A Practical Guide to Implementing SBOM and VEX in Your Software Development Workflow

1.0 The Strategic Imperative for Software Supply Chain Transparency

Managing the risk inherent in third-party software is no longer an optional security practice; it is a strategic necessity. Growing pressure from regulations like NIS2, DORA, and updated compliance standards such as ISO 27001:2022 is compelling organizations to gain control over their software dependencies. For any company that relies on software for its business processes, understanding and managing the risks introduced by these components is paramount. This section establishes the foundational business and security case for adopting the Software Bill of Materials (SBOM) and Vulnerability Exploitability eXchange (VEX) as core pillars of a modern security program.

Define the Modern Software Development Challenge

Most modern software is not built entirely from scratch. Instead, it is assembled from a vast ecosystem of open-source and proprietary components, frameworks, and libraries, collectively known as software dependencies. This assembly-based approach accelerates development but introduces significant complexity and risk. It is a widespread misconception that vulnerability scanning tools like Dependabot alone constitute a comprehensive risk management process. While these tools are useful for identifying known vulnerabilities (CVEs), they fail to assess the full context of risk. According to CISSP guidelines, risk is defined as the product of vulnerability,

asset, and threat. Simple vulnerability scanning is insufficient because it only addresses one part of this equation. A more robust approach is required to achieve true visibility into the software supply chain.

Establish the Core Concept of SBOM

A Software Bill of Materials (SBOM) is a formal, machine-readable record detailing all components within a piece of software. It can be thought of as a "list of ingredients" for a software package, providing a complete inventory of both direct and indirect dependencies and their supply chain relationships. The primary goal of an SBOM is to provide foundational transparency, allowing organizations to analyze the components they are using, map them to known vulnerabilities, and understand their exposure to supply chain risks. This visibility is the first and most critical step toward managing those risks effectively.

Illustrate the Value of SBOM with a Case Study

The 2021 Log4Shell vulnerability provides a powerful real-world example of the value of SBOMs. The vulnerable component, Apache Log4j, is a widely used logging library. Crucially, it was often used as a *transitive dependency*—a dependency of other dependencies—making it extremely difficult to identify without a complete component inventory. As security teams scrambled to respond, the difference between organizations with and without SBOM capabilities became starkly clear. Those with an SBOM could quickly query their software inventory to determine if and where the vulnerable Log4j library was present. In contrast, organizations without this visibility were forced into time-consuming, resource-intensive manual searches, leaving them exposed to active exploitation for prolonged periods.

"Organizations without SBOM capability often had to engage in time-consuming manual searches and risked remaining vulnerable. Organizations with SBOMs were able to report a relatively straightforward and efficient response..."

This incident underscored that in a crisis, speed and accuracy are paramount, and SBOMs provide the data foundation necessary for both. To enable this level of transparency at scale, the industry relies on a set of standardized formats and frameworks.

2.0 Understanding the SBOM and VEX Ecosystem

For software transparency to be effective across organizations and supply chains, a common language is essential. Standardized formats ensure that the data generated by one tool can be understood and consumed by another, enabling the automation required for modern security operations. This section defines the core components of the SBOM and VEX ecosystem, including the key standards and the distinct roles that different teams play in leveraging them for improved security.

Analyze Key SBOM Standards

Several open standards have emerged to structure and share SBOM data. The two most prominent formats are:

- **SPDX (Software Package Data Exchange):** An open standard for communicating SBOM information, including components, licenses, and security details. Its comprehensive and machine-readable format has led to wide adoption. For example, GitHub uses the SPDX format for its dependency graph export feature, making it a readily accessible option for many development teams.
- **CycloneDX:** Another leading, lightweight SBOM standard designed for use in application security contexts and supply chain component analysis. It provides a standardized way to represent software components, their relationships, and associated vulnerabilities.

Introduce Vulnerability Exploitability eXchange (VEX)

An SBOM answers the question, "What is in my software?" VEX answers the follow-up question, "Am I affected by a specific vulnerability in one of my components?" VEX is a companion standard to SBOM that provides a formal statement about the exploitability of a known vulnerability in the context of a specific product. This is a crucial function for reducing noise and prioritizing remediation efforts. A component may contain a vulnerability, but if the vulnerable code is not reachable or used in the final product, it poses no immediate risk. VEX documents communicate this status, allowing security teams to focus on true positives and avoid wasting resources on vulnerabilities that are not exploitable.

Define Key Roles and Responsibilities

The value of SBOM and VEX is realized when it is integrated into the workflows of different teams across an organization. CISA guidance identifies three primary roles in the software ecosystem, each of which benefits from increased transparency.

Role	Description	Benefit from SBOM/VEX
Producers (Developers)	Organizations and teams that build and maintain software. They have direct control over the components used in their products.	Enables automated tracking of dependencies, better response to new vulnerabilities, and improved management of license compliance throughout the development lifecycle.

Choosers (Procurement/Architecture)	Organizations and teams responsible for acquiring, purchasing, or selecting software for use within their enterprise.	Empowers risk-informed procurement decisions by allowing for the evaluation of a product's components and the supplier's security hygiene before acquisition.
Operators (IT/Security Operations)	Organizations and teams that deploy, manage, and monitor software in production environments.	Provides critical visibility into the software running on their networks, enabling rapid assessment of exposure to new vulnerabilities and more targeted, efficient incident response.

With a clear understanding of these core concepts and roles, we can now turn to the practical steps required to implement them in a real-world development workflow.

3.0 Step-by-Step Implementation: A Practical Case Study with Kubernetes Java Client

Theory is valuable, but practical application is essential. This section provides a hands-on walkthrough of the core SBOM and VEX workflow using a real-world example: the open-source Kubernetes Java Client project. This project uses Maven for build management, making it a representative case study for many Java-based applications. This hands-on walkthrough will demonstrate how a single team can embody the roles of both a software 'Producer' (generating the SBOM) and an 'Operator' (analyzing the result for operational risk).

3.1 Step 1: Generating the SBOM

The first step is to create the foundational "list of ingredients." Modern development platforms like GitHub have integrated features that make this process straightforward.

1. **Fork the Repository:** To begin, create your own copy of the target open-source project by forking its repository (e.g., the Kubernetes Java Client on GitHub).
2. **Navigate to Project Insights:** In your forked repository, go to the `Insights` tab.

3. **Enable and View the Dependency Graph:** Select the `Dependency graph` option from the side menu. This feature automatically parses project files like Maven's `pom.xml` to detect and display the project's dependencies.
4. **Export the SBOM:** Click the "Export SBOM" button in the Dependency Graph view. This action generates and downloads a `.json` file containing the complete software bill of materials in the industry-standard SPDX format. This file is the foundation for all subsequent analysis.

3.2 Step 2: Analyzing the SBOM for Vulnerabilities

With the SBOM generated, the next critical step is to scan it for known vulnerabilities. This can be accomplished using either platform-integrated tools or external scanners.

Method A: Using Platform-Integrated Tools

Development platforms like GitHub often perform this analysis automatically. Directly within the Dependency Graph view, the platform will flag components with known vulnerabilities. In the case of the Kubernetes Java Client, this view highlights a high-risk vulnerability in the `com.diffplug.spotless:spotless-maven-plugin` version `1.17.0`.

Method B: Using External Scanners

For more flexibility or as part of an automated pipeline, open-source tools like Google's OSV Scanner can be used. This command-line tool takes an SBOM file as input and checks its components against a comprehensive vulnerability database.

1. Run the scanner against the exported SBOM file with the following command: `osv-scanner --sbom=java.spdx.json`
2. The scanner will output a table of its findings. The results for the `spotless-maven-plugin` would appear as follows:

OSV URL	CVSS	ECOSYSTEM	PACKAGE	VERSION	SOURCE
https://osv.dev/GHSA-7v35-qwwj-p98g	7.5	Maven	com.diffplug.spotless:spotless-maven-plugin	1.17.0	java.spdx.json

3.3 Step 3: Assessing Exploitability and Context

Identifying a vulnerability (a CVE) is only the first part of the process. The crucial next question is, "So what?" To answer this, you must determine if the vulnerability is actually exploitable within the specific context of your project. This requires investigating how the vulnerable component is used.

Investigating the Component's Context

This investigation combines automated assistance with manual verification to ensure a thorough understanding of the potential risk.

- **Automated Assistance:** AI-powered tools like GitHub Copilot can provide immediate insights. By asking how the library is used, we receive a clear, context-aware answer:
- **Manual Verification:** A manual search through the project's files can be used to confirm the dependency's usage, verifying the findings from the automated tool. In this case, the investigation confirms the plugin is only active during the build phase and its code is not included in the final, executable product.

3.4 Step 4: Creating the VEX Document

The VEX document is the formal communication of the findings from Step 3. It serves as an authoritative statement on the exploitability of a vulnerability. While the tooling in this area is still maturing—tools like OpenVex offer basic functionality but can require manual or semi-automated processes—the structure of a VEX statement is well-defined. For architects, this means prioritizing the automation of SBOM generation first, while treating VEX generation as a semi-automated or manual process for critical vulnerabilities. Plan for the development of internal scripts or be prepared to adopt new tooling as the ecosystem matures.

Anatomy of a VEX Statement

Based on our investigation, we can create a VEX statement for the `spotless-maven-plugin` vulnerability. The key components of this VEX statement are:

- **`vulnerability`**: The specific CVE or other vulnerability identifier. Here, it is `CVE-2019-9843`.
- **`products`**: The unique package URL (PURL) identifier for the component, taken directly from the SBOM. In this case, `pkg:maven/com.diffplug.spotless/spotless-maven-plugin@1.17.0`.
- **`status`**: The assessment of the vulnerability's impact. Based on our findings, the correct status is **`not_affected`**.
- **`justification`**: A machine-readable reason for the status. The most appropriate justification here is **`vulnerable_code_not_in_execute_path`**.

- `impact_statement`: A clear, human-readable explanation summarizing the justification. For example: "This vulnerability is not exploitable because the plugin is not used directly in the code but rather the build process."

By following these four steps, you can move from a simple list of dependencies to a context-aware assessment of risk. The next step is to integrate these processes into your organization's daily operations.

4.0 Automating and Integrating into the SDLC

The true power of SBOM and VEX is realized when these practices are scaled from manual exercises to a fully automated system integrated into the development lifecycle. This aligns with the core principles of Secure by Design, particularly the call to "Embrace Radical Transparency and Accountability," where SBOMs serve as the primary mechanism for demonstrating command of the software supply chain. The following maturity model outlines a phased approach to operationalizing these practices.

Phase 1: Foundational Visibility (Manual & Ad-Hoc)

The initial goal is to establish the capability to generate and analyze SBOMs for critical applications. This phase mirrors the hands-on case study in Section 3, focusing on manual or semi-automated processes to build familiarity and demonstrate immediate value. At this stage, teams learn to use platform features or command-line tools to produce an SBOM, scan it for high-priority vulnerabilities, and manually conduct exploitability analysis for the most critical findings.

Phase 2: Automated Generation & Analysis (CI/CD Integration)

In this phase, the process moves from ad-hoc to automated. SBOM generation becomes a standard, non-negotiable artifact of every build within CI/CD pipelines. Tools like the CycloneDX Maven Plugin, Syft, and platform-native features in GitHub Actions or Azure DevOps can automate this process. Concurrently, SBOMs are automatically fed into scanners that correlate component data with vulnerability databases in real-time. This creates a continuous monitoring loop that tracks the risk posture of an application throughout its lifecycle, alerting teams to new risks as they are discovered.

Phase 3: Proactive Enforcement & Integration (Policy as Code)

At full maturity, SBOM and VEX data become an active enforcement mechanism. Policies are defined as code and integrated into the CI/CD pipeline to automatically block builds or deployments that contain unacceptable components. These policies can govern license types, prohibit components from untrusted sources, or flag dependencies with unpatched critical vulnerabilities or end-of-life status. Finally, all SBOM and VEX data is seamlessly integrated into

centralized systems for vulnerability management, asset inventory, and supply chain risk management, turning raw component data into prioritized, context-rich, and actionable tasks for security and development teams.

5.0 Conclusion: Building a More Secure and Transparent Future

This guide has demonstrated that SBOM and VEX are foundational tools for enhancing software security and transparency. By providing a clear "list of ingredients" for software and a mechanism for communicating the real-world exploitability of vulnerabilities, they empower organizations to move from a reactive to a proactive security posture. While challenges remain, particularly in maturing the automation of VEX generation, their value in identifying, prioritizing, and mitigating software supply chain vulnerabilities is indispensable. As development practices continue to evolve, the widespread adoption and integration of these standards are integral conditions for creating a software ecosystem that is truly secure by design.

For more information on implementing these strategies, please contact `realrisk@sbomvex.info` or visit https://sbomvex.info