*i*Journals: International Journal of Software & Hardware Research in Engineering (IJSHRE)
ISSN-2347-4890
Volume 9 Issue 9 September 2021

Securing Java-Based Microservices: A Zero-Trust Architecture Using Spring Boot and Kubernetes

Author: Pradeep Kumar Shrivastav

Affiliation: Divergent Software Labs, Indore

ABSTRACT

Microservices improve modularity and independent deployment, but they also distribute application functionality across many network-reachable components and create additional trust boundaries. This paper synthesizes the security architecture proposed in Divergent Software Labs' "Microservices Security Architecture and Design — Java and Spring Boot" specification. It examines a layered, zero-trust approach for protecting business APIs deployed on Kubernetes, with particular emphasis on identity validation, service-level authorisation, mutual TLS, secrets management, container hardening, secure software delivery, and operational monitoring. The reference design uses Java 17, Spring Boot 2.5.7, Spring Security 5.5.3, Spring Cloud 2020.0.4, and Kubernetes 1.22, reflecting a November 2021 technology baseline. A threat-driven method based on STRIDE and OWASP risks is connected to practical controls and measurable verification activities. The paper argues that security is strongest when gateways provide broad protective controls while each service remains responsible for decisions about its own data. It concludes with implementation priorities and limitations that organisations must reassess before adoption.

Keywords: microservices security, zero trust, Spring Boot, OAuth 2.0, OpenID Connect, JWT, Kubernetes, mutual TLS, DevSecOps

1. INTRODUCTION

Microservices architecture decomposes an application into independently deployable services that communicate over network interfaces. This model can improve scalability, team autonomy, and release flexibility, but it replaces a comparatively concentrated trust boundary with multiple service-to-service and infrastructure interactions. A weakness in one endpoint, workload, dependency, or deployment pipeline may become a route to other components if identity and access controls are inconsistent. NIST's guidance on microservices identifies authentication, access management, secure communication, service discovery, and monitoring as core security concerns [1], while its zero-trust model rejects trust based only on network location [4].

The source specification applies these principles to an illustrative e-commerce domain containing an Order Service and a Payment Service. It defines controls for internet-facing requests, internal calls, data stores, secrets, Kubernetes workloads, and the software delivery pipeline. This paper condenses that design into a research-oriented discussion: the problem and objectives are stated, the architecture and its controls are analysed, and a verification and implementation approach is presented. The analysis remains anchored to the supplied document and its November 2021 baseline; the versions named should not be interpreted as a recommendation to deploy unsupported software today.

2. RESEARCH AIM AND METHOD

The aim is to explain how a layered security architecture can reduce the likelihood and impact of common attacks against Java-based microservices. The paper uses a design-analysis method rather than reporting a new empirical experiment. It reviews the supplied specification's security objectives, trust boundaries, threat catalogue, architecture

decisions, implementation guidance, test plan, risk assessment, and roadmap. The document's threats were identified using STRIDE across trust boundaries and cross-checked against the OWASP API Security Top 10 (2019) and OWASP Top 10 (2021) [18][19].

The design has six stated objectives: reject unauthenticated business requests; ensure users can access only permitted actions and objects; prevent a compromised service from automatically reaching other services; protect confidential data and secrets; permit only reviewed and scanned production artefacts; and detect, attribute, and respond to security-relevant events. These objectives are useful because each is tied to verification evidence rather than being expressed only as a general aspiration.

3. THREAT MODEL AND SECURITY CHALLENGES

The specification assumes an adversary can submit arbitrary internet requests, may possess a legitimate low-privilege account, and may eventually compromise a single pod through a vulnerable dependency. The protected assets include customer and order data, payment operations, identity and cryptographic credentials, the build and deployment pipeline, and audit logs. The design identifies seven important trust boundaries: internet to edge, client to identity provider, gateway to services, service to service, service to data stores or Vault, pipeline to registry and cluster, and workload to Kubernetes API or node.

The threat catalogue covers token forgery and replay, broken object-level authorisation (BOLA/IDOR), function-level privilege escalation, injection and mass assignment, excessive data exposure, denial of service, lateral movement, server-side request forgery (SSRF), secret leakage, vulnerable or malicious dependencies, and insufficient auditability. These threats demonstrate that authentication alone is not sufficient. A valid token can still be used to request another customer's order if the service fails to check ownership. Similarly, a protected external API can remain vulnerable if internal network paths permit broad lateral movement.

A key implication is that controls should be independent and layered. The source specification maps threats to controls C1–C11 and notes that most threats receive more than one mitigating control. BOLA is a notable exception: only the service that owns the data can authoritatively determine whether a principal may access a particular object. Therefore, object-level checks must be implemented and tested inside the relevant service.

4. PROPOSED SECURITY ARCHITECTURE

The reference design separates responsibilities among the edge, API gateway, identity provider, application services, data stores, secrets infrastructure, Kubernetes platform, and monitoring system. A load balancer or web application firewall provides TLS termination, request normalisation, coarse rate and size limits, and volumetric denial-of-service protection. The Spring Cloud Gateway provides route allow-lists, preliminary JWT checks, per-principal rate limiting, request-size controls, removal of spoofable identity headers, and protected upstream connections.

The gateway is not treated as the final authority for business access. Each Spring Boot service independently validates the token and makes endpoint- and object-level authorisation decisions. Data stores use encryption and service-specific, least-privilege credentials. Vault and an internal certificate authority support secret storage, dynamic database credentials, short-lived certificates, and auditing of secret access. Kubernetes provides workload isolation, network segmentation, role-based access control, admission controls, and encryption of secrets at rest. A central logging and security information and event management (SIEM) capability supports correlation, alerting, and forensics.

This division of responsibilities reflects a defence-in-depth principle: the gateway reduces exposure and applies consistent cross-cutting rules, while services enforce rules that require business context. It also limits the consequences of gateway bypass or misconfiguration. Network controls remain valuable, but they supplement rather than replace identity-based authorisation.

© 2021, *i*Journals All Rights Reserved

5. IDENTITY, AUTHENTICATION, AND AUTHORISATION

5.1 User and workload identity

The specification uses OpenID Connect (OIDC) for user authentication and OAuth 2.0 for delegated API access. Public browser and mobile clients use the authorisation-code flow with Proof Key for Code Exchange (PKCE), while the implicit and resource-owner password flows are excluded. The identity tier is expected to support multi-factor authentication for workforce users and high-risk customer operations, exact redirect URI matching, state and nonce validation, refresh-token rotation where applicable, and signing-key rotation.

Access tokens are short-lived signed JWTs, with the source baseline recommending a five-to-fifteen-minute lifetime and an audience restricted to the intended API. Each resource server validates the expected signing algorithm, signature, issuer, audience, expiry and not-before claims, and applicable token type, scopes, and roles. The design warns against accepting a token simply because it parses or has a valid signature. Audience validation prevents a token intended for one API from being replayed at another; issuer validation prevents acceptance of tokens from an untrusted identity realm.

5.2 Fine-grained authorisation

Authorisation combines delegated scopes, organisational roles, and object attributes such as record ownership. Every endpoint must have an explicit rule, and denial is the default. Roles and scopes can determine whether a user may invoke an operation, but they do not prove that the user owns the requested record. For example, an orders.read scope must not permit a customer to change an order identifier and retrieve another customer's order. The service should scope data access to the authenticated principal or perform an equivalent ownership check before returning or modifying the object [18].

5.3 Service-to-service trust

Internal service calls use mutual TLS (mTLS) with certificates issued by an internal CA, together with OAuth 2.0 client-credentials tokens scoped and audience-restricted for the callee. This creates two complementary signals: the certificate authenticates the workload connection, and the access token conveys the caller's application-level authority. Kubernetes NetworkPolicies then deny traffic that is not explicitly permitted. The design prefers service identity and least privilege over broad trust based on namespace or IP address.

6. APPLICATION, PLATFORM, AND DELIVERY CONTROLS

At application level, the source design recommends strict input validation, DTO allow-listing, parameterised data access, safe error responses, response filtering, and restricted management endpoints. These controls reduce injection, mass assignment, information disclosure, and accidental exposure of operational details. SSRF risk is addressed through URL validation and egress restrictions rather than trusting user-supplied destinations.

For transport and data protection, TLS 1.2 is the minimum and TLS 1.3 is preferred in the source baseline [16][22]. Sensitive data should be encrypted at rest, and secrets must not be committed to source control, embedded in container images, or exposed through logs. Runtime retrieval from Vault and dynamic database credentials reduce the time and scope of credential exposure. Because Vault becomes a critical dependency, availability, access auditing, and controlled emergency procedures must be planned.

Container and Kubernetes controls include non-root execution, read-only root filesystems, restricted pod privileges, least-privilege service accounts, NetworkPolicies, image scanning, admission checks, and controlled access to the Kubernetes API. The design also treats the software supply chain as part of the security boundary. A gated CI/CD pipeline should run static application security testing, dependency and container-image scanning, generate a software bill of materials (SBOM), sign images, and prevent unapproved artefacts from reaching production. This approach aligns with secure software development practices described by NIST [26].

© 2021, *i*Journals All Rights Reserved

Logging and incident response complete the preventive controls. Structured security events should support attribution and correlation without logging access tokens, passwords, private keys, or unnecessary personal data. Alerts should feed a SIEM and a documented response process, informed by NIST's incident-handling guidance [24][25]. Monitoring does not prevent an attack by itself; it makes suspicious activity more visible and supports investigation and recovery.

7. VERIFICATION, RISKS, AND IMPLEMENTATION ROADMAP

The specification emphasises negative security testing: tests should prove that invalid requests fail, not merely that valid workflows succeed. The test plan includes missing, expired, malformed, wrongly signed, wrong-issuer, and wrong-audience tokens; missing scopes and roles; cross-user object access; disallowed service-to-service calls; invalid input and mass assignment; excessive request sizes; secret scanning; and rejection of vulnerable or unsigned deployment artefacts. Acceptance criteria include all non-health business endpoints rejecting requests without valid tokens, passing object-ownership tests, and providing evidence that unauthorised east-west traffic is blocked.

The risk assessment acknowledges residual risks. Vulnerabilities continue to emerge in dependencies, so the response depends on SBOM-assisted identification, rapid patching, and rebuilds. The identity provider is a central trust anchor and potential single point of failure. Self-contained JWTs also create a revocation delay: a stolen token may remain usable until expiry. Short lifetimes reduce this window, while token introspection may be reserved for high-risk operations. Network segmentation depends on a Kubernetes CNI that actually enforces NetworkPolicy, and certificate rotation requires operational discipline.

The implementation roadmap is intended for a platform team and two to four service teams over approximately five months, subject to actual capacity. A practical sequence is to establish identity and gateway foundations; implement service-level token and object checks; deploy mTLS, Vault, and network segmentation; harden containers and CI/CD; and then complete security testing, monitoring, incident exercises, and risk-owner acceptance. Each phase should have measurable exit criteria, assigned owners, and evidence retained for review.

8. DISCUSSION

The design's main strength is its explicit allocation of security responsibility. Central controls are useful for consistent routing, rate limiting, and early token checks, but the application service remains accountable for decisions involving its data. The approach also recognises that a service may be compromised and therefore limits lateral movement through independent identity, network, and workload controls. Finally, it integrates software delivery and operations into the security model rather than treating security as a one-time code review.

The principal limitation is that the specification is a reference design, not an empirical evaluation. Its implementation examples were reviewed for consistency but were not compiled or executed in the preparation environment, according to the source document. The security outcomes therefore depend on adoption quality, test coverage, operational maturity, and the validity of stated assumptions. The source versions are also a historical baseline from November 2021. Any real deployment must select currently supported releases and revalidate compatibility, configuration, and security guidance before implementation.

9. CONCLUSION

A secure microservices environment requires more than a gateway and a login mechanism. It requires identity validation at every hop, explicit service-level authorisation, object ownership checks, authenticated internal communication, least-privilege workloads, protected secrets, controlled software delivery, and observable operations. The Java/Spring Boot and Kubernetes architecture described in the source document offers a coherent way to organise these controls around a zero-trust principle: no request is trusted solely because of its origin or because it passed an upstream component. Its value lies in making security requirements testable and in reducing the blast radius when an

© 2021, *i*Journals All Rights Reserved

individual control or service fails. The design should be adopted only after its assumptions are confirmed and its historical technology baseline is updated to supported versions.

REFERENCES

[1] R. Chandramouli, Security Strategies for Microservices-based Application Systems, NIST SP 800-204, 2019. https://doi.org/10.6028/NIST.SP.800-204

[2] R. Chandramouli and Z. Butcher, Building Secure Microservices-based Applications Using Service-Mesh Architecture, NIST SP 800-204A, 2020. https://doi.org/10.6028/NIST.SP.800-204A

[3] S. Rose et al., Zero Trust Architecture, NIST SP 800-207, 2020. https://doi.org/10.6028/NIST.SP.800-207

[4] M. Souppaya, J. Morello, and K. Scarfone, Application Container Security Guide, NIST SP 800-190, 2017. https://doi.org/10.6028/NIST.SP.800-190

[5] D. Hardt (Ed.), The OAuth 2.0 Authorization Framework, RFC 6749, IETF, 2012. https://www.rfc-editor.org/rfc/rfc6749.html

[6] M. Jones, J. Bradley, and N. Sakimura, JSON Web Token (JWT), RFC 7519, IETF, 2015. https://www.rfc-editor.org/rfc/rfc7519.html

[7] N. Sakimura, J. Bradley, and N. Agarwal, Proof Key for Code Exchange by OAuth Public Clients, RFC 7636, IETF, 2015. https://www.rfc-editor.org/rfc/rfc7636.html

[8] B. Campbell et al., OAuth 2.0 Mutual-TLS Client Authentication and Certificate-Bound Access Tokens, RFC 8705, IETF, 2020. https://www.rfc-editor.org/rfc/rfc8705.html

[9] OWASP Foundation, OWASP API Security Top 10 — 2019, 2019. https://api-security.owasp.org/editions/2019/en/0x00-toc/

[10] P. Cichonski et al., Computer Security Incident Handling Guide, NIST SP 800-61 Rev. 2, 2012. https://doi.org/10.6028/NIST.SP.800-61r2

[11] D. Dodson, M. Souppaya, and K. Scarfone, Mitigating the Risk of Software Vulnerabilities by Adopting a Secure Software Development Framework, NIST Cybersecurity White Paper, 2020. https://doi.org/10.6028/NIST.CSWP.04232020

[12] P. K. Shrivastav, Microservices Security Architecture and Design — Java and Spring Boot, Divergent Software Labs, Version 1.0, 30 November 2021 (source document).

© 2021, *i*Journals All Rights Reserved