International Journal of Applied Mathematics

Volume 38 No. 5s, 2025

ISSN: 1311-1728 (printed version); ISSN: 1314-8060 (on-line version)

LEGACY VS MODERN SECURITY HANDLING IN JAVA: A COMPARATIVE STUDY OF OPENSAML, SPRING SECURITY, AND JWT-BASED AUTHENTICATION

[1]Sravan Reddy Kathi,[2] Ayush Dayaram Jaiswal

[1]Bridgeport, Pennsylvania, USA, Independent Researcher, sravanreddykathi55@gmail.com

https://orcid.org/0009-0000-7646-3085

[2]Atlanta, GA, USA, Independent Researcher, ayushjaiswal76@gmail.com

https://orcid.org/0009-0007-5281-1250

Abstract

Enterprises must make important decisions about replacing outdated security features with more secure, scalable, and maintainable alternatives as they update their Java applications. Three popular Java security frameworks—OpenSAML, Spring Security OAuth2, and JWT-based authentication—are compared in this paper. Their architectural models, language integrations, and vulnerability exposure in Java 8 and Java 17 environments are evaluated. The study highlights compatibility, performance, and security risks that arise during version upgrades as it examines at significant shifts from XML-based security to annotation-driven and token-based security. We measure improvements in code maintainability, testability, and CVE reduction using benchmark testing, real-world deployment scenarios, and static analysis tools (e.g., OWASP Dependency-Check, SpotBugs). Our research shows that modern security stacks are much better in terms of performance, ease of integration, and long-term maintainability. However, they also need careful migration planning to avoid regressions. This study fills in a big gap in the literature by systematically comparing old and new security methods in Java across LTS versions. It also provides helpful guidance to developers and architects who plans secure Java Modernization.

Keywords:

Java Security, OpenSAML, Spring Security, JWT Authentication, Identity and Access Management, OAuth2, XML-based Security, Annotation-based Security, CVE Analysis, Java Version Migration, Static Code Analysis, Vulnerability Management, Secure Software Development

1. Introduction

Over the past 20 years, the security architecture of Java-based enterprise applications has evolved a lot. There are now many ways to implement access control, from early XML-based systems to modern annotation-driven and token-based security frameworks. Each of these has its own set of risks. When moving from one version of Java to another, developers and architects have a hard time identifying, maintaining, and upgrading authentication and authorization mechanisms.

Three different Java security paradigms are represented by OpenSAML, Spring Security, and JWT-based custom authentication schemes. OpenSAML was originally used for federated identity systems and mainly depends on XML assertions and signatures. A flexible, extensible, and annotation-driven model that is integrated with modern OAuth2 flows is provided by Spring Security. Stateless, token-based authentication is provided by JWT (JSON Web Token), which is becoming more and more popular in microservices and often does not require external libraries.

Received: August 09, 2025

Volume 38 No. 5s, 2025

ISSN: 1311-1728 (printed version); ISSN: 1314-8060 (on-line version)

The design, performance, and vulnerability exposure of Spring Security OAuth2, JWT-based authentication schemes, and OpenSAML (v5.1.2 and v5.1.4) are compared in this paper. It also highlights how these implementations get affected by Java version upgrades, from Java 8 to Java 17, in terms of tooling support, compatibility, and runtime behavior.

1.1 Research Contributions

This study's main contributions are:

- An architectural comparison of JWT-based authentication, Spring Security OAuth2, and OpenSAML placed side by side.
- An evaluation of the developer work required during the Java migration and compatibility shifts.
- A CVE-based vulnerability analysis of Java versions and across frameworks.
- Guidance for architects on how to choose the best security stack depending on factors like application size, risk tolerance, and futureproofing.

1.2 Paper Organization

Each section of the paper is structured as follows: Related work is addressed in Section 2. The experimental design and evaluation process are described in detail in Section 3. Results on architectural complexity, security posture, and performance are shown in Section 4. Limitations and implications are covered in Section 5. The study is concluded in Section 6 with recommendations for further research.

2. Background and Related Work

Java security frameworks have been a focus of research, particularly as federated identity models and distributed architectures have become more prevalent in enterprise systems. The increasing need for secure token-based communication, fine-grained authorization, and Single Sign-On (SSO) has led developers to often choose between enterprise frameworks like Spring Security, legacy libraries like OpenSAML, and contemporary tokenization techniques like JWT. But a large portion of the scholarly literature is still dispersed, that focuses on specific libraries [1][2], formal security models [3][4], or cryptographic vulnerabilities [5][6] without comparing them to Java version updates or enterprise modernization.

2.1. OpenSAML and Federated Identity

The Shibboleth Consortium's OpenSAML has long served as the foundation for federated identity implementations, especially in the government and higher education sectors [7]. It offers programmatic support for XML signatures, SAML assertions, and schema-based validation. Scholars have emphasized its robust cryptographic design and strong alignment with SAML 2.0 specifications [8]. Nevertheless, a number of studies also note that its XML-based configuration is verbose and complex, which has caused its adoption in microservices-oriented architectures to be slower [9].

The increased overhead in parsing and validating SAML tokens, particularly in high-load environments, was observed in a study by Mosenia et al. [10]. Additionally, forward migration became more difficult as Java developed because many OpenSAML dependencies (such as Apache XML Security and older JAXB-based modules) were deprecated or incompatible [11].

2.2. Spring Security and Declarative Models

For enterprise-grade applications developed using the Spring ecosystem, Spring Security has become the standard option. It provides declarative authorization through annotations, support for OAuth2 client/server, and smooth integration with Spring Boot auto-configuration and servlet filters [12]. Its modular structure was examined by researchers like Mueller et al. [13], who also commended the flexibility it provides through method-level security and pluggable authentication providers.

However, the steep learning curve and configuration complexity are still well-documented issues, especially when it comes to personalizing authentication flows [14]. One common enterprise vulnerability in CI/CD pipelines has been identified as security misconfigurations in Spring Security (e.g., exposing actuator endpoints or failing to validate JWT scopes) [15].

The evolution of Spring Security using newer JDK features like java.util.Optional, improved reflection APIs, and sealed classes [16] evolved post-Java 11 has also been the focus of recent works.

2.3. JWT and Stateless Token Handling

For stateless authentication, JSON Web Tokens (JWTs) have become more and more popular, especially in serverless applications and REST APIs. By avoiding the need for session storage and directly embedding claims in tokens, JWT simplifies client-server interactions [17]. In distributed systems, studies like Bertoni et al. [18] have compared JWT to OAuth2 and SAML, demonstrating JWT's relative effectiveness in terms of token size and parsing speed.

Hard-coded secrets, a lack of expiration, or insufficient signature verification are common implementation errors that affect JWTs despite their efficiency [19][20]. Numerous JWT-related risks are included in OWASP's top 10 API security list, which highlights the necessity of claim validation and rotating secrets [21].

2.4. Comparative Studies and Migration Context

Although there have been individual comparisons between JWT and SAML [22] and Spring Security's OAuth2 and legacy servlet filters [23], few studies place these comparisons in the context of enterprise integration, CI/CD testing automation, or Java version migration. Library compatibility has been significantly impacted by the deprecation of core APIs such as JAXB, CORBA, or sun.misc.* and the increasing modularization of the Java platform post-Java 9 via JPMS [24].

This study offers a comparative, empirical evaluation for OpenSAML, Spring Security, and JWT-based authentication in enterprise Java applications migrating from Java 8 to Java 17, which is different from prior studies. We concentrate on test coverage, vulnerability exposure, configuration effort, and practical migration challenges. Additionally, the work fills a significant gap in the academic and practitioner literature by incorporating DevSecOps tools like SonarQube and OWASP Dependency-Check into the analysis pipeline.

3. Methodology

The operational characteristics of three widespread Java-based security stacks—OpenSAML, Spring Security with OAuth2, and JWT-based custom authentication—across two major Java Long-Term Support (LTS) versions—Java 8 and Java 17—were assessed and compared using the structured experimental design presented in this section. The evaluation focuses on architecture complexity, migration effort, runtime performance, and security posture, and has been inspired by gaps identified from previous research [2][3][6].

Practical situations that arise during enterprise migrations, like cryptographic shifts, API deprecations, and configuration differences between XML-centric and annotation-driven security models, are highlighted in the comparative analysis.

3.1 Security Stack Implementations

One of the target security frameworks was incorporated into each of the three stand-alone microservice-style reference applications that were developed. To ensure test parity and isolate the security stack as the only variable, all services implement the identical business logic including CRUD operations, login workflows, and role-based access controls.

- **OpenSAML (v5.1.2 and v5.1.4):** Complete SAML 2.0 support was implemented, including the use of BouncyCastle for XML digital signature processing, assertion validation, and SP metadata configuration. Reliance on XML configuration and schema-based marshalling/unmarshalling was the contributor of complexity.

- **Spring Security with OAuth2 (Spring Boot 2.5 and 3.2)**: Token-based authentication and role mapping were implemented using OAuth2 Resource Server and Spring Authorization Server. Java configuration was used to set up REST endpoint security and method-level annotations.
- **Custom JWT-Based Authentication**: HS256 and RS256 algorithms were used to implement stateless token-based security. After logging in, JWTs were sent out and validated using a custom filter. To simulate real-world usage, claim validation and updated token logic were added.

Jenkins and SonarQube were used to integrate all implementations into CI/CD pipelines for continuous validation, and all implementations followed the OWASP Top Ten guidelines.

3.2 Java Version Compatibility and Migration Effort

Originally developed on Java 8 (OpenJDK 1.8.0_361), the applications were later moved to Java 17 (OpenJDK 17.0.9). Identification and resolution of version-specific issues, such as deprecated APIs, library incompatibilities, and module system constraints added in Java 9+, was the primary focus of the migration tasks.

Migration metrics included:

- **Build Compatibility**: Success rates for compilations and necessary changes required to the Maven plugin.
- **API Deprecation**: Identified through manual review, jdeps, and jdeprscan.
- **Library Upgrade Requirements**: Including the Jackson, BouncyCastle, and Spring modules.
- **Refactoring Complexity**:Measured by the number of hours reported during the migration and the lines of code that were changed.

To highlight operational burden, tooling gaps, and behavioral inconsistencies between Java versions, the migration path for each stack was documented.

3.3 Security Analysis and CVE Mapping

A two-pronged evaluation was conducted to assess each framework's security posture across versions:

1. **Automated Scanning**:

a. To detect known vulnerabilities in third-party dependencies, use OWASP Dependency-Check v8.4.

b. Use SonarQube v9.9 to evaluate rule violations, code smells, and security hotspots such as weak crypto or hardcoded secrets.

2. **Manual CVE Mapping**:

a. Cross-referenced library versions with the **NIST National Vulnerability Database (NVD)**.

b. Evaluated whether library updates fixed pre-existing CVEs or caused regressions by contrasting Java 8 and Java 17 environments.

This method demonstrated differences in the risk surface and reinforced the significance of version-aware patching, particularly for security-critical modules such as OpenSAML [1][2][8].

3.4 Performance Benchmarking

Apache JMeter 5.6 was used to run a performance benchmarking suite that mimicked enterprise-scale traffic patterns. Every application was load-tested in the same way:

- **Users**: 1000 concurrent sessions
- **Duration**: 90 seconds sustained
- **Endpoints**: Auth, GET/POST for protected resources, refresh token

Metrics collected:

- **Authentication Latency (ms)**: Time from login request to token issuance or SAML response validation.
- **Average API Response Time (ms)**: For both secured and unsecured endpoints.

Volume 38 No. 5s, 2025

ISSN: 1311-1728 (printed version); ISSN: 1314-8060 (on-line version)

- **Throughput (req/sec)**: System-wide request processing rate.
- **Heap Memory Usage (MB)**: Monitored via VisualVM.
- **GC Pause Time (ms)**: Tracked for both **Parallel GC (Java 8 default)** and **G1GC/ZGC (Java 17)** configurations.

Performance comparison under various GC strategies and runtime improvements across Java versions was made possible by this multi-metric approach [14][16].

3.5 Evaluation Dimensions

The following dimensions were used to score and interpret results:

Dimension	Description
Architectural Complexity	Measured in design pattern density, configuration LOC (XML/YAML/Java), and modularization effort.
Migration Effort	Number of breaking changes encountered, dependency upgrades, and developer time invested.
Security Posture	Number of high/critical CVEs, number of passed OWASP checks, and code smell trends.
Performance Impact	Changes in latency, throughput, memory consumption, and garbage collection times.

In Section 4: Results and Analysis, these dimensions are thoroughly covered with the support of radar plots, bar charts, and differential performance tables.

4. Results and Analysis

The empirical results of assessing the three Java security stacks—OpenSAML, Spring Security OAuth2, and JWT-based authentication—across several software quality criteria are shown in this section. Real-world migration scenarios from Java 8 to Java 17 were used in the controlled evaluation. Runtime performance, vulnerability exposure, integration effort, and compatibility with Java versions are among the metrics.

An enterprise-grade reference application was used to implement each security stack, and it was put through the same functional workload and performance profile. This guarantees equality and separates the security mechanism's effect as the main differentiator.

4.1 Performance Metrics

Performance benchmarks were collected by simulating 500 concurrent clients interacting with secured endpoints using Apache JMeter 5.6 and the Java Microbenchmark Harness (JMH). Table 1 displays the relevant performance metrics.

Table 1: Performance Comparison Across Security Stacks

Metric	OpenSAML 5.1.4	Spring Security 6.2	JWT Auth (jjwt 0.11.5)
Average Auth Latency (ms)	124	92	31
Startup Time (sec)	8.7	5.9	4.2
Memory Usage (MB)	610	460	375

Observations:

- Because of its stateless, cache-friendly design and lack of XML parsing overhead, JWT-based authentication demonstrated the lowest latency and resource consumption [9].
- With native support for method-level security and Spring Boot optimizations improving startup efficiency, Spring Security demonstrated balanced performance.

- Despite its strength in identity federation, OpenSAML's processing of cryptographic assertions, metadata resolution, and XML schema validation resulted in high latency and memory usage [1][2].

These findings support the trade-off between runtime efficiency and security richness for example federated trust, particularly in enterprise deployments with limited resources.

4.2 Security Vulnerability Assessment

SonarQube, OWASP Dependency-Check, and manual CVE validation through the NIST NVD were used to conduct security assessments. Table 2 provides a summary of the comprehensive findings from these evaluations.

Table 2: Security Vulnerabilities Before and After Migration

Framework	Critical CVEs (Before)	Critical CVEs (After)	OWASP A10 Exposure
OpenSAML 3.x	5	1	Yes (XML Injection Risk)
Spring Security 5.3	3	0	No
JWT (Legacy Impl)	4	0	Yes (Token Forgery Risk)
JWT (Updated Impl)	-	0	No

Analysis:

- By switching from OpenSAML 3 to 5.1.4, known vulnerabilities were greatly reduced. Nevertheless, unless DEFLATE encoding and strict schema enforcement were specifically set up, residual XML injection risks continued [17].
- With integrated CSRF, CORS, and OAuth2 defenses, Spring Security 6.2 showed excellent OWASP compliance [18]. All known CVEs in critical dependencies (like Jackson and Spring Framework) were fixed after the migration.
- At first, JWT-based implementations exposed hardcoded secret risks and poor token validation. Token expiration policies, improved claim validation logic, and HMAC key rotation helped mitigate these [6][8].

This confirms that when updating security-critical libraries across Java versions, dependency hygiene, secure defaults, and migration-aware patching are crucial.

4.3 Maintainability and Integration Overhead

Using Lines of Code (LOC), Cyclomatic Complexity, and Maintainability Index (MI) through SonarQube, code maintainability and integration complexity were evaluated. Ten Java developers with four or more years of enterprise experience were given a survey to rate perceived effort on a 10-point scale. Table 3 displays the combined quantitative and qualitative results.

Table 3: Maintainability Metrics

Metric	OpenSAML	Spring Security	JWT Auth
Avg. LOC for Integration	~1,100	~800	~450
Configuration Complexity	9	7	3
Maintainability Index (MI)	57	71	84

Developer Insights:

- Because OpenSAML required a lot of XML setup, metadata bootstrapping, and low-level cryptographic handling, it was harder to maintain and took longer for new developers to get up and running.
- Because of filter chains, custom authentication providers, and token introspection, Spring Security had a steep learning curve despite its good modularity.
- Although cross-domain validation necessitated extra orchestration, JWT-based implementations were chosen for their clarity and simplicity, particularly in stateless microservices.

This reflects broader enterprise migration trends, particularly under DevSecOps workflows, toward minimal configuration surfaces and declarative security models [3][6].

4.4 Compatibility with Java Versions

Every security stack—OpenSAML, Spring Security, and JWT—was thoroughly tested for compatibility at both the build-time and run-time levels in the Java 8, Java 11, and Java 17 environments. Table 4 summarizes the findings and identifies runtime exceptions, version-specific problems, and deprecated API usage in the context of migration scenarios.

Table 4: Java Version Compatibility

Framework	Java 8	Java 11	Java 17
OpenSAML 3.x	✅	⚠️ Partial	❌
OpenSAML 5.1.4	✅	✅	✅
Spring Security 5.3	✅	✅	⚠️ Deprecated APIs
Spring Security 6.2	❌	✅	✅
JWT (jjwt 0.11.5)	✅	✅	✅

Observations:

- Outdated XML processing APIs and deprecated JAXP internals caused OpenSAML 3.x to fail under Java 17. On the other hand, OpenSAML 5.1.4 worked well with modular Java configurations and was completely compatible with JPMS.
- Java 11 was supported by Spring Security 5.3, but Java 17 generated warnings about deprecated APIs. This was fixed by upgrading to Spring Security 6.2, but because legacy support had been removed, there were some disruptive changes that got introduced.
- Because of the small dependency surface and lack of deep platform coupling, JWT implementations demonstrated complete compatibility across versions.

These results highlight how crucial proactive dependency audits and migration testing are when planning Java upgrades for enterprise systems that are sensitive in terms of security.

5. Discussion

The experimental results are interpreted in this section based on the real enterprise security procedures used in Java ecosystems. It highlights how secure, scalable modernization initiatives are driven by changing development paradigms, migration viability, and architectural trade-offs.

Volume 38 No. 5s, 2025

ISSN: 1311-1728 (printed version); ISSN: 1314-8060 (on-line version)

5.1 XML-Based Security vs Declarative Models

The findings support long-standing concerns about the cognitive and operational burden that XML-based security frameworks like OpenSAML impose. These systems require complex setups for digital signature verification, metadata trust anchors, and schema validation. A brittle integration surface is created by this complexity, where even small configuration errors can lead to vulnerabilities or runtime errors [19].

On the other hand, frameworks like JWT-based authentication and Spring Security facilitate annotation-driven declarative models like @EnableWebSecurity and @PreAuthorize, which greatly enhance the readability, maintainability, and testability of code. Declarative models reduce the need for verbose and prone to errors web.xml or Spring XML descriptors by enabling finer-grained access control embedded within business logic layers.

This change is in line with developer ergonomics and tooling advancements brought about by Java 11+ as well as more general trends in the modernization of the Java ecosystem. [4] [5].

5.2 Token-Based Authentication: Scalable but Security-Sensitive

Especially in stateless microservice environments, JWT-based security demonstrated excellent runtime performance with minimal integration overhead. However, JWT's effectiveness depends on rigorous operational discipline; its lightweight nature does not guarantee security by default [6][7].

Key considerations include:

- To stop misuse and token replay vulnerabilities like CVE-2015-9235, it is necessary to enforce token expiration, audience scoping, and signature rotation.
- Hardened mechanisms like Azure Key Vault, AWS KMS, or HashiCorp Vault should be used to centralize secret management. In earlier JWT deployments, breaches were often linked to hardcoded or poorly protected secrets [20].

The findings support the idea that, in contrast to frameworks like Spring Security, which provide centralized configuration and opinionated defaults for common security threats, JWT transfers security responsibility to the implementer, even though it can be extremely flexible and performant.

5.3 Incremental Framework Migration Is Practical

The concern about system-wide regressions is a significant barrier to updating legacy security layers. However, the study shows that in enterprise environments with layered architectures, incremental migration techniques are both feasible and desirable:

- Assertion parsers, metadata resolvers, and credential resolvers can be gradually replaced by OpenSAML upgrades that are enclosed within abstraction layers.
- Through SecurityFilterChain segregation and profile-based configurations, Spring Security modules can coexist, enabling rollback flexibility and gradual rollouts.
- Newer microservices allow for the selective introduction of JWT modules, allowing for phased adoption without requiring the refactoring of monolithic systems.

This layered transition model supports risk-aware modernization techniques that are encouraged in the CI/CD literature [21][22] and complies with continuous delivery principles.

5.4 Java 17 Enables Robust Security Posture—but Requires Preparation

Modern authentication architectures benefit from the security-focused improvements provided by the migration to Java 17, a long-term support release:

- JEP 329 offers enhanced key size enforcement and more robust default cryptography algorithms.

- Stricter encapsulation is introduced by the Java Platform Module System (JPMS), which prevents illegal reflective access, a common source of runtime vulnerabilities [23].
- Compact Strings and records are examples of memory optimizations that increase security and runtime efficiency by reducing the attack surface in serialization and deserialization steps.

These advantages are incompatible with the past, though. Older versions of Apache CXF or JAXB and legacy frameworks like OpenSAML 3.x frequently rely on internal or deprecated APIs (such as sun.security.x509.*), which can cause issues during migration. Therefore, before beginning any upgrade project, migration readiness assessments should be conducted, using tools such as jdeprscan and jdeps.

5.5 Broader Implications for Enterprise Security Strategy

According to the study's findings, a one-size-fits-all security strategy is undesirable and impractical. Rather, large-scale enterprises are increasingly adopting contextual hybridization of security frameworks as the standard:

- In heavily regulated or government-integrated systems that need to delegate trust to AD FS or other identity providers, federated identity protocols—like SAML 2.0 via OpenSAML—remain crucial.
- For external-facing APIs, microservices, and mobile integrations where statelessness and elasticity are crucial, OAuth2 and JWT-based solutions are becoming increasingly popular.
- Some enterprises use hybrid approaches, using SAML assertions for backend service federation and JWT tokens for frontend access control [22].

Multi-framework security validations, such as automated scans, secret validation, compatibility testing, and audit trails, must be incorporated into CI/CD pipelines in such environments. Deployment flows should incorporate tools like SonarQube Security Rules, Trivy, and OWASP Dependency-Check to guarantee uniformity and compliance.

6. Conclusion and Future Work

Three well-known Java-based security models—OpenSAML, Spring Security, and JWT-based authentication—were compared in this study in context with the development of the Java platform, with an emphasis on the switch from Java 8 to Java 17. For a thorough understanding of these frameworks' maintainability, vulnerability profile, and operational efficiency, the methodology included migration analysis, performance benchmarking, static and dynamic security auditing, and structured implementation.

The following important conclusions were drawn from the empirical findings:

- Compared to XML-based systems like OpenSAML, declarative, annotation-driven frameworks like Spring Security greatly increase maintainability and reduce human error. Additionally, they are more compatible with modern development processes such as microservice-first architectures and Spring Boot.
- JWT-based solutions need strict security discipline even though they are lightweight and high-performing. As demonstrated by several CVEs found in legacy implementations, poor token management, weak signing keys, and a lack of rotation policies can introduce serious and critical vulnerabilities.
- Significant improvements in language and runtime capabilities are provided by Java 17, such as JPMS-based modular access control, improved garbage collection (ZGC), and sealed classes. Leaner runtime behavior, better cryptographic defaults, and enhanced isolation are made possible by these features, but they also cause incompatibilities with older APIs and libraries, which calls for careful migration planning.
- The study reaffirms the importance of layered and incremental migration strategies backed by strong CI/CD pipelines. While progressively updating the Java runtime and the security stack in enterprise applications, this method guarantees the continuation of business operations.

6.1. Limitations

Although the approach was systematic and based on real-world concerns, this study has several limitations:

Volume 38 No. 5s, 2025

ISSN: 1311-1728 (printed version); ISSN: 1314-8060 (on-line version)

- Framework Scope: Only implementations based on JWT, OpenSAML, and Spring Security were compared. Despite their increasing use in some industries, security platforms like Quarkus Security, Apache Shiro, and Keycloak were not included.
- Controlled Environments: Performance and vulnerability data were gathered from single-tenant, controlled setups that might not accurately represent the intricacies of hybrid cloud deployments, federated identity integration, or multi-tenant systems where issues with data residency and cross-domain trust are important.
- Static tool limitations include the inability to identify dynamic runtime vulnerabilities, especially those brought on by reflection, runtime proxies, or improperly configured security filters, especially in frameworks like Spring Security, despite the widespread use of tools like SonarQube and OWASP Dependency-Check.

6.2. Future Work

Expanding upon this foundation, some possible directions for future research include:

- To capture a wider range of enterprise security requirements, the comparison should be extended to include other security frameworks like Keycloak (OAuth2/OpenID Connect), Apache Shiro, and Quarkus Security.
- With a focus on cross-protocol interoperability (SAML $\leftrightarrow$ OIDC), federated authentication is being evaluated in distributed cloud environments such as AWS Cognito, Azure AD, and GCP Identity.
- Creating CI/CD-integrated automated migration toolchains that identify deprecated APIs, enforce secure token procedures, and guide version-safe dependency upgrades.
- To evaluate real-world exploitability beyond what static tools can detect, security testing should be conducted in adversarial scenarios like replay attacks, simulated token injection, or XML signature wrapping.
- Framework adoption decisions can be influenced by quantitative developer usability studies that assess the cognitive complexity and onboarding friction between security configurations based on annotations and XML.

By providing both guidance and caution as enterprises adapt their security posture in together with platform upgrades, this study provides a useful, risk-informed basis for Java-based security modernization.

References

1. Hardt, D. (2012). The OAuth 2.0 Authorization Framework. IETF RFC 6749.
2. Cantor, S. (2016). OpenSAML Java Documentation. Shibboleth Consortium.
3. Li, N., Mitchell, J. C., & Winsborough, W. H. (2002). Design of a Role-Based Trust-Management Framework. IEEE Symposium on Security and Privacy.
4. Lodderstedt, T., McGloin, M., & Hunt, P. (2013). OAuth 2.0 Threat Model and Security Considerations. IETF RFC 6819.
5. Somorovsky, J. (2016). On the Insecurity of XML Security. USENIX Security Symposium.
6. Mainka, C., Mladenov, V., Schwenk, J., & Wich, T. (2016). Do not trust me: Using malicious IdPs for analyzing and attacking Single Sign-On. IEEE European Symposium on Security and Privacy.
7. Shibboleth Consortium. (2023). OpenSAML Java. Retrieved from https://shibboleth.net
8. Grochowicz, M. (2018). Understanding OpenSAML XML Objects. ACM SIGCSE.
9. Thönes, J. (2015). Microservices. IEEE Software, 32(1), 116-116.
10. Mosenia, A., Sur-Kolay, S., Raghunathan, A., & Jha, N. K. (2017). SAML Token Parsing Performance in Federated Networks. Elsevier Journal of Systems Architecture.
11. Oracle. (2021). Java EE Module Removal in JDK 11. https://openjdk.org/
12. Walls, C. (2018). Spring Security in Action. Manning Publications.
13. Mueller, J., et al. (2020). Secure Development with Spring Boot. Journal of Software Security Engineering.
14. Bezdicek, M. (2021). Misconfiguration pitfalls in Spring Security. OWASP AppSec Conference.
15. OWASP. (2023). Top 10 API Security Risks.
16. Oracle. (2023). Java 17 Enhancements. https://docs.oracle.com/en/java/javase/17/
17. Jones, M., Bradley, J., & Sakimura, N. (2015). JSON Web Token (JWT). IETF RFC 7519.

Volume 38 No. 5s, 2025

ISSN: 1311-1728 (printed version); ISSN: 1314-8060 (on-line version)

18. Bertoni, D., et al. (2019). A Comparative Study of SAML, OAuth and JWT for Secure Token-Based Authorization. IEEE Access.
19. Meyer, E., & Somorovsky, J. (2019). Attacking and Fixing JWT Implementations. USENIX WOOT.
20. OWASP. (2022). JWT Cheat Sheet. https://cheatsheetseries.owasp.org/
21. OWASP API Top 10 2023. https://owasp.org/www-project-api-security/
22. Liu, X., et al. (2019). Security and Performance Comparison of SAML vs JWT. ACM Symposium on Cloud Computing.
23. Wang, Y., & Chen, R. (2020). OAuth2 Evolution in Spring Security. Spring.IO Conference.
24. Mark Reinhold. (2017). The State of the Module System. OpenJDK JSR 376.