popular and are poised to become a major player in the overall software market due to the benefits they afford, such as visibility and worldwide access. They are, without a doubt, essential to the current and next generation of businesses and they have become part of our everyday online lives. In fact, a web application is a worldwide gate accessible not only through standard personal computers but also though different communication devices such as mobile phones and PDAs as shown in figure-2.3. The use of web applications is especially beneficial for a company: with just a little investment, a company can open up a marketing channel that will allow potential clients easy global access to its business 24 hours a day. A typical example of a web application is an online questionnaire or user survey. The end-user client simply completes the online

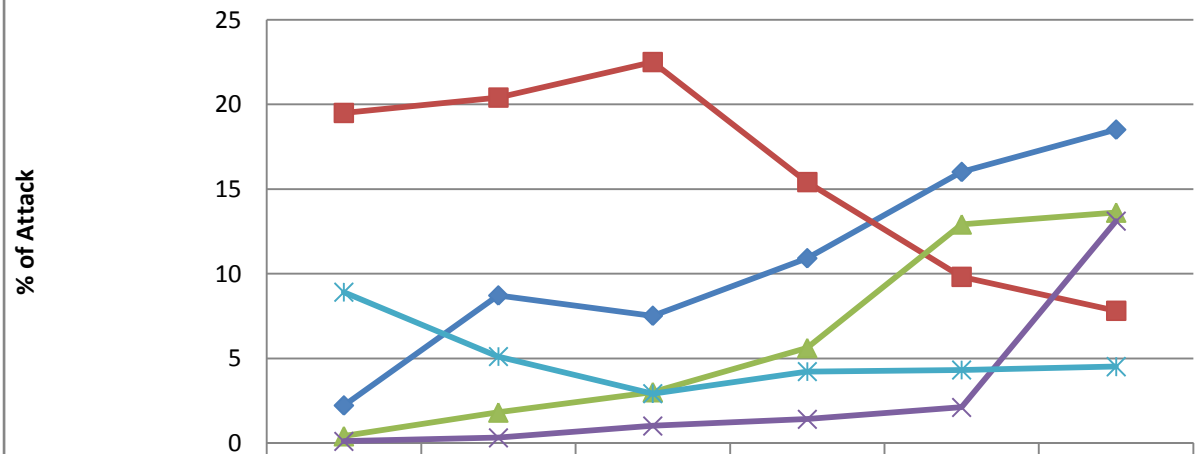
questions by filling in a form that is accessible worldwide through any kind of network device and submits the responses to the application that then collects and stores the data in a database on the server side.

Web applications are present in all aspects of our daily internet use. Common examples are those applications used for searching the internet such as “Google"; for collaborative open source projects as “SourceForge"; for public auctions as “eBay" and many others as well as blogs, webmail, web-forums, shopping carts, e-commerce, dynamic contents, discussion boards and social networks. At the moment, according to survey conducted by cnet.com Gmail, Amazon, yahoo, live search are coming under the top 100 web application [6].The core part of a web application, as stated above, is stored on the server-site within the application server. This core consists of a real computer software program coded in a browser supported programming language such as PHP, ASP, CGI, Perl, Java/JSP, J2EE. Generally, to run the application, you must deploy it in a server and configure it properly. However, the way you install web applications depends on server machine you are using and also the particular application used. In our work we have used J2EE web applications for their good compatibility with our security model. Java 2 Enterprise Edition (this name has since been changed to Java EE version 5.0) is a well-known open source web service platform that uses a distributed multi-tier application model for enterprise java-based applications.

2.3.1 Input Validation-Based Vulnerabilities

The most prominent class of input validation errors are SQL injections. SQL injections are the classes of vulnerabilities in which an attacker causes the web application server to produce HTML documents and database queries, respectively, that the application programmer did not intend. They are possible because, in view of the low-level APIs described above for communication with the browser and database, the application constructs queries and HTML documents via low-level string manipulation and treats un-trusted user inputs as isolated lexical entities. This is especially common in scripting languages such as PHP, which generally do not provide more sophisticated APIs and use strings as the default representation for data and code. Some paths in the application code may incorporate user input unmodified or unchecked into database queries or HTML documents. The modifications/checks of user input on other paths may not adequately constraint the input to function in the generated query or HTML document as the application programmer intended. In that sense, SQL injections are integrity violations in which low-integrity data is used in a high-integrity channel; that is, the browser or the database executes code from an un-trusted user, but does so with the permissions of the application server.

Figure 2.4 : Vulnerability Type Distributions



	2001	2002	2003	2004	2005	2006
XSS	2.2	8.7	7.5	10.9	16	18.5
Buf Overflow	19.5	20.4	22.5	15.4	9.8	7.8
SQL Injection	0.4	1.8	3	5.6	12.9	13.6
PHP Include	0.1	0.3	1	1.4	2.1	13.1
Dir Traverce	8.9	5.1	2.9	4.2	4.3	4.5

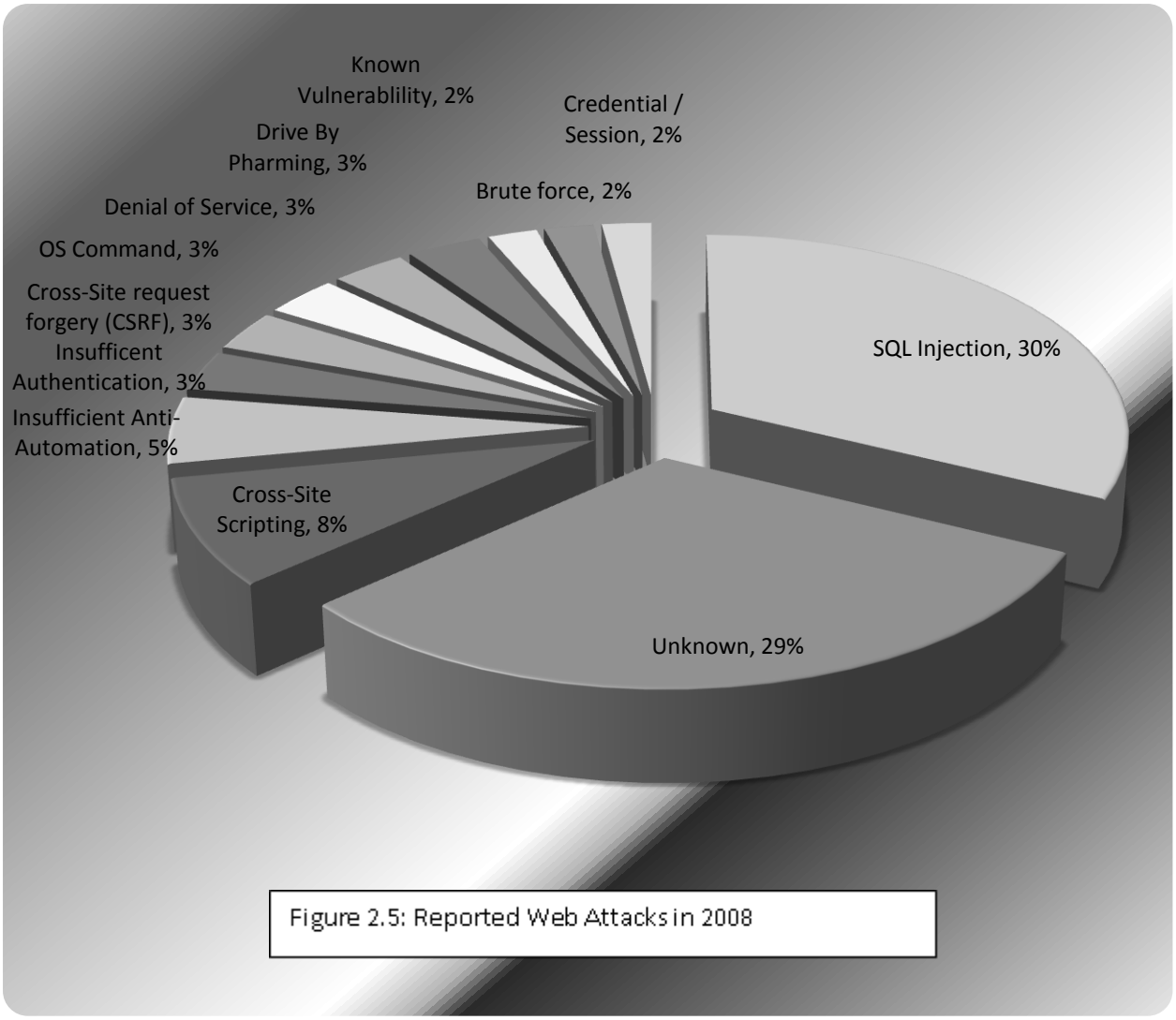
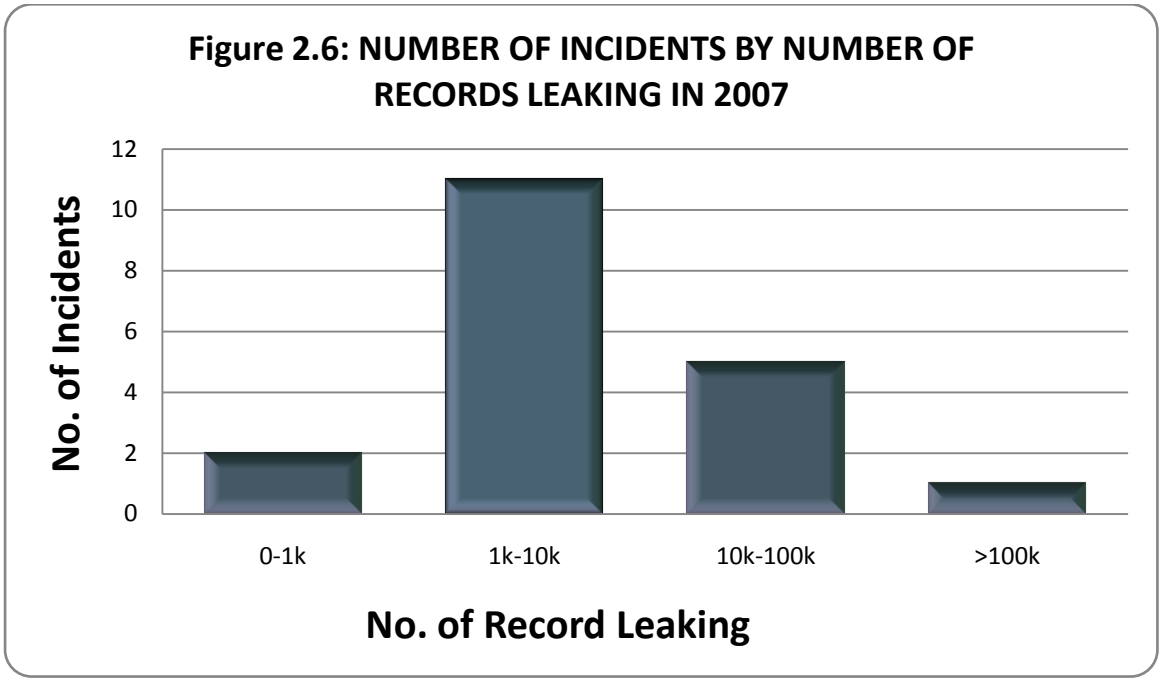


Figure 2.5: Reported Web Attacks in 2008

To highlight how ubiquitous web applications have become and how prevalent their problems are, Figure-2.4 shows, for each year from 2001 to 2006, the percentage of newly reported security vulnerabilities in five vulnerability classes (this data comes from Mitre's report [7]): XSS, SQL injection, PHP file inclusions, buffer overflows, and directory traversals. These were the five most reported vulnerabilities in 2006. All of these except buffer overflows are specific to web applications. Note that SQL injections are consistently at or near the top: 13% of the reported vulnerabilities in 2006, respectively. Some web security analysts speculate that because web applications are highly accessible and databases often hold valuable information, the percentage of SQL injection attacks being executed is significantly higher than the percentage of reported vulnerabilities would suggest. Empirical data supports this hypothesis. Figure-2.5 shows percentages of reported web attacks for the year 2008 (this data comes from the Web Hacking Incidents Database)[9]. Although many

attacks go unreported or even undetected, this chart shows that 30% of the web-based attacks that made the press in 2008 were SQL injections, respectively. This chart also includes attacks on higher-level logic errors, such as weak session IDs.



This kind of attacks can cause severe damage. Typical uses of SQL injection leak confidential information from a database, by-pass authentication logic, or add unauthorized accounts to a database. Figure-2.6 shows the number of reported information leakage attacks in 2007 that leaked the number of records in each of four ranges split by a log-scale (this data comes from the Web Hacking Incidents Database [8]). A news article from May, 2008 gives an example of the kinds of records that Over 1.5 million pages affected by via an SQL injection. It is an attempt to mitigate the impact of the recent waves of SQL injection attacks, and provide more transparency into the approximate number of affected pages [10]. 25th April, 2008 washingtonpost.com sited - Hundreds of thousands of Web sites - including several at the United Nations and in the U.K. government -- have been hacked recently and seeded with code that tries to exploit security flaws in Microsoft Windows to install malicious software on visitors' machines and according to Finnish anti-virus maker F-Secure, the number of hacked Web pages serving up malicious software from this attack may be closer to half a million. Most attacks leaked more than one thousand records, and one leaked more than one hundred thousand records. This illustrates how many users a single attack typically affects.

2.3.2 Most Attacked Organizations

Another aspect we looked into is the type of organizations attackers chose as targets. We found that the largest category of hacked organizations is government and related organizations (Law Enforcement and Politics). Combine those categories with education in 6th place and it appears that the non-commercial sector represents the primary target for hackers (this data comes from the Web Hacking Incidents Database)[9]. Government is a prime target due to ideological reasons, while universities are more open than other organizations. These statistics, however, are biased, to a degree, as the public disclosure requirements of government and other public organizations are much broader than those of commercial organizations

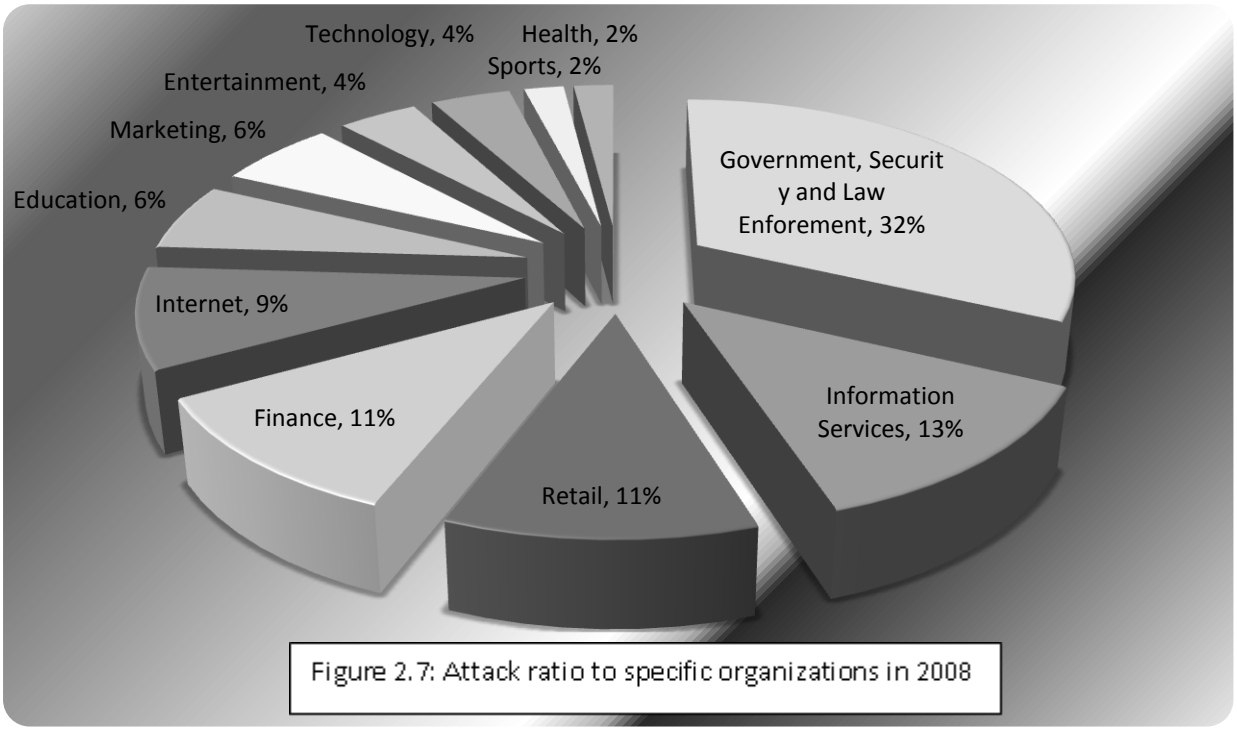


Figure 2.7: Attack ratio to specific organizations in 2008

On the commercial side, Internet-related organizations top the list. This group includes retail shops, comprising mostly e-commerce sites, media companies and pure internet services such as search engines and service providers. It seems that these companies do not compensate for the higher exposure they incur, with the proper security procedures.

2.3.3 SQLIA Recent Scenario

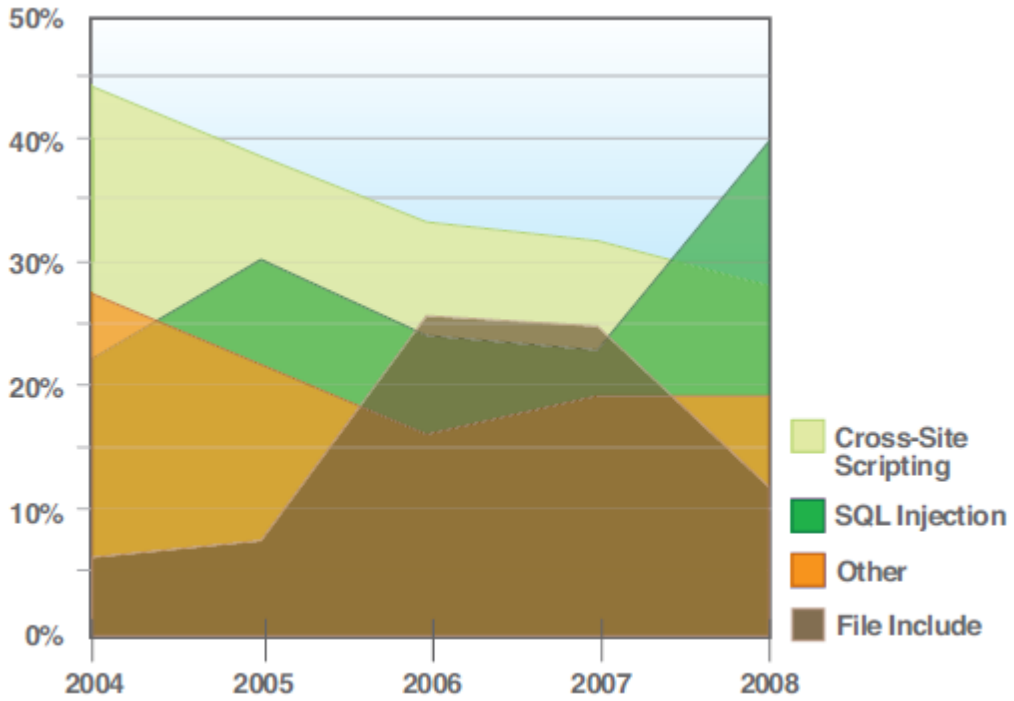


Figure2.8: Web Application Vulnerabilities by Attack Technique, 2004 – 2008 [12]

In 2008, SQL injection replaced cross-site scripting as the predominant Web application vulnerability. In fact, the overall increase of 2008 Web application vulnerabilities can be attributed to a huge spike in SQL injection vulnerabilities, which was up a staggering 134 percent from 2007 shown in Figure 2.8 [12].

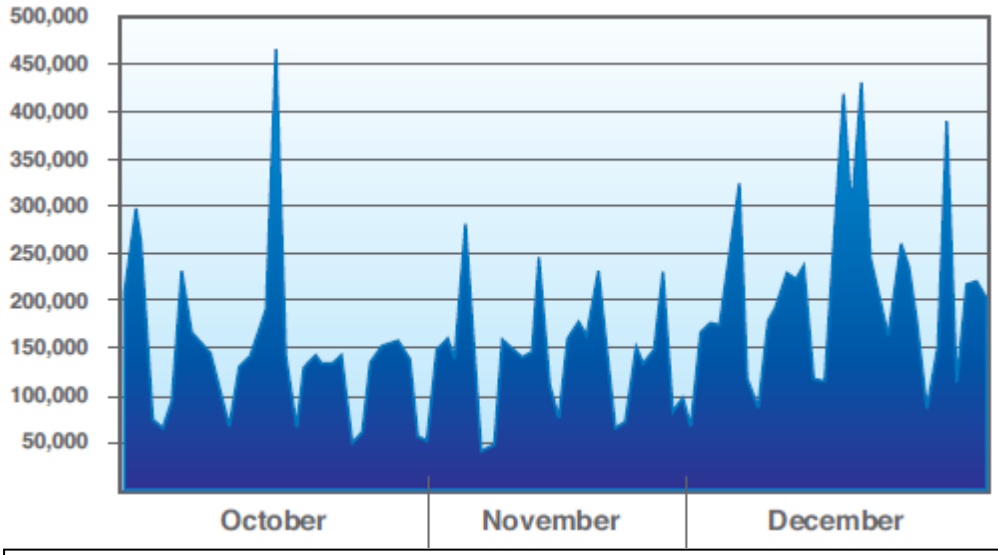


Figure2.9: SQL Injection Attacks Monitored by IBM ISS Managed Security Services, in 4th quarter of 2008 [12]

In the past, most Web server compromises had been one-off, targeted exploitation attempts that steal information or manipulate an application in a way that is beneficial to the attacker. In the fist half of 2008, it has been seen that instead of leveraging SQL injection to steal data, this attack updated the application's back-end data to include iFrames to redirect visitors to malicious Web pages[12]. These attacks targeted many well-known as well as many trusted Web sites also.

2.3.4 Vulnerabilities resolved

Once vulnerabilities are identified it does not necessarily mean they are fixed quickly, or ever. It is interesting to analyze the types and severity of the vulnerabilities that do get fixed (or not) and in what volumes. Some organizations target the easier issues first to demonstrate their progress by vulnerability reduction. Others prioritize the high severity issues to reduce overall risk. Still, resources and security interest are not infinite so some issues will remain unresolved for extended periods of time. Table -2.1 shows the severity of each vulnerability along with the percentage of the vulnerability resolved (this data comes from the Web Hacking Incidents Database)[9].

Classes of Attack	% resolved	Severity
Cross Site Scripting	20%	Urgent
Insufficient Authorization	19%	Urgent
SQL Injection	30%	Urgent
HTTP Response Splitting	75%	Urgent
Directory traversal	53%	Urgent
Abuse of Functionality	28%	Critical
Cross-Site Request forgery	45%	Critical
Session Fixation	45%	Critical
Brute Force	11%	High
Content Spoofing	25%	High
HTTP Response Splitting	30%	High
Information Leakage	29%	High
Predictable Resource Location	26%	High

Table 2.1: Percentage of vulnerabilities resolved in 2008(Sorted by class & severity)

Chapter 3

SQL Injection

SQL injection is a technique that exploits a security vulnerability occurring in the database layer of an application. The vulnerability is present when user input is either incorrectly filtered for string literal escape characters embedded in SQL statements or user input is not strongly typed and thereby unexpectedly executed. It is in fact an instance of a more general class of vulnerabilities that can occur whenever one programming or scripting language is embedded inside another.

3.1 How SQL Injection Attacks (SQLIAs) Work

SQL injection refers to a class of code-injection attacks in which data provided by the user is included in the SQL query in such a way that part of the user's input is treated as SQL code. It is a trick to inject SQL query or command as an input possibly via the web pages. They occur when data provided by user is not properly validated and is included directly in a SQL query. By leveraging these vulnerabilities, an attacker can submit SQL commands directly access to the database. There are two major SQL Injection techniques: i) *access through login page or user input* and ii) *access through URL*.

The first technique is the easiest in which it bypasses the login forms where users are authenticated by using passwords. This kind of technique can be performed
by the attackers through: *'or' condition*, *'having' clause*, *multiple queries* and *extended stored procedure*.

This kind of vulnerability represents serious threats.

SELECT * FROM users WHERE username = ' " & userName & " ' AND password = ' " & userPass & " ' "

If the username and password as provided by the user are used, the query to be submitted to the database takes the form;

SELECT * FROM users WHERE username = 'guest' AND password = 'guestpass'

If the user were to enter [‘ OR 1=1 --] and [ ] instead of [guest] and [guestpass], the query would take the form;

SELECT * FROM users WHERE username = ‘ ‘ OR 1=1 –‘ AND password = ‘ ‘

The query now checks for the conditional equation of [1=1] or an empty password, then a valid row has been found in the *users* table. The first [‘] quote is used to terminate the string and the characters [--] mark the beginning of a SQL comment, and anything beyond is ignored. The query as interpreted by the database now has a tautology and is always satisfied. Thus an attacker can bypass all authentication modules gaining unrestricted access to critical information on the server. SQL injection potentially affects every database on all platform and web application. This attack can be used to gain confidential information, to bypass authentication mechanisms, to modify the database, and to execute arbitrary code.

The second technique can be performed by the attackers through: *manipulating the query string in URL* and using the *SELECT* and *UNION* statements.

When a user enters the following URL:

http://www.mydomain.com/products/products.asp?productid=123

The corresponding SQL query is executed:

SELECT ProductName, ProductDescription FROM Products WHERE ProductNumber = 123

An attacker may abuse the fact that the *ProductID* parameter is passed to the database without sufficient validation. The attacker can manipulate the parameter's value to build malicious SQL statements. For example, setting the value [123 OR 1=1] to the ProductID variable results in the following URL:

http://www.mydomain.com/products/products.asp?productid=123 or 1=1

The corresponding SQL Statement is:

SELECT ProductName, Product Description From Products WHERE ProductNumber = 123 OR 1=1

This condition would always be true and all *ProductName* and *ProductDescription* pairs are returned. The attacker can manipulate the application even further by inserting malicious commands. For example, an attacker can request the following URL:

http://www.mydomain.com/products/products.asp?productid=123;DROP TABLE Products

In this example the semicolon is used to pass the database server multiple statements in a single execution. The second statement is *"DROP TABLE Products*" which causes SQL Server to delete the entire Products table.

An attacker may use SQL injection to retrieve data from other tables as well. This can be done using the SQL *UNION SELECT* statement. The *UNION SELECT* statement allows the chaining of two separate SQL SELECT queries that have nothing in common. For example, consider the following SQL query:

SELECT ProductName, ProductDescription FROM Products WHERE ProductID = '123' UNION SELECT Username, Password FROM Users;

The result of this query is a table with two columns, containing the results of the first and second queries, respectively. An attacker may use this type of SQL injection by requesting the following URL:

http://www.mydomain.com/products/products.asp?productid=123 UNION SELECT user-name, password FROM USERS

In certain circumstances the attacks happened on the database server itself. The security model used by many Web applications assumes that an SQL query is a trusted command. This enables attackers to exploit SQL queries to circumvent access controls, authentication and authorization checks. In some instances, SQL queries may allow access to host operating system level commands. This can be done using stored procedures. Stored procedures are SQL procedures usually bundled with the database server. For example, the extended stored procedure xp_cmdshell executes operating system commands in the context of a Microsoft SQL Server. Using the same example, the attacker can set the value of *ProductID* to be [123; EXEC master..xp_cmdshell dir--], which returns the list of files in the current directory of the SQL Server process.

Even though DBMSs provide access control mechanisms, these mechanisms are not adequate to deal with SQL injection attacks. For that reason, various techniques such as the minimum use of stored procedures, prohibiting display of database server error messages and use of escape sequences for sanitizing user inputs are employed as a quick fix solution. Unfortunately, even these security measures are also inadequate against highly sophisticated SQL injection attacks.

3.2Consequences of SQL Injections Attacks

With SQL injections, cyber-criminals can take complete remote control of the database, with the consequence that they can become able to manipulate the database to do anything they wish, including:

- Insert a command to get access to all account details in a system, including user names and retrieve VNC passwords from registry
- Shut down a database
- Upload files
- Through reverse lookup, gather IP addresses and attack those computers with an injection attack
- Corrupting, deleting or changing files and interact with the OS, reading and writing files
- Online shoplifting e.g. changing the price of a product or service, so that the cost is negligible or free
- Insert a bogus name and credit card in to a system to scam it at a later date
- Delete the database and all its contents

91% of database attacks lead to financial loss [11], but the financial impact can be dwarfed be the long-term damage to an organizations reputation. In fact, research by Ipsos MORI[11] at march 2007 revealed that 58% of consumers would stop using an organization’s services following a security breach involving their personal data.

3.3Classification of SQLIA Techniques

An SQL injection attack has a set of properties, such as assets under threat, vulnerabilities being exploited and attack techniques utilized by threat agents. The detail feature set of every property in the SQL injection attack model is identified in this section.

3.3.1 Attack Intention

When a threat agent utilizes a crafted malicious SQL input to launch an attack, the attack intention is the goal that the threat agent tries to achieve once the attack has been successfully executed.

Identifying Inject-able Parameters [13]: Inject-able parameters are the parameters or the user input fields of the Web applications directly used by server-side program logic to construct SQL statements, which are vulnerable to SQLIA. In order to launch a successful attack, a threat agent must first discover which parameters are vulnerable to SQL injection attack.

Performing database finger-printing [13]: Database finger-print is the information that identifies a specific type and version of database system. Every database system employs a different proprietary SQL language dialect. For example, the SQL language employed by Microsoft SQL server is T-SQL while Oracle SQL server uses PL/SQL. In order for an attack to be succeeded, the attacker must first find out the type of and version of database deployed by a web application, and then craft malicious SQL input accordingly.

Determining database schema [13]: Database schema is the structure of the database system. The schema defines the tables, the fields in each table, and the relationships between fields and tables. Database schema is used by threat agents to compose a correct subsequent attack in order to extract or modify data from database.

Bypassing Authentication [13]: Authentication is a mechanism employed by web application to assert whether a user is who he/she claimed to be. Matching a user name and a password stored in the database is the most common authentication mechanism for web applications. Bypassing authentication enables an attacker to impersonate another application user to gain un-authorized access.

Extracting Data [13]: In most of the cases, data used by web applications are highly sensitive and desirable to threat agents. Attacks with intention of extracting data are the most common type of SQL injection attacks.

Adding or Modifying Data [13]: Database modification provides a variety of gains for a threat agent, for instance, a hacker can pay much less for a online purchase by altering the price of a product in the database. Or, the threads in a online discussion forum can be modified by an attacker to launch subsequent Cross-Site-Scripting attacks.

Performing denial of service [13]*:* 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 [13]: This category refers to certain attack techniques that are employed to avoid auditing and detection by system protection mechanisms.

Executing Remote Commands [13]: Remote commands are executable code resident on the compromised database server. Remote command execution allows an attacker to run arbitrary programs on the server. Attacks with this type of intention could cause entire internal networks being compromised.

Performing privilege escalation [13]: Privileges are described in a set of rights or permissions associated with users. Privilege escalation allows an attacker to gain un-authorized access to a particular asset by associating a higher privilege set of rights with a current user or impersonate a user who has higher privilege.

Downloading File: Downloading files from a compromised database server enable an attacker to view file content stored on the server. If the target web application resides on the same host, sensitive data such as configuration information and source code will be disclosed too.

Uploading File: Uploading files to a compromised database server enable an attacker to store any malicious code onto the server. The malicious code could be a Trojan, a back door or a worm that can be used by an attacker to launch subsequence attack.

3.3.2 Assets

Assets are information or data an unauthorized threat agent attempt to gain.

Database Server Fingerprint: The database server fingerprint contains information about the database system in use. It identifies the specific type and version of the database, as well as the corresponding SQL language dialect. A compromise of this asset may allow attackers to construct malicious code specifically for the SQL language dialect in question.

Database Schema: The database schema describes the server's internal architecture Database structure information such as table names, size, and relationships are defined in the database schema. Keeping this asset private is essential in keeping the confidentiality and integrity of the database data. A compromise in the database schema may allow attackers to know the exact structure of the database, including table, row, and column headings.

Database Data: The database data is the most crucial asset in any database system. It contains the information in the tables described in the database schema, such as prices in an online store, personal information of clients, administrator passwords, etc. A compromise in the database data will usually result in failure of the system's intended functionality, thus, its confidentiality and integrity must be protected.

Host: A host is a discrete node in any network, usually uniquely defined with an IP address. It may have various privileges in a network and may be a database server or a regular computer terminal.

Network: A network interconnects numerous hosts together and allows communication between them. A compromise in a network will most likely compromise every host in the network. Some networks may also be interconnected with other networks, furthering the potential damage, should an attack be successful.

3.3.3 Vulnerabilities

Insufficient Input Validation: Input validation is an attempt to verify or filter any given input for malicious behavior. Insufficient input validation will allow code to be executed without proper verification of its intention. Attackers taking advantage of insufficient input validation can utilize malicious code to conduct attacks.

Privileged Account: A privileged account has a degree of freedom to do what normal accounts can not. Its actions may also be exempt from auditing and validation. This presents vulnerability since a jeopardized privileged account, such as an administrator account, can compromise much more than what a jeopardized regular account can.

Extra Functionality: Extra functionalities meant to provide a broader range of vulnerability, since combinations of this functionality may result in unintended actions. For example, xp_cmdshell is meant to provide users with a way of executing operating system commands, but commonly used to added unauthorized users into the operating system.

3.4 Methodology for a Successful SQLIA

Attack techniques are the specific means by which a threat agent carries out attacks using malicious code. Threat agents may use many different methods to achieve their goals, often combining several of these sequentially or employing them in different varieties.

3.4.1 Attacks Techniques

Tautology: The general goal of a tautology-based attack is to inject code in one or more conditional statements so that they always evaluate to true. The consequences of this attack depend on how the results of the query are used within the application. The most common usages are to bypass authentication pages and extract data. In this type of injection, an attacker exploits an inject-able field that is used in a query's WHERE conditional. Transforming the conditional into a tautology causes all of the rows in the database table targeted by the query to be returned. In general, for a tautology-based attack to work, an attacker must consider not only the inject-able/vulnerable parameters, but also the coding constructs that evaluate the query results. Typically, the attack is successful when the code either displays all of the returned records or performs some action if at least one record is returned.

Example: In this example attack, an attacker submits [' or 1=1 - -] for the *user name* input field (the input submitted for the other fields is irrelevant). The resulting query is:

SELECT status FROM users WHERE user_name='' or 1=1 – AND city= ' '

The code injected in the condition [OR 1=1] transforms the entire WHERE clause into a tautology. The database uses the conditional as the basis for evaluating each row and deciding which ones to return to the application. Because the conditional is a tautology, the query evaluates to true for each row in the table and returns all of them.

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 [- -] after inputs so that remaining queries are not treated as executable code, but comments. This is useful since threat agents may not always know the syntax or fields in the server.

Example: In this example attack, an attacker submits [admin'--] for the *user name* input field (the input submitted for the other fields is irrelevant). The resulting query is:

SELECT * FROM user WHERE username = 'admin'--' AND password = ‘’

The code injected in the condition [admin’-] transform the WHERE clause in that way it is going to log the attacker as admin user if there is a user name called “admin”, because rest of the SQL query will be ignored.

Illegal/Logically Incorrect Query: This attack lets an attacker gather important information about the type and structure of the back-end database of aWeb application. The attack is considered a preliminary, information gathering step for other attacks. The vulnerability leveraged by this attack is that the default error page returned by application servers is often overly descriptive. In fact, the simple fact that an error messages is generated can often reveal vulnerable/inject-able parameters to an attacker. Additional error information, originally intended to help programmers debug their applications, further helps attackers gain information about the schema of the back-end database. When performing this attack, an attacker tries to inject statements that cause a syntax, type conversion, or logical error into the database. Syntax errors can be used to identify inject-able parameters. Type errors can be used to deduce the data types of certain columns or to extract data. Logical errors often reveal the names of the tables and columns that caused the error.

Example: This example attack’s goal is to cause a type conversion error that can reveal relevant data. To do this, the attacker injects the following text [UNION SELECT TOP 1 COLUMN_NAME FROM INFORMATION_SCHEMA.COLUMNS WHERE TABLE_NAME='admin_login'--] into the following URL:

http://www.mydomain.com/products/products.asp?productid=123

The resulting query is:

http://www.mydomain.com/products/products.asp?productid=123 UNION SELECT TOP 1 COLUMN_NAME FROM INFORMATION_SCHEMA.COLUMNS WHERE TABLE_NAME='admin_login'—

The injected query extracts the 1st column name of 'admin_login' table from the information _schema database (assume the application is using Microsoft SQL Server). The query then converts the table name into an integer but this is not a legal type conversion, the database throws an error. For Microsoft SQL Server, the error would be:

" Microsoft OLE DB Provider for ODBC Drivers error '80040e07' [Microsoft][ODBC SQL Server Driver][SQL Server]Syntax error converting the nvarchar value 'login_id' to a column of data type int./index.asp, line 5"

There are two useful pieces of information in this message that aid an attacker. First, the attacker can see that the database is an SQL Server database, as the error message explicitly states this fact. Second, the error message reveals the value of the string that caused the type conversion to occur. In this case, this value is also the name of the first column name of 'admin_login' table in the database: "login_id." A similar strategy can be used to systematically extract the name and type of each column in the database. Using this information about the schema of the database, an attacker can then create further attacks that target specific pieces of information.

Union Query: In union-query attacks, an attacker exploits a vulnerable parameter to change the data set returned for a given query. With this technique, an attacker can trick the application into returning data from a table different from the one that was intended by the developer. Attackers do this by injecting a statement of the form: UNION SELECT <rest of injected query>. Because the attackers completely control the second/injected query, they can use that query to retrieve information from a specified table. The result of this attack is that the database returns a dataset that is the union of the results of the original first query and the results of the injected second query.

Example: Referring to the running example, an attacker could inject the text [’ UNION SELECT username, password from user_info where user_name=’abc’- -] into the login field, which produces the following query:

SELECT * FROM users WHERE login=’’ UNION SELECT password from user_info where user_name=’abc’-- AND pass=’’

Assuming that there is no login equal to ”, the original first query returns the null set, whereas the second query returns data from the “user_info” table. In this case, the database would return column “password” for username “abc.” The database takes the results of these two queries, unions them, and returns them to the application. In many applications, the effect of this operation is that the value for “password” is displayed along with the user information.

Piggy-backed Query: In this attack type, an attacker tries to inject additional queries into the original query. We distinguish this type from others because, in this case, attackers are not trying to modify the original intended query; instead, they are trying to include new and distinct queries that “piggy-back” on the original query. As a result, the database receives multiple SQL queries. The first is the intended query which is executed as normal; the subsequent ones are the injected queries, which are executed in addition to the first. This type of attack can be extremely harmful. If successful, attackers can insert virtually any type of SQL command, including stored procedures,1 into the additional queries and have them executed along with the original query. Vulnerability to this type of attack is often dependent on having a database configuration that allows multiple statements to be contained in a single string.

Example: If the attacker inputs [’; drop table users - -] into the *password* field, the application generates the query:

SELECT accounts FROM users WHERE login=’doe’ AND pass=’’; drop table users -- ’

After completing the first query, the database would recognize the query delimiter (“;”) and execute the injected second query. The result of executing the second query would be to drop table users, which would likely destroy valuable information. Other types of queries could insert new users into the database or execute stored procedures. Note that many databases do not require a special character to separate distinct queries, so simply scanning for a query separator is not an effective way to prevent this type of attack.

System Stored Procedure: SQLIAs of this type try to execute stored procedures present in the database. Today, most database vendors ship databases with a standard set of stored procedures that extend the functionality of the database and allow for interaction with the operating system. Therefore, once an attacker determines which backend database is in use, SQLIAs can be crafted to execute stored procedures provided by that specific database, including procedures that interact with the operating system.

It is a common misconception that using stored procedures to write Web applications renders them invulnerable to SQLIAs. Developers are often surprised to find that their stored procedures can be just as vulnerable to attacks as their normal applications [13]. Additionally, because stored procedures are often written in special scripting languages, they can contain other types of vulnerabilities, such as buffer overflows, that allow attackers to run arbitrary code on the server or escalate their privileges [13]

Example: consider this stored procedure [13]

```
CREATE PROCEDURE DBO.isAuthenticated
        @userName varchar2, @pass varchar2, @pin int
AS
        EXEC("SELECT accounts FROM users
        WHERE login='" +@userName+ "' and pass='" +@password+ "' and
        pin=" +@pin);
GO
```

This example demonstrates how a parameterized stored procedure can be exploited via an SQLIA. In the example, we assume that the query string constructed at lines 5, 6 and 7 of our example has been replaced by a call to the stored procedure defined above. The stored procedure returns a true/false value to indicate whether the user's credentials authenticated correctly. To launch an SQLIA, the attacker simply injects [' ; SHUTDOWN; --] into either the user Name or password fields. This injection causes the stored procedure to generate the following query:

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

At this point, this attack works like a piggy-back attack. The first query is executed normally, and then the second, malicious query is executed, which results in a database shut down. This example shows that stored procedures can be vulnerable to the same range of attacks as traditional application code.

Inference: In this attack, the query is modified to recast it in the form of an action that is executed based on the answer to a true/false question about data values in the database. In this type of injection, attackers are generally trying to attack a site that has been secured enough so that, when an injection has succeeded, there is no usable feedback via database error messages. Since database error messages are unavailable to provide the attacker with feedback, attackers must use a different method of obtaining a response from the database. In this situation, the attacker injects commands into the site and then observes how the function/response of the website changes. By carefully noting when the site behaves the same and when its behavior changes, the attacker can deduce not only whether certain parameters are vulnerable, but also additional information about the values in the database. There are two well known attack techniques that are based on inference. They allow an attacker to extract data from a database and detect vulnerable parameters.

Blind Injection: In this technique, the information must be inferred from the behavior of the page by asking the server true/false questions. If the injected statement evaluates to true, the site continues to function normally. If the statement evaluates to false, although there is no descriptive error message, the page differs significantly from the normally-functioning page.

Timing Attacks: 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 inject-able 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=’’

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

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 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=''

In this attack the SUBSTRING function is used to extract the first character of the first table's name. Using a binary search strategy, the attacker can then ask a series of questions about this character. In this case, the attacker is asking if the ASCII value of the character is greater-than or less-than-or-equal-to the value of *X*. If the value is greater, the attacker knows this by observing an additional 5 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.

Alternate Encodings: In this attack, the injected text is modified so as to avoid detection by defensive coding practices and also many automated prevention techniques. This attack type is used in conjunction with other attacks. In other words, alternate encodings do

not provide any unique way to attack an application; they are simply an enabling technique that allows attackers to evade detection and prevention techniques and exploit vulnerabilities that might not otherwise be exploitable. These evasion techniques are often necessary because a common defensive coding practice is to scan for certain known "bad characters," such as single quotes and comment

To evade this defense, attackers have employed alternate methods of encoding their attack strings (e.g., using hexadecimal, ASCII, and Unicode character encoding). Common scanning and detection techniques do not try to evaluate all specially encoded strings, thus allowing these attacks to go undetected. Contributing to the problem is that different layers in an application have different ways of handling alternate encodings. The application may scan for certain types of escape characters that represent alternate encodings in its language domain. Another layer (e.g., the database) may use different escape characters or even completely different ways of encoding. For example, a database could use the expression char(120) to represent an alternately-encoded character "x", but char(120) has no special meaning in the application language's context. An effective code-based defense against alternate encodings is difficult to implement in practice because it requires developers to consider of all of the possible encodings that could affect a given query string as it passes through the different application layers. Therefore, attackers have been very successful in using alternate encodings to conceal their attack strings.

Example: Because every type of attack could be represented using an alternate encoding, here we simply provide an example of how esoteric an alternatively-encoded attack could appear. In this attack, the following text is injected into the *login* field: [legalUser'; exec(0x73687574646f776e) - - ". The resulting query generated by the application is:

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

This example makes use of the char() function and of ASCII hexadecimal encoding. The char() function takes as a parameter an integer or hexadecimal encoding of a character and returns an instance of that character. The stream of numbers in the second part of the injection is the ASCII hexadecimal encoding of the string "SHUTDOWN." Therefore, when the query is interpreted by the database, it would result in the execution, by the database, of the SHUTDOWN command [13].

3.4.2 Evasion Techniques

Evasion techniques are obscuring techniques employed in an attack to avoid detection by signature-based detection system. In the context of SQL injection detection, a signature is the pattern of known attack strings. SQL injection attack occurs when input string alter the intended syntactical structure of SQL statement. Thus, an attack signature usually consists of one or more SQL keywords, delimiters and expressions. Signature-based detection systems build a database of attack signatures, and then examine input strings against the signature database at runtime in detection of attacks. Evasion techniques obscure input strings, making look different but yielding the same results when executed by a database server.

Sophisticated Matches: One of the common signatures used by such mechanisms is some sort of variant on the famous OR 1=1 attack. Sophisticated matches evasion technique uses alternative expression of *"OR 1=1"*.

Example: *OR 'Unusual' = 'Unusual', OR 'Simple' = 'Sim'+'ple'*, *OR 2 > 1* and *OR 'Simple' BETWEEN 'R' AND 'T'* all have the same effect as "*OR 1=1"*.

Hex Encoding: Hex encoding evasion technique uses hexadecimal encoding to represent a string. For example, the string 'SELECT' can be represented by the hexadecimal number 0x73656c656374, which most likely will not be detected by a signature protection mechanism.

Example: SELECT LOAD_FILE (0x633A5C626F6F742E696E69)
This will show the content of c:\boot.ini

Char Encoding: Char encoding evasion technique uses build-in CHAR function to represent a character.

Example: the string 'SELECT' can be represented by the CHAR function as char(73)+char(65)+"LECT", which make it very difficult for detection system to build a signature that match it.

In-line Comment: In-line comment evasion technique obscures input strings by inserting in-line comments between SQL keywords.

Example: */**/UNION/**/SELECT/**/name* can escape detection from signatures that expects white space between SQL keywords.

Dropping White Space: Dropping white space evasion technique obscures input strings by dropping white space between SQL keyword and string or number literals. For example,

OR'Simple'='Simple' works exactly the same way as *OR*

'Simple' = 'Simple', but has no spaces in it, make it capable of evading any spaces based signature.

Break Words in the Middle: With MySQL, the in-line comments would not work as a replacement for a space. The in-line comments can be used in MySQL to break words in the middle, for instance: *UN/**/ION/**/ SE/**/LECT/**/* is evaluated as *UNION SELECT*.

3.4.3 Countermeasures

There are a number of ways a programmer/system administrator can prevent or counter attacks made on their systems.

Parameterized Query: Parameterized query is parameterized database access API provided by development platform such as PrepareStatement in Java or SQLParameter .NET. Instead of composing SQL by concatenating string , each parameter in a SQL query is declared using place holder and input is provided separately [14].

Least Privilege: The account that an application uses to access the database should have only the minimum permissions necessary to access the objects that it needs to use.

Different Accounts: Use a different database account for a task that requires a different level of privilege.

Customized Error Message: Threat agents may gain access to knowledge through overly informative error messages, yet completely removing error messages makes debugging a difficult task. Customized error messages hinder the reconnaissance progress of threat agents, particularly in deducing specific details such as inject-able parameters, etc.

System Stored Procedure Reduction: Once a threat agent gains knowledge of which back-end server is used, he/she has knowledge of an entire set of system stored procedures that are available. By limiting the system stored procedures one can execute on a server, especially the processes that are not used, one can reduce or even eliminate vulnerabilities that may arise from these stored procedures.

SQL Keyword Escaping: Escape specific SQL keyword or delimiter in the input string.

Input Variable Length Checking: By checking for input variable length, malicious code strings beyond certain length limits will not be applicable. Even if the length limitation is long enough to fit a few additional queries, the inability to input an infinitely long string disables the threat agent from employing evasion techniques such as encoding, and consequently, allows signature based detection mechanisms to intercept simple attacks.

Although these techniques remain the best way to prevent SQL injection vulnerabilities, but their application is problematic in practice. These techniques are prone to human error and are not as rigorously and completely 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

Chapter 4

4.1 Observations

If we closely analyze the code structure of an 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 dynamically generated query then we can determine that the dynamically generated query is a SQLIA or a valid query.

4.2 Related Work

In this section, we list the technique closely related to ours and discuss their advantages and disadvantages.

Framework Support: Recent frameworks for web applications provide a functionality that can be used to prevent SQL injections. An input validator prohibits user input from including meta-characters to avoid SQL injections. But if we want to include meta- characters in the input we can not prevent SQL injections.

Prepare Statement [15]: SQL provides the prepare statement, which separates the values in a query from the structure of SQL. The programmer defines a skeleton of an SQL query and then fills in the holes of the skeleton at runtime. The prepare statement makes it harder to inject SQL queries because the SQL structure can not be changed. To use the prepare statement, we must modify the web application entirely; all the legacy web applications must be re-written to reduce the possibility of SQL injections.

Static Analysis [16]: Wassermann and Su proposed an approach that uses a static analysis combined with automated reasoning. This technique verifies that the SQL queries generated in the application usually do not contain a tautology. This technique is effective only for SQL injections that insert a tautology in the SQL queries, but can not detect other types of SQL injections attacks.

Machine Learning Approach [17]: Valeur *et al.* proposed the use of an intrusion detection system (IDS) based on a machine learning technique. IDS is trained using a set of typical application queries, builds models of the typical queries, and then monitors the application at runtime to identify the queries that do not match the model. The overall IDS quality depends on the quality of the training set; a poor training set would result in a large number of false positives and negatives.

Instruction-Set Randomization [18]: SQLrand provides a framework that allows developers to create SQL queries using randomized keywords instead of the normal SQL keywords. A proxy between the web application and the database intercepts SQL queries and de-randomizes the keywords. The SQL keywords injected by an attacker would not have been constructed by the randomized keywords, and thus the injected commands would result in a syntactically incorrect query. Since SQLrand uses a secret key to modify keywords, its security relies on attackers not being able to discover this key. SQLrand requires the application developer to rewrite code.

Taint-Based Technique [19]: Pietraszek and Berghe modified a PHP interpreter to track taint information at the character level. This technique uses a context-sensitive analysis to reject SQL queries if an un-trusted input has been used to create certain types of SQL tokens. A common drawback of this approach is that they require modifications to the runtime environment, which diminishes the portability.

Combined Static and Dynamic Analysis: Su *et al.* [20] present grammar-based approach to detect and stop queries having SQLIAs by implementing SQLCHECK tool. They mark user supplied portions in queries with a special symbol and augment the standard SQL grammar with production rule. A parser is generated based on the augmented grammar. The parser successfully parses the generated query at runtime, if there are no SQLIAs in the generated queries after adding user inputs. This approach uses a secret key to discover user inputs in the SQL queries. Thus, the security of the approach relies on attackers not being able to discover the key. Additionally, this approach requires the application developer to rewrite code to manually insert the secret keys into dynamically generated SQL queries.

Buehrer *et al* [21]. secure vulnerable SQL statements by comparing the parse tree of a SQL statement before and after input and only allowing SQL statements to execute if the parse trees match. They conducted a study using one real world web application and applied their SQLGUARD solution to each application. They found that their solution stopped all of the SQLIAs in their test set without generating any false positives. While it stopped all of the SQLIAs, their solution required the developer to rewrite all of their SQL code to use their custom libraries, which is an overhead we are trying to eliminate. However, the approach is ineffective, if the user supplied input does not appear at the leaf of the tree.

Halfond *et al*[22]. developed AMNESIA (Analysis for Monitoring and NEutralizing SQL Injection Attack), which is a model based technique that combines the static and dynamic analyses. The tool first identifies hotspots, where SQL queries are issued to database engines. At each hotspot, a query model is developed by using non deterministic finite automata (NDFA). The hotspot is instrumented with monitor code, which matches the dynamically generated query against the query model. If a generated query is not consumed by NDFA, then it is an attack. The accuracy of AMNESIA depends on that of the static analysis. They conducted a study using five real world web applications and two student-created web applications and applied their AMNESIA solution to each application. They found that their solution stopped all of the SQLIAs in their attack set without generating any false positives. Their solution throws an exception for each SQLIA, which the developer handles and builds in attack recovery logic. Unfortunately, to use this technique significant change should be needed in the source code. For run time validation it take a query string and the ID of the hotspot that generated the query, retrieves the SQL-query model for that hotspot using the ID, and checks the query against the model. For that reason it cannot be easily used for an existing application due to the over head of the rewritten of the source code.

In our work we mainly concentrate on the technique describe by AMNESIA and try to remove the necessity of the source code modification as well as to minimize the runtime response time.

4.3 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. Database 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's 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, other wise if we do not found any ordered sequence like the arrived query in our entire database then it's a possible SQLIA. As we stored all the possible structure of the valid SQL query in our database in static analysis phase or its better to say the offline computation phase so it is not possible for the hacker to perform an SQLIA.

As this technique we does 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* [24] 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 [22].

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 multi threaded way, where each thread is responsible to search each individual link list stored the structure of a individual query. If a thread found the correct path it intimate the other threads to terminate as is already performs a successful search. As due to the hardware constant only few numbers of executable cores are available in the processor so that much number of threads cannot run parallel, so we have to predict the possible link list from among many link lists for a quick success. If we closely analyze a web application most of the cases similar type of query is used with different user input value and among the wide variety of queries few queries are frequently used. So while searching if we give preference to paths mostly traversed then the hit ratio will be increased. While start searching, we first pick up the mostly traversed path among the entire possible alternative path to find the desired one quickly. To do this we include a hit count at each of the node of the link list which points the starting node of another link list which stored a valid query structure. When a thread perform a successful search as well as it intimate the other threads to stop searching it also increase the hit count on that link list and ordered the entire link list which store the starting node of the link lists which store the structure on the query in an descending order. To use the threading support we use the JAVA Thread Pool so we can control the number of threads running in a system. So if we issue all the threads at same time the mechanism provided by the JAVA Thread pool only execute the first few threads as we instruct, and wait for these few threads to complete then it initialize the waiting threads in the thread queue, so to search the most traversed path first we store the list according to the hit count in an ascending order [23].

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 a valid query. Then the thread informs all other thread to stop execution as it found a valid query structure. Then it update the hit count field of the node of the main link list which store the starting address of that searched single link list and if the hit count is greater than the hit count of the previous node of the main link list then it swap the node with the previous node. But in this matching process if a mismatch occurs and the node in the link list say's that it is a position of user input then we skip to the next token in the token sequence of the incoming query as well as move forward to the link list and try to match the token with the token in the current pointing node assuming that the previous token is a user input data value. If we found a match at this position we continue our matching process or if a mismatch found at that position then it is clear that the link list we are searching for the similar query structure for the incoming query is not correct list so the thread searching this singly list should stop its execution. The search for the similar structure for the incoming query is done in this fashion in a multithreaded way. If no similar type of structure found in the dataset then it's a possible SQLIA. Figure 4.1 shows node structure on the singly as well as main link list.

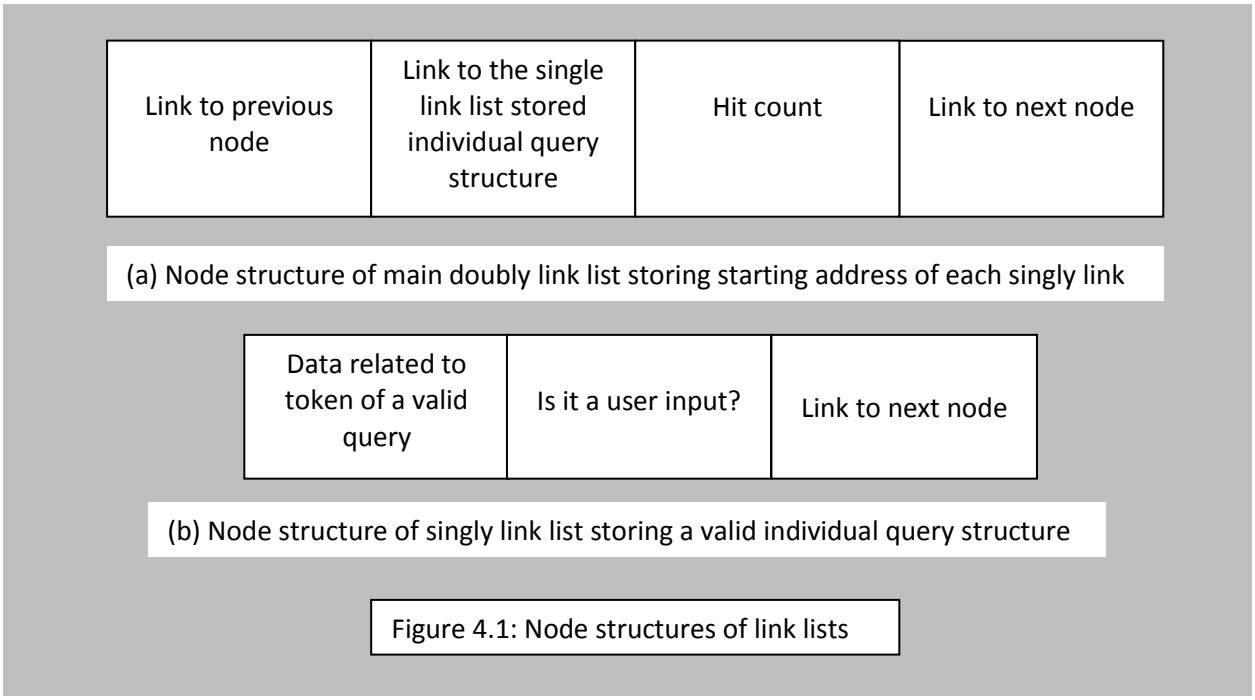


Figure 4.1: Node structures of link lists

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 4.2 shows a sequence of tokens extracted from an incoming query.

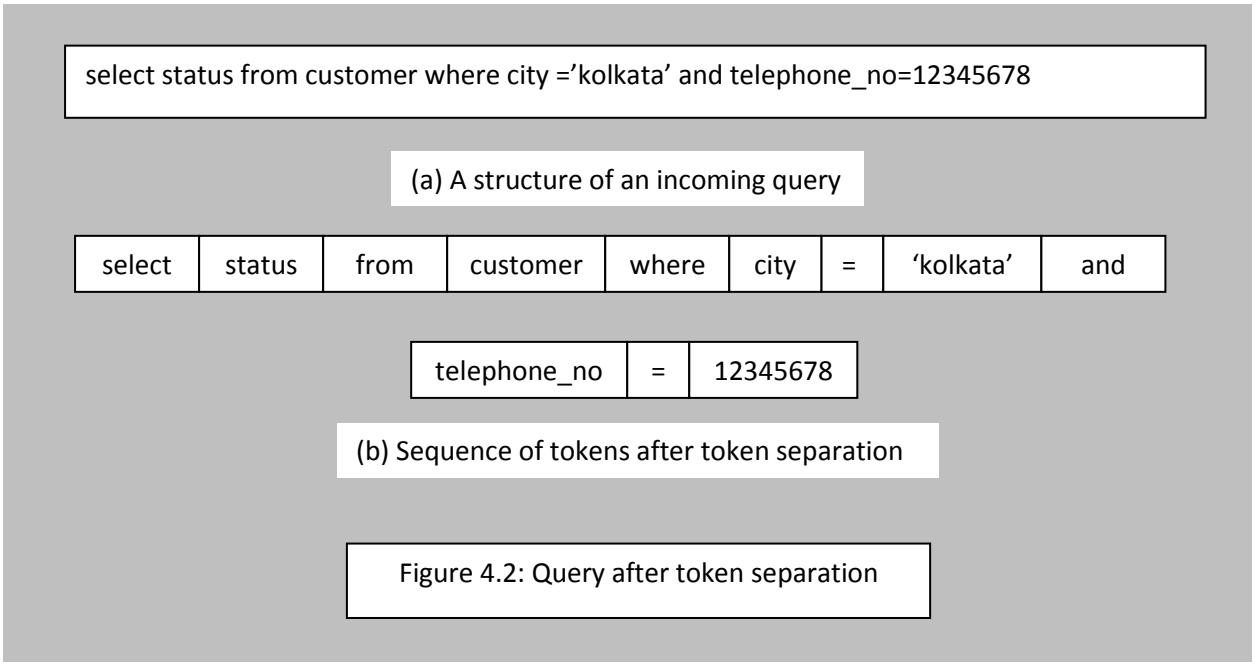


Figure 4.2: Query after token separation

For runtime token matching if we used literal wise matching like others [22][24] then it will be a huge computational over head. For example if there are 'n' literals are in incoming query string and there are 'q' query structures available in the database of the same length of the 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. 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 for example if there are 'n' no of literals and 'm' no of tokens in the query instead of storing 'n' no of values we are storing 'm' no of values where m<<n. So in run time validation the length of a singly link list storing the query having 'm' no of tokens is m. In cases of incoming query after token separation we transform each token into its corresponding integer values then we start our searching. In worst case if we assume that the mismatch occurs at the last position of each and every singly link list and if it is an unsuccessful search then we have to check 'm' no of integer values for each 'q' no of query, so the complexity become O(m*q) instead of O(n*q) where m<<n. So this is a

performance gain. 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 'SELECT' the corresponding ascii decimal values for the literals is S=83, E=69, L=76, E=69, C=67, T=84, now the position of each literal is S=1, E=2, L=3, E=4, C=5, T=6, so after multiplying the ascii value of each literals with its position we have to add the values up. So now the value of the token becomes 83*1+69*2+76*3+69*4+67*5+84*6=1564. So the corresponding integer value of 'SELECT' is 1564. Figure 4.3 shows a valid query translation to its corresponding integer sequence.

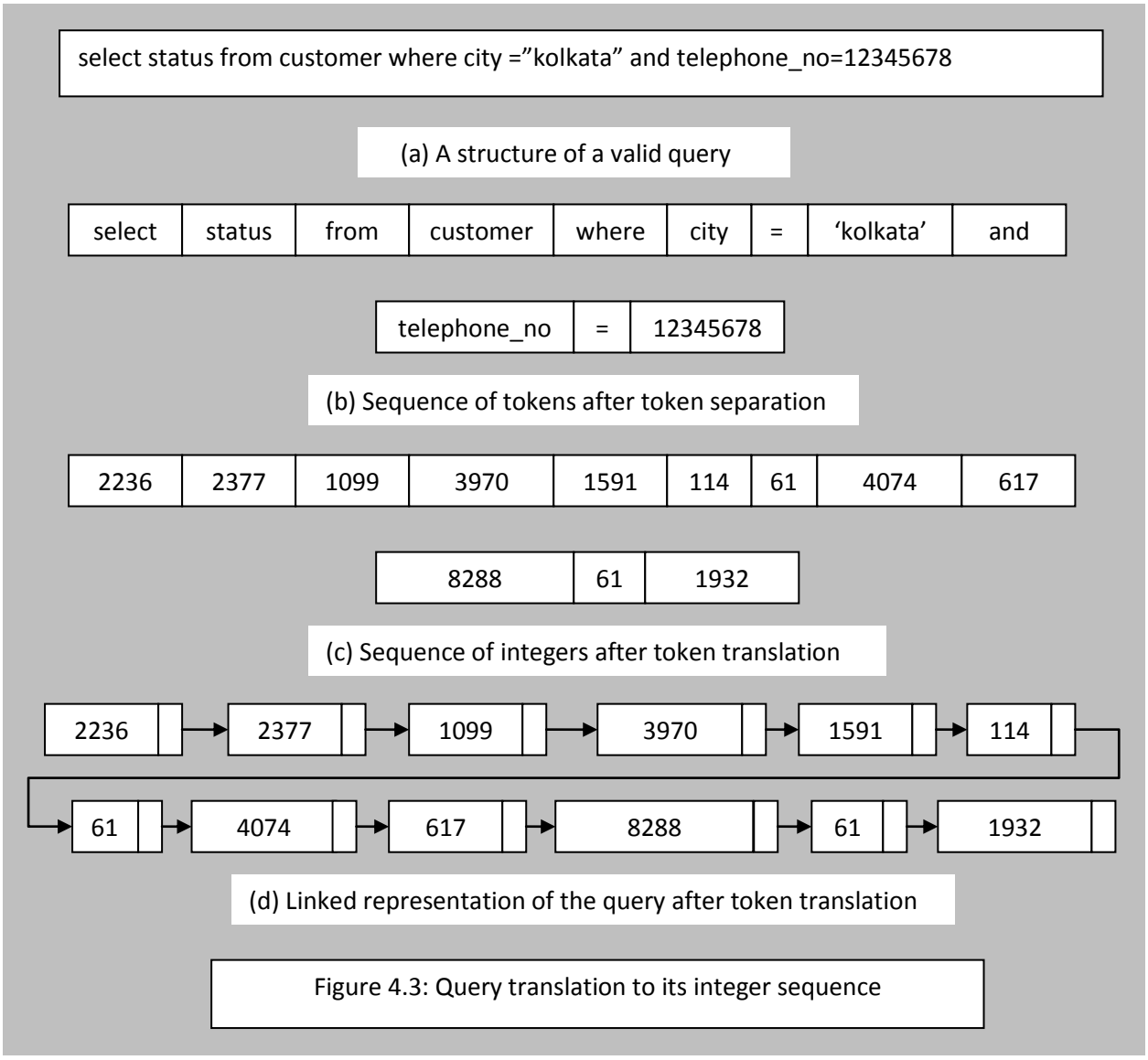


Figure 4.3: Query translation to its integer sequence

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 4.4 shows the group representation of queries having 4 tokens.

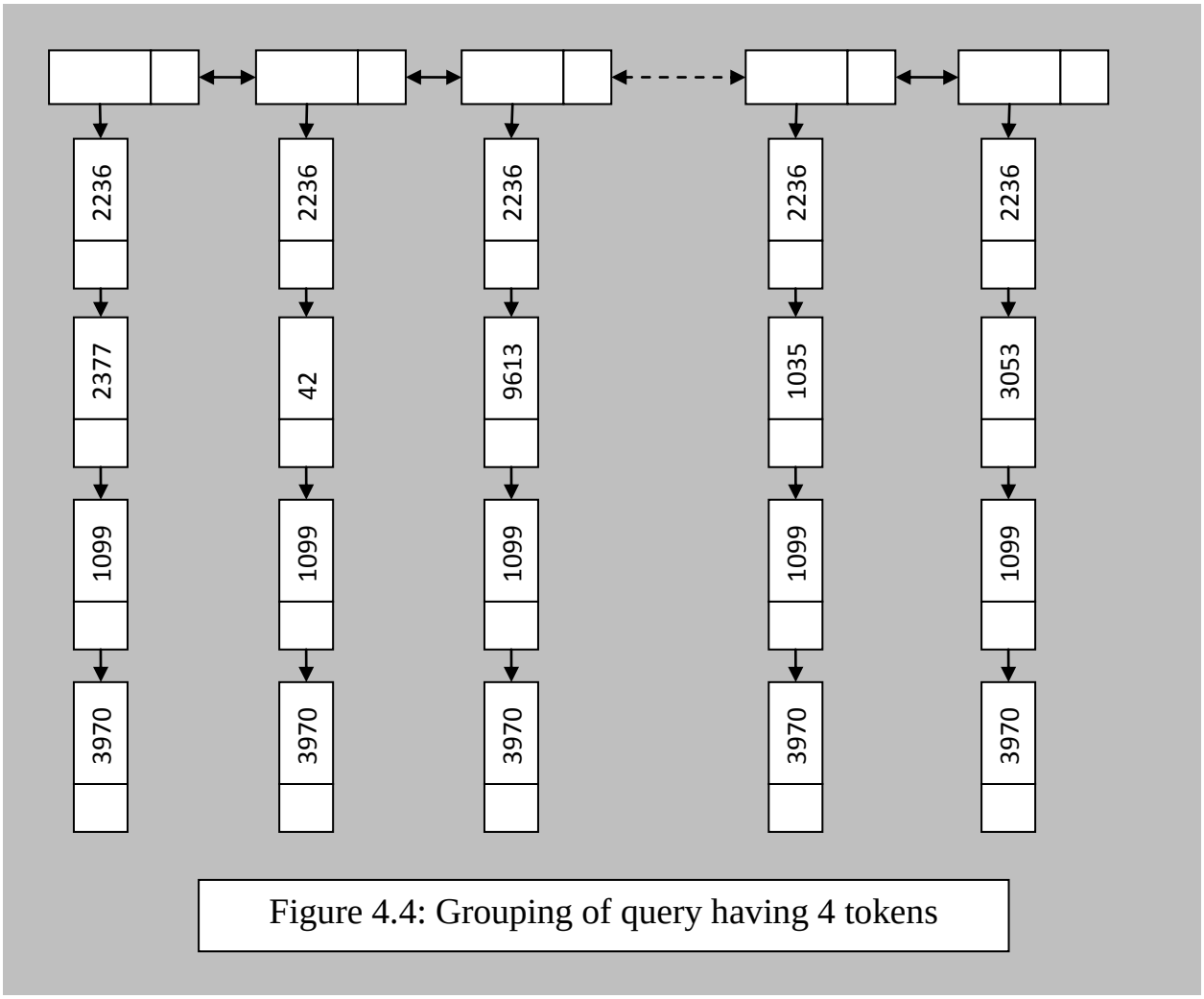


Figure 4.4: Grouping of query having 4 tokens

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' number of tokens then the starting address of the group be assigned to n^{th} 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. Figure 4.5 shows the complete structure to store all valid queries.

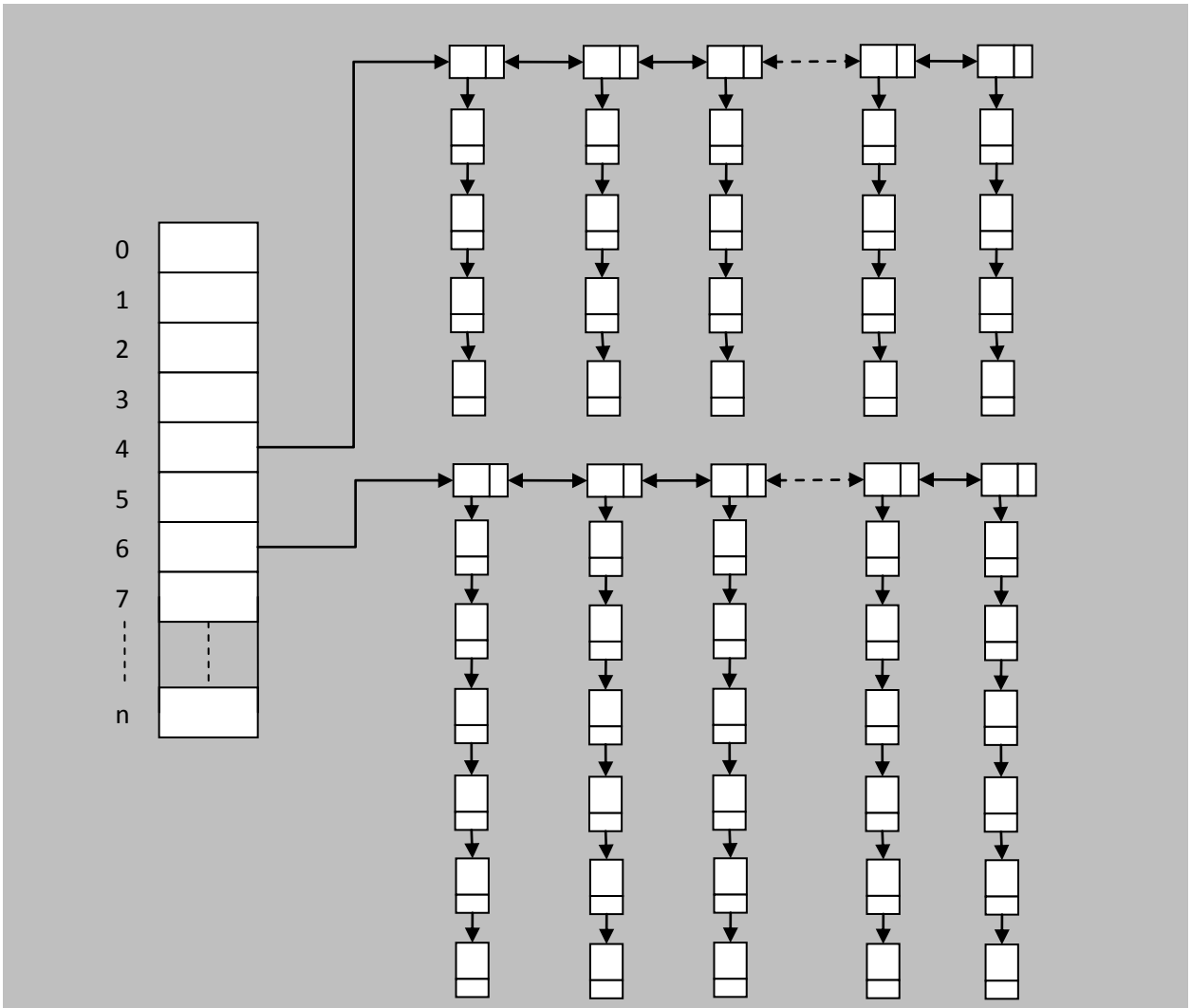


Figure 4.4: Complete structure to store all valid queries

4.4 Proposed System Architecture

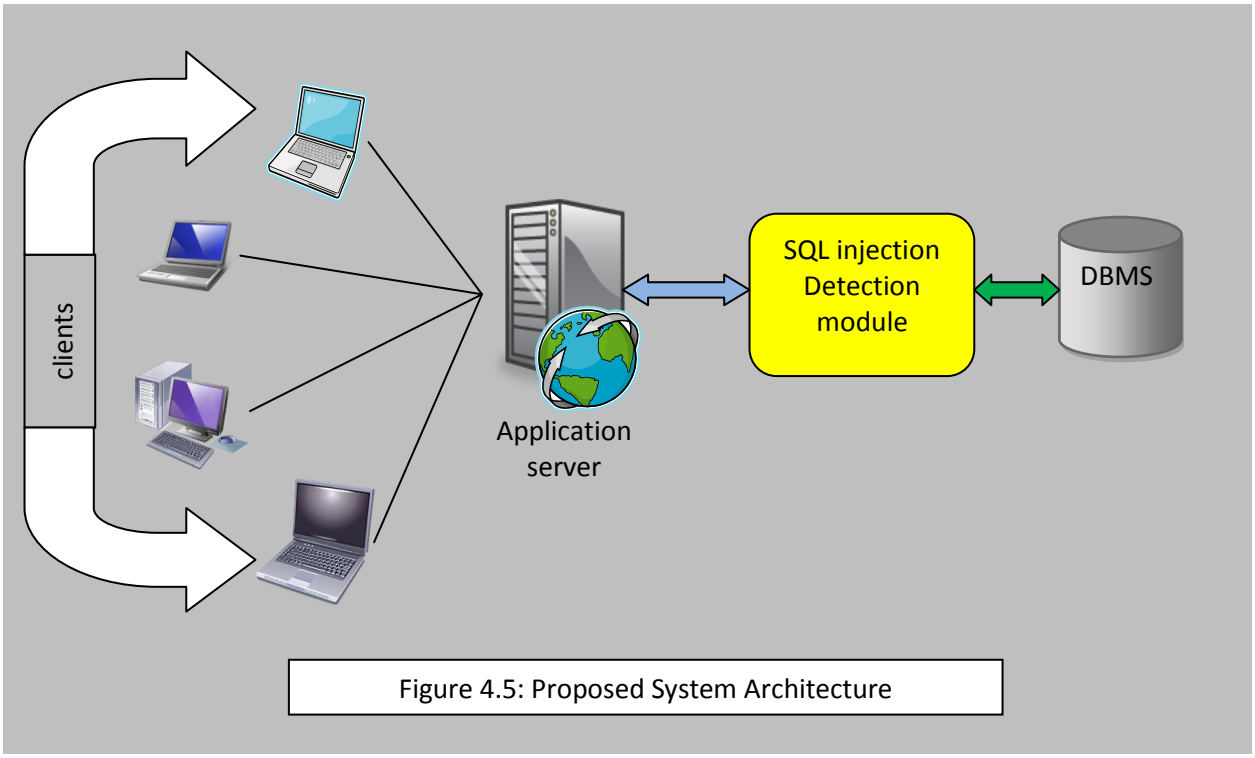


Figure 4.5: Proposed System Architecture

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.

4.5 Algorithm

traceSQL(SQLstmt,SQL_DB_array)

// SQLstmt is a query string

//SQL_DB_array is an array holding the starting address of all groups storing individual array

{

//lex_of_sql( ) separates each token from the SQLstmt and store in token_sequence and returns total number of token in a SQLstmt

//token_sequence is a 2D array where each row represents a token and each cell holds a literal of that token and total number of rows represent total number of tokens it stored

no_of_token=lex_of_SQL(token_sequence,SQLstmt);

//token_to_int( ) convert token into its corresponding integer value

//int_sequence is an integer array where each cell stores an integer value of a token

token_to_int(no_of_token,token_sequence,int_sequence);

//initialize starting node of the double link list to search

search_node=SQL_DB_array[no_of_token];

while(search_node->rlink != NULL) do

{

//start_thread_search( ) search for a ordered sequence of integer provided by int_sequence in the link list whose starting address is stored in search_node.

Create thread start_thread_search(no_of_token, int_sequence, search_node);

search_node=search_node->rlink

}

```
        //wait( ) wait for all the threads to complete
        wait( );
        // unsuccessful_search( ) return true if all threads failed to perform a successful search
        if (unsuccessful_search ( ))
                return faule; //its an SQL injection
        else
                return true; // it's a valid query not a SQL injection
} // end traceSQL

token_to_int(no_of_token, token_sequence, int_sequence )
        //token_sequence is a 2D array where each row represents a token and each cell holds
        a literal of that token and total number of rows represent total number of tokens it
        stored
        //int_sequence is an integer array whose each cell stores an integer value of a token
        //no_of_token the total number of token in token_sequence
        for i=1 to no_of_token do
        {
                j=0; sum=0;
                While(token_sequence[i][j] != END_OF_TOKEN) do
                {
        // END_OF_TOKEN is a special character inserted by lex_of_SQL to
        determine the end of the token. to_decimal( ) returns a decimal ascii value of a
        literal
                        Sum=sum+to_decimal(token_sequence[i][j]);
```

```
                j=j+1;
            }
        }


start_thread_search(no_of_token, int_sequence, search_node)
{
        node=search_node->start_node;
        i=0;
        while(i < no_of_token) do
        {
            if( int_sequence[i] == node->data)
            {
                i=i+1;
                node=node->link;
            }
            else
                if( node->is_user_input)
                {
                    i=i+1;
                    node=node->link;
                }
                else
                    break; //exit from while
        }
```

```
        if(i == no_of_token)
                // it's a successful search
                Search_node->hit_count=search_node->hit_count +1;
                if ( search_node->llink->hit_count < search_node->hitcount)
                {
                        temp=search_node->llink;
                        search_node->llink=temp->llink;
                        temp->llink=search_node;
                        temp->rlink=search_node->rlink;
                        search_node->rlink=temp;
                }
                // found( ) informs all other thread that a successful match found so they stop
                found( );
        else
                // stop the execution of that thread
                terminate( );
} // end of start_thread_search
```

4.6 Analysis Implementation and Results

The main advantage of that algorithm is that it uses the modern multi-core architecture available to minimize response time. And there is always a difference between a prediction and without prediction by calculating hit count it predict the possible list to search to minimize the response time. The main complexity of this algorithm is in three procedures 1st token separation, 2nd token to integer conversion and 3rd the main searching in the link list. In token separation its depend upon the database we are implementing because different database have different key word set, different function name as well as different syntax, and these token separation technique is same for all existing solution to detect SQLIA so here the complexity of token separation is same with all existing solutions. The complexity of token to integer conversion process in O(n) where n is the total number of literals in all tokens of the query because we have to scan each literal exactly once to get its ascii decimal value. In 3rd procedure, the main searching procedure the worst case happens if the mismatch will occur at the end of the every list and it is an unsuccessful search, so we have to traverse the length of every singly link list. If the length of the singly link list is 'm' and there are 'q' no of link list we have to search then the complexity will be O(m*q). In best case we found the desired query structure in the very first link list we traversed then we have to traversed only 'm' no of links where m is the total number of tokens in the query the complexity will be O(m). If we use a literal wise string checking like other methods in worst case it is O(n*q), where n is the total number of literals in the query and n>>m.

We implement this algorithm in java because it support multi-threading technique and through the help of java pool we can also restrict the number of threads to run at a time because only few cores are available in the processor.

The results are given below.

No of SQL	SQL statement
	select branch-name from loan
2	select distinct branch-name from loan
3	select all branch-name from loan
4	select loan-number,branch-name,amount*100 from loan
5	select loan-number from loan where branch-name = ? and amount > ?
6	select loan-number from loan where amount between ? and ?
7	select loan-number from loan where amount = ? and amount >= ?
8	select customer-name,borrower.loan-number,amount from borrower,loan where borrower.loan-number = loan.loan-number
9	select customer-name,borrower.loan-number,amount from borrower,loan where borrower.loan-number = and branch-name = ?
10	select customer-name,borrower.loan-number,amount from borrower,loan where borrower.loan-number = loan.loan-number
11	select customer-name,borrower.loan-number as loan-id,amount from borrower,loan where borrower.loan-number = loan.loan-number
12	select customer-name,t.loan-number,s.amount from borrower as t,branch as s where t.assets > s.assets and s.branch-city = ?
13	select customer-name from customer where customer-street like '%main%'
14	select distinct customer-name from borrower.loan-number = loan.loan-number and branch-name = ? order by customer-name
15	select * from loan order by amountdesc, loan-number asc
16	select customer-name from depositor
17	select customer-name from borrower
18	(select customer-name from depositor)union (select customer-name from borrower)
19	(select customer-name from depositor) union all (select customer-name from borrower)
20	(select distinct customer-name from depositor) intersect (select distinct customer-name from borrower)
21	(select customer-name from depositor) intersect all (select customer-name from borrower)
22	(select distinct customer-name from depositor) except (select customer-name from borrower)
23	(select customer-name from depositor) except all (select customer-name from borrower)

Table 4.1 Input SQL Database [31]

No of Sql	SQL Statement
24	select avg (balance) from account where branch-name = ?
25	select branch-name,avg (balance) from account group by branch-name
26	select branch-name,count (distinct customer-name) from depositor,account where depositor.account-number = account.account-number group by branch-name
27	select branch-name,avg (balance) from account group by branch-name having avg (balance) > ?
28	select avg (balance) from account
29	select count (*) from customer
30	select depositor.cutomer-name,avg (balance) from depositor,account,customer where depositor.account-number = account.account-number and depositor.customer-name = customer.customer-name and customer-city = ?
31	selct loan-number from loan where amount is null
32	select distinct customer-name from borrower where customer-name in (select customer-name from depositor)
33	select distinct customer-name from borrower,loan where borrower.loan-number = loan.loan-number and branch-nume = ? and (branch-name,customer-name) in (select branch-name,customer-name from depositor,account where depositor.account-number = account.account-number)
34	select distinct customer-name from borrower where customer-name not in (select customer-name from depositor)
35	select distinct t.branch-name from branch as t,branch as s where t.assets > s.assets and s.branch-city = ?
36	select branch-name from branch where assets > some (select assets from branch where branch-city = ?)
37	select branch-name from branch where > all (select assets from branch where branch-city = ?)
38	select branch-name from account group by branch-name having avg (balance) >= all (select avg (balance) from account group by branch-name)
39	select customer-name from borrower from brrower where exists (select * from depositor where depositor.customer-name = borrower.customer-name)
40	select distinct s.customer-name from depositor as s where not exists ((select branch-name from branch where branch-city = ?) except (select r.branch-name from depositor as t,account as r where t.account-number = r.account-number and s.customer-name = t.customer-name))
41	select distinct t.customer from depositor t where not unique (select r.customer-name from account,depositor as r where t.customer-name = r.customer-name and r.account-number = account.account-number and account.branch-name = ?)

Table 4.1 Input SQL Database [31]

No of SQL	Injected SQL Statement
1	select loan-number from loan where branch-name = '' or 1 = 1 and amount > 1200
2	select loan-number from loan where branch-name = 'admin' --' and amount > 1200
3	select loan-number from loan where branch-name = 'admin' /*' and amount > 1200
4	select loan-number from loan where branch-name = 'perryridge' and amount > 1200 union select * from loan
4	select loan-number from loan where branch-name = 'perryridge' and amount > 1200 ; select * from loan
5	select loan-number from loan where branch-name = 'perryridge' and amount > 1200; drop table loan
6	select loan-number from loan where branch-name = 'perryridge' and amount > 1200; select load_file (0x633A5C626F6F742E696E69)
7	select loan-number from loan where branch-name = '' or 'simple' = 'simple' and amount > 1200
8	select loan-number from loan where branch-name = 'legalUser' and ASCII (SUBSTRING ((select top 1 name from sysobjects),1,1)) > X WAITFOR 5 -- ’ and amount > 1200
9	select loan-number from loan where branch-name = 'legalUser'; exec (char (0x73687574646f776e)) -- ' and amount > 1200

Table 4.2 Sample Injected SQL Query

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)
1062	12372	11	942	120	457	515

Table 4.3 Performance analysis

No of query	Total false positive	Total false negative
250(valid)+50(SQL Injection)	0	0

Table 4.4 Accuracy Result

System Configuration: Pentium® 4 CPU 3.00GHz, 2 GB RAM

No of Threads used: 2

4.5 Advantages and Limitations

As it is a multi threaded implementation is fully utilized the newly available multi core processors and performs the search quickly. Due to use of hit count techniques the frequently generated SQL will be processed quickly which is a performance gain to the existing available solution. As this scheme validates each dynamically generated SQL at runtime, it increases the runtime overload of the system but reduces the possibility of SQLIA. This technique can only prevent the SQLIA which alter the structure of the existing query but if there is no alteration of existing structure of query then the technique fails to detect such type of attacks. For example if someone has an account in as application maintaining telephone database of an telephone service provider and after authenticate himself as a valid user the hacker try to obtain the information regarding to a his phone number but while sending the information regarding to his account to the application server in the intermediate stage he changed all the data value of his account to the data value of another account, here the application server recognized the data input is a valid and authenticated input so it provides all the information regarding to those data to the hacker which he is not authorized to view.

Chapter 5

5.1 Conclusion

Most web applications employ a middleware technology designed to request information from a relational database in SQL parlance. SQL injection is a common technique hackers employ to attack these web-based applications. These attacks reshape the SQL queries, thus altering the behavior of the program for the benefit of the hacker. In this paper, we present a technique for detecting and preventing SQLIA incidents. The technique abstracts the intended SQL query behavior in an application in the form of an ordered sequence of tokens, as a one-time offline procedure using static analysis of the application code. This database is then validated against the entire different incoming SQL query at runtime to capture all malicious SQL queries, before they are sent to the database server for execution. To minimize searching time and response time it uses the modern processor architecture by perform the searching in a multi threaded way as well as it predict the possible correct list for an incoming query by introducing hit count calculation. This technique also avoid the computational over head of string matching algorithm for token matching by converting the token in its corresponding integer value and store the tokens in terms of its integer values in the database. This technique helps in capturing all the different types and modes of execution of SQLIAs, in a transparent manner requiring no modification to the underlying application source and as well as providing an overall efficiency.

5.2 Future Work

This security module can only processes a single SQL query it cannot handle PL/SQL code blocks so in future we expand our solution to handle PL/SQL code block. As we told that there is a limitation of this technique that it cannot detect such type of attack caused by only changing the data value of a query but not changing the structure of the query so we will try to improve our solution which can eliminate all those problems.

References

[1] Wikipedia, “information security” http://en.wikipedia.org/wiki/Information_security

[2] University of Nevada, Las Vegas, http://www.unlv.edu/infotech/infosec/index.html

[3]Cultural communication, Chicago,

http://www.cultural.com/web/security/infosec.glossary.html

[4] Wikipedia, “AAA protocol”, http://en.wikipedia.org/wiki/AAA_protocol

[5] Wikipedia, “web application”, http://en.wikipedia.org/wiki/Web_application

[6] CNET.com. 2009 Webware 100 winners. 2009. http://www.webware.com/100/

[7] Steve Christey. Vulnerability Type Distributions in CVE. cwe.mitre.org, Oct 2006.

http://cwe.mitre.org/documents/vuln-trends.html.

[8] Ofer Shezaf. The Web Hacking Incidents Database Annual Report 2007. Breach Security Whitepaper, 2007

http://www.breach.com/assets/files/resources/breach_security_labs/2008/02/The%20Web%20Hacking%20Incidents%20Database%20Annual%20Report%202007.pdf

[9] Breach .com, The Web Hacking Incidents Database Annual Report 2009. Breach Security Whitepaper, Feb 2009.

http://www.breach.com/resources/whitepapers/downloads/WP_WebHackingIncidents_2008.pdf

[10] Dancho Danchev, Over 1.5 million pages affected by the recent SQL injection attacks, ZDNet.com,20th May 2008. http://blogs.zdnet.com/security/?p=1150

[11] secerno.com,” SQL Injection Attack: A Security Threat”,

http://www.secerno.com/?pg=SQL-Injection#2

[12] IBM, IBM Internet Security SystemsX-Force 2008 Trend & Risk Report, Jan 2009, http://www-935.ibm.com/services/us/iss/xforce/trendreports/xforce-2008-annual-report.pdf

[13] W. Halfond, J. Viegas, and A. Orso. A Classification of SQL-Injection Attacks and Countermeasures. *Proceedings of the IEEE International Symposium on Secure Software Engineering (ISSSE)*, 2006.

[14] Wikipedia, “SQL injection” http://en.wikipedia.org/wiki/SQL_injection

[15] Stephen Thomas, Laurie Williams. *Using Automated Fix Generation to Secure SQL Statements.* Third International Workshop on Software Engineering for Secure Systems (SESS'07), pages 9-9,May 2007.

[16] G. Wassermann and Z. Su. An Analysis Framework for Security in Web Applications. *In Proceedings of the FSE Workshop on Specification and Verification of Component-Based Systems (SAVCBS)*, pages 70–78, 2004.

[17] F. Valeur, D. Mutz, and G. Vigna. A Learning-Based Approach to the Detection of SQL Attacks. *In Proceedings of the Conference on Detection of Intrusions and Malware andVulnerability Assessment (DIMVA)*, pages 123–140, 2005.

[18] S. Boyd and A. Keromytis. SQLrand: Preventing SQL injection attacks. *In Proceedings of the Applied Cryptography and Network Security (ACNS)*, pages 292–304, 2004.

[19] T. Pietraszek and C. Vanden Berghe. Defending against Injection Attacks through Context-Sensitive String Evaluation. *In Proceedings of the 8th International Symposium on Recent Advances in Intrusion Detection (RAID)*, pages 124–145, 2005.

[20] Z. Su and G. Wassermann. The Essence of Command Injection Attacks in Web Applications. *Annual Symposium on Principles of Programming Languages (POPL)*, pages 372–382, 2006.

[21] G. Buehrer, B. W. Weide, and P. A. G. Sivilotti, "Using Parse Tree Validation to Prevent SQL Injection Attacks," 5th International Workshop on Software Engineering and Middleware, pages 106-113, 2005

[22] W. Halfond and A. Orso. AMNESIA: Analysis and Monitoring for NEutralizing SQL-Injection Attacks. *In Proceedings of the 20th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 174–183, 2005.

[22] M. Muthuprasanna, Ke Wei, Suraj Kothari. Eliminating SQL Injection Attacks - *A Transparent Defense Mechanism. Eighth IEEE International Symposium on Web Site Evolution (WSE'06)*,pages 22-32,Sept,2006.

[23] Anderson Bailey. The Secret of Java Thread Pools. developer.amd.com, Nov 2006.

http://developer.amd.com/documentation/articles/Pages/1121200683.aspx

[24] William G.J. Halfond and Alessandro Orso. Preventing SQL Injection Attacks Using AMNESIA. *Proceedings of the 28th international conference on Software engineering*. Pages 795-798, May 2006

[25] Asmawi, Aziah; Sidek, Zailani Mohamed; Razak, Shukor Abd. System architecture for SQL injection and insider misuse detection system for DBMS. *International Symposium on Information Technology*. Pages 1 - 6 Aug 2008

[26] Bertino, Elisa, Kamra, Ashish; Early, James P. Profiling Database Application to Detect SQL Injection Attacks. *IEEE International Performance, Computing, and Communications Conference*. Pages 449 – 458, April 2007

[27] Xiang Fu; Xin Lu, Peltsverger, B. Shijun Chen, Kai Qian, Lixin Tao. A Static Analysis Framework For Detecting SQL Injection Vulnerabilities. *31st Annual International Computer Software and Applications Conference.* Pages 87 – 96 July 2007.

[28] William G. J. Halfond, Alessandro Orso. Combining Static Analysis & Runtime Monitoring to Counter SQL-Injection Attacks. *SIGSOFT Software Engineering Notes Volume 30 Issue 4.* July 2005

[29] R. Ezumalai, G. Aghila. Combinatorial Approach for Preventing SQL Injection Attacks. *IEEE International Advance Computing Conference*. Pages 1212 – 1217, March 2009.

[30] A. Dasgupta, V. Narasayya,M. Syamala. A Static Analysis Framework for Database Applications. *IEEE 25th International Conference on Data Engineering*. Pages 1403 – 1414, March 2009.

[31] Book- Silberschatz, Korth, Sudarshan. Database system concepts, 4th Edition