IJARSCT
International Journal of Advanced Research in Science, Communication and Technology
International Open-Access, Double-Blind, Peer-Reviewed, Refereed, Multidisciplinary Online Journal
ISSN: 2581-9429
Volume 6, Issue 11, April 2026
Impact Factor: 8.2

Software Supply Chain Security: From Dependency Vulnerabilities to Secure CI/CD and Runtime Threat Mitigation

Prof. Anuja Phapale[1], Pranav Kubal[2], Ritesh Korade[3], Chinmay Kalokhe[4]
Professor, Department of Information Technology[1]
Student, Department of Information Technology[2,3,4]
AISSMS Institute of Information Technology, Pune, Maharashtra, India

Abstract: *The process of software development currently depends extensively on third-party software libraries and automated pipelines used for delivering software to production environments. This approach, although effective in boosting software development processes, is increasingly posing a threat in the sense that most organizations are still unable to cope with the associated risks. In this regard, we analyze software supply chain security by focusing on three key aspects of the problem: propagation of vulnerabilities within dependency trees, development of secure CI/CD pipelines, and detection of threats during live operations. The literature review provides insights into the attack landscape of SSCS, which include dependency confusion, typosquatting, package tampering, and build system compromise. Based on these insights, we identify appropriate defense mechanisms for every attack class along with the governance structures in place, namely SLSA, NIST SP 800-204D, and OWASP SCVS. Building on the results of our research, we propose a new SSC security architecture – SSC-Shield – which includes an automated approach to software bill of materials, vulnerability scanning, secure CI/CD pipeline deployment, and machine learning-driven runtime monitoring. Evaluation showed that the approach is able to reduce average vulnerability exposure window time by 60.9%...*

Keywords: Software Supply Chain Security, Dependency Vulnerabilities, CI/CD Pipeline Security, DevSecOps, SLSA, SBOM, STRIDE, Zero Trust, Provenance Attestation

I. INTRODUCTION

Modern software is rarely developed as an independent effort. For example, an enterprise application typically relies on hundreds, if not thousands, of external libraries stored in public repositories like npm, PyPI, Maven Central, and RubyGems. According to a Synopsys study published in 2024, over 96% of commercial software codebases use some pieces of open-source software, while more than 84% are likely to have at least one security issue within them. This approach speeds up the development process but carries the full security history of all used components, their developers, and their repository systems.

The concept of the software supply chain (SSC) refers to all the elements and processes involved in the creation and distribution of software products. It includes the source code, external dependencies, build systems, compilers, and distribution pipelines. In this respect, any breach in the supply chain can be easily distributed along with each subsequent release of every dependent product.

The problem is real. As shown by the SolarWinds supply chain attacks in 2020, a compromised build process can inject a backdoor into 18,000 downstream customers, including multiple federal departments. In 2021, the Log4Shell flaw impacted virtually every Java-based product because of transitively included third-party libraries. XZ Utils is another example from 2024 when a sophisticated attacker spent several years building trust within an open-source ecosystem prior to carrying out the attack.

DOI: 10.48175/IJARSCT-36610

IJARSCT
International Journal of Advanced Research in Science, Communication and Technology
International Open-Access, Double-Blind, Peer-Reviewed, Refereed, Multidisciplinary Online Journal
ISSN: 2581-9429
Volume 6, Issue 11, April 2026
Impact Factor: 8.2

Nevertheless, there is still a noticeable disparity between the sophistication of the attacks and the level of defence implemented by businesses. Existing literature focuses on either one component of the attack process, or a combination of them, but does not cover the complete integration of all the measures into a coherent lifecycle-wide approach.

II. BACKGROUND AND RELATED WORK

A. The Dependency Risk Landscape

Decan et al. examined more than 1.4 million npm packages within eight years and discovered that around 7.5% of RubyGems and 6.93% of npm packages have a single known vulnerability. It should be noted that the impact of the transitive dependencies is non-linear because a seemingly harmless package can have critical vulnerabilities hidden deep in its dependencies. Until recently, most scanning tools focused on analysing direct dependencies; thus, these hidden vulnerabilities remained unnoticed.

According to Zimmermann et al., one malicious npm package can affect as much as 52% of the whole package graph due to high levels of interdependence. Such a system allows a malicious agent who gains control of one highly connected node to cause an enormous amount of harm in an effort disproportionately small. Moreover, research demonstrates that the median delay in applying a security fix can last for 80 days; however, such delays can be reduced to a mere 12 days through Dependabot.

B. Supply Chain Attack Categories

Gokkaya et al. offer the most comprehensive classification to date, identifying nine major attack categories. Table I summarises the six most prevalent types alongside their MITRE ATT&CK mappings and severity assessments.

Table 1: Software Supply Chain Attack Taxonomy

Attack Vector	Target	ATT&CK ID	Severity
Dependency Confusion	Package Registry	T1195.001	Critical
Typosquatting	Developer Intent	T1195.001	High
Malicious Pkg Injection	Open-Source Repo	T1195.002	Critical
Build System Compromise	CI/CD Infrastructure	T1195.003	Critical
Artifact Tampering	Binary/Artifact	T1553.002	High
Maintainer Hijacking	Identity/Account	T1078	Critical

Dependency confusion makes use of misconfigurations in the resolution process for determining the source from which to download a package between the public registry and the private one, and this vulnerability was exploited by Birsan successfully against 35 large companies, such as Apple, Microsoft, and PayPal, without resorting to any type of malware. In typosquatting, the attackers make use of developers' typos when naming their packages, and in 2023, there were more than 5,000 possible targets in the PyPI repository.

C. Governance Frameworks

Governance of risks within supply chains is covered by ISO 28000 and NIST SP 800-161r1 in broad terms. As far as frameworks useful for software engineers go, the best options include SLSA, OpenSSF Scorecard, and OWASP SCVS. The framework NIST SP 800-204D focuses on DevSecOps scenarios, prioritizing automation and code for provenance and policies.

IJARSCT
International Journal of Advanced Research in Science, Communication and Technology
International Open-Access, Double-Blind, Peer-Reviewed, Refereed, Multidisciplinary Online Journal
ISSN: 2581-9429
Volume 6, Issue 11, April 2026
Impact Factor: 8.2

III. DEPENDENCY VULNERABILITIES AND MITIGATION

A. How Risk Propagates

Dependency risk has two dimensions. First, direct risk involves a vulnerability in a library that is directly imported into the application and is measured by its CVSS score. Second, transitive risk entails a vulnerability in libraries of libraries, which may be incorporated into the application without developers knowing they are there. This second kind of risk was demonstrated perfectly in the case of the Log4Shell vulnerability, where the flaw was hidden deep down within various third-party applications as well as cloud-based solutions

B. Key Structural Challenges

Dependency conflicts caused by diamonds occur when two different libraries use different versions of the same library, leading to the need for the package manager to choose between them. If the package manager chooses a particular version, the patched security bug might go unnoticed and be present in the final software. In addition, the processing delays in the NVD database mean that in 2024, at least 4,000 vulnerabilities had not been reviewed in full even after 90 days since their discovery - a problem for any tool only using this database. Phantom dependencies and abandoned dependencies increase the risks.

C. Mitigation Strategies

Bills of Materials for software (SBOM) serve as the cornerstone for secure dependencies. U.S. President Biden's Executive Order 14028 required all software purchased for federal agencies to be delivered with a SBOM since visibility was a must to gain control over the situation. There are mainly two SBOM formats in use: SPDX (the ISO/IEC 5962:2021 standard) and CycloneDX, which has a built-in Vulnerability Exploitability eXchange (VEX) extension provided by OWASP, giving additional information about whether the certain CVE is really exploitable on this particular product. Multi-source scanning services checking NVD, OSV, GitHub Advisory Database and Snyk at the same time lower the false negatives rate by 23%.

IV. SECURE CI/CD PIPELINE DESIGN

A. DevSecOps Integration

According to Singh, the cost of incidents is lowered by 47 percent and the number of vulnerabilities detected after deployment decreases by 63 percent when security is incorporated into every phase of the pipeline. This makes economic sense since the cost of fixing a vulnerability discovered during development is much lower than that of fixing a vulnerability after deployment. Table II shows the phases of development mapped to the security controls and associated tools used.

Table 2: DevSecOps Pipeline Security Controls

Stage	Security Control	Toolchain
Code Commit	Pre-commit hooks, secret detection	git-secrets, gitleaks
Build	SAST, SCA, IaC scanning	Semgrep, Snyk, Checkov
Test	DAST, fuzzing, IAST	OWASP ZAP, AFL, Contrast
Artifact	Signing, SBOM generation	Sigstore, Syft, Grype
Deploy	Policy-as-code, admission control	OPA/Gatekeeper, Kyverno
Monitor	Runtime telemetry, threat intel	Falco, Prometheus, SIEM

B. STRIDE Threat Analysis of CI/CD Pipelines

When the STRIDE framework is applied to the elements in a software development pipeline, the build server becomes the riskiest phase. By compromising it, the attacker gains control and is able to tamper with any build produced by the

IJARSCT
International Journal of Advanced Research in Science, Communication and Technology
International Open-Access, Double-Blind, Peer-Reviewed, Refereed, Multidisciplinary Online Journal
ISSN: 2581-9429
Volume 6, Issue 11, April 2026
Impact Factor: 8.2

system, precisely as happened during the attacks on SolarWinds and 3CX. A study of 50,000 GitHub Actions workflows concluded that 12.8% of them were at risk from injection of arbitrary expression via user-provided input, making it possible for pull requests to steal sensitive data undetected.

Table 3: STRIDE Threat Analysis of CI/CD Pipeline Components

Component	STRIDE	Primary Threat
Source Repository	S, T	Unauthorised branch push, force-push
Build Server	T, E, D	Pipeline injection, secret leakage
Artifact Store	T, R	Artifact substitution, unsigned binaries
Deploy Env	E, D	Privilege escalation, rollback attacks
Secrets Manager	S, I	Credential theft, token rotation failures

C. The SLSA Framework

The SLSA framework provides four progressive levels for assurance on the provenance, integrity, and reproducibility of software artifacts. At level one, there is creation of the provenance metadata by the build system. At level two, the provenance metadata is digitally signed using the build service. At level three, there is a clean build environment for every build process. At level four, there is deterministic building requiring dual reviews. Table 4 summarizes the requirements at each level of the SLSA and the associated threat class.

Table 4: SLSA Maturity Levels

L	Description	Requirements	Mitigates
1	Build Integrity	Provenance exists	No source attestation
2	Authenticated Provenance	Signed provenance, hosted build	Forged provenance
3	Hardened Builds	Isolated, ephemeral builds	Compromised build host
4	Reproducible Builds	Hermetic, two-party review	Insider threats

D. NIST SP 800-204D Guidance

Chandramouli et al. categorize pipeline governance strategies into four categories: (1) provenance and integrity by means of cryptography signing and tamper-evident logging; (2) policy enforcement through policy-as-code gates in technologies such as the Open Policy Agent framework; (3) secrets management employing ephemeral dynamic credentials without any established privileges; and (4) continual verification by automating the CIS Benchmark and STIG compliance testing process at each build cycle.

V. RUNTIME THREAT MITIGATION

A. Why Build-Time Controls Are Insufficient

Nevertheless, even stringent controls implemented at the time of building are not sufficient when the program is deployed. Hackers can take advantage of logical vulnerabilities that emerge only when specific runtime environments are reached, use harmless-looking system calls to achieve malicious ends, or utilize zero-day exploits against which no fix currently exists. According to the 2023 Falco Cloud Native Threat Report, one-third of all breaches detected on cloud environments have taken advantage of the runtime behavior of an application.

IJARSCT
International Journal of Advanced Research in Science, Communication and Technology
International Open-Access, Double-Blind, Peer-Reviewed, Refereed, Multidisciplinary Online Journal
ISSN: 2581-9429
Volume 6, Issue 11, April 2026
Impact Factor: 8.2

B. Runtime Detection Techniques

eBPF (Extended Berkeley Packet Filter) facilitates efficient kernel-based monitoring of system calls without making changes to the program code. Falco and Tetragon, for instance, leverage eBPF technology to establish a behavioural baseline for each container, using an observation window before beginning the comparison of current behaviour to the baseline. Since the detection process relies on behavior rather than signatures, this kind of monitoring detects novel attacks and zero-days that could not be detected by signature-based approaches.

The Graph Neural Network model has shown its potential in detecting anomalies in microservices. By modelling the communication between services as a time-varying graph, the GNN model, trained on historic data, predicts the likelihood of any given edge existing in the graph. If the probability of a certain edge being present deviates from the predicted value by more than three standard deviations, the model flags an alert. It reached a precision score of 94.3% and a recall score of 91.7%, outperforming threshold-based methods on the DARPA OpTC dataset.

RASP agents inject directly into the code itself and observe all the calls to security APIs at run-time. Since it has full knowledge of the call stack and also sees the data being manipulated, RASP is able to detect SQL injections, path traversal, and deserialization attacks, even if they have been obfuscated for bypassing the network-level detection mechanisms.

In the container and Kubernetes context, some important security controls to harden the systems include seccomp profiles (whitelist of system calls which the container will make, around 50–80 out of around 400 calls), mandatory access controls using AppArmor or SELinux, Pod Security Standards requiring non-root containers and read-only root file systems, and network policies for micro-segmentation.

VI. SSC-SHIELD: A PROPOSED INTEGRATED FRAMEWORK

A. Design Principles

The SSC-Shield architecture is founded upon four core pillars. Lifecycle Coverage refers to how security policies will cover all lifecycle stages of software development without treating any lifecycle stage as being intrinsically secure. Automation First ensures that automation and enforcement of security controls is integrated into standard development processes, leaving human intervention as a last resort for any exceptions. Evidence-Based Trust entails validating artifacts or dependencies through cryptography rather than relying on any preconceived notion of their safety.

B. Framework Architecture

SSC-Shield consists of four interconnected layers targeting distinct phases of the supply chain lifecycle. Table V summarises the layers and their key components.

Table 5: SSC – Shield Framework Layers

L	Layer	Key Components
1	Dependency Security	SBOM (CycloneDX), multi-source scanning, safe-config solver
2	CI/CD Security	SLSA L3, signed artifacts, STRIDE-mitigated gates, secrets mgmt
3	Runtime Security	eBPF monitoring, GNN anomaly detection, RASP, container hardening
4	Governance	Policy-as-code (OPA), continuous risk assessment, SBOM lifecycle mgmt

Layer 1 (Dependency Security) is triggered in every dependency resolution. An SBOM is created automatically in CycloneDX, multi-source CVE scanning is performed on the SBOM, and all packages within a vulnerable version range are highlighted with severity ratings along with suggested upgrades. Layer 2 (CI/CD Security) includes guarantees of provenance and integrity throughout the entire process of building, including signed commits, ephemeral

IJARSCT
International Journal of Advanced Research in Science, Communication and Technology
International Open-Access, Double-Blind, Peer-Reviewed, Refereed, Multidisciplinary Online Journal
ISSN: 2581-9429
Volume 6, Issue 11, April 2026
Impact Factor: 8.2

and secure build environments according to the SLSA Level 3 standard, signature-less artifact signing using Sigstore, and the use of dynamically provided secrets. Layer 3 (Runtime Security) uses eBPF profiling to detect deviations in the behavior of each workload, complemented with GNN-based detection and RASP instrumentation. Layer 4 (Governance) ensures visibility across all layers with a policy engine based on OPA, a continuous risk dashboard integrating risk data from all operational layers, and SBOM management tracking the history of artifacts' provenance and vulnerabilities throughout their lifetime.

C. Enforcement Procedure

Prior to the deployment of any artifact, three checks are performed by SSC-Shield on it before moving to the next step: vulnerability scanning based on SBOM using multiple sources, attestation verification based on the configured trust store, and finally the evaluation of OPA policy. Any failed check means deployment does not proceed and there is a report about the incident. On success of the above steps, deployment takes place with an eBPF monitoring tool that monitors run-time behavior and detects and isolates any deviation from the baseline.

VII. EVALUATION

A. Setup and Results

The SSC-Shield technique was tested on three open-source project profiles having varying complexities of dependency structures, such as Java Spring Boot Micro-service Project (having 340 direct dependencies), Node.js REST API (having 1,247 direct dependencies), and Python ML Pipeline (having 82 direct dependencies). The analysis was performed for these projects both with and without the help of SSC-Shield using the vulnerability information obtained from OSV in Q1 2025. It was found that the use of SSC-Shield helped to reduce the average exposure time of vulnerabilities from 98 days to 38.3 days - an improvement of 60.9%.

Table 6: SSC – Shield v/s Baseline (Key Metrics)

Metric	A-Base	A-Shield	B-Base	B-Shield
Known CVEs detected	47	47	183	183
CVEs auto-remediated	0	39	0	147
Critical unpatched	12	2	31	5
Mean exposure (days)	91	34	104	41
False positive rate	—	4.1%	—	5.7%

The 4-6% false-positive rate associated with automated remediation aligns well with previous empirical research conducted on Dependabot. The eBPF-based runtime monitor was able to detect three anomalies in the experiments that included dependency confusion injection, container escape, and unusual DNS exfiltration. There were no false negatives for any attack injected.

B. Performance Overhead

The eBPF monitoring mechanism generated median CPU overhead of 1.7% and median memory overhead of 43 MB per node, which comfortably remained within reasonable limits for operational use. The latency in build pipeline for each release saw an increase of about 87 seconds on account of SCA scanning and generation of SLSA attestations. However, this overhead was more than compensated for by the removal of manual security review process which consumed about 6.4 hours for every release cycle.

IJARSCT
International Journal of Advanced Research in Science, Communication and Technology
International Open-Access, Double-Blind, Peer-Reviewed, Refereed, Multidisciplinary Online Journal
ISSN: 2581-9429
Volume 6, Issue 11, April 2026
Impact Factor: 8.2

VIII. CHALLENGES AND FUTURE DIRECTIONS

A. Open Challenges

There are three major structural challenges that are difficult to solve. First, SPDX, CycloneDX, SLSA, OWASP SCVS, and OpenSSF Scorecard each solve a piece of the supply chain puzzle but via distinct data models and toolchain integration techniques. So far, there is not any universal format for sharing SBOMs that is fully integrated into all package registries. Secondly, continuously analyzing risks and preventing vulnerabilities while minimizing developer overhead means conducting incremental graph analysis - that is, making changes only to parts of the dependency tree that are relevant to a newly published advisory - something that is the subject of ongoing research. Finally, there seems to be no clear solution yet to balancing rigorous pipeline security against delivery speed.

B. Future Research Directions

A blockchain-backed provenance approach using build attestations on an immutable blockchain would result in transparency and tampering-resistance of supply chain histories. This was found to be possible at gas costs of only $0.003 per attestation on the Polygon network, thus ruling out cost considerations. Zero trust pipeline identities based on SPIFFE/SPIRE frameworks would address lateral movement opportunities for attackers, thanks to the privilege removal aspect. AI-powered vulnerability prediction, which takes into account code churns, author networks, and activity on issue trackers to identify vulnerabilities even before disclosure of CVEs, has been successfully tested on held-out data sets with AUC greater than 0.83. Autonomous patch and redeploy processes, initiated automatically upon disclosure of CVEs and regression tests selected using artificial intelligence, would define the future of autonomous security.

IX. CONCLUSION

In summary, software supply chain security has emerged as one of the most pressing issues in contemporary software engineering. The interplay between open-source environments, automation pipelines, and cloud-based infrastructures has generated a security landscape of immense scope and complexity, which experienced adversaries take advantage of, recognizing that compromising just one supplier can have disastrous consequences on thousands of consumers.

This work has made multiple tangible contributions to addressing this issue: a model for vulnerability propagation via transitive dependencies that takes into account non-linearity; an attack taxonomy annotated according to the MITRE ATT&CK framework that covers passive attacks on infrastructure and prolonged social engineering; a threat matrix based on the STRIDE methodology, pinpointing build servers as a prime target for attacks; a comparative assessment of SLSA, NIST SP 800-204D, and OWASP SCVS; and finally, SSC-Shield, a multi-layered framework that yielded a 60.9% decrease in vulnerability exposure time at minimal performance costs.

The secure delivery of software necessitates more than merely upgrading tools. It entails embracing security engineering as a fundamental aspect of the software lifecycle, automating and monitoring it, providing empirical evidence, and foregoing development speed for the sake of security.

REFERENCES

[1] S. Anasuri, "Secure software supply chains in open-source ecosystems: Governance frameworks and practical approaches," arXiv preprint arXiv:2312.04782, 2023.

[2] R. Chandramouli, M. Bartock, B. Grayson, and D. Sutton, "Strategies for the integration of software supply chain security in DevSecOps CI/CD pipelines," NIST Special Publication 800-204D, NIST, Gaithersburg, MD, 2024.

[3] S. Dhandapani, "STRIDE-based threat modelling of CI/CD pipeline components for software supply chain security," in Proc. IEEE ICSSA, 2025, pp. 112-121.

[4] B. Gokkaya, L. Ma, and E. Bodden, "Software supply chain: A comprehensive taxonomy of attacks, mitigation strategies, and future directions," ACM Comput. Surv., vol. 57, no. 4, 2025.

IJARSCT
International Journal of Advanced Research in Science, Communication and Technology
International Open-Access, Double-Blind, Peer-Reviewed, Refereed, Multidisciplinary Online Journal
ISSN: 2581-9429 Volume 6, Issue 11, April 2026 Impact Factor: 8.2

[5] A. Marquez, D. Romero, and J. García, "Dependency vulnerability analysis in modern software ecosystems: Metrics, tools, and recommendations," J. Syst. Softw., vol. 211, p. 112023, 2024.
[6] H. Mohayeji, M. Keshavarz, and A. Zakerolhosseini, "Automated vulnerability mitigation via Dependabot: An empirical study of patch latency and effectiveness," in Proc. MSR 2025, 2025, pp. 88-98.
[7] D. Patel, "SLSA adoption in CI/CD pipelines: A practitioner's guide," in Proc. IEEE DevOps Summit, 2025, pp. 45-54.
[8] B. Singh, "Implementing DevSecOps practices in CI/CD pipelines: A quantitative assessment," Int. J. Inf. Secur., vol. 19, no. 6, pp. 611-628, 2020.
[9] A. Decan, T. Mens, and E. Constantinou, "On the impact of security vulnerabilities in the npm package dependency network," in Proc. MSR 2019, Montreal, Canada, 2019, pp. 181-191.
[10] M. Zimmermann, C.-A. Staicu, C. Tenny, and M. Pradel, "Small world with high risks: A study of security threats in the npm ecosystem," in Proc. 28th USENIX Security Symp., 2019, pp. 995-1010.
[11] A. Birsan, "Dependency confusion: How I hacked into Apple, Microsoft and dozens of other companies," Medium, Feb. 2021.
[12] P. Abate, R. Di Cosmo, G. Gousios, and S. Zacchiroli, "Dependency solving is still hard, but we are getting better at it," in Proc. SANER 2020, London, ON, Canada, 2020, pp. 547-551.
[13] B. Krebs, "U.S. agencies, companies compromised by sophisticated Trojanised software update," KrebsOnSecurity, Dec. 2020.
[14] J. Breen et al., "Log4Shell: RCE 0-day exploit found in Log4j," LunaSec Blog, Dec. 2021.
[15] S. Mirhosseini and C. Parnin, "Can automated pull requests encourage software developers to upgrade out-of-date dependencies?" in Proc. ASE 2017, Urbana-Champaign, IL, 2017, pp. 84-94.