

**In-Course Assessment 3**  
**IT3122 Computer Security**

**Group Members :**

M.M.F. Azha – 2021/ICT/48

M.N.S. Nasakath – 2021/ICT/49

M.M.Sajan - 2021/ICT/56

M.I.F.Nusha – 2021/ICT/108

A.S.Salma – 2021/ICT/116

## Introduction

### 1. Cross-Site Scripting

XSS involves an attacker injecting malicious scripts (usually JS) into content that is delivered to other users' browsers. The browser executes this scripts because it trusts the website.

**Impact :** Attackers can steel session cookies, deface websites, redirect users to malicious sites, or steal sensitive information (e.g Credentials) from authenticated users.

**Mechanism :** Occurs when a web application takes untrusted data and includes it in a web page without proper validation or encoding.

### 2. SQL Injection (SQLi)

SQLi is a vulnerability that allows an attacker to interfere with the queries that an application makes to its database.

**Impact :** Attackers can bypass authentication, view, modify, or delete sensitive data in the database, and in some cases, gain administrative control over the database server or the entire system.

**Mechanism :** Occurs when user-supplied data is directly concatenated into a SQL query string, rather than being handled via prepared statements.

### 3. Cross-Site Request Forgery (CSRF)

CSRF is an attack that tricks an authenticated user into unknowingly submitting a malicious request to a web application they are logged into.

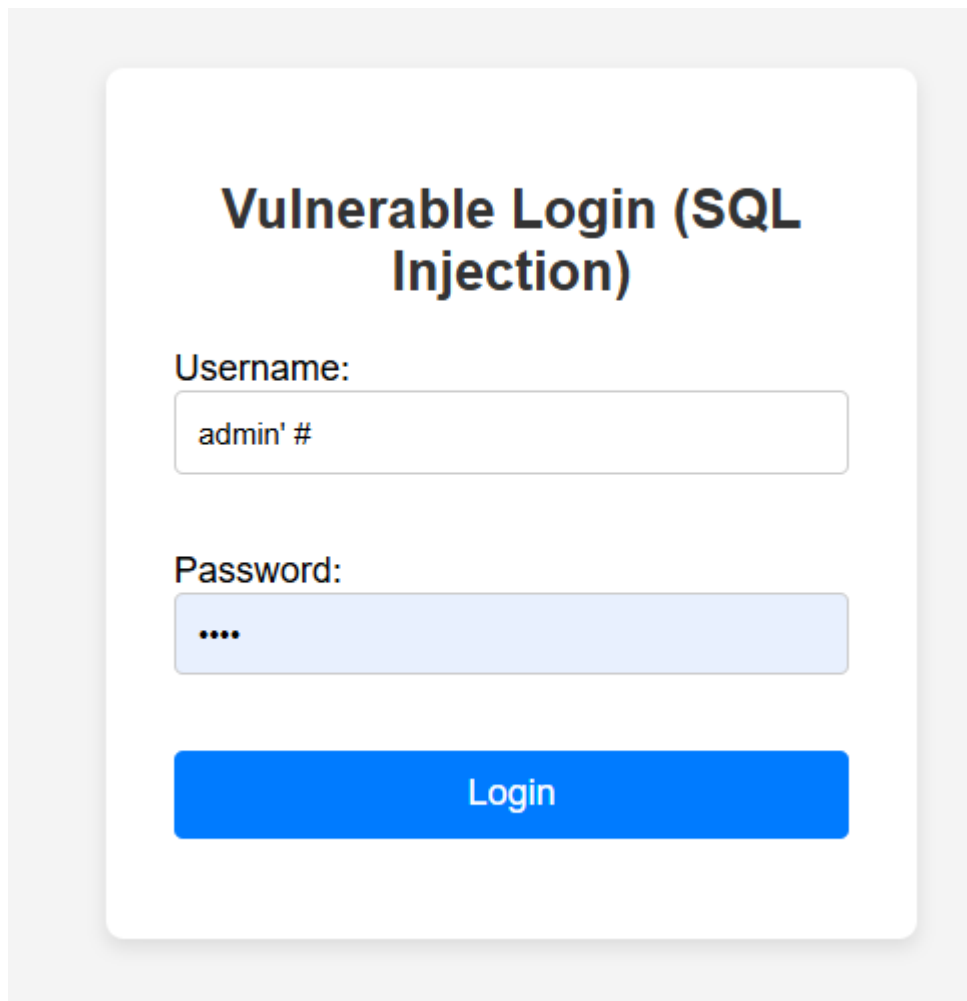
**Impact :** The attack forces the victim's browser to perform actions on the application (e.g: Changing a password, transferring funds, making purchases) as if the victim intended to do them.

**Mechanism :** The application trusts the authenticated user's session cookie and does not verify the origin of the request using anti-CSRF tokens or same-site cookie policies.

For this project implementation and demonstration we have created a database in SQL and used Xampp server.

1. Database created named ica3
2. Created a table named Users, comments.
3. Inserted some data for Users table.

### **Task 1 : SQL Injection**



A login form titled "Vulnerable Login (SQL Injection)". It features a "Username:" label above a text input field containing "admin' #". Below this is a "Password:" label above a password input field with four dots. A blue "Login" button is at the bottom.

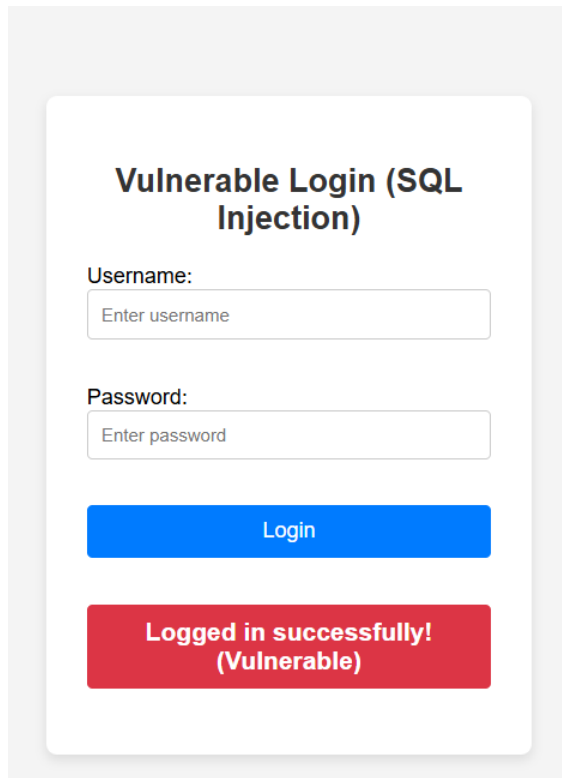
**Vulnerable Login (SQL Injection)**

Username:

Password:

Login

After clicked the login button :



**Vulnerable Login (SQL Injection)**

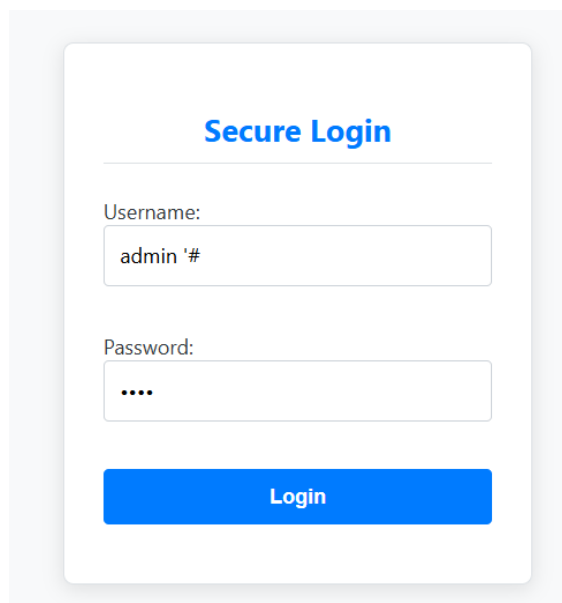
Username:

Password:

**Login**

**Logged in successfully!  
(Vulnerable)**

Secure demo :



**Secure Login**

Username:

Password:

**Login**

The image shows a web form titled "Secure Login" in blue text. Below the title is a horizontal line. There are two input fields: "Username:" with a placeholder "Enter username" and "Password:" with a placeholder "Enter password". Below these is a blue "Login" button. At the bottom, there is a red error message box that says "Login failed.".

### **Mechanism :**

When you use PDO prepared statements in PHP, the mechanism prevents SQL Injection by treating all user input as literal characters, regardless of what they are. This is why characters like single quotes (') and hash symbols (#) are treated as normal characters, not SQL commands.

This protection works through a two-step process:

**Preparation:** The SQL query structure (SELECT \* FROM users WHERE username = :u AND password = :p) is sent to the database first, entirely separate from user data. The database engine compiles this statement, marking the spots (:u and :p) where data will be inserted later. The database establishes the intent of the query at this point.

**Execution and Parameter Binding:** The user's input (\$u and \$p) is sent to the database separately. The database engine inserts this data only into the predefined spots.

Because the user input never alters the structure of the SQL command itself, any special characters within the input are handled strictly as the data they represent, preventing them from being interpreted as executable SQL instructions.

## **Task 2 : Cross-Site Scripting**

### 1. Reflected XSS

#### **A. Reflected XSS**

Enter a name (Try: `<script>alert('Reflected XSS');</script>` )

Name:

Say Hello

## XSS Demonstration

This page demonstrates three types of XSS vulnerabilities.

### A. Reflected XSS

Enter a name (Try: `<script>alert('Reflected XSS');</script>` )

Name:

Say Hello

Vulnerable Output (Neutralized): Input treated as safe plain text. Payload blocked.

Secure Output: Hello `<script>alert('Reflected')</script>`

### Mechanism :

In the context of a vulnerable reflected XSS implementation (unlike the secure code provided above), the browser executes a malicious script because the application fails to sanitize or encode user input before it is rendered on the page.

**Input Submission:** A user submits input (e.g., `<script>alert('XSS')</script>`) via a URL or form.

**Server Reflection (Vulnerable PHP):** The server-side code directly embeds this input into the HTML output without any modification, for example: `echo "<div>Hello " . $name . "</div>";`.

**Browser Interpretation:** When the browser receives the HTML, it finds the `<script>` tags and interprets them as a standard part of the webpage's structure, not just text data.

**Execution:** The browser trusts the source of the HTML (the legitimate website's server) and proceeds to execute the embedded JavaScript code, leading to the XSS attack.

## 2. Stored XSS

### B. Stored XSS

Post a comment (Try: `<img src=x onerror=alert('Stored XSS')>`). The input is saved to the database.

Comment:

`<b>Bold Text</b><script>alert('Stored')</script>`

Post Comment



After clicked the Stored XSS :

### B. Stored XSS

Post a comment (Try: `<img src=x onerror=alert('Stored XSS')>`). The input is saved to the database.

Comment:

Leave a comment

Post Comment

Vulnerable Display (Neutralized): Input treated as safe plain text. Payload blocked.

Secure Display: `<b>Bold Text</b><script>alert('Stored')</script>`

### Mechanism :

In a stored XSS vulnerability scenario (unlike the secure example code you provided which uses `htmlentities()` to prevent this), the malicious script is

injected directly into the application's database and runs every time the page loads for any user who views the comment.

**Injection and Storage:** An attacker submits a malicious payload (e.g., `<script>...</script>`) via a comment form. The vulnerable application takes this raw input and saves it directly into the database without any encoding or sanitization. The script is now stored persistently on the server.

**Persistent Threat:** Every time a user, including administrators or regular visitors, loads the page to view recent comments, the server fetches the malicious script from the database.

**Execution:** Because the application vulnerably outputs the raw content to the browser, the browser interprets and executes the script as legitimate webpage code. This means the attack is automatically triggered for every visitor, making it a widespread and persistent threat.

The secure code you provided effectively blocks this by encoding the output using `htmlentities()`, neutralizing the script before the browser can execute it.

### 3. DOM Based XSS :

#### C. DOM-Based XSS

Check the URL. Add the payload to the URL fragment/hash and refresh:

```
#<img src=x onerror=alert('DOM XSS')>
```

Vulnerable JS (innerHTML Neutralized): Input treated as safe plain text. Payload blocked.

Secure JS (textContent): `<img src=x onerror=alert('DOM')>`

## **Mechanism :**

In a DOM-based XSS attack scenario, the JavaScript running on the page is tricked into handling data from the URL (specifically the `window.location.hash`, or the part of the URL after the `#`) in an unsafe manner.

Here is the mechanism of a vulnerable DOM-based XSS:

**Data Source:** The JavaScript code takes data directly from an untrusted source, such as `window.location.hash`.

**Insecure Sink:** The script then writes this data into the HTML using an insecure sink method, typically `.innerHTML`.

**Execution:** The browser sees that the trusted JavaScript is using `.innerHTML` to write content. It interprets the input as actual HTML structure and executes any embedded scripts within it.

The secure code you provided earlier avoided this vulnerability by using `.textContent` instead of `.innerHTML`. This forces the browser to treat the input as simple text rather than executable HTML, preventing the malicious script from running.

## **Solution :**

In the provided secure code examples that use PHP to prevent XSS, the function `htmlspecialchars()` is the key defense mechanism.

`htmlspecialchars()` works by converting characters that have special meaning in HTML into their safe, non-executable HTML entity representations.

For example, when a user submits a payload like `<script>alert('XSS')</script>`, the `htmlspecialchars()` function converts it into:

`<` becomes `&lt;`;

`>` becomes `&gt;`;

`'` becomes `&apos;` or `&#039;`; (depending on flags used)

`"` becomes `&quot;`; (depending on flags used)

The resulting output in the HTML source code looks like this:

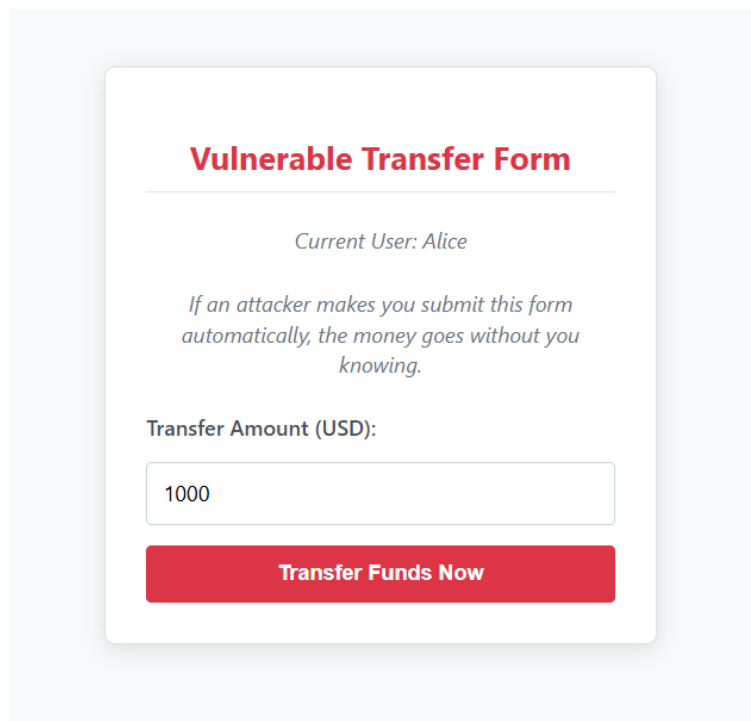
html

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

Use code with caution.

When the browser reads this, it simply displays the text exactly as it appears (e.g., `<script>alert('XSS')</script>`), treating it as harmless text on the page rather than executable code. This effectively neutralizes the attack.

### Task 3 : CSRF



**Vulnerable Transfer Form**

---

*Current User: Alice*

*If an attacker makes you submit this form automatically, the money goes without you knowing.*

Transfer Amount (USD):

**Transfer Funds Now**

After clicked the transfer funds now button :

Money Transferred! (Vulnerable - No Token Check)

## Vulnerable Transfer Form

---

*Current User: Alice*

*If an attacker makes you submit this form automatically, the money goes without you knowing.*

Transfer Amount (USD):

1000

Transfer Funds Now

## Solution Demo :

### Secure Transfer Form

Current User: **Alice**

This form is protected by a CSRF token.

Transfer Amount (USD):

**Transfer Funds Securely**

**Current CSRF Token:** 005bf52b7c616e89e0734007fe84b3041a066fe9562c166430752351c62ab630

After clicked the button transfer funds securely

**Money Transferred! (Secure - Token Valid)**

### Secure Transfer Form

Current User: **Alice**

This form is protected by a CSRF token.

Transfer Amount (USD):

**Transfer Funds Securely**

**Current CSRF Token:** 005bf52b7c616e89e0734007fe84b3041a066fe9562c166430752351c62ab630

In the provided secure CSRF (Cross-Site Request Forgery) demonstration code, the fund transfer successfully proceeded only because a security check confirmed the

**Token Generation:** When the secure form page is initially loaded, the server generates a unique, unpredictable anti-CSRF token and stores it in the user's secure server-side session. This same token is also embedded as a hidden field within the HTML form itself.

**Submission:** When the user legitimately clicks the submit button, both the transfer details and the hidden token are sent to the server via a POST request.

**Verification:** The server-side code immediately compares the received csrf\_token value from the \$\_POST data with the expected token stored in the \$\_SESSION.

The transfer is executed only if these two tokens match exactly.

An attacker's malicious link or hidden form on a different website cannot forge this unique, session-specific token. The attacker doesn't know the token value, so their forged request will fail the hash\_equals() check, and the transfer will be blocked.