**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

## Methodology

The key steps are:

* understand the 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](#_Toc180450521)

[1.1 summary and objectives of the project 2](#_Toc180450522)

[1.2 Methodology 3](#_Toc180450523)

[1.3 key findings 3](#_Toc180450524)

[2 Introduction 7](#_Toc180450525)

[3 Literature Review 7](#_Toc180450526)

[4 Methodology 7](#_Toc180450527)

[5 Mitigation 8](#_Toc180450528)

[5.1 Use Framework Security Features 8](#_Toc180450529)

[5.2 Regular Security Audits and Testing 8](#_Toc180450530)

[6 Vulnerabilities study 9](#_Toc180450531)

[6.1 Broken Access Control 9](#_Toc180450532)

[6.1.1 Description 9](#_Toc180450533)

[6.1.2 Occurrence 9](#_Toc180450534)

[6.1.3 Risk Measure 9](#_Toc180450535)

[6.1.4 Implementation example 9](#_Toc180450536)

[6.2 Cryptographic Failures 9](#_Toc180450537)

[6.2.1 Description 9](#_Toc180450538)

[6.2.2 Occurrence 10](#_Toc180450539)

[6.2.3 Risk Measure 10](#_Toc180450540)

[6.2.4 Implementation example 10](#_Toc180450541)

[6.3 Injection 10](#_Toc180450542)

[6.3.1 Description 10](#_Toc180450543)

[6.3.2 Occurrence 10](#_Toc180450544)

[6.3.3 Risk Measure 10](#_Toc180450545)

[6.3.4 Implementation example 10](#_Toc180450546)

[**6.4** Insecure Design 14](#_Toc180450547)

[6.4.1 Description 14](#_Toc180450548)

[6.4.2 Occurrence 14](#_Toc180450549)

[6.4.3 Risk Measure 15](#_Toc180450550)

[6.4.4 Implementation example 15](#_Toc180450551)

[6.5 Security Misconfiguration 15](#_Toc180450552)

[6.5.1 Description 15](#_Toc180450553)

[6.5.2 Occurrence 15](#_Toc180450554)

[6.5.3 Risk Measure 15](#_Toc180450555)

[6.5.4 Implementation example 15](#_Toc180450556)

[6.6 Vulnerable and Outdated Components 16](#_Toc180450557)

[6.6.1 Description 16](#_Toc180450558)

[6.6.2 Occurrence 16](#_Toc180450559)

[6.6.3 Risk Measure 16](#_Toc180450560)

[6.6.4 Implementation example 16](#_Toc180450561)

[6.7 Identification and Authentication Failures 17](#_Toc180450562)

[6.7.1 Description 17](#_Toc180450563)

[6.7.2 Occurrence 17](#_Toc180450564)

[6.7.3 Risk Measure 17](#_Toc180450565)

[6.7.4 Implementation example 17](#_Toc180450566)

[6.8 Software and Data Integrity Failures 17](#_Toc180450567)

[6.8.1 Description 17](#_Toc180450568)

[6.8.2 Occurrence 18](#_Toc180450569)

[6.8.3 Risk Measure 18](#_Toc180450570)

[6.8.4 Implementation example 18](#_Toc180450571)

[6.9 Security Logging and Monitoring Failures 18](#_Toc180450572)

[6.9.1 Description 18](#_Toc180450573)

[6.9.2 Occurrence 18](#_Toc180450574)

[6.9.3 Risk Measure 18](#_Toc180450575)

[6.9.4 Implementation example 19](#_Toc180450576)

[6.10 Server-Side Request Forgery (SSRF) 19](#_Toc180450577)

[6.10.1 Description 19](#_Toc180450578)

[6.10.2 Occurrence 19](#_Toc180450579)

[6.10.3 Risk Measure 19](#_Toc180450580)

[6.10.4 Implementation example 19](#_Toc180450581)

[7 Discussion 21](#_Toc180450582)

[8 Conclusion 21](#_Toc180450583)

[9 References 21](#_Toc180450584)

[10 Appendices 23](#_Toc180450585)

[11 Acknowledgments 23](#_Toc180450586)

[12 Executive Summary 23](#_Toc180450587)

# 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
  + **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

#### Easy Detection:

* + 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.

#### Medium Difficulty Detection:

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

#### High Difficulty Detection:

* + Conduct a manual review of business logic in the application to find subtle access control flaws.
  + **Mitigation**: Perform regular code reviews and audits, including testing for business logic vulnerabilities.

## 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

#### Easy Detection:

* + 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.

#### Medium Difficulty Detection:

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

#### High Difficulty Detection:

* + Conduct a security audit to uncover vulnerabilities in custom cryptographic implementations.
  + **Mitigation**: Regularly update cryptographic libraries and use vetted libraries instead of custom solutions.

## Injection

### Description

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

### Occurrence

Common in SQL, NoSQL, and command injection scenarios.

### Risk Measure

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

### Implementation example

#### Easy Detection (‘OR’):

* + Use SQL injection payloads (user 1' OR '1'='1) in login field.
  + **Mitigation**: Use parameterized queries and prepared statements.

#### XSS :

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

|  |  |  |
| --- | --- | --- |
|  | Vulnerable script | Secured script |
| script |  |  |
| Test |  | A screenshot of a computer  Description automatically generated |
| Result |  | A screenshot of a computer  Description automatically generated |

#### 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, to clean user inputs before rendering them.
  + Avoid using functions that execute code directly from user input, like eval() 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., escape quotes).
  + 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, you can add:

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;

#### High Difficulty Detection:

* + Analyze source code for vulnerable query construction, especially in complex applications.
  + **Mitigation**: Conduct regular code reviews and security testing, including fuzzing.

## 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

#### Easy Detection:

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

#### Medium Difficulty Detection:

* + Analyze application workflows for security gaps during user interactions.
  + **Mitigation**: Conduct security reviews during the design phase, involving security experts.

#### High Difficulty Detection:

* + Perform a comprehensive architecture review to identify inherent security flaws.
  + **Mitigation**: Utilize frameworks 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

#### Easy Detection:

* + Use security scanning tools (e.g., Nmap) to check for open ports or default credentials.
  + **Mitigation**: Regularly review configurations and change default settings.

#### Medium Difficulty Detection:

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

#### High Difficulty Detection:

* + Analyze cloud infrastructure configurations for compliance with security policies.
  + **Mitigation**: Utilize Infrastructure as Code (IaC) practices to enforce secure configurations.

## 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

#### Easy Detection:

* + Scan the application with tools like Snyk to find outdated dependencies.
  + **Mitigation**: Regularly update dependencies and monitor for vulnerabilities.

#### Medium Difficulty Detection:

* + Review the dependency tree for known vulnerabilities using tools like npm audit.
  + **Mitigation**: Establish a regular update schedule for dependencies.

#### High Difficulty Detection:

* + 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

#### Easy Detection:

* + Test for weak passwords or lack of account lockout mechanisms.
  + **Mitigation**: Implement strong password policies and multi-factor authentication (MFA).

#### Medium Difficulty Detection:

* + Analyze session management logic for vulnerabilities (e.g., session fixation).
  + **Mitigation**: Use secure session management practices, including token expiration.

#### High Difficulty Detection:

* + 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

#### Easy Detection:

* + Check if application updates are performed without validation (e.g., file downloads without checks).
  + **Mitigation**: Implement cryptographic checksums to verify software integrity.

#### Medium Difficulty Detection:

* + Analyze the update mechanism for security flaws (e.g., lack of secure channels).
  + **Mitigation**: Use secure transmission methods and validate all data inputs.

#### High Difficulty Detection:

* + Conduct a thorough review of the software update processes and integrity verification methods.
  + **Mitigation**: Implement comprehensive integrity checks and monitoring systems.

## 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

#### Easy Detection:

* + Review application logs for missing events or anomalies.
  + **Mitigation**: Implement comprehensive logging for all security-related events.

#### Medium Difficulty Detection:

* + Analyze log configurations for security and compliance.
  + **Mitigation**: Regularly review and test logging configurations.

#### High Difficulty Detection:

* + Perform a security audit to evaluate the effectiveness of logging and monitoring systems.
  + **Mitigation**: Establish incident response plans and regular monitoring reviews.

## 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

#### Easy Detection:

* + Attempt to submit a malicious URL that points to internal resources (e.g., http://localhost:8080).
  + **Mitigation**: Validate and sanitize all input URLs before processing.

#### Medium Difficulty Detection:

* + Use testing tools to send crafted requests to the application.
  + **Mitigation**: Implement network segmentation to limit server access.

#### High Difficulty Detection:

* + Conduct a thorough review of the codebase to find vulnerable request handling.
  + **Mitigation**: Apply strict access controls and regular security

# 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.

# 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).