ISSN: 2754-6705

Journal of Mathematical & Computer Applications

SCIENTIFIC Research and Community

Review Article

Open Access

Mitigating Cross-Site Scripting (XSS) and Cross-Site Request Forgery (CSRF) in Java Web Applications

Tirumala Ashish Kumar Manne

USA

ABSTRACT

Cross-Site Scripting (XSS) and Cross-Site Request Forgery (CSRF) are among the most critical security threats targeting Java web applications. These vulnerabilities exploit trust relationships between users and applications, often leading to session hijacking, data theft, or unauthorized actions. Despite advancements in secure frameworks, many applications remain vulnerable due to improper input validation, insecure coding practices, and lack of awareness. This paper provides a comprehensive study of XSS and CSRF threats within Java-based ecosystems, including their mechanisms, common sources, and practical impact. It emphasizes mitigation strategies using well-established tools and frameworks such as OWASP Java Encoder, Spring Security, and secure HTTP headers. By analyzing real-world case studies and secure design patterns, the article outlines actionable techniques to prevent exploitation. It also explores the integration of security testing tools like OWASP ZAP and Burp Suite into the development lifecycle. The goal is to equip developers, architects, and security professionals with a clear roadmap to safeguard applications against these persistent threats. This work contributes to the broader effort of embedding security by design in Java web application development and encourages a proactive approach to defending against XSS and CSRF attacks.

***Corresponding author**

Tirumala Ashish Kumar Manne, USA.

Received: October 12, 2022; **Accepted:** October 20, 2022, **Published:** October 27, 2022

Keywords: Java Web Applications, Cross-Site Scripting (XSS), Cross-Site Request Forgery (CSRF), OWASP, Spring Security, Secure Coding, Java EE Security

Introduction

Web applications have become integral to modern enterprise and consumer services, yet they remain vulnerable to common and well-documented attacks such as Cross-Site Scripting (XSS) and Cross-Site Request Forgery (CSRF). These vulnerabilities, listed in the OWASP Top 10 for over a decade, continue to pose serious risks to data confidentiality, integrity, and user trust [1]. XSS exploits the trust a user has in a specific site by injecting malicious scripts, whereas CSRF takes advantage of the trust that a site has in a user's browser, forcing unauthorized actions on behalf of authenticated users [2].

Java-based web applications, due to their widespread adoption and reliance on frameworks like Spring MVC and Java EE, are frequent targets for such attacks. While these platforms provide baseline protections, insecure configurations and poor coding practices often negate built-in defenses [3]. Insecure input handling, improper output encoding, and lack of token-based validation mechanisms contribute significantly to the problem. This paper explores the root causes and impacts of XSS and CSRF in Java web applications. It further presents a systematic approach to mitigation, highlighting the use of libraries like OWASP Java Encoder and Spring Security, backed by practical examples and real-world scenarios.

Understanding the Threats

Cross-Site Scripting (XSS) and Cross-Site Request Forgery (CSRF) are critical security vulnerabilities that exploit the inherent trust in web applications. Understanding their mechanisms and impact is essential for developing effective defenses, particularly in Java-based environments.

Cross-Site Scripting (XSS)

XSS is a vulnerability that allows attackers to inject malicious scripts into content delivered to users. These scripts execute in the victim's browser with the same permissions as the legitimate application, enabling session hijacking, credential theft, or redirection to malicious sites. XSS is typically categorized into three types:

Reflected XSS: Malicious input is reflected off a web server and executed immediately, often via a crafted URL.

Stored XSS: Malicious code is stored on the server (e.g., in a database) and served to users over time.

DOM-based XSS: Injection occurs on the client side via JavaScript manipulation of the DOM without server interaction [4].

In Java applications, XSS often arises from failing to encode output in JSPs or from insecure use of JavaScript and AJAX within frameworks like JSF or Spring MVC [5].

Cross-Site Request Forgery (CSRF)

CSRF exploits the trust a web application has in a user's browser. If a user is authenticated, an attacker can trick the browser into submitting a request (e.g., transferring funds, changing passwords)

without the user's knowledge. The vulnerability hinges on the automatic inclusion of authentication credentials like cookies in HTTP requests [6].

Java web applications that rely solely on session-based authentication without token-based validation are particularly susceptible. Until recently, CSRF protection in Java EE was limited, leaving developers to implement their own token mechanisms or rely on frameworks like Spring Security to provide CSRF tokens [7].

Root Causes in Java Web Applications

Despite the availability of modern security frameworks, Java web applications often remain vulnerable to XSS and CSRF due to a combination of design flaws, developer oversight, and legacy architectural practices. Identifying these root causes is essential for targeted mitigation.

Common Development Mistakes

Many XSS and CSRF vulnerabilities arise from developers failing to apply secure coding practices consistently. For XSS, a key issue is improper handling of user input particularly when unvalidated data is directly rendered into HTML, JavaScript, or CSS contexts without proper output encoding [8]. Developers often assume that server-side validation alone is sufficient, neglecting the need for contextual encoding in client-facing content [9].

Insecure Use of JSP, JSTL, and Custom Tags

Java Server Pages (JSP) and Java Server Pages Standard Tag Library (JSTL) are widely used in legacy Java applications. However, they render user input without automatic encoding unless explicitly configured. Many custom tag libraries developed without proper encoding mechanisms further increase the risk of reflected or stored XSS [10].

Misconfigured or Missing Security Headers

HTTP headers such as Content-Security-Policy, X-Content-Type-Options, and X-Frame-Options provide an additional layer of protection against injection and clickjacking attacks. However, Java developers often overlook these headers due to lack of awareness or reliance on default server configurations that do not enforce them [11].

Inadequate Input and Output Validation

A fundamental security lapse is the lack of centralized input validation and output encoding mechanisms. Java EE and older versions of Spring provided limited built-in validation, leading to custom, inconsistent implementations. This inconsistency makes it difficult to enforce a uniform security policy and increases the risk of introducing vulnerabilities during future code changes [12].

These root causes reflect the need for security-aware development practices and the integration of secure libraries and frameworks from the early stages of the software development life cycle.

Mitigation Strategies

Effectively defending Java web applications against Cross-Site Scripting (XSS) and Cross-Site Request Forgery (CSRF) requires a multi-layered security strategy. This section outlines technical countermeasures that developers can adopt to reduce the attack surface and strengthen application resilience.

XSS Mitigation

Output Encoding: Proper output encoding ensures that untrusted data is safely inserted into HTML, JavaScript, or CSS contexts without being interpreted as code. Developers should use context-aware encoders such as the OWASP Java Encoder library, which automatically escapes characters based on the output location (HTML element, attribute, or JavaScript) [13].

Input Sanitization: Although output encoding is preferred, input sanitization is useful for limiting input to a strict format (e.g., numeric IDs or email addresses). Libraries such as Apache Commons Validator can be integrated into Java applications to enforce input rules and prevent malformed data from being processed [14].

Content Security Policy (CSP): A properly configured CSP header restricts the sources from which scripts can be executed, thus mitigating the risk of XSS—even if an injection succeeds. Although not a Java-specific control, it complements server-side protections [15].

Secure Templating Engines: Modern templating engines like Thyme leaf and Free Marker support automatic output encoding. Developers should avoid using JSP with script let code and instead adopt these safer alternatives [16].

CSRF Mitigation

Synchronizer Token Pattern (STP): In this approach, the server generates a unique CSRF token for each user session. The token is included in all forms and verified upon submission, ensuring that requests originate from legitimate users. This is the most commonly recommended approach and is natively supported by Spring Security [17].

Double Submit Cookies: This method involves placing the CSRF token in both a cookie and a request parameter or header. The server compares both values to validate the request. Though not as robust as STP, it's suitable in stateless REST APIs [18].

Same Site Cookie Attribute: The Same Site attribute instructs browsers not to send cookies on cross-origin requests, thereby reducing the CSRF threat in modern browsers. Setting Same Site=Lax or Strict on session cookies in Java web apps offers a lightweight defense [19].

Framework-Level Protection: Spring Security and other modern frameworks offer built-in CSRF protection that can be enabled with minimal configuration. For example, Spring automatically inserts and validates CSRF tokens in forms and headers when properly enabled via the HttpSecurity.csrf(). enable () API [17].

These mitigation strategies, when applied together, form a robust defense-in-depth model. Developers are encouraged to integrate them early in the development lifecycle and automate security validation as part of CI/CD pipelines.

Secure Framework Usage

Modern Java frameworks offer built-in mechanisms to mitigate XSS and CSRF vulnerabilities, but effective use requires proper configuration and awareness of secure development patterns. Leveraging these frameworks significantly reduces the likelihood of introducing security flaws, provided best practices are followed throughout the development lifecycle.

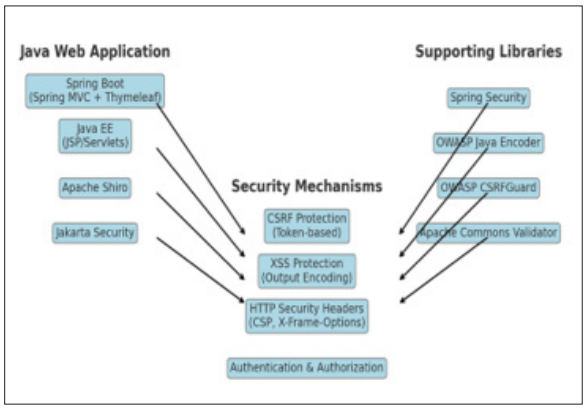

Figure 1: Secure Framework Usage

CSRF and XSS Protection in Spring Boot & Spring Security

Spring Security provides out-of-the-box protection against CSRF through token-based validation. When enabled, the framework generates a unique CSRF token per session and automatically injects it into HTML forms. Incoming POST, PUT, DELETE, or PATCH requests are validated for the token's presence and correctness, ensuring that state-changing requests are not forged [20]. Spring MVC encourages the use of Thyme leaf, a secure templating engine that automatically escapes HTML output, reducing the risk of XSS [21].

Spring also supports the integration of Content-Security-Policy headers and other security-related HTTP headers via configuration classes or filters, offering layered protection against common threats.

Role of Java EE Security Features

Java EE (now Jakarta EE) provides foundational support for authentication and authorization but offers limited native protection against XSS and CSRF. Developers using JSP and Servlets must implement their own CSRF token systems or integrate third-party libraries such as OWASP CSRFGuard [22]. For XSS mitigation, Java EE does not enforce output encoding by default in JSPs or EL expressions, leaving it up to developers to apply contextual escaping using custom tag libraries or external encoders [23].

Using Apache Shiro and Jakarta Security

Apache Shiro is a Java security framework focused on authentication, authorization, cryptography, and session management. While it does not include native CSRF or XSS protections, it integrates well with servlet-based Java applications and can be used alongside filters and encoders to enhance security posture [24]. Jakarta Security (formerly Java EE Security API) introduced more modular and declarative security features starting with Java EE 8, including HTTP authentication mechanisms and context-aware identity propagation. While not explicitly designed for CSRF or XSS protection, its integration with CDI and expression languages allows more secure access control patterns [25].

Using secure frameworks reduces the likelihood of developer-induced vulnerabilities and encourages standardized security practices. However, developers must still audit configurations, avoid disabling security features, and keep frameworks up to date to benefit from ongoing improvements.

Testing and Validation

Rigorous testing and validation are essential for identifying and mitigating XSS and CSRF vulnerabilities in Java web applications. Security testing must be incorporated throughout the software development life cycle (SDLC) to ensure robust protection, particularly as modern threats evolve in complexity.

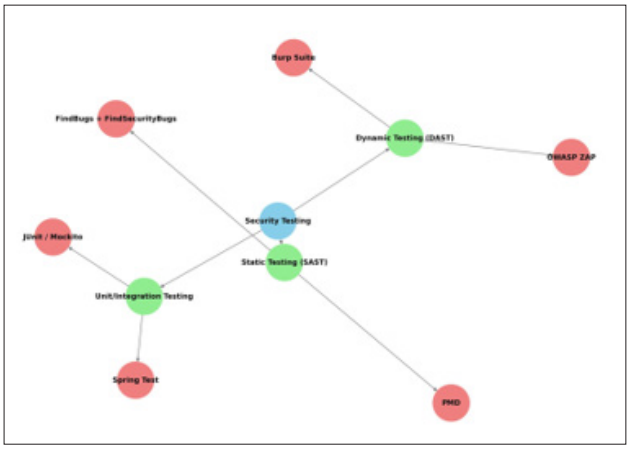

Figure 2: Testing and Validation

Static and Dynamic Analysis Tools

Static Application Security Testing (SAST) analyzes source code or bytecode for security vulnerabilities without executing the program. Tools like Find Bugs with the FindSecurityBugs plugin and PMD are widely used in Java ecosystems to detect potential XSS injection points and unsafe coding patterns [26]. These tools help enforce secure coding standards before deployment.

Dynamic Application Security Testing (DAST), on the other hand, evaluates the application during runtime. Tools like OWASP ZAP and Burp Suite simulate attacks on live applications to uncover runtime vulnerabilities, such as reflected XSS or CSRF exploits [27].

OWASP ZAP and Burp Suite

OWASP ZAP is an open-source DAST tool designed specifically to find vulnerabilities in web applications. It features active and passive scanning, spidering, and fuzzing capabilities. Developers can integrate ZAP into CI/CD pipelines for automated security validation [28].

Burp Suite, widely used for professional penetration testing, supports scanning for XSS and CSRF flaws through intercepting proxies and rule-based scanners. It allows testers to manipulate HTTP headers, parameters, and cookies to uncover insecure implementations [29].

Unit and Integration Testing for Security

Java developers are encouraged to write automated unit and integration tests to validate security-critical components. Frameworks like JUnit, combined with Spring Test or Mockito, can be used to verify that CSRF tokens are generated and validated properly, or that user-generated content is safely encoded before being rendered.

Test cases should include edge conditions such as submitting malformed scripts or sending unauthorized requests to ensure XSS and CSRF protections behave correctly under all circumstances [30].

Integrating both automated and manual testing strategies ensures that vulnerabilities are caught early and mitigated effectively. A security-aware testing culture improves code quality and significantly reduces the likelihood of security breaches in production environments.

Best Practices and Developer Guidelines

Mitigating Cross-Site Scripting (XSS) and Cross-Site Request Forgery (CSRF) in Java web applications requires not only technical controls but also a disciplined approach to secure software engineering. Developers must adhere to industry best practices and adopt a security-first mindset throughout the application development life cycle.

Secure Coding Checklists

Using secure coding checklists during development helps enforce consistency and reduce human error. OWASP's Secure Coding Practices Quick Reference Guide offers a lightweight checklist covering output encoding, input validation, and proper use of authentication tokens [31]. For Java applications, developers should always encode dynamic content in HTML, avoid inline scripts, and disallow dangerous input such as <script> tags or event handlers.

Code Review and Security Audits

Manual code reviews remain one of the most effective ways to detect complex security flaws. Teams should integrate security-focused reviews as part of their pull request and release processes, with checklists targeting CSRF token management, input sanitization, and output encoding. Periodic third-party security audits further strengthen defenses by identifying flaws that internal teams may overlook [32].

Secure Defaults and Least Privilege

Applications should be configured with secure defaults. This includes enabling CSRF protection in frameworks like Spring Security, disabling unnecessary HTTP methods, and setting Http Only, Secure, and Same Site flags on cookies. Least privilege principles should be applied to user roles, application permissions, and external integrations [33].

Continuous Integration of Security Checks

Incorporating security scanning tools in CI/CD pipelines ensures early detection of vulnerabilities. Tools like OWASP Dependency-Check can identify vulnerable dependencies in Java projects, while static analysis tools like Spot Bugs enforce secure coding patterns [34]. Integration of these tools helps shift security left in the SDLC.

Developer Training and Awareness

Regular training sessions are crucial for maintaining developer awareness of evolving security threats. Training should include hands-on exercises in identifying and fixing XSS and CSRF vulnerabilities using real code examples. In addition, developers should stay updated with secure coding guidelines issued by OWASP, CERT, and vendor-specific advisories [35].

Conclusion

Cross-Site Scripting (XSS) and Cross-Site Request Forgery (CSRF) remain persistent threats to Java web applications, despite the evolution of secure frameworks and tools. This article explored the mechanisms, root causes, and practical mitigation strategies to address these vulnerabilities. By leveraging secure libraries such as OWASP Java Encoder and frameworks like Spring Security, developers can effectively implement defense-in-depth strategies. Additionally, integrating secure coding practices, static and dynamic analysis tools, and proper validation mechanisms strengthens the security posture of applications. The importance of adopting secure defaults, using templating engines with automatic encoding, and performing regular security audits cannot be overstated. Framework-level protections must be properly configured and supplemented with real-world testing using tools like OWASP ZAP and Burp Suite. Ultimately, a security-aware development culture backed by ongoing education, rigorous validation, and adherence to best practices is essential. Addressing XSS and CSRF proactively not only protects application data and user trust but also fosters long-term resilience in increasingly complex threat landscapes.

References

1. (2017) OWASP Foundation, OWASP Top 10 - 2017: The Ten Most Critical Web Application Security Risks https://owasp.org/www-project-top-ten/2017/Top_10.
2. M Jakobsson, S Myers (2006) Phishing and Countermeasures: Understanding the Increasing Problem of Electronic Identity Theft. Wiley-Interscience https://www.researchgate.net/publication/298091713_Phishing_and_Counter-Measures_Understanding_the_Increasing_Problem_of_Electronic_Identity_Theft.
3. A Kieyzun, P J Guo, K Jayaraman, M D Ernst (2009) Automatic creation of SQL injection and cross-site scripting attacks Proc 31st Int Conf. Software Engineering Vancouver https://www.researchgate.net/publication/38003076_Automatic_Creation_of_SQL_Injection_and_Cross-Site_Scripting_Attacks.
4. Y Nadji, P Saxena, D Song (2009) Document Structure Integrity: A Robust Basis for Cross-site Scripting Defense Proc. 16th Network and Distributed System Security Symp. https://www.ndss-symposium.org/wp-content/uploads/2017/09/Document-Structure-Integrity-A-Robust-Basis-for-Cross-site-Scripting-Defense-Yacin-Nadji.pdf.
5. A Barth, C Jackson, J C Mitchell (2009) Securing Frame Communication in Browsers. Communications of the ACM 52:83-91.
6. T Jim, N Swamy, M Hicks (2007) Defeating Script Injection Attacks with Browser-Enforced Embedded Policies Proc. 16th Int. Conf. World Wide Web (WWW) https://www.cs.umd.edu/~mwh/papers/jssecurity.pdf.
7. E F Cureton (2019) Spring Security in Action Manning Publications https://www.oreilly.com/library/view/spring-security-in/9781617297731/.
8. G Wassermann Z Su (2008) Static Detection of Cross-Site Scripting Vulnerabilities Proc. 30th Int. Conf. Software Engineering (ICSE), Leipzig, Germany https://dl.acm.org/doi/10.1145/1368088.1368112.
9. M Howard, D LeBlanc (2003) Writing Secure Code. Microsoft Press https://ptgmedia.pearsoncmg.com/images/9780735617223/samplepages/9780735617223.pdf.
10. D Gollmann (2008) Securing Web Applications Information Security Technical Report. 13:1-9.
11. L Kohn Felder, P Garg (2007) The Threats to Our Products. Microsoft Secure Windows Initiative (SWI) https://www.scribd.com/document/379619841/STRIDE-docx.
12. (2019) OWASP Foundation. OWASP ESAPI for Java Project https://owasp.org/www-project-enterprise-security-api/.
13. (2019) OWASP Foundation. OWASP Java Encoder Project https://owasp.org/www-project-java-encoder/.
14. (2019) Apache Software Foundation. Commons Validator https://commons.apache.org/proper/commons-validator.
15. S Stamm, B Sterne, G Markham (2010) Reining in the Web

with Content Security Policy Proc. 19th Int. Conf. World Wide Web (WWW) https://archives.iw3c2.org/www2010/proceedings/www/p921.pdf.

16. D Winterfeldt (2016) Secure Coding with Java: Developing Secure Web Applications. Wiley https://www.researchgate.net/publication/365662909_Defensive_programming_Developing_a_web_application_with_a_secure_coding_practices.
17. B J Evans (2017) Spring Security 4.2 Reference. Pivotal Software https://docs.spring.io/spring-security/site/docs/4.2.x/reference/html/index.html.
18. L Desmet, F Piessens, W Joosen (2012) Double Submit Cookies for Stateless CSRF Protection IEEE Security & Privacy 10:27-34.
19. M West (2016) Same Site Cookies Explained Google. IETF Draft https://web.dev/articles/samesite-cookies-explained.
20. B J Evans (2017) Spring Security 4.2 Reference, Pivotal Software https://docs.spring.io/spring-security/site/docs/4.2.x/reference/html/index.html.
21. U Dahan, A Orenstein (2014) Spring MVC Cookbook. Packt Publishing https://www.oreilly.com/library/view/spring-mvc-cookbook/9781784396411/pr01.html.
22. (2019) OWASP Foundation. OWASP CSRFGuard Project https://owasp.org/www-project-csrfguard/.
23. R Monson-Haefel, D. Chappell (2008) Java Platform, Enterprise Edition: A Practical Guide. Addison-Wesley https://www.e-booksdirectory.com/listing.php?category=437.
24. K Merkel (2014) Apache Shiro Essentials. Packt Publishing https://www.business-vox.com/catalog/book/docid/88850199?_locale=en.
25. A Gupta, R Broeck (2011) Pro Java EE Security and Identity Management. Apress https://link.springer.com/chapter/10.1007/978-1-4842-8214-4_33.
26. LP Etzkorn (2014) Introduction to Software Engineering. CRC Press https://www.taylorfrancis.com/books/mono/10.1201/9781315371665/introduction-software-engineering-ronald-leach.
27. (2014) OWASP Foundation. OWASP Testing Guide v4 https://owasp.org/www-project-web-security-testing-guide/.
28. S Gupta, R Kumar (2015) Secure Software Development: An Agile Perspective. International Journal of Computer Applications 117:1-7.
29. D Stuttard, M Pinto (2011) The Web Application Hacker's Handbook: Finding and Exploiting Security Flaws, 2nd ed., Wiley https://www.beiruteyecenter.com/uploads/3794_1008_4334.pdf.
30. NR Mead (2009) Security Requirements Engineering for Software Systems: Case Studies in Support of Secure Software Development. ACM SIGSOFT Software Engineering Notes 34:1-7.
31. M Shema (2010) The OWASP Secure Coding Practices Quick Reference Guide. OWASP Foundation https://owasp.org/www-project-secure-coding-practices-quick-reference-guide/.
32. G McGraw (2006) Software Security: Building Security In. Addison-Wesley https://bayanbox.ir/view/8807759492001540187/Software-Security-Building-Security-In-by-Gary-McGraw-z-lib.org.pdf.
33. M Howard, S Lipner (2006) The Security Development Lifecycle. Microsoft Press https://www.researchgate.net/publication/234792172_The_Security_Development_Lifecycle.
34. (2019) OWASP Foundation, OWASP Dependency-Check Project https://owasp.org/www-project-dependency-check/.
35. (2019) CERT Coordination Center, Secure Coding Standards for Java. Carnegie Mellon University https://wiki.sei.cmu.edu/.