A

MAJOR PROJECT REPORT

On

IMPROVING WEB APPLICATION TESTING

Submitted in Partial fulfillment of the requirement

For the Degree Of

MASTER OF TECHNOLOGY

(SOFTWARE ENGINEERING)

Submitted By:

PRIYANKA GUPTA

ROLL NO- 2K12/SWE/21

Under the Guidance of:

Mrs. Abhilasha Sharma

DEPARTMENT OF COMPUTER ENGINEERING

DELHI TECHNOLOGICAL UNIVERSITY

2012-2014

CERTIFICATE

DELHI TECHNOLOGICAL UNIVERSITY

(Govt. of National Capital Territory of Delhi)

BAWANA ROAD, DELHI-110042

Date: ____________________

This is to certify that the thesis entitled ***Improving Web Application Testing*** submitted by *Priyanka Gupta (Roll Number: 2K12/SWE/21*), in partial fulfilment of the requirements for the award of degree of Master of Technology in Software Engineering, is a work carried out by her under my guidance.

____

Mrs. Abhilasha Sharma

Assistant Professor

Department of Computer Engineering

Delhi Technological University

Delhi

Signature……………………………….

ACKNOWLEDGEMENT

I take this opportunity to express my deepest gratitude and appreciation to all those who have helped me directly or indirectly towards the successful completion of this thesis.

Foremost, I would like to express my sincere gratitude to my guide ***Mrs. Abhilasha Sharma***, *Assistant Professor, Department of Computer Engineering, Delhi Technological University, Delhi* whose benevolent guidance, constant support, encouragement and valuable suggestions throughout the course of my work helped me successfully complete this thesis. Without her continuous support and interest, this thesis would not have been the same as presented here.

Besides my guide, I would like to thank the entire teaching and non-teaching staff in the Department of Computer Engineering, DTU for all their help during my course of work.

Priyanka Gupta
Master of Technology
(Software Engineering)
College Roll No. 2K12/SWE/21
Department of Computer Engineering
Delhi Technological University
Bawana Road, Delhi-110042

Signature………………………………

ABSTRACT

Web and web application are part and parcel of current use of internet. Web applications are used in simple applications like photo album, discussion forum to on-line store, auctions and internet banking. Many corporate processes rely on a web application used in intranet or internet environment. Web sites are dynamic, static, and most of the time a combination of both. Web sites need protection in their database to assure security. Online data theft has recently become a very serious issue, and recent cases have been widely publicized over concerns for the confidentiality of personally identifiable information. Web applications vulnerabilities allow attackers to perform malicious actions that range from gaining unauthorized account access to obtaining sensitive data. As early as 2002, the CSI & FBI survey reported that more than 50% of online databases experience a security breach each year. As a matter of fact, Injection Flaws – and particularly SQL Injections (SQLI) – appear among the OWASP's Top Ten most critical web applications vulnerabilities list [35].

The primary focus of our research was to secure the web applications and improve the way in which the web applications are tested. So that every loop hole in an application and the attacks that are injected on the web application can be detected. Therefore, to secure the web applications we have developed reliable black box vulnerability scanner for detecting SQLI vulnerabilities which is called as SQLIVS (SQL injection vulnerability scanner). The black box approach is based on simulation of SQLI attacks against web applications. It analyzes the value submitted by users through HTML forms, URL parameters whether clean or normal parameters and look for possible attack patterns. SQLIVS proposes a simple and effective method to accurately detect and prevent SQLIV by using SQL query parameters. To further improve the web application testing process I have introduced a new paradigm called as QUIT paradigm that focuses on the key features of regression testing both in software and web applications. This paradigm also tells about the dissimilarities that exist between the regression testing criteria for software and web application. The proposed technique for SQLIVS showed promising results as compared to other techniques. Two new essential features have been added in this technique, one of which handles the phenomenon of login form disappearance and the other feature provides an expandable payload which gives the opportunity to the user to add new attack patterns to SQLIVS database. This way SQLIVS can be easily extended to support different and new SQLI attacks.

List of Figure(s)

List of Table(s)

Table of Contents

CHAPTER 1

INTRODUCTION

CHAPTER 1
INTRODUCTION

1.1 OVERVIEW

In today's world, our regular life is totally dependent upon Internet. Web and web application are part and parcel of current use of internet. Web applications are used in simple applications like photo album, discussion forum to on-line store, auctions and internet banking. Many corporate processes rely on a web application used in intranet or internet environment.

Web applications are getting more and more important role in ordinary life, business, industry, government etc. As the importance of the application grows the importance to protect them from an unauthorized usage, assure data integrity and many other security aspects also increases. With the speedy involvement of the Internet in our everyday life, it is very important to make it secure, with the tremendous growth of internet and network technologies there is an increase in the number of attacks and threats.

Why There Is A Need To Improve Web Application Testing?

The use of the internet for accomplishing important tasks, such as transferring a balance from a bank account, always comes with a security risk. Today's web applications strive to keep their users data confidential and after years of doing secure business online, these companies have become experts in information security. The database systems behind these secure web applications store non-critical data along with sensitive information, in a way that allows the information owners quick access while blocking break-in attempts from unauthorized users.

Personally identifiable information (PII) can be found in bank accounts, retirement or investment accounts, and credit card accounts. The volume of this information is often great as a result of institutions having so many customers and users. Therefore, the ideal way to store and retrieve all of this information is in a database. To satisfy people's desire to access information from this database from anywhere at 6 anytime, the natural network of choice is the Internet. The feature of having such convenient access is precisely how we run into security issues. The Internet was first created without security in mind, because it was not an issue at the time. It was a simple, open way of communicating and sharing ideas in the academic setting. The modern version of the Internet, however, is accessible by anybody including those with dubious moral values. There needs to be protection of the PII in these databases. People want their names, addresses, phone numbers, credit card numbers, and social security numbers private and protected. Now, more than ever, we need to strengthen the security of the systems which use these databases and make them secure.

So therefore, we need to improve the way we test our web applications, so that every loop hole and every weak area in a web application can be detected, prevented and secured in an efficient way from the attacks and vulnerabilities that are being created from recent past and are still progressing.

1.2 WEB APPLICATION ARCHITECTURE

In order to understand the attacks that are injected on the web applications, there will be a brief explanation on the architecture and processes of web applications.

Although web application can be classified as programs running on a web browser, web applications generally have a Tree-tier construction as shown in Figure 1.1[55].

- **Presentation Tier:** receives the user's input data and shows the result of the processed data to the user. It can be thought of as the Graphic User Interface (GUI). Flash, HTML, JavaScript, etc. are all part of the presentation tier which directly interact with the user.
- **CGI Tier:** also known as the Server Script Process is located in between the presentation tier and database tier. The data inputted by the user is processed and stored into the database. The database sends back the stored data to the CGI tier which is finally sent to the presentation tier for viewing. Therefore, the data processing within the web application is done at the CGI Tier. It can be programmed in various server script languages such as JSP, PHP, ASP, etc.
- **Database Tier:** stores and manages all of the processed user's input data. All sensitive data of web applications are stored and managed within the database. The database tier is responsible for the access of authenticated users and the authorisation given to every user.

1.3 A.A.A ARCHITECTURE

To protect the web applications from the attacks we need to understand the importance of the key features that are needed to remain intact with the web application for its security.

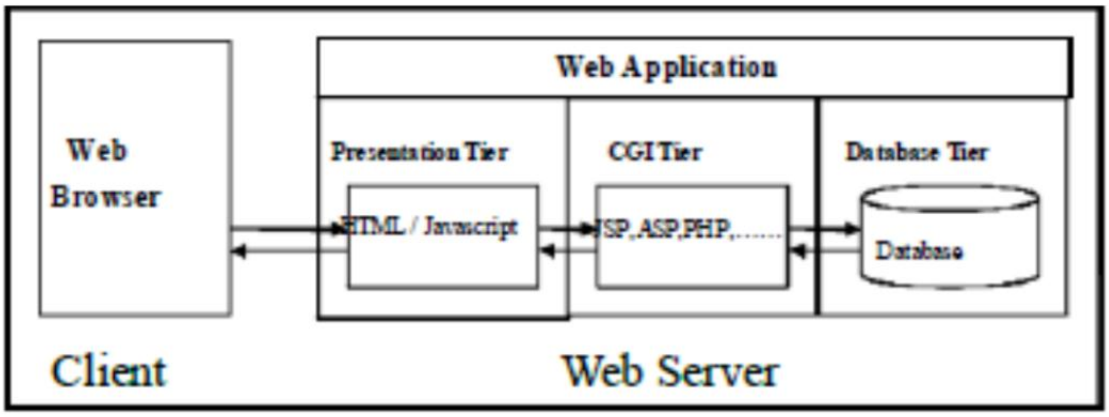


Figure 1.1: Web Application Architecture [55]

This architecture tells us about the key components and importance of them with respect to web applications security. This architecture and components specification of software and hardware system architecture 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 [60]. These are the three basic issues that are encounter 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 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, originator, or 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.

1.4 VULNERABILITIES, RISKS AND THREATS

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

- **Risk:** combination of the likelihood of an event ant its impact [31].
- **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 [31].
- **Vulnerability:** if web application security is applied to a weakness in a system which allows an attacker to violate the integrity of that system. The weakness of an asser or group of assets can be exploited by one or more threats. Vulnerabilities may result from different reasons such as weak passwords, software bugs, virus, other malware, script code injection or a SQL injection [31].

1.5ATTACKS ON WEB APPLICATION

Web applications are extremely popular today. Nearly all information systems and business applications (e-commerce, banking, transportation, web mail, blogs, etc) are now built as web-based database applications. They are so exposed to attacks that any existing security vulnerability will most probably be uncovered and exploited, which may have a highly negative impact on users. Web applications contain a mix of traditional flaws (e.g., ineffective authentication and authorization mechanisms) and web-specific vulnerabilities (e.g., using user-provided inputs in SQL queries without proper sanitization).

Traditional security mechanisms like network firewalls, intrusion detection systems (IDS), and use of encryption can protect the network but cannot mitigate attacks targeting web applications, even assuming that key infrastructure components such as web server and database management systems (DBMS) are fully secure. Hence, hackers are moving their focus from network to web applications where poor programming code represents a major risk. This can be confirmed by numerous vulnerability reports.

In 2007 the Open Web Application Security Project released its ten most critical web application security vulnerabilities based on data provided by [49]. This report ranked XSS as the most critical vulnerability, followed by Injection Flaws, particularly SQL injection.

Here, we will briefly describe some of the most common vulnerabilities in web applications (reader can refer to the OWASP Top 10 List, which tracks the most critical vulnerabilities in web applications):

- **Cross-Site Scripting (XSS):** XSS vulnerabilities allow an attacker to execute malicious JavaScript code as if the application sent that code to the user. This is the first most serious vulnerability of the OWASP Top 10 List.
- **SQL Injection:** SQL injection vulnerabilities allow one to manipulate, create or execute arbitrary SQL queries. This is the second most serious vulnerability on the OWASP Top 10 List.
- **Code Injection:** Code injection vulnerabilities allow an attacker to execute arbitrary commands or execute arbitrary code. This is the third most serious vulnerability on the OWASP Top 10 List.
- **Broken Access Controls:** A web application with broken access controls fails to properly define or enforce access to some of its resources. This is the tenth most serious vulnerability on the OWASP Top 10 List.

As shown in figure 1.2 SQL-injections was ranked number one of the most dangerous software errors 2011 (CWE and SANS April-13), and it is still in the top of critical security risks 2013 (Top 10 2013, May-13). SQL-injections are not only one of the most critical attacks; it is also one of the most popular data extraction techniques (X-force trend and risk report, May-13). In January to June 2011 SQL-injections accounted for 68 percent of the total number of web-application attacks (see Figure 1.2) (Cyber security report, May-13). So there is a need to detect and prevent web applications from the SQL injection attacks.

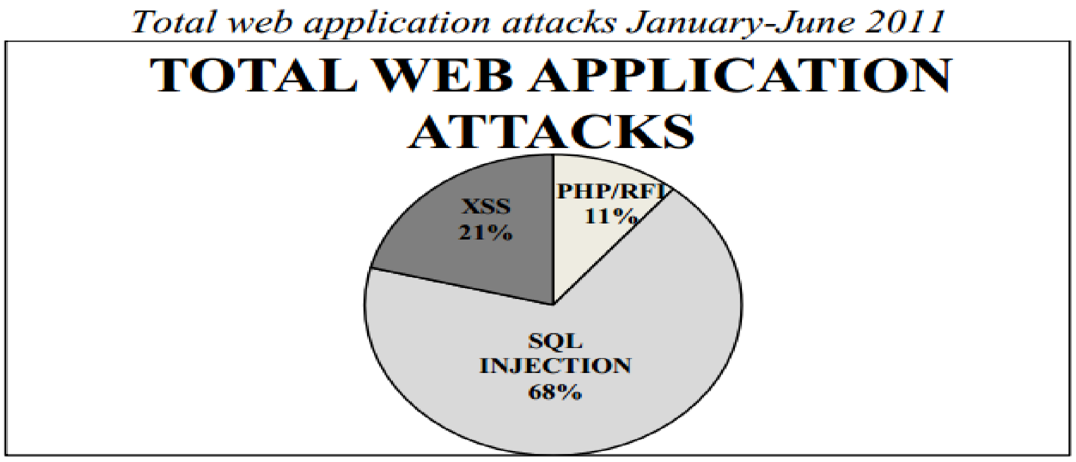


Figure 1.2: Total Web Application Attacks [49]

1.6 NEED FOR REGRESSION TESTING AFTER DETECTION OF SQLI ATTACKS

Web applications are getting more and more important role in ordinary life, business, industry, government etc. As the importance of the application grows the importance to protect them from an unauthorized usage, assure data integrity and many other security aspects also increases.

A web application is prone to attacks because of poor programming practices. Hence hackers are moving their focus from network to these vulnerable web applications. Out of all the attack types, SQL injection attack is the most serious and critical attack type as discussed in section 1.5. So dealing with SQL injection attacks is a main concern now. Therefore new techniques are being developed to detect these SQL injection attacks, these techniques also tries to secure the web applications and prevent the SQL injection attacks to occur.

Whenever these attacks are detected, they are fixed so that the web application keeps working, and provides all the required services correctly. When errors are being fixed the web applications do undergo modification for proper working. After modifications are done in a web application there is a need for regression testing to ensure that modifications do not lead to adverse effects.

Since regression testing is a testing activity that is performed to ensure that changes do not harm the existing behaviour of the web application. Hence whenever SQL injection attacks are detected in a web application, these attacks needs to be fixed for proper running of web application thereby calling for regression testing so that these fixings do not harm the existing behaviour of the web application.

1.7 ORGANISATION OF THESIS

This thesis consists of 6 chapters. The rest of the thesis is organised as:

- **Chapter 2** provides description about SQL injection.
- **Chapter 3** provides description about the QUIT paradigm that has been introduced for regression testing of software and web applications.
- **Chapter 4** provides review of literature in order to create an adequate framework for conducting this research work.
- **Chapter 5** deals with the research methodology followed to achieve the goals of the work.
- **Chapter 6** provides the results of the thesis work, & discussion about the result.
- Conclusion and Future Work.

CHAPTER 2

SQL INJECTION

CHAPTER 2
SQL INJECTION

2.1 INTRODUCTION

In today's world, application-level vulnerabilities, which are believed to account for 70% to 90% of overall flaws, are now the main focus of attackers and researchers. Online applications (websites and services) are especially at risk due to their universal exposure and their extensive use of the firewall-friendly HTTP protocol. Moreover, database security is too often overlooked in favour of web and application server security, resulting in backend databases being a major target for attackers which are able to use them as easy entry points to organizations' networks.

- **SQL Injection:** A common break-in strategy is to try to access sensitive information from a database by first generating a query that will cause the database parser to malfunction, followed by applying this query to the desired database. Such an approach to gaining access to private information is called *SQL injection*.

Since databases are everywhere and are accessible from the internet, dealing with SQL injection has become more important than ever. Although current database systems have little vulnerability, the Computer Security Institute discovered that every year about 50% of databases experience at least one security breach. The loss of revenue associated with such breaches has been estimated to be over four million dollars.

To get a better understanding of SQL injection, we need to have a good understanding of the kinds of communications that take place during a typical session between a user and a web application. Figure 1.1 shows the typical communication exchange between all the components in a typical web application system.

A web application, based on the model as shown in figure 1.1, takes text as input from users to retrieve information from a database. Some web applications assume that the input is legitimate and use it to build SQL queries to access a database. Since these web applications do not validate user queries before submitting them to retrieve data, they become more susceptible to SQL injection attacks. For example, attackers, posing as normal users, use maliciously crafted input text containing SQL instructions to produce SQL queries on the web application end. Once processed by the web application, the accepted malicious query may break the security policies of the underlying database architecture because the result of the query might cause the database parser to malfunction and release sensitive information.

2.2 BACKGROUND ON SQL INJECTION ATTACKS

SQL injection vulnerabilities have been described as one of the most serious threats for Web applications [9]. Web applications that are vulnerable to SQL injection may allow an attacker to gain complete access to their underlying databases. Because these databases often contain sensitive consumer or user information, the resulting security violations can include identity theft, loss of confidential information, and fraud. In some cases, attackers can even use an SQL injection vulnerability to take control of and corrupt the system that hosts the Web application. Web applications that are vulnerable to SQL Injection Attacks (SQLIAs) are widespread—a study by Gartner Group on over 300 Internet Web sites has shown that most of them could be vulnerable to SQLIAs. In fact, SQLIAs have successfully targeted high-profile victims such as Travelocity, FTD.com, and Guess Inc.

Intuitively, an **SQL Injection Attack (SQLIA)** occurs when an attacker changes the intended effect of an SQL query by inserting new SQL keywords or operators into the query. This informal definition is intended to include all of the variants of SQLIAs reported in literature and presented in this paper. Interested readers can refer to [66] for a more formal definition of SQLIAs. Here, we define two important characteristics of SQLIAs that we use for describing attacks: injection mechanism and attack intent.

2.2.1 Injection Mechanism

Malicious SQL statements can be introduced into a vulnerable application using many different input mechanisms. In this section, we explain the most common mechanisms.

- ***Injection through user input*:** In this case, attackers inject SQL commands by providing suitably crafted user input. A Web application can read user input in several ways based on the environment in which the application is deployed. In most SQLIAs that target Web applications, user input typically comes from form submissions that are sent to the Web application via HTTP *GET* or *POST* requests [63]. Web applications are generally able to access the user input contained in these requests as they would access any other variable in the environment.

- ***Injection through cookies*:** Cookies are files that contain state information generated by Web applications and stored on the client machine. When a client returns to a Web application, cookies can be used to restore the client’s state information. Since the client has control over the storage of the cookie, a malicious client could tamper with the cookie’s contents. If a Web application uses the cookie’s contents to build SQL queries, an attacker could easily submit an attack by embedding it in the cookie [63].
- ***Injection through server variables*:** Server variables are a collection of variables that contain HTTP, network headers, and environmental variables. Web applications use these server variables in a variety of ways, such as logging usage statistics and identifying browsing trends. If these variables are logged to a database without sanitization, this could create SQL injection vulnerability [63]. Because attackers can forge the values that are placed in HTTP and network headers, they can exploit this vulnerability by placing an SQLIA directly into the headers. When the query to log the server variable is issued to the database, the attack in the forged header is then triggered.
- ***Second-order injection*:** In second-order injections, attackers seed malicious inputs into a system or database to indirectly trigger an SQLIA when that input is used at a later time. The objective of this kind of attack differs significantly from a regular (i.e., first order) injection attack. Second-order injections are not trying to cause the attack to occur when the malicious input initially reaches the database. Instead, attackers rely on knowledge of where the input will be subsequently used and craft their attack so that it occurs during that usage. To clarify, we present a classic example of a second order injection attack [5]. In the example, a user registers on a website using a seeded user name, such as “admin’ --”. The application properly escapes the single quote in the input before storing it in the database, preventing its potentially malicious effect. At this point, the user modifies his or her password, an operation that typically involves
- Checking that the user knows the current password and
- Changing the password if the check is successful. To do this, the Web application might construct an SQL command as follows:

  queryString="UPDATE users SET password=’" + newPassword + "’
  WHERE userName=’" + userName + "’ AND password=’" +oldPassword + "’"

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

UPDATE users SET password=’newpwd’

WHERE userName= ’admin’--’ AND password=’oldpwd’

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

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

2.2.2 Attack Intent

Attacks can also be characterized based on the goal, or *intent* of the attacker. Therefore, each of the attack type definitions that we provide in Section 2.3 includes a list of one or more of the attack intents defined in this section [63].

- **Identifying inject-able parameters:** The attacker wants to probe a Web application to discover which parameters and user-input fields are vulnerable to SQLIA.
- **Performing database finger-printing:** The attacker wants to discover the type and version of database that a Web application is using. Certain types of databases respond differently to different queries and attacks, and this information can be used to “fingerprint” the database. Knowing the type and version of the database used by a Web application allows an attacker to craft database specific attacks.
- **Determining database schema:** To correctly extract data from a database, the attacker often needs to know database schema information, such as table names, column names, and column data types. Attacks with this intent are created to collect or infer this kind of information.
- **Extracting data:** These types of attacks employ techniques that will extract data values from the database. Depending on the type of the Web application, this information could be sensitive and highly desirable to the attacker. Attacks with this intent are the most common type of SQLIA.

- **Adding or modifying data:** The goal of these attacks is to add or change information in a database.
- **Performing denial of service:** These attacks are performed to shut down the database of a Web application, thus denying service to other users. Attacks involving locking or dropping database tables also fall under this category.
- **Evading detection:** This category refers to certain attack techniques that are employed to avoid auditing and detection by system protection mechanisms.
- **Bypassing authentication:** The goal of these types of attacks is to allow the attacker to bypass database and application authentication mechanisms. Bypassing such mechanisms could allow the attacker to assume the rights and privileges associated with another application user.
- **Executing remote commands:** These types of attacks attempt to execute arbitrary commands on the database. These commands can be stored procedures or functions available to database users.
- **Performing privilege escalation:** These attacks take advantage of implementation errors or logical flaws in the database in order to escalate the privileges of the attacker. As opposed to bypassing authentication attacks, these attacks focus on exploiting the database user privileges.

2.3 EXAMPLE APPLICATION

Before we start talking about various types of attacks, we will show an example application that contains SQL injection vulnerability. We use this example in the next section to provide attack examples. Note that the example refers to a fairly simple vulnerability that could be prevented using a straightforward coding fix. We use this example [63] simply for illustrative purposes because it is easy to understand and general enough to illustrate many different types of attacks.

The example in Figure 2.1 implements the login functionality for an application. It is based on similar implementations of login functionality that we have found in existing Web-based applications. The code in the example uses the input parameters login, pass, and pin to dynamically build an SQL query and submit it to a database.

For example, if a user submits login, password, and pin as “doe,” “secret,” and “123,” the application dynamically builds and submits the query:

SELECT accounts FROM users WHERE

login='doe' AND pass='secret' AND pin=123

If the login, password, and pin match the corresponding entry in the database, doe's account information is returned and then displayed by function displayAccounts(). If there is no match in the database, function displayAuthFailed() displays an appropriate error message.

```
String login, password, pin, query
login = getParameter("login");
password = getParameter("pass");
pin = getParameter("pin");
Connection conn.createConnection("MyDataBase");
query = "SELECT accounts FROM users WHERE login='" +
        login + "' AND pass='" + password +
        "' AND pin=" + pin;
ResultSet result = conn.executeQuery(query);
if (result!=NULL)
    displayAccounts(result);
else
    displayAuthFailed();
```

Figure 2.1: Example of Servlet Implementation [63]

2.4 SQL INJECTION ATTACKS

Here, we present and discuss the different kinds of SQL injection attacks known to date. For each attack type, we provide a descriptive *name*, one or more *attack intents*, a *description* of the attack, an attack *example*, and a set of *references* to publications and Web sites that discuss the attack technique and its variations in greater detail. The different types of attacks are generally not performed in isolation; many of them are used in combination or sequentially, depending on the specific goals of the attacker. Note also that there are countless variations of each attack type. For space reasons, we do not present all of the possible attack variations but instead present a single representative example.

2.4.1 Tautologies:

Attack Intent - Bypassing authentication, identifying inject-able parameters, extracting data.

Description - 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 *login* input field (the input submitted for the other fields is irrelevant). The resulting query is:

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

The code injected in the conditional (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. In our example, the returned set evaluates to a non null value, which causes the application to conclude that the user authentication was successful. Therefore, the application would invoke method displayAccounts() and show all of the accounts in the set returned by the database [5, 38, 50, 51].

2.4.2 Illegal/Logically Incorrect Queries:

Attack Intent - Identifying inject-able parameters, performing database finger-printing, extracting data.

Description - This attack lets an attacker gather important information about the type and structure of the back-end database of Web 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 into input field *pin*: "convert (int,(select top 1 name from sysobjects where xtype='u'))". The resulting query is:

SELECT accounts FROM users WHERE login='' AND pass='' AND pin= convert (int,(select top 1 name from sysobjects where xtype='u'))

In the attack string, the injected select query attempts to extract the first user table (xtype='u') from the database's metadata table (assume the application is using Microsoft SQL Server, for which the metadata table is called sysobjects). The query then tries to convert this table name into an integer. Because 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 SQL Server (0x80040E07) Error converting nvarchar value 'CreditCards' to a column of data type int."*

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 user-defined table in the database: "CreditCards." 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 [5, 10, 38].

2.4.3 Union Query:

Attack Intent - Bypassing Authentication, extracting data.

Description - 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 cardNo from CreditCards where acctNo=10032 - -" into the login field, which produces the following query:

SELECT accounts FROM users WHERE login='' UNION SELECT cardNo from CreditCards WHERE acctNo=10032 -- AND pass='' AND pin=

Assuming that there is no login equal to "", the original first query returns the null set, whereas the second query returns data from the "CreditCards" table. In this case, the database would return column "cardNo" for account "10032." 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 "cardNo" is displayed along with the account information [5, 50, 51].

2.4.4 Piggy Backed Queries:

Attack Intent – Extraction of data, addition or modification of data, performing denial of service, execution of remote commands.

Description - 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 *pass* field, the application generates the query:

SELECT accounts FROM users WHERE login='doe' AND pass=''; drop table users --' AND pin=123

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 [5, 38, 50].

2.4.5 Stored Procedures:

Attack Intent - Performing privilege escalation, performing denial of service, executing remote commands.

Description - 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 [38, 6]. 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 [15].

```
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
```

Figure 2.2: Stored Procedures for Checking Credentials [63]

Example - 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 in Figure 2.2. 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 userName or password fields. This injection causes the stored procedure to generate the following query:
SELECT accounts FROM users WHERE login=’doe’ AND pass=’ ’; SHUTDOWN; -- *AND pin=*
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 [5, 17, 45, 50, 51].

2.4.6 Inference:

Attack Intent - Identifying inject-able parameters, extraction of data, determining database schema.

Description - 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 behaviour 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. Researchers have reported that with these techniques they have been able to achieve a data extraction rate of 1B/s.

Blind Injection - In this technique, the information must be inferred from the behaviour 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=’’ AND pin=0

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

Now, let us consider two scenarios. In the first scenario, we have a secure application, and the input for *login* is validated correctly. In this case, both injections would return login error messages, and the attacker would know that the *login* parameter is not vulnerable. In the second scenario, we have an insecure application and the *login* parameter is vulnerable to injection. The attacker submits the first injection and, because it always evaluates to false, the application returns a login error message. At this point however, the attacker does not know if this is because the application validated the input correctly and blocked the attack attempt or because the attack itself caused the login error. The attacker then submits the second query, which always evaluates to true. If in this case there is no login error message, then the attacker knows that the attack went through and that the *login* parameter is vulnerable to injection. The second way inference based attacks can be used is to perform data extraction. Here we illustrate how to use Timing based inference attack to extract a table name from the database. In this attack, the following is injected into the *login* parameter: ‘‘legalUser’ and ASCII(SUBSTRING((select top 1 name from sysobjects),1,1)) > *X* WAITFOR 5 --’’. This produces the following query: SELECT accounts FROM users WHERE login=’legalUser’ and ASCII(SUBSTRING((select top 1 name from sysobjects),1,1)) > *X* WAITFOR 5 -- ’ AND pass=’’ AND pin=0 .In this attack the SUBSTRING function is used to extract the first character of the first 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 [7, 32].

2.4.7 Alternate Encodings:

Attack Intent - Evading detection.

Description - 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 operators. To evade this defence, 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 defence 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 [38] 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='' AND pin=

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 [5, 38].

CHAPTER 3

QUIT PARADIGM FOR REGRESSION TESTING OF SOFTWARE AND WEB APPLICATION

CHAPTER 3
QUIT PARADIGM FOR REGRESSION TESTING OF SOFTWARE AND WEB APPLICATION

3.1 INTRODUCTION

Web applications are getting more and more important role in ordinary life, business, industry, government etc. As the importance of the application grows the importance to protect them from an unauthorized usage, assure data integrity and many other security aspects also increases. With the speedy involvement of the Internet in our everyday life, it is very important to make it secure, with the tremendous growth of internet and network technologies there is an increase in the number of attacks and threats [56].

Web applications can be composed of heterogeneous self contained web services. Such applications are usually modified to fix errors or to enhance their functionality. After modifications, regression testing is essential to ensure that modifications do not lead to adverse effects.

Regression testing is a testing activity that is performed to ensure that changes do not harm the existing behaviour of the software or the web application. Test suites tend to grow in size as software / web application evolves, often making it too costly to execute entire test suites. A number of different approaches have been developed to maximise the value of the accrued test suite. The regression testing process is altogether different technique in software and web application. The purpose, motive and the issues that occur during the regression testing of software and web application are different [22].

Why There Is A Need To Do Regression Testing?

Originally, web sites were constructed from a collection of web pages containing text documents and intercom interconnected via hyper links. Recently, the dramatic evolution of web technology has led to web applications that can be built by integrating different components from variety of sources, residing on distributed hardware platforms, and running concurrently on heterogeneous networks. The construction of systems from different types of software components faces various challenges such as maintaining performance, reliability and availability of those systems. But the validation of such web applications remains a major challenge.

A Web application might invoke multiple web services located on different servers with no design, source code or interface available. This forces designer to use black-box notions to select relevant web services from the pool of services found on the internet. With the increasing number of periodic publishing of web services, web applications need more often to be updated so that they select the most optimal and reliable service. Moreover, web systems are usually exposed to structural changes and modifications. These changes require us to retest the web application in order to provide confidence that the system functionality and unmodified parts have not been adversely affected by the modifications [21].

Regression testing refers to selecting tests from the test suite generated during the initial development phase and to adding new tests to address enhancements and additions. Regression testing (also referred to as program revalidation) is carried out to ensure that no new errors (called regression errors) have been introduced into previously validated code (i.e., the unmodified parts of the program). Although regression testing is usually associated with system testing after a code change, regression testing can be carried out at unit, integration or system testing levels [57].

Regression testing is acknowledged to be an expensive activity. It consumes large amounts of time as well as effort, and often accounts for almost half of the software maintenance costs. The extents to which time and effort are being spent on regression testing are exemplified by a study that reports that it took 1000 machine-hours to execute approximately 30,000 functional test cases for a software product. It is also important to note that hundreds of man-hours are spent by test engineers to oversee the regression testing process; that is to set up test runs, monitor test execution, analyze results, and maintain testing resources, etc [47].

3.2 REGRESSION TESTING PROCESS

Regression testing is an important and expensive activity that is undertaken every time a program is modified to ensure that the modifications do not introduce new bugs into previously validated code.

Regression test selection (RTS) techniques select a subset of valid test cases from an initial test suite (T) to test that the affected but unmodified parts of a program continue to work correctly. Use of an effective regression test selection technique can help to reduce the testing costs in environments in which a program undergoes frequent modifications. Regression test selection essentially consists of two major activities [57]:

- **Identification of the affected parts** - This involves identification of the unmodified parts of the program that are affected by the modifications.
- **Test case selection** - This involves identification of a subset of test cases from the initial test suite T which can effectively test the unmodified parts of the program. The aim is to be able to select the subset of test cases from the initial test suite that has the potential to detect errors induced on account of the changes.

3.2.1 Regression Testing of Software

As shown in figure 3.1 Rothermel and Harrold [20] have formally defined the regression test selection problem as follows:

- Let P be an application program and P′ be a modified version of P.
- Let T be the test suite developed initially for testing P.
- An RTS technique aims to select a subset of test cases T′ ⊆ T to be executed on P′, such that every error detected when P′ is executed with T is also detected when P′ is executed with T′.

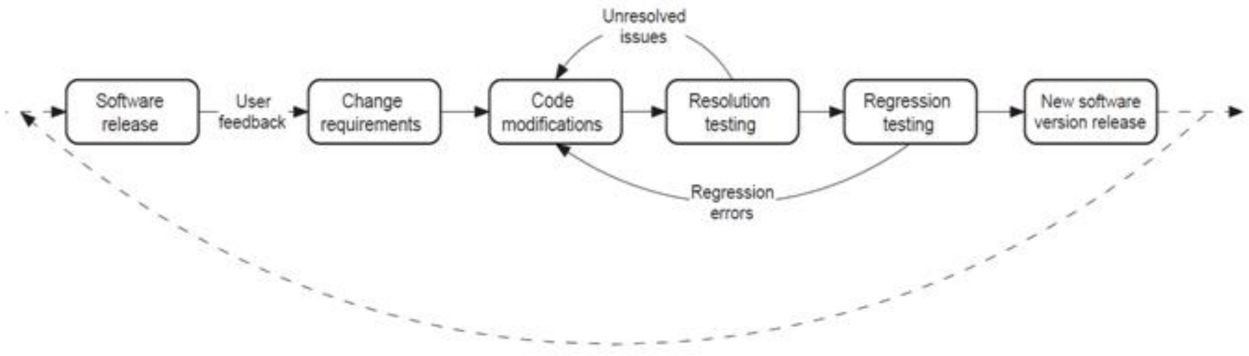


Figure 3.1: Activities That Takes Place during Software Maintenance and Regression Testing [57]

3.2.2 Regression Testing of Web Application

Regression Testing is a process which tests a system once again to ensure that it still functions as expected by specification [1]. The reason for this renewed testing activity is usually performed when changes are done to a system W producing a modified version W′. Regression testing is a way to test the modified W′ using a test set T used previously to test the original system W.

The selection of suitable test cases from T can be made in different ways and a number of regression-testing methods have been proposed. These methods are based on different objectives and techniques, such as: procedure and class firewalls [26,46]; semantic differencing [11]; textual differencing [19]; slicing-based data-flow technique [21,47]; test case reduction [42,43]; and safe algorithm based on program's control graph [22].

Typical regression testing procedures follow five main steps:

1) Identify the modifications made to W;
2) Test selection step: using the results in step 1, select T′ µ T, a set of tests that may reveal modification-related errors in W′;
3) If necessary, create new tests for W′ and append to T′. These may include new functional tests required by changes in specifications, and/or new structural tests required by applicable coverage criteria;
4) Run T′ on W′, to provide confidence about W′'s correctness with respect to T′; and finally,
5) Create T″, a new test set/history for W′.

Further, test histories should be maintained. The system's test history identifies, for each test, its input, and output and execution history. An execution history consists of a list of components and their internal states exercised by the test. Moreover, [22] emphasized the creation of a test history for the test suite so that a regression test can be applied.

3.3 QUIT PARADIGM FOR REGRESSION TESTING

While focusing on the regression testing process in software and web application certain key features have been identified and a new paradigm has been introduced called as the QUIT Paradigm that focuses on the Quality Aspect, User Expectation, Issues and techniques used in the regression testing of software and web application. The QUIT Paradigm features are presented in a tabular format listed in table 3.1 that describes all the key features of regression testing in software and web application. This Paradigm also helps us to understand and know that how the regression testing varies in software and web application in terms of requirement by every feature.

Serial Number	**Criteria**	**Regression Testing Of Web Application**	**Regression Testing Of Software**
1.	**Quality Aspect**	➢ Structural quality ➢ Content ➢ Timeliness ➢ Accuracy & Consistency ➢ Response Time & Latency ➢ Performance ➢ Security ➢ Underlying Complex Architecture ➢ Usability	➢ Conformance Quality ➢ Gap Quality ➢ Value Quality
2.	**User Expectation**	➢ Testing the web application for ease of navigation. ➢ Content checking ➢ Search and help menu should be easy to understand ➢ Fast navigation ➢ Proper links connectivity.	➢ Performing variety of tasks with the software to record the success of the user in performing each task ➢ How fast the user goes ➢ What mistakes they make. ➢ Where they get confused ➢ What solution paths to follow ➢ How many learning trials are involved
3.	**Issues Occurring**	➢ Rapid change in data ➢ Complex multitier architecture ➢ Multiple intermingled language ➢ Hard to apply traditional testing techniques as	➢ Difficult to maintain the reliability of the software ➢ Difficult to minimize the cost of software ➢ regression testing ➢ Time consuming as software are huge in size.

		most of the web services are developed by third party. ➢ Difficult fault localization ➢ Complex test cases so involve lot to human interaction. ➢ Concurrency issue ➢ May or may not produce deterministic output. ➢ Cost issues	➢ Maintaining quality is difficult ➢ Getting meaningful result is slow ➢ Its repetitive, dull and tedious job. ➢ Best test cases are saved for the last. ➢ Research on this is limited, so limited knowledge.
4.	**Techniques Used**	➢ Testing based on user session data ➢ Based on slicing ➢ Based on system models ➢ Based on control flow techniques ➢ Based on testability measures ➢ Based on applying agents ➢ Based on prioritizing user session data ➢ Based on leveraging user session data ➢ Using mm path approach	➢ Testing based on minimization techniques focusing on the coverage criteria ➢ Testing based on operation abstraction ➢ Based on delta debugging technique ➢ Based on logic criteria ➢ Based on regression test selection ➢ Based on prioritization techniques

Table 3.1: QUIT Paradigm for Regression Testing of Software and Web Application

3.4 QUIT PARADIGM FEATURES

As already known regression testing is an important and expensive activity that is undertaken every time a program is modified to ensure that the modifications do not introduce new bugs into previously validated code.

A number of different approaches have been developed to maximise the value of the accrued test suite. The regression testing process is altogether different technique in software and web application. The purpose, motive and the issues that occur during the regression testing of software and web application are different.

Through the QUIT paradigm we can see that there are four key features while talking about the regression testing process, they are Quality Aspect, User Expectation, Issues and Techniques. As in Table1, we can see that all the four criteria are in some way different in regression testing of software and web application respectively. The quality criterion is there in both but what is demanded from that quality is different. What the user expects from it, what are the issues faced during the process and the techniques used in both are quite different.

So, using this paradigm we are getting so much of knowledge regarding the regression testing of software and web application. It also tells us about the similarities and the contrasts that are there in both, the regression testing software and web application.

3.4.1 Quality Aspect

Quality in business, engineering and manufacturing has a pragmatic interpretation as the non-inferiority or superiority of something; it is also defined as fitness for purpose. Quality is a perceptual, conditional, and somewhat subjective attribute and may be understood differently by different people.

Consumers may focus on the specification quality of a product/service, or how it compares to competitors in the marketplace. Producers might measure the conformance quality, or degree to which the product/service was produced correctly. Support personnel may measure quality in the degree that a product is reliable, maintainable, or sustainable. Simply put, a quality item (an item that has quality) has the ability to perform satisfactorily in service and is suitable for its intended purpose.

Quality aspect has a different value both in regression testing of web application and software. They vary in there wants when applied to both.

A. Quality Aspect For Regression Testing of Web Application:

- **Structural quality:** The website should be well connected for easy navigation and all external and internal links should be working [16].
- **Content:** HTML code should be valid and content matches what is expected [16].
- **Timeliness:** a web application changes rapidly. The change has to be identified and highlighted and tested [16].
- **Accuracy and consistency:** content should be consistent over time and data should be accurate [16].
- **Response Time and latency:** Server response times to user's requests should be within the accepted limits for that particular application [16].
- **Performance:** performance should be acceptable under different usage loads [16].
- **Security:** with the expanding amount of applications such as e-commerce sites and e-banking, security has become a major issue [16].

B. Quality Aspect For Regression Testing of Web Application

- **Conformance quality:** performance measurement against established standard [20].
- **Gap quality:** capturing the content to which the services meet customer expectations [20].
- **Value quality:** ROI (return on investment) and benefits reaped by customers based on service pricing [20].

3.4.2 User Expectation

User expectation refers to the consistency that users expect from the products. The user expectation means differently in both regression testing of web application and software. What the user wants from the product when the regression testing process is completed on the web application and software are dissimilar in one way or the other. All these points have to be kept in mind while performing regression testing on web application and software.

A. User Expectation From Regression Testing of Web Application

- **Testing the web application for ease of navigation:** when a user works on the web application, it should be easy to navigate
- **Content checking:** when checking the contents on the web application, a user should be able to understand it properly, easily and it should be user friendly.

- **Search and help menu:** these search option and help menu should be easy to understand for a user and should be easy to use.
- **Fast navigation:** whenever a user is using a web application, it should have a fast access and even the navigation should be fast.
- **Proper links connectivity:** the links on a web application should be proper, i.e. every link should connect to another trusted web pages. Links should not be broken.

B. User Expectation From Regression Testing of Software

- **Performing variety of tasks:** with the software to record the success of the user in performing each task, so that it can be known that how accurate and easy it was to use the software.
- **How fast the user goes:** while using software the user expects it to run at a fast speed and do all the required work efficiently.
- **What mistakes they make:** while using software all the mistakes that a user can make or actually makes should be noted properly, so that while performing regression testing all this can be rectified.
- **Where they get confused:** all the points and places in software should be marked and noted wherever a user it finding difficulty to use the software.
- **What solution paths to follow:** when using software if by any how a user gets stuck, a solution path should be provided so that the user can come out of it.
- **How many learning trials are involved:** how many learning trials should be provided to a user for understanding software

3.4.3 Issues Occurring

While performing regression testing of web application and software there are many issues that occur during the process. These issues occur mostly due to the rapid growth of internet and the increasing complexity of it. Even the issues that occur during the regression testing process of web application and software are quite different from each other.

A. Issues Occurring In Regression Testing of Web Application

- **Rapid change in data:** as lots and lots of information is added on the internet thereby creating problem to test this huge amount of data.

- **Complex multitier architecture:** since the web applications are developed using different architecture and then merging all the techniques together to make a web application, creates lot of problem while regression testing these.
- **Multiple intermingled languages:** every part of a web application is written in one or the other language, making it difficult to regression test the web application.
- **Hard to apply traditional testing techniques as most of the web services are developed by third party:** Web services often reside in remote locations and are developed by a third-party, making it hard to apply the traditional white-box regression testing techniques that require analysis of source code
- **Difficult fault localization:** Modifications can happen across multiple services, which can make fault localisation difficult.
- **Complex test cases so involve lot to human interaction:** High interactivity in web applications may result in complex test cases that may involve human interaction
- **Concurrency issue:** distributed systems often contain concurrency issues.
- **May or may not produce deterministic output:** as there is concurrency issue in regression testing a web application which finally creates a problem with producing a deterministic output.
- **Cost issues:** as to regression test these complex web application is problematic and long process thereby requiring lot of money to regression test them.

B. Issues Occurring In Regression Testing of Software

- **Difficult to maintain the reliability of the software:** Since now a day we have huge software that has long coding so even after regression testing software it’s difficult to maintain its reliability.
- **Difficult to minimize the cost of software regression testing:** as regression testing is an ongoing process. It starts at the unit level and continues even in the maintenance phase of software thereby requiring a lot of money to regression test the data. 50% of the actual software cost is spent on the regression testing of software.
- **Time consuming as software are huge in size:** since software is big in size having huge coding in it so regression testing it is quite time consuming.
- **Maintaining quality is difficult:** as software is big in size so regression testing them is difficult thereby maintaining the quality of the software is also difficult.
- **Getting meaningful result is slow:** since software is huge in size so while regression testing this big software takes time thereby producing the required results slowly.

- **Its repetitive, dull and tedious job:** performing regression testing on software is a repetitive process has to be done whenever changes are made to software thereby making it a dull and tedious job.
- **Best test cases are saved for the last:** usually while regression testing the software the test cases are arranged in certain orders ranging from easily produced test cases to test case that are difficult to produce. Hence getting the most difficult test cases will be given late.
- **Research on this is limited, so limited knowledge:** as research on regression testing is limited, and new approaches are still in developing process so thereby making testing process restricted in some way or the other.

3.4.4 Techniques Used

A lot of techniques have been developed for regression testing of both web application and software. These techniques are used to enhance and improve the process of regression testing. But even the techniques used are dissimilar for regression testing of both web application and software.

A. Techniques Used For Regression Testing of Web Application:

- **Testing based on user session data**

  Sebastian Elbaum et. al. (2003) [52] This paper explores the notion that user session data gathered as users operate web applications can be successfully employed in the testing of those applications, particularly as those applications evolve and experience different usage profiles. We report results of an experiment comparing new and existing test generation techniques for web applications, assessing both the adequacy of the generated tests and their ability to detect faults on a point-of-sale web application. Our results show that user session data can produce test suites as effective overall as those produced by existing white-box techniques, but at less expense. Moreover, the classes of faults detected differ somewhat across approaches, suggesting that the techniques may be complimentary.
- **Testing based on slicing**

  Lei Xu et. al. (2003) [36] In order to carry through the regression testing quickly and effectively, this paper make the simplification with the method of slicing. Firstly, they analyze the possible changes in the Web applications and the influences produced by these changes, discussing in the direct-dependent and indirect-dependent way; next, we give the regression testing method based on slicing emphasized on the indirect-dependent among data, i.e., obtaining the dependent set of changed variables by Forward and Backward Search Method and generating the testing suits thereby greatly improving the quality and efficiency of the regression testing since the testing suits are tightly related to the changes and there are no lack or redundancies in the testing suits.

➢ **Testing based on leveraging user session data**

Sebastian Elbaum et. al. (2005) [53] Presented several techniques for using user session data gathered as users operate Web applications to help test those applications from a functional standpoint. They report results of an experiment comparing these new techniques to existing white-box techniques for creating test cases for Web applications, assessing both the adequacy of the generated test cases and their ability to detect faults on a point-of-sale Web application. Our results show that user session data can be used to produce test suites more effective overall than those produced by the white-box techniques considered; however, the faults detected by the two classes of techniques differ, suggesting that the techniques are complementary.

➢ **Testing based on system models**

Abbas Tarhini et. al. (2006) [1] The technique defines web services as self-contained component-based applications residing at separate locations and communicating using XML-encoded messages using SOAP interfaces. The communication using message exchange may also be time constrained. The services provided by a web service are shared using WSDL specifications. The authors have modelled a web application in two hierarchical levels to avoid state explosion. In the first level, the interaction of the components with the main application is modelled using a Timed Labelled Transition System (TLTS). Each node in a TLTS represents a component, and an edge joining two nodes represents a transition between the two components. The internal behaviour of each component is modelled in the second level. Each node in the second-level TLTS represents a state of the component that is being modelled. The authors have proposed an RTS technique which selects all relevant test cases that test the side-effects of adding, removing or fixing an operation or a timing constraint in an existing component based on an analysis of the constructed two-level TLTS models. The technique in is safe because it selects every test case that produces a different behavior in the modified system. However, this technique cannot strictly be considered as a pure RTS technique because the analysis involves generation of test cases as an intermediate step.

➢ **Testing based on control flow techniques**

Michael Ruth et. al. (2007) [39] Proposed safe RTS techniques for web services based on analysis of control flow models. It is a gray-box technique because it does not require the source code of the web services. Instead, their approach assumes that the component web service providers would provide the following information as metadata along with a service release: WSDL specification, a set of test cases, CFGs for the web services, and test coverage information. Their technique requires that each procedure in a web service is modelled as a CFG at the service developer side. Their technique also assumes that the method calls to other services are decided statically. The CFGs for all the individual procedures are then combined to form a global CFG. When a web service is modified, then a global CFG is also constructed for the modified web service. Each node in a CFG stores a hash code of the corresponding statement. These techniques are safe, and comparatively more precise when compared to others.

- **Testing based on applying agents**

  Lei Xu et. al. (2007) [37] This paper first analyzed the necessity and feasibility of the automatic and intelligent testing for Web applications; Then, they discuss several scenes of applying Agent into Web application testing, such as using Agent to obtain users' visiting actions, carry out performance testing, regression testing and usability evolvement; next, we adopt Agent to execute the testing, including the testing process and the detailed actions, so as to monitor, manage and handler exceptions during the whole testing execution. Thus, in this way, the web application testing can be completed more automatically and intelligently.

- **Testing based on prioritizing user session data**

  Sreedevi Sampath et. al. (2008) [56] Proposed several new test suite prioritization strategies for web applications and examine whether these strategies can improve the rate of fault detection for three web applications and their pre-existing test suites. They prioritize test suites by test lengths, frequency of appearance of request sequences, and systematic coverage of parameter-values and their interactions. Experimental results show that the proposed prioritization criteria often improve the rate of fault detection of the test suites when compared to random ordering of test cases. In general, the best prioritization metrics either (1) consider frequency of appearance of sequences of requests or (2) systematically cover combinations of parameter-values as early as possible.

- **Testing based on mm path approach**

  Jingxian Gu et. al. (2008) [30] Web application contains many components, which makes it become component-based Web application. This paper focuses on this kind of Web applications and constructs three dependency graphs based on structure relations and message call relations. Then we improve the path-based integration testing method, propose an extended MM-path approach and use this approach to find out testing paths of component-based Web application.

- **Testing based on testability measures**

  Nadia Alshahwan et. al. (2009) [44] Proposed a framework for the collection of testability measures, during the automated testing process, called as ‘in-testing’ measure collection. The measures gathered in this way can take account of dynamic and content driven aspects of web applications, such as form structure, client-side scripting and server-side code. Their goal is to capture measurements related to on-going testing activity, indicating where additional testing can best lead to higher overall coverage. They denote a form of ‘web testability’ measures. This paper presents an implementation of a prototype Web Application Testing Tool, WATT, illustrating the in-testing measure collection approach on 34 forms taken from 14 real world web applications. Although the results are preliminary, they highlight some interesting features of the systems under test and the way in which measures can be used to draw the tester’s attention to them.

B. Techniques Used For Regression Testing of Software

- **Testing based on minimization techniques focusing on the coverage criteria**

  M. J. Harrold et. al. (1993) [40] Presented a heuristic for the minimal hitting set problem with the worst case execution time of O (|T| * max (|Ti|)). Here |T| represents the size of the original test suite, and max (|Ti|) represents the cardinality of the largest group of test cases among T1…..Tn.

  Y. Chen et. al. (1996) [58] Applied GE and GRE heuristics and compared the results to that of the HGS (Harrold-Gupta-Soffa) heuristic. The GE and GRE heuristics can be thought of as variations of the greedy algorithm that is known to be an effective heuristic for the set cover problem. He defined essential test cases as the opposite of redundant test cases. If a test requirement r_i can be satisfied by one and only one test case, the test case is an essential test case. On the other hand, if a test case satisfies only a subset of the test requirements satisfied by another test case, it is a redundant test case.

  D. Jeffrey et. al. (2005) [12] extended the heuristic so that certain test cases are selectively retained. This `selective redundancy' is obtained by introducing a secondary set of testing requirements. When a test case is marked as redundant with respect to the first set of testing requirements, Jeffrey considered whether the test case is also redundant with respect to the second set of testing requirements. If it is not, the test case is still selected, resulting in a certain level of redundancy with respect to the first set of testing requirements. The empirical evaluation used branch coverage as the first set of testing requirements and all-uses coverage information obtained by data-flow analysis. The results were compared to two versions of the heuristic, based on branch coverage and def-use coverage. The results showed that, while their technique produced larger test suites, the fault detection capability was better preserved compared to single-criterion versions of the heuristic.

- **Testing based on operation abstraction**

  M. Harder et. al. (2003) [41] Approached test suite minimisation using operational abstraction. An operational abstraction is a formal mathematical description of program behaviour. While it is identical to formal specifications in form, an operational abstraction expresses dynamically observed behaviour. He used the widely studied Daikon dynamic invariant detector to obtain operational abstractions. Daikon requires executions of test cases for the detection of possible program invariants. Test suite minimisation is proposed as follows: if the removal of a test case does not change the detected program invariant, it is rendered redundant. They compared the operational abstraction approach to branch coverage based minimisation. While their approach resulted in larger test suites, it also maintained higher fault detection capability. Moreover, he also showed that test suites minimised for coverage adequacy can often be improved by considering operational abstraction as an additional minimisation criterion.

- **Based on delta debugging technique**

  A. Leitner et. al. (2007) [2] Proposed somewhat different version of the minimisation problem. They start from the assumption that they already have a failing test case, which is too complex and too long for the human tester to understand. Note that this is often the case with randomly generated test data; the test case is often simply too complex for the human to establish the cause of failure. The goal of minimisation is to produce a shorter version of the test case; the testing requirement is that the shorter test case should still reproduce the failure. This minimisation problem is interesting because there is no uncertainty about the fault detection capability; it is given as a testing requirement. He applied the widely studied delta-debugging technique to reduce the size of the failing test case.

- **Based on logic criteria**

  J. M. Kaminski et. al. (2002) [23] investigated the use of a logic criterion to reduce test suites while guaranteeing fault detection in testing predicates over Boolean variables. From the formal description of fault classes, it is possible to derive a hierarchy of fault classes. From the hierarchy, it follows that the ability to detect a class of faults may guarantee the detection of another class. Therefore, the size of a test suite can be reduced by executing only those test cases for the class of faults that subsume another class, whenever this is feasible.

- **Based on regression test selection**

  Jiang Zheng et. al. (2006) [29] paper we present the application of the lightweight Integrated - Black-box Approach for Component Change Identification (I-BACCI) Version 3 process that select regression tests for applications that use COTS components. Two case studies, examining a total of nine new component releases, were conducted at ABB on products written in C/C++ to determine the effectiveness of I-BACCI. The results of the case studies indicate this process can reduce the required number of regression tests at least 70% without sacrificing the regression fault exposure.

  L. White et. al. (2008) [33] This paper investigates situations when data-flow paths are longer, and the testing of modules and components only one level away from the changed elements may not detect certain regression faults; an extended firewall considers these longer data paths. We report empirical studies that show the degree to which an extended firewall detected more faults, and how much more testing was required to achieve this increased detection.

  L. C. Briand et. al. (2009) [34] In this paper presents a methodology and tool to support test selection from regression test suites based on change analysis in object-oriented designs. We assume that designs are represented using the Unified Modelling Language (UML) and we propose a formal mapping between design changes and a classification of regression test cases into three categories: Reusable, Re-testable, and Obsolete. We provide evidence of the feasibility of the methodology and its usefulness by using our prototype tool on an industrial case study and two student projects.

- **Based on prioritization techniques**

  D. Jeffery et. al. (2006) [13] In this paper, we present a new approach to prioritize test cases based on the coverage requirements present in the relevant slices of the outputs of test cases. We present experimental results comparing the effectiveness of our prioritization approach with that of existing techniques that only account for total requirement coverage, in terms of ability to achieve high rate of fault detection. Our results present interesting insights into the effectiveness of using relevant slices for test case prioritization.

  Adam Smith et. al. (2007) [3] This paper presents a tool that constructs tree based models of a programs behaviour during testing and employs these trees while reordering and reducing a test suite. Using dynamic call tree or a calling context tree, the test reduction component identifies a subset of the original tests that covers the same call tree paths. The prioritization technique reorders a test suite so that it covers the call tree paths more rapidly than the initial ordering. For a chosen case study application, the experimental results show that a prioritized suite achieves coverage much faster and a reduced test suite contains 45% fewer and consumes 82% less time.

Adam Smith et. al. (2009) [4] this paper uses the Harrold Gupta Soffa, delayed greedy, traditional greedy, and 2-optimal greedy algorithms for both test suite reduction and prioritization. Even though reducing and reordering a test suite is primarily done to ensure that testing is cost-effective, these algorithms are normally configured to make greedy choices with coverage information alone. This paper extends these algorithms to greedily reduce and prioritize the tests by using both test cost (e.g., execution time) and the ratio of code coverage to test cost. An empirical study with eight real world case study applications shows that the ratio greedy choice metric aids a test suite reduction method in identifying a smaller and faster test suite. The results also suggest that incorporating test cost during prioritization allows for an average increase of 17% and a maximum improvement of 141% for time sensitive evaluation metric called coverage effectiveness.

CHAPTER 4

LITERATURE REVIEW

CHAPTER 4
LITERATURE REVIEW

4.1 INTRODUCTION

A Literature Review is a body of text that aims to identify, evaluate, and synthesize the critical points of current knowledge and methodological contributions to a particular topic by the author. Literature reviews are secondary sources, and as such, do not report any new or original experimental work. It should give a theoretical basis for the research and helps to determine the nature of research. It identifies what is already known about an area of research.

For the present study, literature has been collected from different sources such as journal articles, Internet and conference proceeding paper.

4.2 REVIEWS ON SQL INJECTION TECHNIQUES

D. Scott et. al. (2002) [14] Security Gateway is a proxy filtering system that enforces input validation rules on the data flowing to a Web application. Using their Security Policy Descriptor Language (SPDL), developers provide constraints and specify transformations to be applied to application parameters as they flow from the Web page to the application server. Because SPDL is highly expressive, it allows developers considerable freedom in expressing their policies. However, this approach is human-based and, like defensive programming, requires developers to know not only which data needs to be filtered, but also what patterns and filters to apply to the data.

Y. Huang et. al. (2003) [64] WAVES implements a machine learning method in building attack requests. It analyzes the response page to verify SQLIVs and modify the attack request for deep injection. It is better than traditional penetration test methods by improving the attack methodology, but it cannot test all the vulnerable spots.

C. Gould et. al. (2004) [8] JDBC-Checker is a technique for statically checking the type correctness of dynamically-generated SQL queries. This technique was not developed with the intent of detecting and preventing general SQLIAs, but can nevertheless be used to prevent attacks that take advantage of type mismatches in a dynamically-generated query string. JDBC-Checker is able to detect one of the root causes of SQLIA vulnerabilities in code improper type checking of input. However, this technique would not catch more general forms of SQLIAs because most of these attacks consist of syntactically and type correct queries. The primary drawback of this technique is that its scope is limited to detecting and preventing tautologies and cannot detect other types of attacks.

S. W. Boyd et. al. (2004) [54] SQLrand is an approach based on instruction-set randomization. SQLrand provides a framework that allows developers to create queries using randomized instructions instead of normal SQL keywords. A proxy filter intercepts queries to the database and de-randomizes the keywords. SQL code injected by an attacker would not have been constructed using the randomized instruction set. Therefore, injected commands would result in a syntactically incorrect query. While this technique can be very effective, it has several practical drawbacks. First, since it uses a secret key to modify instructions, security of the approach is dependent on attackers not being able to discover the key. Second, the approach imposes a significant infrastructure overhead because it requires the integration of a proxy for the database in the system.

Y. Huang et. al. (2004) [65] WebSSARI detects input validation related errors using information flow analysis. In this approach, static analysis is used to check taint flows against preconditions for sensitive functions. The analysis detects the points in which preconditions have not been met and can suggest filters and sanitization functions that can be automatically added to the application to satisfy these preconditions. The WebSSARI system works by considering as sanitized input that has passed through a predefined set of filters. In their evaluation, the authors were able to detect security vulnerabilities in a range of existing applications. The primary drawbacks of this technique are that it assumes that adequate preconditions for sensitive functions can be accurately expressed using their typing system and that having input passing through certain types of filters is sufficient to consider it not tainted. For many types of functions and applications, this assumption is too strong.

W. G. Halfond et. al. (2005) [61] AMNESIA is a model-based technique that combines static analysis and runtime monitoring. In its static phase, AMNESIA uses static analysis to build models of the different types of queries an application can legally generate at each point of access to the database. In its dynamic phase, AMNESIA intercepts all queries before they are sent to the database and checks each query against the statically built models. Queries that violate the model are identified as SQLIAs and prevented from executing on the database. In their evaluation, the authors have shown that this technique performs well against SQLIAs. The primary limitation of this technique is that its success is dependent on the accuracy of its static analysis for building query models. Certain types of code obfuscation or query development techniques could make this step less precise and result in both false positives and false negatives.

F. Valeur et. al. (2005) [18] propose the use of an Intrusion Detection System (IDS) to detect SQLIAs. Their IDS system is based on a machine learning technique that is trained using a set of typical application queries. The technique builds models of the typical queries and then monitors the application at runtime to identify queries that do not match the model. They have shown that their system is able to detect attacks with a high rate of success. However, the fundamental limitation of learning based techniques is that they can provide no guarantees about their detection abilities because their success is dependent on the quality of the training set used. A poor training set would cause the learning technique to generate a large number of false positives and negatives.

R. McClure et. al. (2005) [48] and W. R. Cook et. al. (2005) [62] developed SQL DOM and Safe Query Objects, use encapsulation of database queries to provide a safe and reliable way to access databases. These techniques offer an effective way to avoid the SQLIA problem by changing the query-building process from an unregulated one that uses string concatenation to a systematic one that uses a type-checked API. Within their API, they are able to systematically apply coding best practices such as input filtering and rigorous type checking of user input. By changing the development paradigm in which SQL queries are created, these techniques eliminate the coding practices that make most SQLIAs possible. Although effective, these techniques have the drawback that they require developers to learn and use a new programming paradigm or query-development process. Furthermore, because they focus on using a new development process, they do not provide any type of protection or improved security for existing legacy systems.

Z. Su et. al. (2006) [67] developed SQLCheck that check queries at runtime to see if they conform to a model of expected queries. In these approaches, the model is expressed as a grammar that only accepts legal queries. In SQLCheck, the model is specified independently by the developer. Both approaches use a secret key to delimit user input during parsing by the runtime checker, so security of the approach is dependent on attackers not being able to discover the key. Additionally, the use of these two approaches requires the developer to either rewrite code to use a special intermediate library or manually insert special markers into the code where user input is added to a dynamically generated query.

Y. Kosuga et. al. (2007) [66] presented a technique SANIA for detecting SQLIV in web applications in development and debugging phase which using the following procedures. 1) Sania intercepts the SQL queries between a web application and a database and then, collects normal SQL queries between client and web applications and between the web application and database, and analysis the vulnerabilities. 2) It automatically generates elaborate attacks according to the syntax and semantics of the potentially vulnerable spots in the SQL queries. 3) After attacking with the generated code, it collects the SQL queries generated from the attack. 4) Sania compares the parse trees of the intended SQL query and those resulting after an attack to assess the safety of these spots. 5) Finally, it determines whether the attack succeeded or not. By analyzing the syntax in the parse tree of SQL queries, it is possible to generate precise pinpoint attack requests.

Shahriar et. al. (2008) [27] proposed a Mutation-based SQL Injection vulnerabilities Checking (testing) tool (MUSIC) that automatically generates mutants for the applications written in Java Server Pages (JSP) and performs mutation analysis. Mutation is the act of intentionally modifying a program's code, then re-running a suite of valid unit tests against the mutated program. Mutation testing is a method of fault-based software testing, which involves modifying programs' source code or byte code in small ways. Mutation testing is done by selecting a set of mutation operators and then applying them to the source program one at a time for each applicable piece of the source code. The result of applying one mutation operator to the program is called a mutant. These mutants are killed by a test case if it is causes different end output or different intermediate state between the original program and a mutant. Otherwise the mutant is remaining alive. Additional test cases should be generated for killing live mutants. Authors proposed nine mutation operators to inject SQLIV in source code of application which four of them inject faults into generated SQL queries and remaining five of operators inject faults into the API method calls. However, MUSIC is very simple and effective way for testing SQL queries having simple form, but it cannot address the SQLIV of stored procedures.

Haiyan Wu et. al. (2011) [28] presents a new test method called SMART, which automatically tests SQLIVs in web applications. SMART analyzes the SQL queries generated by web applications and uses a structure matching validation mechanism to determine whether SQLIVs exist. Comprehensive experiments show that SMART is effective in finding SQLIVs. Testing the web applications with SMART, the security against SQL injection can be greatly improved.

Zoran Djuric et. al. (2013) [68] developed a reliable black-box vulnerability scanner for detecting SQLI vulnerability SQLIVDT (SQL Injection Vulnerability Detection Tool). The black-box approach is based on simulation of SQLI attacks against web applications. Thus, the scope of analysis is limited to HTTP responses and HTML pages received from the application server. In order to achieve efficient SQLI vulnerability detection, an efficient algorithm for HTML page similarity detection is used. The proposed tool showed promising results as compared to six well-known web application scanners.

CHAPTER 5

PROPOSED METHODOLOGY

CHAPTER 5
PROPOSED METHODOLOGY

5.1 INTRODUCTION

The first step for successful SQL injection exploitation is to find the vulnerable piece of code which will allows performing the injection. The primary focus of our research was to secure the web applications and improve the way in which the web applications are tested. So that every loop hole in an application and the attacks that are injected on the web application can be detected.

Therefore, to secure the web applications we have developed reliable black box vulnerability scanner for detecting SQLI vulnerabilities which is called as SQLIVS (SQL injection vulnerability scanner). The black box approach is based on simulation of SQLI attacks against web applications. It analyzes the value submitted by users through HTML forms, URL parameters whether clean or normal parameters and look for possible attack patterns. SQLIVS proposes a simple and effective method to accurately detect and prevent SQLIV by using SQL query parameters.

We have coded SQLIVS for four SQL injection attacks namely Tautologies, piggy backed query, inference and stored procedures. It is very important to mention that SQLIVS user can add new attack patterns to the SQLIVS database. This way SQLIVS can be easily extended to support different and new SQLI attacks. There are numerous variations of each SQLI attack type. This is why we present only a few most important supported by SQLIVS.

The proposed technique for SQLIVS showed promising results as compared to other techniques. Two new essential features have been added in this technique, one of which handles the phenomenon of *login form disappearance* and the other feature provides an *expandable payload* which gives the opportunity to the user to add new attack patterns to SQLIVS database. This way SQLIVS can be easily extended to support different and new SQLI attacks.

5.2 WORKING OF SQLIVS

In figure 5.1, we have shown the flow chart of SQLIVS. This flow chart tells us that how scanner is working and on what parameters. We have used Web Proxy in our tool. A **Web proxy** help to bypass client-side restrictions, providing full control of the requests sent to servers. Additionally, they offer greater visibility of the response from the server, providing greater chances of detecting subtle vulnerabilities that could remain undetected if visualized in the Web browser. This tool analysis only the dynamic code of an HTML page, and skips the static code of the page. The dynamic code consists of the inputs forms and the URL parameters of a page.

SQLIVS uses **web proxy**, so can enforce input validation rules on the data flowing to a web application. Using this method filters can be applied on the data of input forms, URLs and cookies also.

Our tool tries to identify points in a web application that can be used to inject malicious code. Then perform attacks that target these points and monitor how the application responses to the generated attacks.

Most of the scanners work only on normal parameters but SQLIVS works on both the type of parameters ***Clean*** and ***Normal*** URL parameters. In the case of clean URL parameters, first we perform the SQL injection attacks on all the URLs detected, and then check for response:

1) if the HTTP response code = 200, then the hold is successfully transferred to other page but even then there are chances of potential SQL injection attack as sometimes its transferred to a vulnerable pages because of insertion of spoof data.
2) If the HTTP response code = 404, then it is okay, the page is not found but no SQL injection attack will occur.
3) If the HTTP response code >=500 but <=599, they serious risk, SQL injection attack detected.

In case of normal URL parameter, we again perform all the attacks and make changes to the URL and try to run it again, if it navigates to another pages then SQL injection attack is detected otherwise not.

The Input forms are detected and at every entry point the SQL injection attacks are performed and the response is checked. If user is unable to login then output is shown as error not found. But if user logs in and on the navigated page the logout button or similar words in the created dictionary are found then it’s a successful login. Other case could be that when navigation is made to the other page if no such word is found, then the output will show as potential successful login but with spoof data. So that the phenomenon of “Form Disappearance” can be handled and checks can be put so that on SQL injection attack occurs.

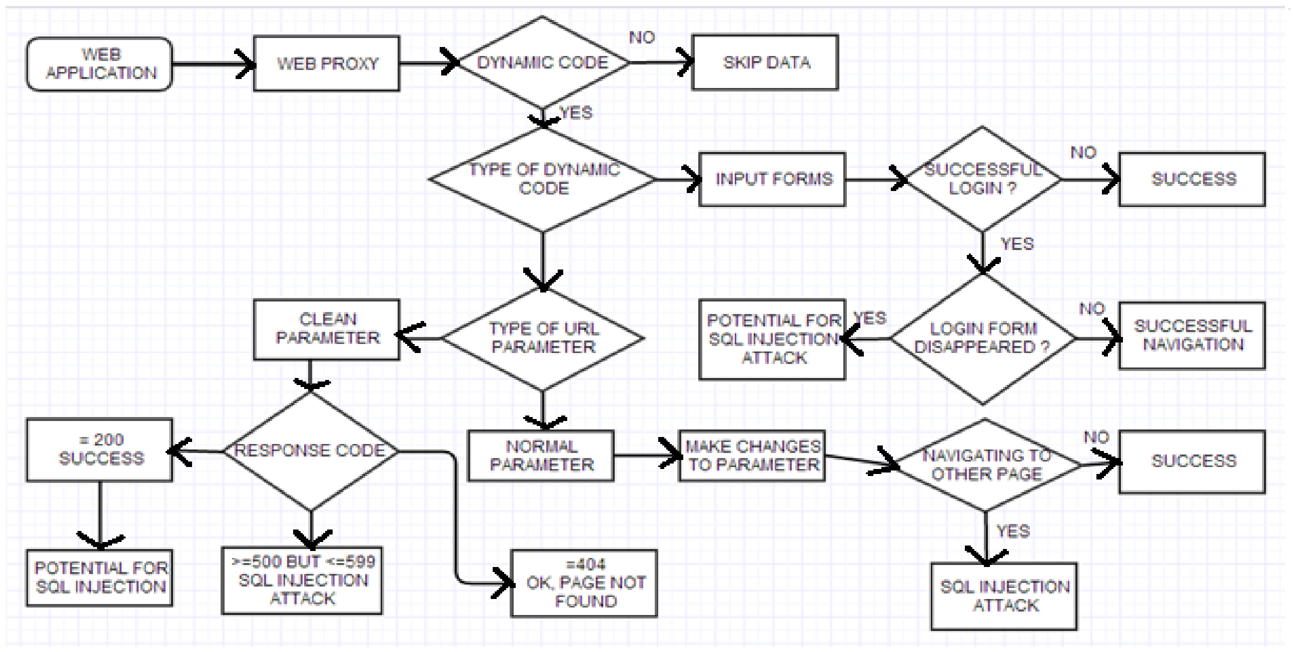


Figure 5.1: Flow Chart for Working of SQLIVS

5.3 PROPOSED ALGORITHM

Step 1: detect dynamic code of an HTML page and skip the static code.

Step 2: detect all the input forms on an HTML page and inject SQL injection attacks on each form and check for the response

- If (!logged in)

  Then

  OUTPUT: error not found

- If (logged in)

  Then

  Search for keywords like (Sign out, logout, successful ….)

  If(keyword found)

  Then

  OUTPUT: success

  Else

  OUTPUT: potential login with spoof data

Step 3: detect the URL parameters

If URL is of the form (…/…./…/), inject the SQL injection attack on them

Then

Check for HTTP response code

- If (response code = 200)

  Then

  OUTPUT: successful, but potential for SQL injection attack

- Else if (response code = 404)

  Then

  OUTPUT: error, page not found

- Else if (response code >=500 && <=599)

  Then

  OUTPUT: SQL injection attack and the attack type

Step 4: if detected URL is of the form (…..=….?...../….?....=…..), inject the SQL injection attack on them

Then

Add junk values to the URL or remove the parts after "/" symbol and run it again

If (navigates to other page)

Then

OUTPUT: SQL injection attack and the attack type

5.4 FEATURES OF SQLIVS

1) SQLIVS **uses web proxy** which helps to bypass client-side restrictions, providing full control of the requests sent to servers. Additionally, they offer greater visibility of the response from the server, providing greater chances of detecting subtle vulnerabilities that could remain undetected if visualized in the Web browser [31].

2) Since SQLIVS **web proxy**, therefore does not face the Near-Duplicate Issue which is a major concern for web crawling based SQLI scanners [25].

3) SQLIVS provides an **expandable dictionary**, so that all the new keywords can be added to SQLIVS database that defines the successful login into a page, and hence ease the process.

4) SQLIVS **works on Clean URL parameters** also, which provide an upper hand to this tool as most of the scanners do not check clean URL parameter, and hence misses most of the SQLI vulnerabilities.

5) SQLIVS provides an **expandable payload** therefore SQLIVS user can add new attack patterns to the SQLIVS database. This way SQLIVS can be easily extended to support different and new SQLI attacks.

6) SQLIVS handles the phenomenon of **login form disappearance** thereby more SQL injection vulnerabilities can be discovered.

CHAPTER 6

RESULT AND DISCUSSION

CHAPTER 6

RESULTS AND DISCUSSION

6.1 INTRODUCTION

All the experimental work is done in Java using the official NetBeans IDE 8.0 and system information is as follows, OS-Window 8 single language, processor-Intel(R) core(TM) i5 3337U CPU@1.8 GHz, Physical memory-4.00 GB, and System type- 64 bit processor. Here we discuss and show the result of our experiment**.**

6.2 COMPARATIVE ANALYSIS

When the technique used for SQLIVS was compared with other known techniques, it showed promising results. We evaluate the techniques presented in chapter 4 using several different criteria. We first consider which attack types each technique is able to address. Finally, we evaluate the deployment requirements of each technique.

6.2.1 Comparison of Techniques Based On Attack Types

In table 6.1 we have shown the comparison of the detection focused techniques based on the attack types that were discussed in chapter 2. Evaluation results for detection based techniques were taken from [63], all the detection based approaches are techniques that detect the SQLI attacks at runtime.

Most of the detection-focused techniques perform fairly uniformly against the various attack types. The three exceptions are the IDS based approach [18], whose effectiveness depends on the quality of the training set used, Java Dynamic Tainting [59], whose performance is negatively affected by the fact that its tainting operations allow input to be used without regard to the quality of the check, and Tautology-checker [24], which by definition can only address tautology-based attacks.

The proposed technique for SQLIVS showed better results when compared to other detection focused techniques.

Detection Technique	**Tautologies**	**Piggy Backed Query**	**Inference**	**Stored Procedure**
AMNESIA	●	●	●	X
CSSE	●	●	●	X
IDS	◌	◌	◌	◌
Java Dynamic Tainting	–	–	–	-
SQL Check	●	●	●	X
SQL Guard	●	●	●	X
SQL Rand	●	●	●	X
Tautology Checker	●	X	X	X
Web App Hardening	●	●	●	X
Proposed Technique(for SQLIVS)	●	●	●	◌

Table 6.1: Comparison of Detection-Focused Techniques With Respect To Attack Types.

Symbols: '●' defines that detection is possible, 'x' defines that detection is impossible

'◌' defines detection is partially possible, '–'defines that there is no relation

6.2.2 Comparison of Techniques Based On Deployment Requirements

Each of the techniques has different deployment requirements. We need to determine the effort and infrastructure required to use the technique. Evaluation for each technique has to be done with respect to the following criteria: (1) Does the technique require developers to modify their code base? (2) What is the degree of automation of the detection aspect of the approach? (3) What is the degree of automation of the prevention aspect of the approach? (4) What infrastructure (not including the tool itself) is needed to successfully use the technique? The result of this classification has been taken from [63] and is summarized in Table 6.2.

Detection Technique	**Source code Adjustment**	**Attack Detection**	**Attack Prevention**	**Additional Elements**
AMNESIA	Not Needed	Automatic	Automatic	N/A
CSSE	Not Needed	Automatic	Automatic	Custom PHP Interpreter
IDS	Not Needed	Automatic	Report Generation	IDS System Training Set
Java Dynamic Tainting	Not Needed	Automatic	Automatic	N/A
SQL Check	Needed	Partially Automatic	Automatic	Key Management
SQL Guard	Needed	N/A	Automatic	N/A
SQL Rand	Needed	Automatic	Automatic	Proxy Developer Learning Key Management
Tautology Checker	Not Needed	Automatic	Source Code Adjustment	N/A
Web App Hardening	Not Needed	Automatic	Automatic	Custom PHP Interpreter
Proposed Technique(for SQL IVS)	**Not Needed**	**Automatic**	**Report Generation**	**Proxy**

Table 6.2: Comparison of Techniques With Respect To Deployment Requirements.

CONCLUSION AND FUTURE WORK

The primary focus of our research was to secure the web applications and improve the way in which the web applications are tested. So that every loop hole in an application and the attacks that are injected on the web application can be detected.

Therefore, to secure the web applications we have developed reliable black box vulnerability scanner for detecting SQLI vulnerabilities which is called as SQLIVS (SQL injection vulnerability scanner). The black box approach is based on simulation of SQLI attacks against web applications. It analyzes the value submitted by users through HTML forms, URL parameters whether clean or normal parameters and look for possible attack patterns. SQLIVS proposes a simple and effective method to accurately detect and prevent SQLIV by using SQL query parameters.

To further improve the web application testing process I have introduced a new paradigm called as QUIT paradigm that focuses on the key features of regression testing both in software and web applications. This paradigm also tells about the dissimilarities that exist between the regression testing criteria for software and web application.

The proposed technique for SQLIVS showed promising results as compared to other techniques. Two new essential features have been added in this technique, one of which handles the phenomenon of *login form disappearance* and the other feature provides an *expandable payload* which gives the opportunity to the user to add new attack patterns to SQLIVS database. This way SQLIVS can be easily extended to support different and new SQLI attacks.

In future we can add the new SQLI attacks patterns to make it more beneficial tool for SQLIV detection. Secondly, it should be expanded to handle more web application attacks such as XSS, code injection etc. Moreover new and highly efficient detection techniques should be implemented in SQLIVS to enhance its functionality and usage. Finally, new innovative features should be created that can be added to SQLIVS to make it more user friendly.

REFERENCES

[1] A. Tarhini, H. Fouchal, and N. Mansour, "Regression Testing Web Services-Based Applications", In AICCSA Proceedings of the IEEE International Conference on Computer Systems and Applications, 2006.

[2] A. Leitner, M. Oriol, A. Zeller, I. Ciupa, and B. Meyer, "Efficient Unit Test Case Minimization", In Proceedings of the 22nd IEEE/ACM international conference on Automated Software Engineering, 2007.

[3] A. Smith, J. Geiger, G. M. Kapfhammer, and M. L. Soffa, "Test Suite Reduction and Prioritization with Call Trees", In Proceedings of the IEEE/ACM International Conference on Automated Software Engineering, 2007.

[4] A. M. Smith and G. M. Kapfhammer, "An Empirical Study Of Incorporating Cost Into Test Suite Reduction And Prioritization", In Proceedings of the 24th Symposium on Applied Computing, 2009.

[5] C. Anley, "Advanced SQL Injection In SQL Server Applications", White paper, Next Generation Security Software Ltd., 2002.

[6] C. A. Mackay, "SQL Injection Attacks and Some Tips on How to Prevent them", Technical report, The Code Project, 2005.

[7] C. Anley, "(more) Advanced SQL Injection", White paper, Next Generation Security Software Ltd., 2002.

[8] C. Gould, Z. Su, and P. Devanbu, "JDBC Checker: A Static Analysis Tool for SQL/JDBC Applications", In Proceedings of the 26th International Conference on Software Engineering, 2004.

[9] D. Aucsmith, "Creating and Maintaining Software that Resists Malicious Attack", distinguished lecture series, 2004.

[10] D. Litchfield, "Web Application Disassembly with ODBC Error Messages", Technical document, Stake, Inc., 2002.

[11] D. Binkley, "Semantic Guided Regression Cost Reduction", In IEEE Transactions on Software Engineering 23, 1997.

[12] D. Jeffrey and N. Gupta, "Test Suite Reduction with Selective Redundancy", In Proceedings of the 21st IEEE International Conference on Software Maintenance, 2005.

[13] D. Jeffrey and N. Gupta, “Test Case Prioritization Using Relevant Slices”, In Proceedings of the 30th Annual International Computer Software and Applications Conference, IEEE 2006.

[14] D. Scott and R. Sharp, “Abstracting Application-level Web Security”, In Proceedings of the 11th International Conference on the World Wide Web, 2002.

[15] E. M. Fayo, “Advanced SQL Injection In Oracle Databases”, Technical report, Information Security, Black Hat Briefings, Black Hat USA, 2005.

[16] E. Miller, “Website Testing”, White Paper, 2005.

[17] F. Bouma, “Stored Procedures Are Bad, Okay”, Technical report, Asp.Net Weblogs, 2003.

[18] 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 and Vulnerability Assessment (DIMVA), 2005.

[19] F.I. Vokolos and P.G. Frankl, “Pythia: A Regression Test Selection Tool Based On Textual Differencing”, In Proceedings of the 3rd International Conference on Reliability, Quality and Safety of Software-Intensive Systems, 1997.

[20] G. Rothermel and M. Harrold, “Selecting Tests And Identifying Test Coverage Requirements For Modified Software”, in proceedings of the International Symposium on Software Testing and Analysis, 1994.

[21] G. Rothermel, M.J. Harrold, and J. Dehia, “Regression Test Selection for C++”, In Software Testing verification and reliability, 2000.

[22] G. Rothermel and M. J. Harrold, “A Safe, Efficient Algorithm for Regression Test Selection”, In Proceedings of the conference on software maintenance. CA, 1993.

[23] G. K. Kaminski and P. Ammann, “Using Logic Criterion Feasibility To Reduce Test Set Size While Guaranteeing Fault Detection”, In Proceedings of International Conference on Software Testing, Verification, and Validation, IEEE 2009.

[24] 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, 2004.

[25] Gurmeet Singh Manku, Arvind Jain, Anish Das Sarma, “Detecting Near Duplicates for Web Crawling”, in world wide web conference, 2007.

[26] H. Leung and L. White, “A Firewall Concept in both Control-Flow and Data-Flow In Regression Integration Testing”, In Proceedings of the Conference on Software Maintenance, 1992.

[27] H. Shahriar and M. Zulkernine, “MUSIC: Mutation-based SQL Injection Vulnerability Checking,” in Proceedings the Eighth International Conference on Quality Software, IEEE 2008.

[28] Haiyan Wu and Guozhu Gao chunyu Miao, “Test SQL Injection Vulnerabilities in Web Applications Based on Structure Matching”, 20th International Conference on Computer Science and Network Technology, 2011.

[29] J. Zheng, B. Robinson, L. Williams, and K. Smiley, “A Lightweight Process For Change Identification And Regression Test Selection In Using Cots Components”, In Proceedings of the 5th International Conference, IEEE 2006.

[30] Jingxian Gu, Lei Xu, Baowen Xu, Hongji Yang, “An Extended MM-Path Approach to Component-based Web Application Testing”, 12th IEEE International Workshop on Future Trends of Distributed Computing Systems, 2008.

[31] Justin Clarke, “SQL injection attack and defence”, IEEE 2009.

[32] K. Spett, “Blind SQL injection”, White paper, SPI Dynamics, Inc., 2003.

[33] L. White, K. Jaber, B. Robinson, and V. Rajlich, “Extended Firewall for Regression Testing: an experience report”, Journal of Software Maintenance and Evolution, 2008.

[34] L. C. Briand, Y. Labiche, and S. He, “Automating Regression Test Selection Based on UML Designs”, Journal of Information and Software Technology, 2009.

[35] Laxman Singh, Sushmit M Vishwakarma, Yashvant Ugalmugale and Prasad Naikwade, “Detection and Prevention of SQL Injection Attacks on Database Using Web Services”, International Journal of Emerging Technology and Advanced Engineering, Volume 4, Issue 4, 2014.

[36] Lei Xu, Baowen Xu, Zhenqiang Chen, Jixiang Jiang, and Huowang Chen, “Regression Testing For Web Applications Based On Slicing”, In Proceedings of the 27th Annual International Computer Software and Applications Conference, 2003.

[37] Lei Xu, Baowen Xu, “Applying Agent into Intelligent Web Application Testing”, International Conference on Cyber worlds, 2007.

[38] M. Howard and D. LeBlanc, “Writing Secure Code”, Microsoft Press, Redmond, Washington, second edition, 2003.

[39] M. Ruth and S. Tu, “A Safe Regression Test Selection Technique For Web Services”, In Proceedings of the Second International Conference on Internet and Web Applications and Services, IEEE Computer Society, 2007.

[40] M. J. Harrold, R. Gupta and M. L. Soffa, “A Methodology for Controlling The Size of A Test Suite”, ACM Transactions on Software Engineering and Methodology, 1993.

[41] M. Harder, J. Mellen, and M. D. Ernst, “Improving Test Suites via Operational Abstraction”, In Proceedings of the 25th International Conference on Software Engineering, 2003.

[42] N. Mansour and K. El-Fakih, “Simulated Annealing and Genetic Algorithms for Optimal Regression Testing”, In Journal of Software Maintenance, Vol. 11, 1999.

[43] N. Mansour and R. Bahsoon, “Reduction-Based Methods and Metrics for Selective Regression Testing”, In Information and Software Technology, Volume 44, Number 7, 2002.

[44] Nadia Alshahwan, Mark Harman, Alessandro Marchetto and Paolo Tonella, “Improving Web Application Testing Using Testability Measures”, IEEE 2009.

[45] P. Finnigan., “SQL Injection and Oracle - Parts 1 & 2”, Technical Report, Security Focus, 2002.

[46] P. Hisa, X. Li, D.C. Kung, C.T. Hsu, Y. Toyoshima L. Li, and C. Chen, “A Technique For The Selective Revalidation Of OO Software”, In Journal of Software Maintenance 9, 1997.

[47] R. Gupta, M.J. Harrold, and M.L. Sofa, “Program Slicing-Based Regression Testing Techniques”, in Software Testing verification and reliability, 1996.

[48] R. McClure and I. Kruger, “SQL DOM: Compile Time Checking of Dynamic SQL Statements”, In Proceedings of the 27th International Conference on Software Engineering, 2005.

[49] Rahul Johari, Pankaj Sharma, “A Survey on Web Application Vulnerabilities (SQLIA, XSS) Exploitation and Security Engine for SQL Injection”, International Conference on Communication Systems and Network Technologies, 2012.

[50] S. McDonald, “SQL Injection: Modes of attack, defence, and why it matters”, White paper, GovernmentSecurity.org, April 2002.

[51] S. Labs, “SQL Injection”, White paper, SPI Dynamics, Inc., 2002.

[52] S. Elbaum, S. Karre, and G. Rothermel, “Improving Web Application Testing with User Session Data”, In Proceedings International Conference Software Engineering, May 2003.

[53] S. Elbaum, S. Karre, and G. Rothermel and Marc Fisher, “Leveraging User-Session Data to Support Web Application Testing ”, IEEE transactions on software engineering, volume 31, no. 3, march 2005.

[54] S. W. Boyd and A. D. Keromytis, “SQLrand: Preventing SQL Injection Attacks”, In Proceedings of the 2nd Applied Cryptography and Network Security (ACNS) Conference, June 2004.

[55] Shruti Bangre, Alka Jaiswal, "SQL Injection Detection and Prevention Using Input Filter Technique", International Journal of Recent Technology and Engineering, Volume-1, Issue-2, 2012.

[56] Sreedevi Sampath, Renee C. Bryce, Gokulanand Viswanath, Vani Kandimalla, A. Gunes Koru, "Prioritizing User-session-based Test Cases for Web Applications Testing", International Conference on Software Testing, Verification, and Validation, 2008

[57] Swarnendu Biswas and Rajib Mall, "Regression Test Selection Techniques: A Survey", Informatica 35, 2011.

[58] T. Y. Chen and M. F. Lau, "Dividing strategies for the optimization of a test suite", Information Processing Letters, 1996.

[59] V. Haldar, D. Chandra, and M. Franz, "Dynamic Taint Propagation for Java", In Proceedings 21st Annual Computer Security Applications Conference, 2005.

[60] Vinay Kumar S.B, Manjula N Harihar, "Diameter-based Protocol in the IP Multimedia Subsystem", International Journal of Soft Computing and Engineering, Volume 1, Issue-6, 2012.

[61] W. G. Halfond and A. Orso, "AMNESIA: Analysis and Monitoring for Neutralizing SQL-Injection Attacks", In Proceedings of the IEEE and ACM International Conference on Automated Software Engineering, 2005.

[62] W. R. Cook and S. Rai, "Safe Query Objects: Statically Typed Objects as Remotely Executable Queries", In Proceedings of the 27th International Conference on Software Engineering, 2005.

[63] William G.J. Halfond, Jeremy Viegas, and Alessandro Orso, "A Classification of SQL Injection Attacks and Countermeasures", IEEE 2006.

[64] Y. Huang, S. Huang, T. Lin, and C. Tsai, "Web Application Security Assessment by Fault Injection and Behaviour Monitoring", In Proceedings of the 11th International World Wide Web Conference, 2003.

[65] Y. Huang, F. Yu, C. Hang, C. H. Tsai, D. T. Lee, and S. Y. Kuo, "Securing Web Application Code by Static Analysis and Runtime Protection", In Proceedings of the 12th International World Wide Web Conference, 2004.

[66] Y. Kosuga, K. Kernel, M. Hanaoka, M. Hishiyama, and Y. Takahama, "Sania: syntactic and semantic analysis for automated testing against SQL injection," in Proceedings the Computer Security Applications Conference, 2007.

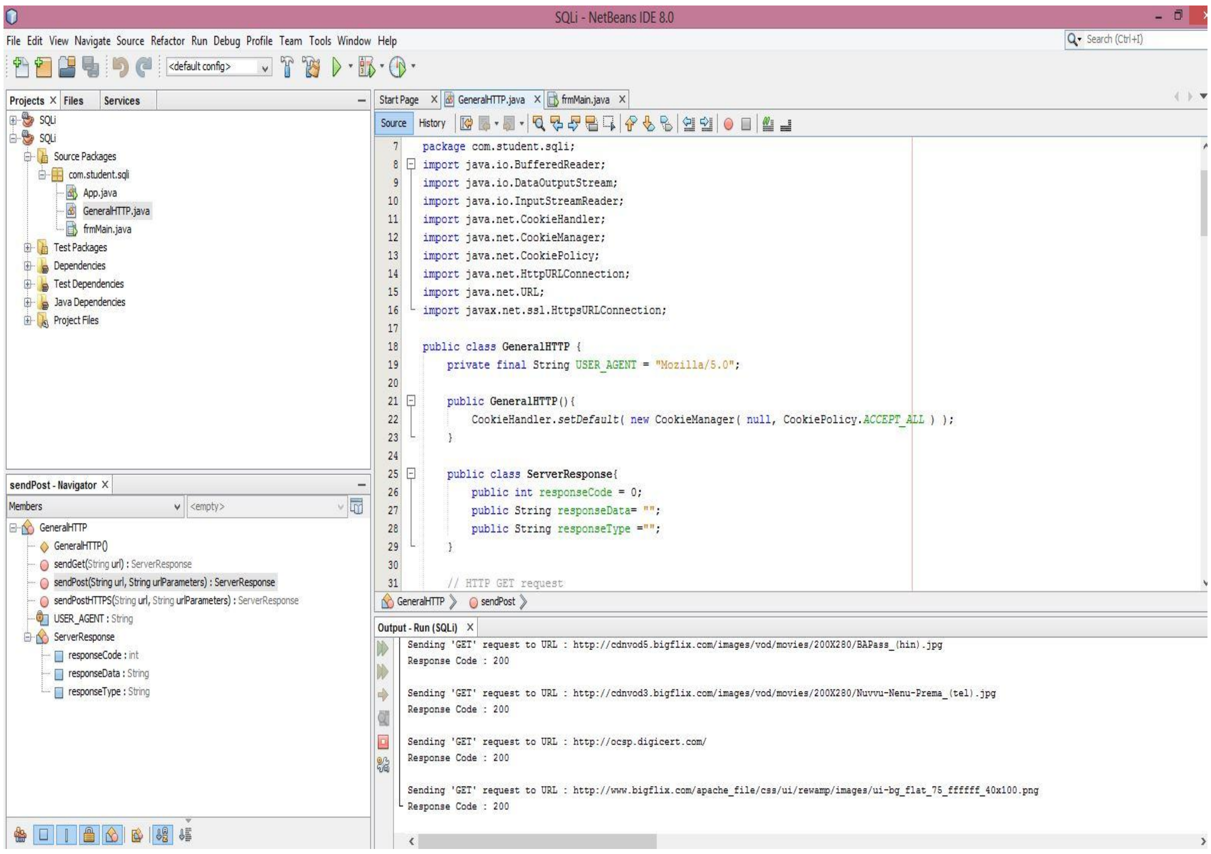
[67] Z. Su and G. Wassermann, “The Essence of Command Injection Attacks in Web Applications”, in the 33rd Annual Symposium on Principles of Programming Languages (POPL 2006), 2006.

[68] Zoran Djurica, “Black-box Testing Tool for Detecting SQL Injection Vulnerabilities”, IEEE 2013.

APPENDIX A

APPENDIX A

In figure A.1, a screenshot of code running on NetBeans 8.0 has been shown. This code is implemented in JAVA and run to detect the SQL injection attacks on a web application. It has two classes; one is a general class that defines the HTTP GET and POST requests, and looks for HTTP response code. The other class is the main class in which all the SQLIVS algorithm is implemented.

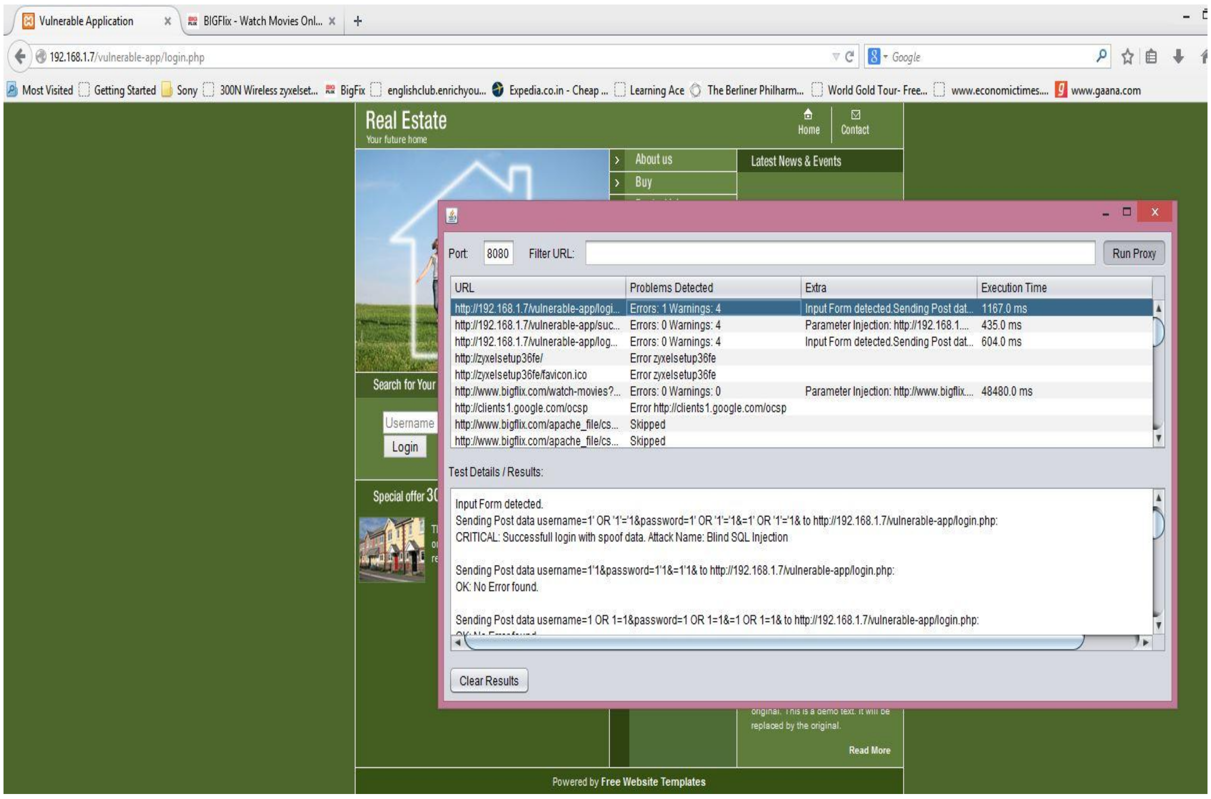


```java
package com.student.sqli;
import java.io.BufferedReader;
import java.io.DataOutputStream;
import java.io.InputStreamReader;
import java.net.CookieHandler;
import java.net.CookieManager;
import java.net.CookiePolicy;
import java.net.HttpURLConnection;
import java.net.URL;
import javax.net.ssl.HttpsURLConnection;

public class GeneralHTTP {
    private final String USER_AGENT = "Mozilla/5.0";

    public GeneralHTTP(){
        CookieHandler.setDefault( new CookieManager( null, CookiePolicy.ACCEPT_ALL ) );
    }

    public class ServerResponse{
        public int responseCode = 0;
        public String responseData= "";
        public String responseType ="";
    }

    // HTTP GET request
```

```
Sending 'GET' request to URL : http://cdnvod5.bigflix.com/images/vod/movies/200X280/BAPass_(hin).jpg
Response Code : 200

Sending 'GET' request to URL : http://cdnvod3.bigflix.com/images/vod/movies/200X280/Nuvvu-Nenu-Prema_(tel).jpg
Response Code : 200

Sending 'GET' request to URL : http://ocsp.digicert.com/
Response Code : 200

Sending 'GET' request to URL : http://www.bigflix.com/apache_file/css/ui/rewamp/images/ui-bg_flat_75_ffffff_40x100.png
Response Code : 200
```

Figure A.1: Screenshot of Code Running On NetBeans 8.0

In figure A.3, a screenshot of output screen and the running web application is shown. The running web application is a vulnerable application that uses web proxy and runs on the local host using XAMPP tool that provides this platform. On the output screen all the detected vulnerabilities of the web application are shown. Here we can see that approximately 9 warnings are there, that were detected while running the vulnerable application.



URL	Problems Detected	Extra	Execution Time
http://192.168.1.7/vulnerable-app/logi...	Errors: 1 Warnings: 4	Input Form detected.Sending Post dat...	1167.0 ms
http://192.168.1.7/vulnerable-app/suc...	Errors: 0 Warnings: 4	Parameter Injection: http://192.168.1....	435.0 ms
http://192.168.1.7/vulnerable-app/log...	Errors: 0 Warnings: 4	Input Form detected.Sending Post dat...	604.0 ms
http://zyxelsetup36fe/	Error zyxelsetup36fe		
http://zyxelsetup36fe/favicon.ico	Error zyxelsetup36fe		
http://www.bigflix.com/watch-movies?...	Errors: 0 Warnings: 0	Parameter Injection: http://www.bigflix....	48480.0 ms
http://clients1.google.com/ocsp	Error http://clients1.google.com/ocsp		
http://www.bigflix.com/apache_file/cs...	Skipped		
http://www.bigflix.com/apache_file/cs...	Skipped		

Test Details / Results:

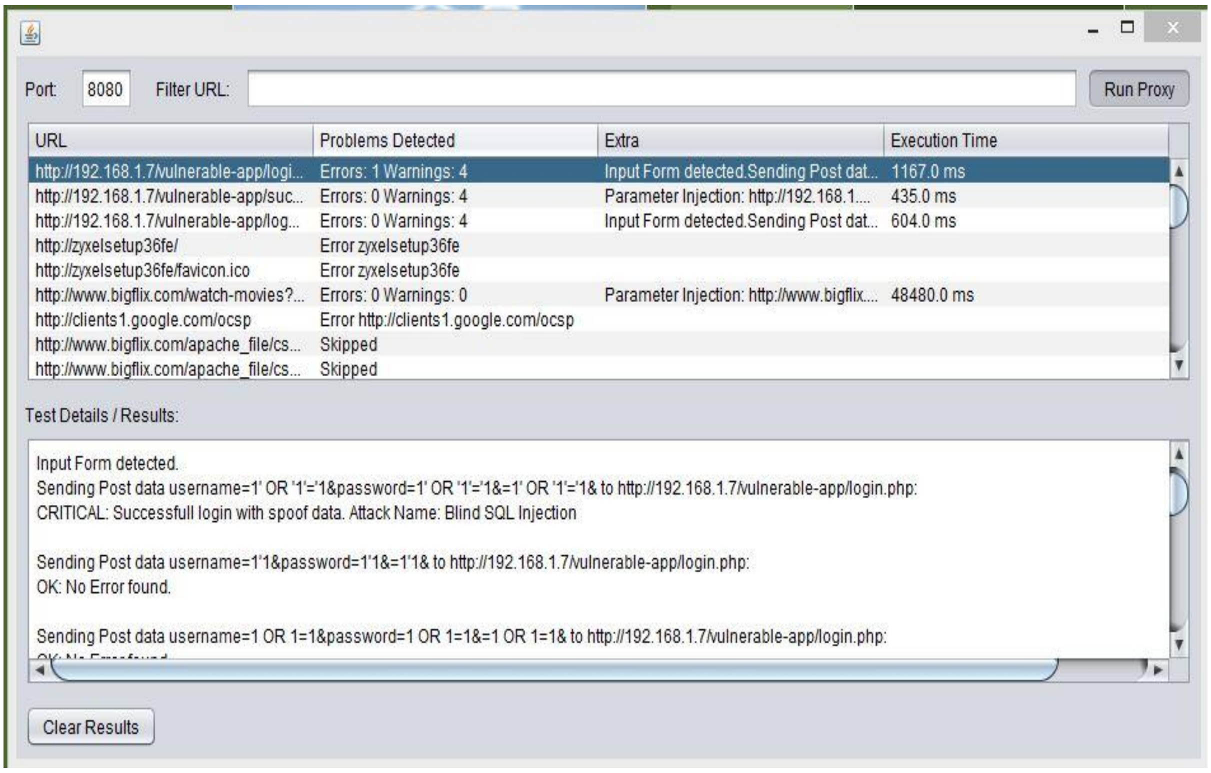
```
Input Form detected.
Sending Post data username=1' OR '1'='1&password=1' OR '1'='1&=1' OR '1'='1& to http://192.168.1.7/vulnerable-app/login.php:
CRITICAL: Successfull login with spoof data. Attack Name: Blind SQL Injection

Sending Post data username=1'1&password=1'1&=1'1& to http://192.168.1.7/vulnerable-app/login.php:
OK: No Error found.

Sending Post data username=1 OR 1=1&password=1 OR 1=1&=1 OR 1=1& to http://192.168.1.7/vulnerable-app/login.php:
```

Figure A.2: Screenshot of Output Screen and Running Web Application

In figure A.3, a screenshot of output screen with detected vulnerabilities is shown. Here on the output screen shows a port number, for now it is 8080. This port number is for Mozilla Firefox, the port number can be changed and can be put for other search engines. This output screen has a “Run Proxy” button, when we press this button; the SQLIVS tool provides us with a web proxy. The result of the tool has four parts:

1) URL: that shows all URLs that are run on the web application.
2) Problems Detected: this tells us about the number of errors or warnings detected, the status of test either running or pending and shows “Skipped” wherever it encounters static data.
3) Extra: this tells us about the information that is retrieved when an error or warning is detected. When we click on the details it takes the link to “Test Details/Results” where full detailed error or warning code is written with the attack type and HTTP response code
4) Execution Time: this shows the total time taken to execute a URL in milliseconds.

We can clear the result part using the “Clear Result” button.



URL	Problems Detected	Extra	Execution Time
http://192.168.1.7/vulnerable-app/logi...	Errors: 1 Warnings: 4	Input Form detected.Sending Post dat...	1167.0 ms
http://192.168.1.7/vulnerable-app/suc...	Errors: 0 Warnings: 4	Parameter Injection: http://192.168.1....	435.0 ms
http://192.168.1.7/vulnerable-app/log...	Errors: 0 Warnings: 4	Input Form detected.Sending Post dat...	604.0 ms
http://zyxelsetup36fe/	Error zyxelsetup36fe		
http://zyxelsetup36fe/favicon.ico	Error zyxelsetup36fe		
http://www.bigflix.com/watch-movies?...	Errors: 0 Warnings: 0	Parameter Injection: http://www.bigflix....	48480.0 ms
http://clients1.google.com/ocsp	Error http://clients1.google.com/ocsp		
http://www.bigflix.com/apache_file/cs...	Skipped		
http://www.bigflix.com/apache_file/cs...	Skipped		

Test Details / Results:

```
Input Form detected.
Sending Post data username=1' OR '1'='1&password=1' OR '1'='1&=1' OR '1'='1& to http://192.168.1.7/vulnerable-app/login.php:
CRITICAL: Successfull login with spoof data. Attack Name: Blind SQL Injection

Sending Post data username=1'1&password=1'1&=1'1& to http://192.168.1.7/vulnerable-app/login.php:
OK: No Error found.

Sending Post data username=1 OR 1=1&password=1 OR 1=1&=1 OR 1=1& to http://192.168.1.7/vulnerable-app/login.php:
```

Figure A.3: Screenshot of Output Screen with Details of Detected Vulnerabilities