**Élaboration de démonstrateurs d’attaques Web : OWASP Top 10 Vulnerabilities (2021)**

**Development of Web Attack Demonstrators : OWASP Top 10 Vulnerabilities (2021)**

|  |  |
| --- | --- |
| **Project supervisors:** | **Project team** |
| **Christophe Kiennert, Gregory Blanc** | **Antoine Pennamen, Salma El Bougrini** |

|  |  |
| --- | --- |
| **Mars 2025** | **Telecom Sud-Paris**  **MS-Cybersecurity** |

# Project abstract

A white and black text on a white background

Description automatically generated

(typically 150-250 words)

## summary and objectives of the project

Every year, web servers experience attacks with various consequences: data theft, server downtime, identity theft, etc., leading to financial losses and damage to the company's reputation. The OWASP Foundation regularly publishes the Top Ten, which is a ranking of the 10 most critical vulnerabilities for web applications.

This project aims to study the vulnerabilities highlighted in the latest Top Ten (from 2021) and to implement a number of demonstrators to illustrate the exploitation of certain vulnerabilities through attacks on real-world web applications.





Two deliverables are expected:

— A set of demonstrators

— A set of explanatory documents

## Methodology

The key steps are:

* understand the attack concept (threats and vulnerabilities)
* propose an implementation (recent environment, remote operation
* propose a fix for the vulnerability to render the attack ineffective

## key findings

**Table of contents**

[1 Project abstract 2](#_Toc181358834)

[1.1 summary and objectives of the project 2](#_Toc181358835)

[1.2 Methodology 3](#_Toc181358836)

[1.3 key findings 3](#_Toc181358837)

[2 Introduction 8](#_Toc181358838)

[3 Literature Review 8](#_Toc181358839)

[4 Methodology 8](#_Toc181358840)

[5 Mitigation 9](#_Toc181358841)

[5.1 Use Framework Security Features 9](#_Toc181358842)

[5.2 Regular Security Audits and Testing 9](#_Toc181358843)

[6 Vulnerabilities study 10](#_Toc181358844)

[6.1 Broken Access Control 10](#_Toc181358845)

[6.1.1 Description 10](#_Toc181358846)

[6.1.2 Occurrence 10](#_Toc181358847)

[6.1.3 Risk Measure 10](#_Toc181358848)

[6.1.4 Implementation example 10](#_Toc181358849)

[6.1.4.1 Violation of the principle of least privilege or deny by default: 10](#_Toc181358850)

[6.1.4.2 Unauthorized access enumeration 10](#_Toc181358851)

[6.1.4.3 IDOR (Indirect Direct Object Reference) 10](#_Toc181358852)

[6.2 Cryptographic Failures 11](#_Toc181358853)

[6.2.1 Description 11](#_Toc181358854)

[6.2.2 Occurrence 11](#_Toc181358855)

[6.2.3 Risk Measure 11](#_Toc181358856)

[6.2.4 Implementation example 11](#_Toc181358857)

[6.2.4.1 TLS Encryption: 11](#_Toc181358858)

[6.2.4.2 Deprecated algorithm (md5 instead of sha256 or bcrypt): 11](#_Toc181358859)

[6.2.4.3 Public Key Infrastructure: 11](#_Toc181358860)

[6.3 Injection 12](#_Toc181358861)

[6.3.1 Description 12](#_Toc181358862)

[6.3.2 Occurrence 12](#_Toc181358863)

[6.3.3 Risk Measure 12](#_Toc181358864)

[6.3.4 Implementation example 12](#_Toc181358865)

[6.3.4.1 SQLi : 12](#_Toc181358866)

[6.3.4.1.1 Basic SQL Injection 12](#_Toc181358867)

[6.3.4.1.2 Blind SQL Injection 13](#_Toc181358868)

[6.3.4.1.3 Union-Based SQL Injection 13](#_Toc181358869)

[6.3.4.1.4 Error Based SQL Injection 13](#_Toc181358870)

[6.3.4.1.5 Example of implementation of SQLi: 14](#_Toc181358871)

[6.3.4.1.6 SQLi Mitigation strategies: 19](#_Toc181358872)

[6.3.4.2 XSS : 20](#_Toc181358873)

[6.3.4.2.1 Reflected XSS (Non-Persistent XSS) 20](#_Toc181358874)

[6.3.4.2.2 Stored XSS (Persistent XSS) 20](#_Toc181358875)

[6.3.4.2.3 DOM-based XSS 20](#_Toc181358876)

[6.3.4.2.4 Example of implementation of reflected XSS: 21](#_Toc181358877)

[6.3.4.2.5 Example of implementation of stored XSS: 26](#_Toc181358878)

[Scenario: A Blog Commenting System 26](#_Toc181358879)

[Step 1: Create a Simple Comment Submission Form 26](#_Toc181358880)

[Step 2: PHP Code to Handle Comment Submission (submit\_comment.php) 27](#_Toc181358881)

[Step 3: Display Comments (view\_comments.php) 27](#_Toc181358882)

[Step 4: Inject Malicious Code 27](#_Toc181358883)

[6.3.4.2.6 XSS Mitigation Strategies 28](#_Toc181358884)

[6.4 Insecure Design 31](#_Toc181358885)

[6.4.1 Description 31](#_Toc181358886)

[6.4.2 Occurrence 31](#_Toc181358887)

[6.4.3 Risk Measure 31](#_Toc181358888)

[6.4.4 Implementation example 31](#_Toc181358889)

[6.4.4.1 Data Exposure: 31](#_Toc181358890)

[6.4.4.2 Insufficient Authentication Requirements: 31](#_Toc181358891)

[6.4.4.3 Architecture review: 31](#_Toc181358892)

[6.5 Security Misconfiguration 32](#_Toc181358893)

[6.5.1 Description 32](#_Toc181358894)

[6.5.2 Occurrence 32](#_Toc181358895)

[6.5.3 Risk Measure 32](#_Toc181358896)

[6.5.4 Implementation example 32](#_Toc181358897)

[6.5.4.1 Unnecessary enabled features: 32](#_Toc181358898)

[6.5.4.2 Insufficient application / server configuration reviews: 32](#_Toc181358899)

[6.5.4.3 Default accounts still enabled: 32](#_Toc181358900)

[6.6 Vulnerable and Outdated Components 33](#_Toc181358901)

[6.6.1 Description 33](#_Toc181358902)

[6.6.2 Occurrence 33](#_Toc181358903)

[6.6.3 Risk Measure 33](#_Toc181358904)

[6.6.4 Implementation example 33](#_Toc181358905)

[6.6.4.1 Outdated components and dependencies: 33](#_Toc181358906)

[6.6.4.2 Dependency tree: 33](#_Toc181358907)

[6.6.4.3 Code reviews: 33](#_Toc181358908)

[6.7 Identification and Authentication Failures 34](#_Toc181358909)

[6.7.1 Description 34](#_Toc181358910)

[6.7.2 Occurrence 34](#_Toc181358911)

[6.7.3 Risk Measure 34](#_Toc181358912)

[6.7.4 Implementation example 34](#_Toc181358913)

[6.7.4.1 Automated attacks on passwords (Bruteforce): 34](#_Toc181358914)

[6.7.4.2 Weak Session Management Logic: 34](#_Toc181358915)

[6.7.4.3 Code reviews: 34](#_Toc181358916)

[6.8 Software and Data Integrity Failures 35](#_Toc181358917)

[6.8.1 Description 35](#_Toc181358918)

[6.8.2 Occurrence 35](#_Toc181358919)

[6.8.3 Risk Measure 35](#_Toc181358920)

[6.8.4 Implementation example 35](#_Toc181358921)

[6.8.4.1 Data Integrity: 35](#_Toc181358922)

[6.8.4.2 Untrusted Dependencies: 35](#_Toc181358923)

[6.8.4.3 Software Integrity: 35](#_Toc181358924)

[6.10 Security Logging and Monitoring Failures 36](#_Toc181358925)

[6.10.1 Description 36](#_Toc181358926)

[6.10.2 Occurrence 36](#_Toc181358927)

[6.10.3 Risk Measure 36](#_Toc181358928)

[6.10.4 Implementation example 36](#_Toc181358929)

[6.10.4.1 Security Logging Policy: 36](#_Toc181358930)

[6.10.4.2 Misconfigured Alerts: 36](#_Toc181358931)

[6.10.4.3 Quick Alert / Response: 36](#_Toc181358932)

[6.11 Server-Side Request Forgery (SSRF) 37](#_Toc181358933)

[6.11.1 Description 37](#_Toc181358934)

[6.11.2 Occurrence 37](#_Toc181358935)

[6.11.3 Risk Measure 37](#_Toc181358936)

[6.11.4 Implementation example 37](#_Toc181358937)

[6.11.4.1 Illegitimate access to internal resources: 37](#_Toc181358938)

[6.11.4.2 Network Segmentation: 37](#_Toc181358939)

[6.11.4.3 Denial of service: 37](#_Toc181358940)

[7 Discussion 38](#_Toc181358941)

[8 Conclusion 38](#_Toc181358942)

[9 References 38](#_Toc181358943)

[10 Appendices 40](#_Toc181358944)

[11 Acknowledgments 40](#_Toc181358945)

[12 Executive Summary 40](#_Toc181358946)

# Introduction

* Background information on the topic.
* Purpose and objectives of the project.
* Scope of the report.
* Significance of the study.

# Literature Review

* Summary of existing research and literature related to the project.
* Identification of gaps in knowledge that your project addresses.

# Methodology

* understand the the attack concept (threats and vulnerabilities) through documentation:
  + OWASP 2021 : Website: [owasp.org](https://owasp.org)
  + PortSwigger. (2021). Burp Suite Documentation. Available at: https://portswigger.net/burp/documentation
* propose an implementation (recent environment, remote operation
  + **Use a Controlled Environment** :Set up a local environment and/or use a virtual machine to host 2 websites:
    - the vulnerable web application (ensure it’s not exposing real users or systems to risk)
    - The corrected version mitigating the vulnerabilities
  + **Choose and install deveplopement tools :**
    - HTML, PHP, Javascript, MySQL, Apache
    - Github for versionning and collaboration
  + **Development of the applications components**
    - First the vulnerable web application
    - Second, propose a fix for each studied vulnerability to render the attacks ineffective
  + **Tests:**

Use tools to help identify vulnerabilities and mitigate them:

* + - **Burp Suite**: A powerful web vulnerability scanner.
    - **OWASP ZAP**: An open-source web application security scanner.
* Document:

Keep a record of the studied vulnerabilities, how to exploit them, and potential remediation steps.

# Mitigation

Describe general methods and guidance for mitigating vulnerabilities.

Specific and detailed ones are described in the vulnerabilities study.

## Use Framework Security Features

* **Purpose**: Leverage built-in security features provided by web frameworks.
* **Implementation**:
  + Many modern frameworks (e.g., React, Angular, Vue) automatically escape output or provide security features to help mitigate XSS risks.
  + Review your framework’s documentation for recommended practices.

## Regular Security Audits and Testing

* **Purpose**: Identify and fix vulnerabilities before they can be exploited.
* **Implementation**:
  + Conduct regular security audits of your codebase to find potential XSS vulnerabilities.
  + Use automated security testing tools (like OWASP ZAP or Burp Suite) to scan for vulnerabilities.
  + Implement manual code reviews focusing on input handling and output rendering.

# Vulnerabilities study

## Broken Access Control

### Description

Broken access control occurs when users can act outside their intended permissions, allowing unauthorized actions.

### Occurrence

Commonly found in web applications where user roles are poorly defined or enforced.

### Risk Measure

Risk Level: High. Exploitation can lead to data breaches, unauthorized data manipulation, and complete system compromise.

### Implementation example

#### Violation of the principle of least privilege or deny by default:

* Attempt to access a restricted resource (e.g., admin dashboard) by manipulating the URL (e.g., /admin).
* **Mitigation**: Implement strict role-based access controls and server-side validation of permissions. Apply Least Privilege Principle.

#### Unauthorized access enumeration

* + Use a testing tool to enumerate endpoints and check for unauthorized access (e.g., Burp Suite).
  + **Mitigation**: Use security headers (X-Content-Type) and enforce access control checks at all endpoints.

#### IDOR (Indirect Direct Object Reference)

* + This can occur when a web application or application programming interface uses an identifier for direct access to an object in an internal database but does not check for access control or authentication (<http://foo.com/doc/1234>)
  + **Mitigation**:
    - **Replace Direct IDs with Indirect Identifiers**: Use UUIDs, hash-based identifiers, or tokens rather than easily guessable numeric IDs.
    - **Access Tokens**: Generate temporary tokens for sensitive data requests, which are only valid for a specific user and action.

## Cryptographic Failures

### Description

Cryptographic failures involve insecure cryptographic practices, leading to data exposure or manipulation.

### Occurrence

Occurs when sensitive data is improperly encrypted or when weak algorithms are used.

### Risk Measure

Risk Level: High. Can result in sensitive data exposure, leading to identity theft or fraud.

### Implementation example

#### TLS Encryption:

* + Inspect traffic with tools like Wireshark to find unencrypted sensitive data (e.g., passwords).
  + **Mitigation**: Enforce encryption (e.g., TLS) for data in transit and at rest.

#### Deprecated algorithm (md5 instead of sha256 or bcrypt):

* + Analyze code for hard-coded keys or weak encryption algorithms.
  + **Mitigation**: Use strong encryption standards (e.g., AES) and secure key management practices.

#### Public Key Infrastructure:

* + Is the received server certificate and the trust chain properly validated?
  + **Mitigation**: Check PKI chain validation

## Injection

### Description

Injection vulnerabilities occur when untrusted data is sent to an interpreter, leading to execution of unintended commands.

### Occurrence

An application is vulnerable to injection attacks when:

* User input is not validated, filtered, or sanitized by the application.
* Dynamic queries or non-parameterized calls without context-aware escaping are used directly in the interpreter:
  + Query: SELECT \* FROM users WHERE username = '" + user\_input + "';.
  + Injection: SELECT \* FROM users WHERE username = 'anything' OR '1'='1';
* Hostile data is directly used or concatenated in SQL queries or system commands.
  + Exemple command:

<?php

$domain = $\_GET['domain'];

system("ping -c 4 " . $domain);

?>

* + Exemple Injection:

example.com; ls

### Risk Measure

Risk Level: High. Can lead to data loss, data corruption, or unauthorized access to systems.

### Implementation example

#### SQLi :

SQL injection (SQLi) is a security vulnerability that allows attackers to inject malicious sql quotes into fields processed in the database. Here are four example types of SQLi:

##### Basic SQL Injection

* **Description**: Malicious user would inject simple code to alter database queries directly
* **Impact**: Bypassing WHERE clause to skip a signup procedure.
* **Example**: A connection (login/password) field where a malicious user would inject code like “ ‘ OR ‘1’ = ‘1 “ modifying the query to be always true.

##### Blind SQL Injection

* **Description**: Used when there is no error message. An attacker would use SQL queries like Time based SQLi (SLEEP) or conditional statements (TRUE/FALSE) to check the response from the application.
* **Impact**: Dumping of the entire database, data leaks, unauthorized access
* **Example :** http://example.com/login.php?username=admin' OR IF(1=1, SLEEP(5), 0) –
  + **admin'**: Ends the string for the username.
  + **OR IF(1=1, SLEEP(5), 0)**: This condition will always be true (1=1). If true, the server waits for 5 seconds due to the SLEEP(5) function. If false, it does nothing (0).
  + **--**: This comments out the rest of the SQL query, effectively ignoring the password check.

##### Union-Based SQL Injection

* **Description**: An attacker would use the UNION operator to combine the original SELECT statement with a new SELECT on another table.
* **Impact**: Extracting data from other tables.
* **Example**: http://example.com/search.php?product=shoes' UNION SELECT username, password FROM users --
  + **shoes'**: The attacker terminates the string to inject their SQL.
  + **UNION SELECT**: This combines the results of the original query with another SELECT query.
  + **username, password**: The attacker specifies the columns they want to retrieve from the users table.
  + **FROM users**: This indicates the source of the data for the second query.
  + **--**: This comments out the rest of the SQL query, ignoring any additional clauses.
  + **SQL Query:** SELECT \* FROM products WHERE name = 'shoes' UNION SELECT username, password FROM users --';

##### Error Based SQL Injection

* **Description**: An attacker would exploit error messages to gather database information by using incorrect syntax or invalid column names.
* **Impact**: Reveal database structure in error messages.
* **Example**: in the username field on the login page ' OR uname LIKE '%
  + If the application does not properly handle the SQL syntax and returns an error, an attacker would exploit this to gather information from the database.
  + If the injection is successful we will receive an error message like this: **Fatal error**: Uncaught mysqli\_sql\_exception: Unknown column 'uname' in 'where clause' in /Users/antoinepennamen/PFE/vulnerable\_website/login.php:27

##### Example of implementation of SQLi:

* Strong script

Une image contenant texte, capture d’écran

Description générée automatiquement

* + check that the form has been sent with the POST method
  + check if the username and the password fields are not empty with the isset php function (Boolean, true if not null else false)
  + use the php prepare() function to create a prepared statement ($stmt) and send it to the database to use it later.
  + add a security layer with bind\_param() to link $username to a string, which will prevent any SQL injection. It will be bind to the $stmt as a parameter
  + verify if the user exists.
  + check if the password is correct
  + handle error message
* Test

Une image contenant texte, capture d’écran, Police, nombre

Description générée automatiquement

* Result

Une image contenant texte, capture d’écran, Police

Description générée automatiquement

* Vulnerable website script

Une image contenant texte, capture d’écran

Description générée automatiquement

There is no prepared statement, instead we directly process the query, which is vulnerable to SQL injection,

Exemple : if username = anything’ OR 1=’1’ – the query becomes:

SELECT \* FROM users WHERE username = 'anything' OR 1='1' -- AND password = 'a'

The attacker can login in as the first user of the users table.

* Test

Une image contenant texte, capture d’écran, Police, nombre

Description générée automatiquement

* Result

Une image contenant texte, capture d’écran, Police, carte de visite

Description générée automatiquement

##### SQLi Mitigation strategies:

From OWASP best practices:

**Use Prepared Statements (Parameterized Queries)**

Prepared statements separate SQL code from data, ensuring that user input is treated as data only. This significantly reduces the risk of SQL injection.

**Stored Procedures**

Stored procedures are executed on the database server and can encapsulate SQL logic, reducing the risk of injection. Ensure that the stored procedure is defined safely and does not concatenate user inputs into dynamic SQL statements.

**Input Validation and Sanitization**

Validate and sanitize all user inputs. This can include:

* Checking for expected data types (e.g., integers, strings).
* Using whitelists to allow only valid characters.
* Removing or escaping harmful characters.

**Least Privilege Principle**

Configure your database accounts with the least privileges. For example, if a user only needs to read data, do not grant it permissions to modify or delete data.

**Web Application Firewalls (WAF)**

A WAF can help detect and block SQL injection attempts by analyzing incoming traffic and filtering out malicious requests.

**Error Handling**

Avoid displaying detailed error messages that reveal database information. Use generic error messages for users and log the detailed errors for developers to review.

#### XSS :

Cross-Site Scripting (XSS) is a security vulnerability that allows attackers to inject malicious scripts into web pages viewed by other users. potentially enabling attackers to:

* Steal sensitive information, such as cookies, session tokens, or login credentials.
* Impersonate users or hijack user sessions.
* Redirect users to malicious sites.
* Deface web pages or change their behavior

There are 3 main types of XSS:

##### Reflected XSS (Non-Persistent XSS)

* **Description**: The malicious script is not stored on the server but is reflected off a web server via a URL or HTTP request.
* **Impact**: The attack occurs when a user clicks a specially crafted link that includes the malicious script, which is then executed in their browser.
* **Example**: An attacker sends a victim a link that includes a script in a query parameter. When the victim clicks the link, the script runs.

##### Stored XSS (Persistent XSS)

* **Description**: The malicious script is permanently stored on the target server (e.g., in a database, message forum, or comment field).
* **Impact**: When a user accesses the affected page, the script runs in their browser, potentially stealing cookies, session tokens, or other sensitive information.
* **Example**: An attacker posts a malicious comment on a blog. When other users view the comment, the script executes.

##### DOM-based XSS

* **Description**: The vulnerability exists in the client-side code (JavaScript) rather than on the server. The page's JavaScript modifies the DOM (Document Object Model : programming interface for web documentslike an HTML or XML file), allowing programming languages to manipulate the content, structure, and style of web pages dynamicallyand executes the injected script.
* **Impact**: This type can occur when user input is used to manipulate the page without proper validation or sanitization.
* **Example**: A web page uses document.location to read a URL parameter and directly inserts it into the page, allowing an attacker to execute a script.

##### Example of implementation of reflected XSS:

In the website, go to the “contact” form, where the user can fill a form with his name, email and a message:

* Secured script v1

Une image contenant texte, capture d’écran, logiciel, Logiciel multimédia

Description générée automatiquement

We sanitize user input with the sanitize\_data function, which replaces the <script> attribute by its equivalent HTML entity. This will prevent the user to inject malicious javascript inside <script> tags.

* Test

Une image contenant texte, capture d’écran, Police, nombre

Description générée automatiquement

* Result

Une image contenant texte, capture d’écran, conception

Description générée automatiquement

As the user didn’t use <script> tags, he was able to bypass the sanitization function by injecting javascript code inside an HTML attribute (onerror) which will be executed because the image he tried to load doesn’t exist.

* Strong script

Une image contenant texte, capture d’écran, logiciel, Logiciel multimédia

Description générée automatiquement

We use the php function htmlspecialchars which converts special characters like (< , < , & , “ , ‘) into HTML entities, blocking the attacker possibilities to inject any malicious code into the fields.

* Test

A screenshot of a computer

Description automatically generated

* Result

A screenshot of a computer

Description automatically generated

* Vulnerable script

Une image contenant texte, capture d’écran, logiciel, Logiciel multimédia

Description générée automatiquement

* Test

Une image contenant texte, capture d’écran, Police, nombre

Description générée automatiquement

* Result

Une image contenant texte, capture d’écran, Police

Description générée automatiquement

Une image contenant texte, capture d’écran, Police

Description générée automatiquement

##### Example of implementation of stored XSS:

* Stored XSS (Cross-Site Scripting) occurs when an attacker injects malicious code that is stored on the server (such as in a database) and served to other users who access that data.This type of vulnerability can be particularly harmful because:
* The malicious code can affect all users who view the page.
* The code remains active on the page until the database entry is removed.
* It can lead to data theft, session hijacking, and unauthorized actions.
* **Mitigation**: To prevent this vulnerability, always sanitize and encode user inputs before displaying them:

php

Copy code

echo "<p><strong>" . htmlspecialchars($row['username']) . ":</strong> " . htmlspecialchars($row['comment']) . "</p>";

Using functions like htmlspecialchars() ensures that special characters are rendered as text, not HTML or JavaScript, preventing the execution of malicious code.

### Scenario: A Blog Commenting System

In this example, a vulnerable commenting system does not sanitize or encode user input before displaying it. This allows an attacker to inject JavaScript code into their comment, which is then stored in the database and displayed to every user viewing that comment.

#### Step 1: Create a Simple Comment Submission Form

html

<!DOCTYPE html>

<html>

<head>

<title>Comment Section</title>

</head>

<body>

<h2>Leave a Comment</h2>

<form action="submit\_comment.php" method="post">

<label for="username">Username:</label>

<input type="text" name="username" id="username" required><br><br>

<label for="comment">Comment:</label><br>

<textarea name="comment" id="comment" rows="4" cols="50" required></textarea><br><br>

<input type="submit" value="Submit">

</form>

<h2>Comments</h2>

<div id="comments">

<!-- Comments will be displayed here -->

</div>

</body>

</html>

#### Step 2: PHP Code to Handle Comment Submission (submit\_comment.php)

PHP script to handle the form submission and store the comment in a database without sanitizing the input (which creates the vulnerability).

php

<?php

// Database connection

$mysqli = new mysqli("localhost", "user", "password", "database");

// Get the form data

$username = $\_POST['username'];

$comment = $\_POST['comment'];

// Insert comment into the database

$sql = "INSERT INTO comments (username, comment) VALUES ('$username', '$comment')";

$mysqli->query($sql);

// Redirect back to the comments page

header("Location: view\_comments.php");

?>

#### Step 3: Display Comments (view\_comments.php)

The comments are displayed on this page without escaping or sanitizing, which allows JavaScript injection.

php

Copy code

<?php

// Database connection

$mysqli = new mysqli("localhost", "user", "password", "database");

// Fetch comments from the database

$result = $mysqli->query("SELECT username, comment FROM comments");

// Display each comment

while ($row = $result->fetch\_assoc()) {

echo "<p><strong>" . $row['username'] . ":</strong> " . $row['comment'] . "</p>";

}

?>

#### Step 4: Inject Malicious Code

An attacker could submit a comment like this:

html

<script>alert('Stored XSS Attack!');</script>

When a user views the comments on the view\_comments.php page, this script will execute, displaying an alert box.

Try to do a real-world attack, the script could be more malicious, such as stealing cookies, logging keystrokes, or redirecting users to a malicious website.

##### XSS Mitigation Strategies

To prevent XSS attacks:

* **Input Validation and sanitization**: Validate and sanitize user inputs.

**Purpose**:

* + Ensure that user inputs conform to expected formats before processing them.
  + Remove or neutralize potentially harmful content from user inputs.

**Implementation**:

* + Use whitelisting to define acceptable input formats (e.g., only allow certain characters, lengths, or types).
  + Reject any input that does not meet these criteria.
  + Validate input on both the client and server sides.
  + Use libraries specifically designed for sanitization, such as DOMPurify for JavaScript or htmlspecialchars, to clean user inputs before rendering them.
  + Avoid using functions that execute code directly from user input, like eval (Evaluate a string as PHP code) or innerHTML, without proper sanitization.
* **Output Encoding**: Encode data before rendering it in the browser.

**Purpose**:

* + Encode data before rendering it in the browser, so any injected scripts are treated as data rather than executable code.

**Implementation**:

* + Use context-specific encoding:
* **HTML Encoding**: Convert characters like <, >, and & to their HTML entities (&lt;, &gt;, &amp;).
* **Attribute Encoding**: Encode data when inserting it into HTML attributes (e.g. using quotes).
* **JavaScript Encoding**: Encode data when it’s included in JavaScript contexts (e.g. backslashes (\), and newlines (\n)).
  + Libraries like OWASP’s Java Encoder or other language-specific libraries can help automate this process.
* **Content Security Policy (CSP) and X-XSS-Protection:** Implement a CSP to restrict sources of executable scripts in .htaccess if Apache server, otherwise in all php files. X-XSS-Protection header instructs the browser to activate its built-in XSS protection features.

**Purpose**:

* + CSP : Define which sources of content are allowed to be loaded by the browser, reducing the risk of XSS attacks.
  + X-XSS-Protection can help block some reflected XSS attacks by detecting potential vulnerabilities.

**Implementation**:

* + Set the Content-Security-Policy HTTP header to specify allowed sources for scripts, styles, images, and other resources. For example:

Content-Security-Policy: default-src 'self'; script-src 'self' <https://trusted.cdn.com> (to be replaced by trusted sites like Cloudflare, Amazon CloudFront, Google CDN, jsDelivr,CDNJS)

* + Regularly review and update your CSP to ensure it reflects your application's needs.
  + The X-XSS-Protection header can be added in your web server configuration or in your application code, depending on your setup. Here’s how to add it in Apache environment: In your .htaccess file or in the server configuration file, to activate the browser’s built-in XSS filter,and block the page from rendering if an XSS attack is detected:

Header set X-XSS-Protection "1; mode=block"

* **HttpOnly and Secure Cookies**: Use these flags to protect cookies from being accessed via JavaScript.

**Purpose**:

* + Protect cookies from being accessed via JavaScript, especially session cookies.

**Implementation**:

* + Use the HttpOnly flag on cookies to prevent JavaScript access. This means cookies cannot be read through document.cookie.
  + Use the Secure flag to ensure cookies are only sent over HTTPS connections, protecting them from interception in transit.
  + Example of setting a cookie with these flags in HTTP response headers:

Set-Cookie: sessionId=abc123; HttpOnly; Secure;

## Insecure Design

### Description

Insecure design refers to flaws in application design that fail to consider security risks.

### Occurrence

Often seen in applications lacking a security-focused design phase.

### Risk Measure

Risk Level: Medium to High. Can lead to multiple security vulnerabilities if not addressed during the design phase.

### Implementation example

#### Data Exposure:

* + Review application features for unnecessary data exposure (e.g., error messages revealing stack traces).
  + **Mitigation**: Adopt secure design principles and perform threat modeling.

#### Insufficient Authentication Requirements:

* + Only requiring a weak password for highly sensitive actions or accounts without implementing multi-factor authentication (MFA).
  + **Mitigation**: use password managers, strong password policy or MFA.

#### Architecture review:

* + Perform a comprehensive architecture review to identify inherent security flaws.
  + **Mitigation**: Utilize frameworks, unit tests and best practices for secure design.

## Security Misconfiguration

### Description

Security misconfiguration occurs when default settings are not changed or when security controls are misconfigured.

### Occurrence

Common in cloud services, web servers, and application settings.

### Risk Measure

Risk Level: Medium. May lead to unauthorized access and data exposure.

### Implementation example

#### Unnecessary enabled features:

* + Use security scanning tools (e.g., Nmap) to check for unnecessary enabled features (e.g., ports, webpages, webservices).
  + **Mitigation**: Regularly review configurations and change default settings.

#### Insufficient application / server configuration reviews:

* + Conduct manual reviews of application settings and server configurations.
  + **Mitigation**: Implement automated configuration management tools.

#### Default accounts still enabled:

* + Default accounts and their passwords are still enabled and unchanged.
  + **Mitigation**: Disable default accounts or change passwords.

## Vulnerable and Outdated Components

### Description

Using outdated libraries or components that have known vulnerabilities can expose applications to attacks.

### Occurrence

Common in projects that rely on third-party libraries or frameworks.

### Risk Measure

Risk Level: Medium to High. Can lead to exploitation if vulnerabilities in components are publicly known.

### Implementation example

#### Outdated components and dependencies:

* + Scan the application with tools (Nmap, Snyk, Shodan) to find outdated dependencies / components.
  + **Mitigation**: Continuously inventory the versions of both client-side and server-side components (e.g., frameworks, libraries) and their dependencies using tools defined above.

#### Dependency tree:

* + Review the dependency tree using tools like Roave Security Advisories, npm-audit or composer which will prevent the installation of dependencies with known vulnerabilities.
  + **Mitigation**: Establish a regular update schedule for dependencies.

#### Code reviews:

* + Conduct a manual code review to identify indirect dependencies and their vulnerabilities.
  + **Mitigation**: Adopt a policy for evaluating and vetting third-party components.

## Identification and Authentication Failures

### Description

Failures in user authentication and session management can allow attackers to gain unauthorized access.

### Occurrence

Often found in applications with weak password policies or improper session handling.

### Risk Measure

Risk Level: High. Can lead to account takeovers and unauthorized actions.

### Implementation example

#### Automated attacks on passwords (Bruteforce):

* + Test for weak passwords or lack of account lockout mechanisms.
  + **Mitigation**: Implement strong password policies, multi-factor authentication (MFA) and account lockout or CAPTCHA. Prevent username enumeration and avoid giving hints about valid usernames or email addresses in error messages.

#### Weak Session Management Logic:

* + Analyze session management logic for vulnerabilities (e.g. exposes session identifier in the URL, reuse session identifier after successful login).
  + **Mitigation**: Use secure session management practices, including token expiration.

#### Code reviews:

* + Review authentication code for insecure implementations or patterns.
  + **Mitigation**: Conduct regular security assessments and code reviews focused on authentication.

## Software and Data Integrity Failures

### Description

Failures related to the integrity of software and data can allow unauthorized modifications.

### Occurrence

Common in applications that lack proper integrity checks for their software and data.

### Risk Measure

Risk Level: Medium. Can lead to tampering with application logic or data.

### Implementation example

#### Data Integrity:

* + Data is not protected against integrity violation
  + **Mitigation**: Ensure that unsigned or unencrypted serialized data is not sent to untrusted clients without some form of integrity check or digital signature to detect tampering or replay of the serialized data

#### Untrusted Dependencies:

* + Application relying on plugins, libraries, or modules from untrusted sources or repositories.
  + **Mitigation**: Ensure libraries and dependencies, such as npm or Maven, are consuming trusted repositories. If you have a higher risk profile, consider hosting an internal known-good repository that's vetted.

#### Software Integrity:

* + Check if application updates are performed without validation (e.g., file downloads without checks).
  + **Mitigation**: Implement cryptographic checksums to verify software integrity. Sign code and scripts to ensure that only authorized, unmodified code is executed. This can prevent attackers from tampering with files in the software distribution process.

## Security Logging and Monitoring Failures

### Description

Inadequate logging and monitoring can prevent timely detection of security breaches.

### Occurrence

Common in applications that do not maintain detailed logs of user activity or security events.

### Risk Measure

Risk Level: Medium. Delays in breach detection can lead to more extensive damage.

### Implementation example

#### Security Logging Policy:

* + Auditable events, such as logins, failed logins, and high-value transactions, are not logged.
  + **Mitigation**: Ensure all login, access control, and server-side input validation failures can be logged with sufficient user context to identify suspicious or malicious accounts and held for enough time to allow delayed forensic analysis.

#### Misconfigured Alerts:

* + Poorly configured alert thresholds or excessive false positives can lead to alert fatigue, where security teams begin ignoring alerts, potentially missing critical events (e.g. An attacker could move large amounts of sensitive data to an external source without leading to a data breach alert).
  + **Mitigation**: Configure alerts to trigger for anomalous activity (e.g., multiple failed login attempts, access to restricted files) and use threshold-based alerts to reduce false positives. Ensure alerts are sent to the right personnel in real time.

#### Quick Alert / Response:

* + Penetration testing and scans by dynamic application security testing tools (such as Nmap) do not trigger quick alerts.
  + **Mitigation**: Establish effective monitoring and alerting such that suspicious activities are detected and responded to quickly.

## Server-Side Request Forgery (SSRF)

### Description

SSRF vulnerabilities allow attackers to send unauthorized requests from a server to internal resources.

### Occurrence

Often found in web applications that fetch resources without proper validation of input URLs.

### Risk Measure

Risk Level: High. Can lead to access of sensitive data or internal services.

### Implementation example

#### Illegitimate access to internal resources:

* + Attempt to submit a malicious URL that points to internal resources (e.g., http://127.0.0.1:8080).
  + **Mitigation**: Validate and sanitize all input URLs before processing. Maintain detailed logs of outgoing requests, including the requested URLs, response codes, and payloads. Regularly audit these logs for suspicious activity.

#### Network Segmentation:

* + Use testing tools (SSRFmap) to send crafted requests to the application.
  + **Mitigation**: Implement network segmentation and firewall policies to limit server access from external users to the internal network.

#### Denial of service:

* + An attacker can use SSRF to send requests to internal services, leading to resource exhaustion or other denial-of-service conditions.
  + **Mitigation**: Rate limiting can help mitigate the impact of SSRF attacks by controlling the number of requests a user can make within a certain timeframe.

# Discussion

* Analysis of the results in the context of the objectives.
* Comparison with findings from the literature review.
* Implications of the findings for theory, practice, or policy.

# Conclusion

* Summary of key findings.
* Reiteration of the significance of the study.
* Suggestions for future research or recommendations based on findings.

# References

1. **OWASP Foundation.** (2021). *OWASP Top Ten 2021: The Ten Most Critical Web Application Security Risks*. Available at: https://owasp.org/www-project-top-ten/
2. **OWASP Foundation.** (2021). *OWASP Cheat Sheet Series*. Available at: https://cheatsheetseries.owasp.org/

**Cybersecurity Literature**

1. **Shostack, A.** (2014). *Threat Modeling: Designing for Security*. Wiley.
   * This book discusses various vulnerabilities and how to design systems with security in mind.
2. **Hawkins, J.** (2021). *Web Application Security: A Beginner's Guide*. McGraw-Hill Education.
   * This resource provides insights into web application security, vulnerabilities, and mitigation strategies.

**Security Blogs and Whitepapers**

1. **Krebs, B.** (2021). *Krebs on Security*. Available at: <https://krebsonsecurity.com/>
   * A blog that provides insights and analyses on security incidents and vulnerabilities.
2. **OWASP Foundation.** (2021). *OWASP Security Blog*. Available at: https://owasp.org/www/blog/
   * Regular updates on vulnerabilities and security best practices from the OWASP community.

**Security Testing Tools**

1. **PortSwigger.** (2021). *Burp Suite Documentation*. Available at: https://portswigger.net/burp/documentation
   * Documentation for using Burp Suite, a popular tool for web application security testing.
2. **Snyk.** (2021). *Snyk Documentation*. Available at: https://docs.snyk.io/
   * Guidance on using Snyk for identifying vulnerabilities in open-source dependencies.

**Cybersecurity Training Resources**

1. **Coursera.** (2021). *Web Application Security Courses*. Available at: <https://www.coursera.org/courses?query=web%20application%20security>
   * A variety of courses focusing on web application security principles and practices.
2. **Udemy.** (2021). *Web Application Security: The Complete Guide*. Available at: https://www.udemy.com/course/web-application-security-the-complete-guide/
   * An online course that covers a comprehensive range of web security topics.
3. **Cybrary.** (2021). *Web Application Security Courses*. Available at: https://www.cybrary.it/course/web-application-security/
   * Training resources focused on web application security and the OWASP Top Ten.

**OWASP (Open Web Application Security Project)**:

* OWASP provides comprehensive resources on web security risks and best practices, including guidelines for preventing XSS attacks.
* Website: [owasp.org](https://owasp.org)

**MDN Web Docs (Mozilla Developer Network)**:

* MDN offers detailed documentation on web technologies, including security headers like CSP.
* Website: [developer.mozilla.org](https://developer.mozilla.org)

**W3C (World Wide Web Consortium)**:

* The W3C provides specifications and best practices for web technologies, including the CSP specification.
* Website: [w3.org](https://www.w3.org)

**NIST (National Institute of Standards and Technology)**:

* NIST provides guidelines and standards for security practices, which can include web application security.
* Website: [nist.gov](https://www.nist.gov)

**Books on Web Security**:

* Titles like "Web Application Security: A Beginner's Guide" and "The Web Application Hacker's Handbook" cover many security concepts in detail.

**Coding References**:

* Php: <https://www.php.net/manual/fr> , <https://www.pierre-giraud.com/php-mysql-apprendre-coder-cours/session-definition-utilisation/>
* html: <https://outils-javascript.aliasdmc.fr/encodage-caracteres-scientifiques/encode-caractere-2261-html-css-js-autre.html>

# Appendices

* + Additional materials that support the report (e.g., raw data, questionnaires, detailed calculations).

# Acknowledgments

* + Recognition of individuals or organizations that contributed to the project.

# Executive Summary

* + A condensed version of the report, summarizing key points for decision-makers (if the report is for an audience that requires quick insights).