BLACK DUCK®

WHITE PAPER

MANAGING TRANSITIVE DEPENDENCIES IN OPEN SOURCE SOFTWARE

INSIGHTS AND ESSENTIAL PRACTICES

When developers import open source or third-party components to build software applications, they typically declare the known dependencies—the libraries or modules that a project directly uses in its code. Commonly used libraries and frameworks include React, Express.js, and Django.

But those dependencies often have dependencies of their own, known as indirect or transitive dependencies, and they are the libraries or other software that the direct dependencies utilize. For example, while jQuery itself does not have many direct dependencies, projects using jQuery often pull in other libraries that have their own dependencies, creating chains of transitive dependencies.

So if you're using a jQuery plug-in such as `jquery-ui-extended` for advanced UI components, that plug-in might depend on

- **jQuery itself,** a direct dependency, because the plug-in needs jQuery to function
- **`moment.js`,** a library for handling dates and times, if the UI components involve date pickers or calendars
- **`timezone-data.js`**, which `moment.js` might depend on to accurately handle time zones

In this scenario, `jquery-ui-extended` indirectly (or transitively) depends on `timezone-data.js` through `moment.js`. And if you include `jquery-ui-extended` in your project, you're implicitly including `moment.js` and `timezone-data.js` as well, even if you don't explicitly declare them.

Simple applications may have only a few dependencies, perhaps a handful of libraries or frameworks. But an enterprise-level system or a large web application can have hundreds or thousands of dependencies, many of them transitive. These dependencies can create complex chains that are nearly impossible to detect, manage, and test without automated tooling.

TRANSITIVE DEPENDENCIES AND SOFTWARE SUPPLY CHAIN SECURITY

The 2025 "Open Source Security and Risk Analysis" (OSSRA) report includes several key findings that highlight the importance of identifying and tracking transitive dependencies.

- **Prevalence:** 64% of the open source components identified in the OSSRA report were transitive dependencies
- **Complexity:** Open source software is more complex than ever, and the number of open source files in an average application has increased three-fold since 2020
- **Security risks:** 81% of codebases contain high- or critical-risk vulnerabilities, nearly half of which were introduced by transitive dependencies
- **License conflicts:** 56% of codebases have license conflicts, the majority caused by incompatible transitive dependencies

Five Common Problems and Risks

The findings above emphasize the necessity of managing transitive dependencies in open source software. If left unidentified and untracked, these dependencies can manifest in critical issues that are more difficult and expensive to resolve, whether related to security, license compliance, maintenance and support, or a combination thereof. Below are five problem areas that intransitive dependencies can introduce if you're not managing them carefully.

Security Vulnerabilities

Transitive dependencies can introduce weaknesses such as unpatched open source, zero-day vulnerabilities, and malicious dependencies, which can be exploited by attackers. The infamous vulnerable version of Apache Log4j was most likely introduced via a transitive dependency—when developers chose popular libraries like Spring Boot or Elasticsearch, they also included Log4j by default. In fact, 11 codebases in the 2025 OSSRA report contained vulnerable versions of Log4j. Such vulnerabilities can expose entire projects to potential threats, making it crucial to regularly update and monitor dependencies to mitigate risks.

License Compliance Issues

Hidden license terms, license incompatibility, and custom and variant licenses are all issues that transitive dependencies can introduce. Transitive dependencies may have different or more restrictive license terms that lead to compliance or legal issues. Developers are often unaware of transitive dependencies and their licenses, so unintentional violations of intellectual property laws can occur, leading to costly litigation and damage to the organization's reputation. Customers and partners may lose trust in an organization's ability to deliver software that's reliable, secure, and compliant, causing long-term impact on business relationships and market position.

Maintenance and Support Issues

Poorly maintained dependencies can increase technical debt and slow down development teams. Red flags include open source projects that have a low number of contributors, as well as any component that hasn't been updated in several years, has a history of unresolved security issues, or has restrictive or legally undesirable licenses. These can lead to security vulnerabilities and operational issues such as incompatibility with newer versions of other components or the overall application architecture, and this can cause system instability, performance degradation, and increased maintenance costs. Developers may need to spend considerable time and resources to fix issues or find alternative components, which can delay timelines and increase costs.

Furthermore, transitive dependencies can make applications more complex and harder to manage. For example

- **Complex dependency graph:** Additional layers of dependencies can make it harder to track and manage all components
- **Version conflicts:** Two different direct dependencies could rely on different versions of the same transitive dependency, leading to version conflicts that could break the application
- **Build and deployment complications:** Ensuring all dependencies are correctly resolved and compatible can complicate build and deployment processes

Inconsistent or Inaccurate SBOMs

Transitive dependencies create complex relationships that are difficult to track manually, and that can be problematic when creating and maintaining Software Bills of Material (SBOMs). Without automation, it's nearly impossible to detect a dependency that's nested several layers deep in an application. Outdated or unsupported transitive dependencies can introduce security risks and compliance issues, and further complicate the accuracy of an SBOM.

Performance and Reliability Issues

Transitive dependencies can introduce performance bottlenecks and reliability issues in several ways.

- **Dependency bloat:** An application includes more dependencies than necessary, increasing its size and leading to longer build times, higher memory usage, and slower performance. The OSSRA report revealed that there are 911 components in the average software application. How many of those components are necessary?
- **Runtime errors and crashes:** Outdated dependencies can negatively impact the reliability of an application.
- **Complexity in dependency management:** Tracking and updating all dependencies is more difficult as the chain of transitive dependencies becomes longer and more complex. This helps explain why 91% of codebases analyzed in the OSSRA report contained outdated components.

Essential Practices for Mitigating Risks

Mitigating the risks associated with transitive dependencies is crucial for the security and integrity of any organization's software supply chain. The following activities are essential for detecting, tracking, and managing all components in an application.

Maintain a Comprehensive Inventory of Dependencies

Visibility is step one. You can't manage or fix what you don't know you have. Thus, keeping a comprehensive inventory of all open source dependencies is crucial, regardless of how they were included or what language or package manager is being used. This includes identifying dependencies from

- Build automation tools
- Custom scripts and workflows
- AI coding assistants
- Manual inclusion by developers
- Libraries linked statically or dynamically
- Container images

Implement Continuous Monitoring

Continuous monitoring of all software components, including dependencies, is necessary to detect new vulnerabilities, exposed secrets, malware, and malicious packages. A software composition analysis (SCA) tool such as Black Duck® SCA can help you analyze the severity of risk from all these issues quickly and easily, without the need to rescan projects for dependencies.

Establish and Implement Policies

Defined policies help your organization maintain a secure and compliant software supply chain without impeding the development process. Policies around proactive assessment, continuous monitoring, secure development practices, and compliance reporting take the guesswork out of individual cases by standardizing processes and responses. Policies can define when scans are run, what dependencies are allowed based on their risk metrics, what results trigger workflows, and what reports are generated.

Automate Enforcement of Policies

Automation is key to managing transitive dependencies effectively. Tools that discover and track dependencies can help teams use minimal and well-maintained libraries while helping to keep the dependencies updated on a regular basis.

Integrating automated tools into the software development life cycle (SDLC) and DevOps pipelines helps block components with known vulnerabilities or license issues from entering production, and also provides remediation guidance to resolve issues quickly. Consider the following integrations when looking to automate tooling:

- **IDE plug-ins** to evaluate dependencies before they're committed to projects
- **Source code manager (SCM) integrations** that trigger scans on pull requests
- **CI/CD pipeline integrations** that scan at build time, blocking builds when vulnerable code is detected
- **Binary repository integrations** to analyze release artifacts before they ship and create an internal repository of approved components

Evaluate and Track IP and License Obligations

Ensuring compliance with licenses is crucial to preventing legal and security issues. This means surfacing both declared and nested licenses and copyright data to evaluate for conflicts, no matter how many levels deep a dependency is. SCA tools like Black Duck SCA use multiple detection methods, including snippet analysis and binary analysis, to identify all open source components and their associated licenses, even if they're not explicitly declared. This helps with the creation of a complete and accurate SBOM, another essential practice for managing risks and ensuring compliance. Additionally, Black Duck SCA's automation capabilities can expedite the process by providing simplified views of license requirements and generating necessary notices files, reducing the burden on development teams and minimizing legal risk.

Generate and Validate SBOMs

SBOMs are critical for managing software risk, and they are becoming requirements in several industries. From a practical standpoint, they simplify the management of complex software supply chains by providing a clear and comprehensive inventory of every component in an application, including open source and commercial dependencies.

For SBOMs to be of value, however, they must be complete and accurate. A complete and accurate SBOM enables you to quickly identify and assess the impact of newly discovered vulnerabilities, especially in frequently updated open source components. An incomplete SBOM, on the other hand, can hinder identification of vulnerabilities, leading to delayed or ineffective patching and increased exposure to potential attacks. The same holds true for legal and compliance issues.

Because of their value in enhancing transparency, managing vulnerabilities, ensuring compliance, and improving risk management, SBOMs are increasingly required by regulatory bodies. The Cyber Resilience Act requires SBOMs from any IoT manufacturer selling in the European Union. An SBOM is required for each product, capturing every software component used in a device, including both proprietary and open source components. Detailed metadata must also be kept on version numbers, licensing information, and known vulnerabilities.

In the U.S., the National Institute of Standards and Technology (NIST) has been a strong advocate for the use of SBOMs, providing published definitions, purposes, standards, and guidelines. NIST Special Publication 800-218, "Improving the Security of the Software Supply Chain," provides detailed recommendations for using SBOMs. Additionally

- The Federal Drug Administration requires SBOMs in healthcare and medical devices
- FinTech providers must track third-party libraries in payment processing software
- Car manufacturers must track components in infotainment systems

Understand Package Manager Behavior

Because transitive dependencies are automatically included by package managers such as npm and Maven, it's critical to know how these dependencies are resolved and what you can do to exclude unwanted ones. It's possible, for instance, to use package managers to check for outdated components (e.g., `npm outdated` checks the registry to see if any installed packages are out-of-date) and identify and remove packages that are unused or deprecated (e.g., `npm prune` removes dependencies not listed in package.json).

Train Developers

Educate developers on how to identify and evaluate transitive dependencies in their applications, and train them on the use of security tools such as SCA to detect vulnerabilities and license issues. Several steps can be taken to achieve this.

- **Educational workshops and training sessions** using real-world examples and hands-on use of security tools and techniques
- **Documentation and best practices** of dependency management, including common pitfalls and how to avoid them, as well as proactive steps for how to add, update, and remove dependencies
- **Code reviews and peer collaboration** that include checks for transitive dependencies and ensure they're necessary and secure
- **Security awareness programs** can be incorporated into developer onboarding, and security champions embedded in development teams can provide ongoing guidance and support
- **Community and resources** such as the Open Source Security Foundation and the National Vulnerability Database provide valuable information and tools
- **Practical exercises and simulations** where developers can practice identifying and managing open source risks in a controlled environment

Follow an Established Framework

Instead of reinventing the wheel, follow an established security framework that emphasizes open source software dependency management.

Secure Software Development Life Cycle

The Secure Software Development Life Cycle (SSDLC) emphasizes integrating security management throughout the entire development process. It also highlights the need for inventorying and tracking all open source components, regular updates and patching, thorough security assessments, ensuring license compliance, and using an automated SCA tool.

Supply Chain Levels for Software Artifacts

The Supply Chain Levels for Software Artifacts (SLSA) offers a common vocabulary, criteria for assessing artifact (component) trustworthiness, and actionable steps for software producers and consumers to enhance security. It focuses on the integrity of software components and ensuring that they're as secure as possible by focusing on provenance (the record of how a software component was created), attestations (statements of fact about the security of software components), trustworthy platforms, and levels of assurance (a tiered system representing more stringent security requirements and greater confidence).

Secure Software Development Framework

Established by NIST, the Secure Software Development Framework (SSDF) is a set of software development best practices based on documentation from organizations such as the Business Software Alliance, OWASP, and SAFECode. The SSDF stresses open source risk management, policy and process requirements, attestation and verification (including SBOMs), transparency, and collaboration.

READY TO GET STARTED?

The number of dependencies in a software application can range from just a handful to a few thousand, depending on the application's complexity, programming language, and the project's requirements. Effectively managing every component and dependency is crucial to maintaining the security, compliance, and reliability of the application—and the integrity of your software supply chain.

Black Duck helps customers manage the security and license risks associated with open source software, regardless of size or complexity. We offer

- Dependency analysis
- Binary analysis
- Snippet analysis
- File and directory analysis
- Container scanning
- Security vulnerability management
- Malicious package detection
- Component health insights
- License compliance
- SBOM generation and management in standardized formats like SPDX and CycloneDX
- Automation and integration throughout the SDLC, including IDEs, SCMs, CI/CD pipelines, and binary repositories
- Custom policy configuration

[See how Black Duck can help you secure your software supply chain from the threats posed by transitive dependencies.]

ABOUT BLACK DUCK

Black Duck® meets the board-level risks of modern software with True Scale Application Security, ensuring uncompromised trust in software for the regulated, AI-powered world. Only Black Duck solutions free organizations from tradeoffs between speed, accuracy, and compliance at scale while eliminating security, regulatory, and licensing risks. Whether in the cloud or on premises, Black Duck is the only choice for securing mission-critical software everywhere code happens. With Black Duck, security leaders can make smarter decisions and unleash business innovation with confidence. Learn more at www.blackduck.com.