|  |
| --- |
|  |
| Analysis Report  FudanSELab/train-ticket |
| |  |  | | --- | --- | | Yuval Sigura | 1/14/24 | |

Contents

[Server-Side and Client-Side Code Injection (OWASP A3: Injection) 2](#_Toc156163760)

[Vulnerability Report: Broken Access Control - IDOR (OWASP A5): 4](#_Toc156163761)

[Broken Access Control: CSRF (OWASP A5) 7](#_Toc156163762)

[Broken Access Control: Hardcoded Credentials (OWASP A2) 9](#_Toc156163763)

[Server-Side Request Forgery (SSRF) - OWASP A10: 11](#_Toc156163764)

[Tools been used in the project 14](#_Toc156163765)

# Server-Side and Client-Side Code Injection (OWASP A3: Injection)

Executive Summary:

This report identifies two high-severity vulnerabilities in the application: server-side code injection and client-side code injection. These vulnerabilities map directly to vulnerabilities A3: Injection and A4: Insecure Design in the OWASP Top 10 list. Exploitation could allow attackers to execute arbitrary code, leading to data theft, system compromise, and other serious consequences. Immediate remediation is strongly recommended.

Vulnerability Details:

1. Server-Side Code Injection (OWASP A3: Injection)

Summary:

* Vulnerable endpoint: /api/v1/verifycode/generate
* Vulnerability type: Server-side code injection
* Severity: **High**
* Confidence: Firm

Description:

The application appears to evaluate user input in the /api/v1/verifycode/generate endpoint as code. This was demonstrated by injecting sleep commands in the URL query string, resulting in delayed response times.

Impact:

Attackers could exploit this vulnerability to execute arbitrary code on the server, leading to potential:

* Data theft (user data, sensitive files)
* System compromise (gaining unauthorized access, modifying system files)
* Defacement of the website
* Denial-of-service attacks

Recommendations:

* Implement proper input validation and sanitization:
  + Check user input against a whitelist of allowed characters and patterns.
  + Reject any input that doesn't conform to the expected format.
  + Example (PHP):

PHP

function validateInput($input) {

$allowedChars = "abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ0123456789\_";

if (ctype\_alnum(str\_replace('\_', '', $input)) && strlen($input) <= 32) {

return true;

} else {

return false;

}

}

* Use parameterized queries or prepared statements:
  + Separate code from data to prevent injection attacks.
  + Example (PHP and MySQLi):

PHP

$stmt = $mysqli->prepare("SELECT \* FROM users WHERE username = ?");

$stmt->bind\_param("s", $username);

$stmt->execute();

$result = $stmt->get\_result();

2. Client-Side Code Injection (OWASP A4: Insecure Design)

Summary:

* Vulnerable file: client\_collect.js
* Vulnerability type: Client-side code injection
* Severity: High
* Confidence: Firm

Description:

The client\_collect.js JavaScript file appears vulnerable to code injection through the YsbCaptcha cookie. Malicious code injected into the cookie could be executed when the script is loaded.

Impact:

Attackers could exploit this vulnerability to execute arbitrary JavaScript code on the client-side, leading to potential:

* Cross-site scripting (XSS) attacks (stealing cookies, session tokens, redirecting users)
* Stealing sensitive information via web browser vulnerabilities
* Manipulating the user interface for malicious purposes

Recommendations:

* Validate and sanitize cookies:
  + Ensure cookies only contain valid data using appropriate validation and encoding techniques.
  + Example (JavaScript):

JavaScript

function sanitizeCookie(cookieValue) {

return encodeURIComponent(cookieValue.replace(/[^a-zA-Z0-9\_.-]/g, ''));

}

* Implement a Content Security Policy (CSP):
  + Restrict script execution from untrusted sources.
  + Example (HTTP header):

Content-Security-Policy: script-src 'self'

Additional Recommendations:

* Conduct thorough security testing using automated tools and manual techniques.
* Implement a vulnerability management process to track and remediate vulnerabilities promptly.
* Educate developers on secure coding practices to prevent similar vulnerabilities in the future.

OWASP Top 10 Mapping:

This report specifically addresses vulnerabilities related to OWASP Top 10 categories A3: Injection and A4: Insecure Design. By addressing these vulnerabilities, you will be taking significant steps towards making your application more secure and resilient against these common attack vectors.

Conclusion:These vulnerabilities pose a significant security risk to the application. It is strongly recommended to address them immediately by implementing the suggested recommendations. Failure to do so may result in serious consequences for your users and your application.

Vulnerability Report: Broken Access Control - IDOR (OWASP A5):

Executive Summary:

This report identifies a high-severity Broken Access Control vulnerability (A5 in OWASP Top 10) in the API of an order service. The vulnerability allows unauthorized deletion of orders due to inadequate authorization checks. Immediate remediation is strongly recommended.

Affected Endpoints:

* GET /api/v1/orderservice/order/{order\_id}
* DELETE /api/v1/orderservice/order/{order\_id}

Detailed Findings:

* Description: The DELETE endpoint does not enforce proper authorization checks, enabling unauthenticated users to delete orders.
* Impact: Unauthorized deletion of orders, data loss, potential disruption of business operations.
* POC:

GET /api/v1/orderservice/order/ HTTP/1.1

Host: 139.196.152.44:32677

DELETE /api/v1/orderservice/order/19269289-d445-455c-af41-a2510b0cb93d HTTP/1.1

Host: 139.196.152.44:32677

2. Missing Security Headers:

* Description: Response headers lack essential security headers like Strict-Transport-Security, Content-Security-Policy, and Referrer-Policy.
* Impact: Increased susceptibility to attacks like XSS, clickjacking, and data leakage.

3. Cross-Site Scripting (XSS) (OWASP A7):

* Description: Potential XSS vulnerabilities exist due to user input in cookies (YsbCaptcha and JSESSIONID).
* Impact: Attackers could inject malicious scripts, steal sensitive information, or perform unauthorized actions.

4. Cross-Origin Resource Sharing (CORS) Misconfiguration:

* Description: Server allows requests from any origin (Access-Control-Allow-Origin: \*), potentially broadening attack surfaces.
* Impact: Unintended cross-origin access, increasing risk of unauthorized data access.

5. Sensitive Data Exposure:

* Description: Order details (e.g., accountId, contactsDocumentNumber) are exposed in responses.
* Impact: Potential leakage of sensitive information, violating confidentiality and privacy.

6. Insecure Session Cookie Handling:

* Description: Session cookie JSESSIONID lacks secure attributes, exposing the application to session-related attacks.
* Impact: Attackers could potentially hijack user sessions and compromise user accounts.

Recommendations:

* Implement robust authorization checks for the DELETE endpoint, ensuring only authorized users can perform deletions.
* Enforce user authentication and authorization mechanisms throughout the API.
* Add missing security headers (e.g., Strict-Transport-Security, Content-Security-Policy, Referrer-Policy) to protect against common web attacks.
* Implement proper input validation and sanitization for all user-provided data, especially in cookies, to prevent XSS attacks.
* Review CORS configuration and restrict access to authorized origins.
* Protect sensitive data in responses through encryption or masking.
* Set secure attributes for session cookies (e.g., HttpOnly, Secure) to mitigate session hijacking risks.

Conclusion:

This report highlights significant vulnerabilities in the API that must be addressed promptly to safeguard the application's security and protect sensitive data. Implementing the recommended mitigation measures is crucial to prevent unauthorized access, data breaches, and potential business disruptions.

# Broken Access Control: CSRF (OWASP A5)

**Executive Summary**

A critical vulnerability has been identified in the application's security configuration. Cross-Site Request Forgery (CSRF) protection has been disabled, leaving the application susceptible to CSRF attacks. This vulnerability is categorized as high priority and falls under the OWASP Top 10 category of "A10:2021-Server-Side Request Forgery (SSRF)". Immediate remediation is strongly recommended.

**Vulnerability Details**

**Severity:** High **Category:** Cross-Site Request Forgery (CSRF) **OWASP Top 10 Classification:** A10:2021-Server-Side Request Forgery (SSRF) **POC Code:**

Java

@Override protected void configure(HttpSecurity httpSecurity) throws Exception {

httpSecurity.httpBasic().disable()

// Close default CSRF

.csrf().disable() // Vulnerability: CSRF protection disabled

// Close session

.sessionManagement().sessionCreationPolicy(SessionCreationPolicy.STATELESS)

.and()

.authorizeRequests()

.antMatchers(HttpMethod.GET, "/api/v1/adminbasicservice/adminbasic/stations").permitAll();

}

**Possible Outcomes:**

* Unauthorized actions on behalf of authenticated users
* Data theft or modification
* Account takeover
* Financial losses
* Reputational damage

**Recommendations**

1. **Enable CSRF Protection:**
   * Remove the .csrf().disable() line from the code to enable Spring Security's default CSRF protection.
   * If disabling CSRF protection is unavoidable, implement alternative measures (e.g., custom CSRF tokens).
2. **Implement CSRF Tokens:**
   * Generate unguessable CSRF tokens for state-changing requests.
   * Include the tokens in both forms and corresponding HTTP requests.
   * Validate the tokens on the server-side to ensure authenticity.
3. **Review Sensitive Operations:**
   * Ensure that sensitive operations are not performed using GET requests.
   * Use POST, PUT, or DELETE methods for state-changing actions.
4. **Penetration Testing:**
   * Conduct thorough penetration testing to identify and address any additional vulnerabilities.
5. **Security Awareness:**
   * Educate developers and security teams about CSRF attacks and prevention techniques.

**Additional Considerations**

* Regularly review and update security configurations.
* Stay informed about the latest security threats and best practices.
* Implement a comprehensive security testing and monitoring program.

**Conclusion**

Addressing this CSRF vulnerability is crucial to protect the application and its users from potential attacks. By implementing the recommended measures, you can significantly reduce the risk of CSRF exploits and enhance the overall security posture of the application.

**Sources**

1. <https://github.com/FudanSELab/serverless-trainticket>

2. <https://pt.stackoverflow.com/questions/309618/spring-security-bloqueia-pedidos-post-apesar-das-configura%C3%A7%C3%B5es>

# Broken Access Control: Hardcoded Credentials (OWASP A2)

**Executive Summary**

A critical vulnerability has been discovered in the Train Ticket app due to the presence of hardcoded credentials in the source code. This vulnerability falls under the OWASP Top 10 category A7: Identification and Authentication Failures. It enables attackers to gain unauthorized access to user accounts, potentially leading to data breaches, fraud, and system compromise. Immediate remediation is imperative to mitigate this high-impact risk.

**Vulnerability Details**

**Severity:** Critical **Category:** A7: Identification and Authentication Failures (OWASP Top 10) **POC Code:**

Java

User user = User.builder()

.userId("4d2a46c7-71cb-4cf1-b5bb-b68406d9da6f")

.username("fdse\_microservice")

.password(passwordEncoder.encode("111111"))

.build();

User admin = User.builder()

.userId(UUID.randomUUID().toString())

.username("admin")

.password(passwordEncoder.encode("222222"))

.roles(new HashSet<>(Arrays.asList("ROLE\_ADMIN")))

.build();

**Possible Outcomes:**

* Unauthorized access to user accounts, including privileged accounts
* Data breaches, including theft of personal information and payment details
* Fraudulent activities, such as booking unauthorized tickets or performing financial transactions
* System compromise, potentially leading to complete control of the application
* Compromise of other systems if credentials are reused

**Recommendations**

**Immediate:**

* Revoke hardcoded credentials by removing them from the source code and invalidating them.
* Prompt affected users to reset their passwords through a secure mechanism.

**Long-Term:**

* Implement secure storage for credentials using external configuration files, secrets management solutions, or environment variables.
* Minimize access to credentials, granting access only to authorized personnel with secure access controls.
* Conduct regular code reviews to identify and remediate hardcoded credentials before deployment.
* Utilize static analysis tools to automatically detect hardcoded credentials and other vulnerabilities during development.

**Conclusion**

The presence of hardcoded credentials in the Train Ticket app poses a severe security risk. It's crucial to implement the recommended remediation measures to enhance the app's security posture and safeguard user data and system integrity.

**Sources**

1. <https://github.com/phamquiluan/train-ticket>

# Server-Side Request Forgery (SSRF) - OWASP A10:

**Executive Summary**

Overview

This category is added from the Top 10 community survey (#1). The data shows a relatively low incidence rate with above average testing coverage and above-average Exploit and Impact potential ratings. As new entries are likely to be a single or small cluster of Common Weakness Enumerations (CWEs) for attention and awareness, the hope is that they are subject to focus and can be rolled into a larger category in a future edition.

Description

SSRF flaws occur whenever a web application is fetching a remote resource without validating the user-supplied URL. It allows an attacker to coerce the application to send a crafted request to an unexpected destination, even when protected by a firewall, VPN, or another type of network access control list (ACL).

As modern web applications provide end-users with convenient features, fetching a URL becomes a common scenario. As a result, the incidence of SSRF is increasing. Also, the severity of SSRF is becoming higher due to cloud services and the complexity of architectures.

**How to Prevent**

Developers can prevent SSRF by implementing some or all the following defense in depth controls:

**From Network layer**

* Segment remote resource access functionality in separate networks to reduce the impact of SSRF
* Enforce “deny by default” firewall policies or network access control rules to block all but essential intranet traffic.  
  *Hints:*  
  ~ Establish an ownership and a lifecycle for firewall rules based on applications.  
  ~ Log all accepted *and* blocked network flows on firewalls (see [A09:2021-Security Logging and Monitoring Failures](https://owasp.org/Top10/A09_2021-Security_Logging_and_Monitoring_Failures/)).

**From Application layer:**

* Sanitize and validate all client-supplied input data
* Enforce the URL schema, port, and destination with a positive allow list
* Do not send raw responses to clients
* Disable HTTP redirections
* Be aware of the URL consistency to avoid attacks such as DNS rebinding and “time of check, time of use” (TOCTOU) race conditions

Do not mitigate SSRF via the use of a deny list or regular expression. Attackers have payload lists, tools, and skills to bypass deny lists.

**Possible Outcomes:**

* Unauthorized access to sensitive internal resources or services
* Data exfiltration from internal systems or third-party services
* Remote command execution on internal systems in some cases

**Recommendations**

**Immediate:**

* **Whitelist Allowed Hosts:**
  + Create a list of authorized and trusted service URLs.
  + Reject any tripId values that don't match the whitelisted URLs.
* **Validate Input Thoroughly:**
  + Sanitize tripId to allow only valid characters (alphanumeric, hyphens, etc.).
  + Enforce strict length limits for tripId.
* **Use Pre-Configured URLs:**
  + Remove direct construction of URLs from user input.
  + Hardcode service URLs or fetch them from a secure configuration source.

**Additional Considerations:**

* **Implement Additional Controls:**
  + Consider using a web application firewall (WAF) with SSRF protection.
  + Monitor network traffic for unusual patterns that might indicate SSRF attacks.
* **Regularly Review Whitelists:**
  + Keep whitelists updated as service URLs change.
* **Educate Developers:**
  + Raise awareness about SSRF risks and secure coding practices.
* **Penetration Testing:**
  + Conduct regular penetration testing to identify potential SSRF vulnerabilities.

**Example Code Fix (Java):**

Java

// Whitelist of allowed service URLs

List<String> allowedServiceUrls = Arrays.asList(

"https://trusted-service1.example.com/api/v1/travelservice/trips/",

"https://trusted-service2.example.com/api/v1/travel2service/trips/"

);

// ...

String validatedTripId = sanitizeTripId(tripId); // Sanitize input

String requestUrl = null;

if (allowedServiceUrls.contains(validatedTripId)) {

requestUrl = validatedTripId + tripId;

} else {

// Handle invalid or unauthorized tripId

}

// Proceed with request only if requestUrl is valid

**Conclusion**

Addressing this SSRF vulnerability is crucial to protect the application from unauthorized access, data breaches, and potential remote command execution. Implementing the recommended measures will significantly enhance the application's security posture and mitigate the risk of exploitation.

# Tools been used in the project

#### Static Analysis Tools:

SonarQube:

* Analyzes code for vulnerabilities, bugs, and code quality issues.
* Supports multiple programming languages.
* Integrates with development environments and CI/CD pipelines.

Docker-Scout:

* Specifically scans Docker images for vulnerabilities and misconfigurations.
* Checks for outdated packages and common security issues.
* Provides actionable remediation guidance.

GitHub Action - CodeQL:

* Integrates static analysis into GitHub workflows for continuous code scanning.
* Uses CodeQL, a powerful query language for identifying vulnerabilities.
* Offers customizable queries and reporting.

#### Vulnerability Scanning and Network Mapping Tools:

Shodan:

* Search engine for discovering devices and services connected to the internet.
* Reveals potential security risks in exposed systems.
* Used for reconnaissance and threat intelligence.

Nmap:

* Network scanning tool for identifying hosts, open ports, services, and vulnerabilities.
* Versatile for network discovery, mapping, and vulnerability assessment.
* Web Application Security Testing Tools:

Burp Suite Professional:

* Comprehensive suite for web application security testing.
* Includes features for intercepting proxy, vulnerability scanning, manual testing tools, and reporting.
* Widely used for manual and automated vulnerability discovery and exploitation.

Key Differences:

Static analysis tools examine code without executing it, focusing on code structure and potential flaws.

Vulnerability scanning tools actively probe systems or applications to detect known vulnerabilities.

Web application security testing tools specialize in identifying vulnerabilities in web applications, often involving manual exploration and exploitation techniques.