



Why Precision Matters in Managing Open Source Software





Why Precision Matters

Open source software use is widespread. Sonatype's 2018 State of the Software Supply Chain report shows that the average application includes over 100 open source components, which make up 80-90% of an application's codebase. In 2018, alone, the average number of npm package downloads by Javascript developers was 7 billion each week and Java download requests weighed in at 146 billion for the year – a 68% increase over the previous year.

There are good reasons for the growing popularity of open source by software teams. Open source components provide critical functionality to development teams, accelerating time to market, while dramatically lowering development costs. If even just 50% of an application is comprised of open source, replacing it with custom code would require an organization to either double its development team or double its development time. Neither are acceptable options in today's world.

As open source consumption has grown, so has the awareness that its unmanaged use brings risk from

security vulnerabilities and restrictive licenses in those components. Today, 1-in-8 open source components contains a known security vulnerability, and the average time from exposure to exploit has decreased from 45 days to 3. The cadence of modern software development has outpaced organizational capacity for manually identifying risky components granular enough to mitigate that risk. Capitalizing on this growing need, several vendors have entered the market offering varying capabilities to address the problem. This paper will examine the different approaches used by these companies.

You Can't Protect What You Can't See

A basic tenet of any security program is to understand the risks present in your systems and applications, then prioritize risk mitigation steps. For open source risks, this requires organizations to have full visibility to all of the open source code in an application. Without this, you cannot protect the applications from known risks such as improper licenses or security vulnerabilities.

The Software Bill of Materials

Every manufacturing process requires a bill of materials; a listing of every part required to build an item. This helps companies standardize on parts and reduce costs, across their supply chain. It also allows them to quickly address quality issues. When a part is faulty (think of a car's airbag) they are able to quickly determine which units are affected and prioritize remediation steps.

The same is true for software. With a software Bill of Materials (BoM) an organization can track every external "part" (or component) they use, and better resolve issues when a "faulty" component arises. For security use cases, knowing the precise version of each component in play is critical, since vulnerabilities often affect only specific versions of a component. Precision allows teams to triage risks quickly, reduce unnecessary rework, and stay one step ahead of adversaries.

How Software is Built

As previously noted, open source software has changed how software is built. While 15 years ago it was common to build an application "from scratch", the adoption of open source has advanced to the point that open source comprises the majority of the average application's codebase. Developers maintain open source components in their workspaces and in repositories, defaulting to open source over writing custom code. This allows open source to enter the code base in three ways:

- ▶ **Direct Dependencies** – Open source components are often added directly to the codebase by software engineers. This is the only way to use open source components in programming languages that do not use "package managers", like C and C++.
- ▶ **Declared Dependencies** – In programming languages that use package managers, software engineers may declare that a component is required in a manifest used by the build process. This will cause the build tool

to import the component from a binary repository like Nexus Repository.

- ▶ **Transitive Dependencies** – These occur when a declared dependency requires additional open source components to run properly.

License and security risk can occur in components added to an application by any of these three methods, so it's important to account for components added by any method.

The Consequences of Inaccurate BoMs

An incomplete BoM that is missing components can result in security risks. Thousands of security vulnerabilities are disclosed in open source each year. Exploits for these are often publicly available within days of the disclosure, allowing a simplified attack vector for even inexperienced attackers. Such was the case with the well known Equifax breach, as well as breaches at the Canada Revenue Agency, Okinawa Power, and The University of Delaware. From a licensing standpoint, if a missed component was published under a restrictive open source license like GPL, the “derivative works” comprising the rest of the application could put IP at risk. Imagine a large manufacturing company having to release code that gives them a market advantage to the open source community and competitors.

An imprecise BoM occurs when components are mis-identified, either by selecting the wrong open source project or the incorrect version of the correct project. The former case brings with it the same risk as with missing the component entirely. The latter case adds risk that an organization will improperly assume a vulnerability does not affect a component, or that it falsely affects a component (resulting in unnecessary rework).

Creating a complete and precise BoM is not a simple process. In earlier times, a partial BoM might be created by asking the development manager for a list. The manager would write down the components she was aware of, but the list was almost always incomplete. Individual developers would likely not have complete listings either, as they are under constant pressure to deliver required functionality by a specific date. Using open source is a natural part of their workdays, and manually making note

of each library and version is an unacceptable (and unreliable) practice. Further, even if a developer noted each component added to the code base, they are unlikely to know the dozens of transitive dependencies automatically added to the build.

Building a BoM

An automated process for generating a bill of material (BoM) and monitoring for risk reduces the burden on developers, allows compliance, security, and engineering teams to agree on policies, and enforce those policies. The more precise and complete the Bill of Material, the more useful and actionable the results. First, we will look at a quick but incomplete approach, then at a slower but still imprecise approach.

Build Process Monitoring for Declared Dependencies

The easiest way to build a BoM is to simply look at which components the developer has declared as required for a functioning application (declared dependencies) in the package manifest. Tools using this approach identify components by looking at packaging instructions in files or manifests and looking up records on the file names alphanumerically. In other words, this simply repeats what the developer has listed and confirms what the organization believes exists in the software supply chain.

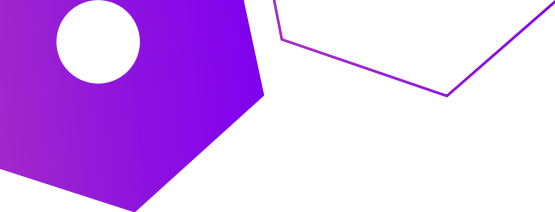
While this is certainly the fastest way to generate a BoM, it is deficient in many ways:

Completeness

- ▶ Package managers are blind to any direct dependencies added to the codebase by developers. Since this approach can only restate what is listed in the package manager, it will miss these components.
- ▶ This approach cannot be used for software languages that do not use package managers, like C and C++. Enterprises developing code in multiple languages may require multiple tools.

Precision

- ▶ A package manager provides “coordinates” for a component so the build system knows from which location



to fetch the component (e.g., org.apache.struts:struts-core:1.0). Duplication of coordinates is a common problem. Just because the package manager thinks it has a file with the right namespace, doesn't mean it's not an altered one, from another repo. Even a one character difference can create either a false positive or a false negative - misidentifying the component completely.

- ▶ There are often unknown components inserted by the packaging systems that cannot be identified alphanumerically.
- ▶ Multiple packaging systems are commonly used, which simply compounds the issues above – creating more noise and a BoM that leaves developers chasing after vulnerabilities that may not be present.

Source Code Signatures for Direct Dependencies

Another simple way to create a BoM is to review source code and calculate hashes for each file. These hashes are compared to a database of hashes for known (inventoried) source code and code snippets. While this is an improvement on simply monitoring the build process and will often catch components added directly by developers, it too has shortcomings.

DevOps Compatibility

- ▶ Timeliness - DevOps environments require accurate and immediate feedback. Calculating hashes and comparing those to a vendor database in the cloud takes time. This method may also produce “partial matches” (see below) that require manual review, further delaying results.

Completeness

- ▶ Modified libraries may not be detected. Developers often add, remove, or modify open source components. This will cause the hashes to not match those previously calculated by the vendor, resulting in either missing components or requiring manual review to identify the correct component and version.

Precision

- ▶ Incorrect matching can mistakenly confuse similar components. This is because common coding patterns

exist, making different pieces of source code appear similar. Source code matching is based on snippet analysis which can result in many false positives.

- ▶ Redundant matches result because a given piece of source code, especially in open source, often exists in multiple projects and, within a project, likely exists in many versions of a component. This results in a BoM with duplicated components. When a vulnerability is reported against that component, it too, appears multiple times.

A Different Approach: Advanced Binary Fingerprinting

Development and security groups, particularly DevOps teams, need speed, accuracy, and precision. The inability of above approaches to conduct a complete and precise listing of all components in an application led Sonatype to invest the necessary time and capital to invent new techniques, processes, and algorithms in data science. Advanced Binary Fingerprinting allows Sonatype to identify all of the open source components in an application quickly and precisely, avoiding the false positives and false negatives common in other methodologies and enabling a mature Source Composition Analysis solution in the DevOps environment.

Here are two examples of the result:

- ▶ **Java** – An application is shipped as a Java Enterprise Archive (EAR) artifact. The EAR contains three Java web applications (WAR). Each of these contain numerous JAR files, including one with a Java JAR file created by patching (forking) an open source project and recompiling it to create a new custom JAR. With Sonatype Advanced Binary Matching, we can identify all nested files successfully. In addition, our patented algorithm will trigger the similarity result and indicate a close partial match to the forked open source including the project version.
- ▶ **JavaScript** – An application is shipped as a self-extracting zip file. The file contains a number of JavaScript files, including one with a dependency for jQuery. However, this file was renamed, and has therefore lost any identifying information to the jQuery dependency. Using Sonatype's proprietary matching, the component is identified, and traced back to the original jQuery dependency, including the specific version.

Precision Matters in Data Intelligence

Once you can precisely identify components, the next step is assuring that the metadata describing the attributes of a component are sufficiently precise. This supports DevOps environments by allowing decisions about a component's acceptability (compared to a policy) to be adjudicated through automated processes instead of by human review.

As with component identification, searching through various databases for metadata such as security disclosures by the name of a component will inevitably produce poor information. Recognizing this, Sonatype invests heavily in a team of data experts that conduct proprietary research to develop deep intelligence on components. This team leverages public and private data feeds, mines popular repositories (like GitHub), and monitors project web sites to provide information on more vulnerabilities than sources like the National Vulnerability Database or mailing lists. Further, the team is able to surface information to users more quickly, identifying those vulnerabilities weeks earlier than other sources. More importantly, the data researchers never rely solely on those information sources without a thorough, independent review of the software risk.

For licensing, this means not just relying on the license declarations made by the project, but also examining headers within the source code. For security defects, this means not relying on the reported problem but identifying the root cause of a vulnerability and documenting paths to remediation or alternative resolution. For other attributes, this means both quantitative and qualitative assessments of architectural and project integrity.

Enforce Policies with Confidence

Precision is the only way to empower teams to make better decisions, so that they can scale faster with controls that are flexible enough to reflect the policies of the organization in the context of the applications that are being developed.

Every organization, every team, and every application is fundamentally unique. Therefore, you need to ask yourself the following question: at what point in the software lifecycle do you want to examine the attributes of a component?

- ▶ Every time a developer downloads a new component?
- ▶ Whenever a development team produces a build?
- ▶ During pre-release testing?
- ▶ When applications are ready for production?

Which is best?

The answer, of course, is that all of them are important because the world of modern software development is not one size fits all. Indeed, every situation is different and therefore it is critical to have component intelligence tools that are flexible enough to add value at every stage of the DevOps tool chain. This includes the developer IDE, build systems, repository managers, CI/CD tools, image constructors, delivery orchestrators, and production runtime environments.

Of course, ensuring the security of an application does not end when the application is deployed. Software security is never permanent. The discovery of a new attack vector or the disclosure of a new vulnerability in an open source component can instantly change the security profile of an application. That means preserving a bill of materials and monitoring the ecosystem for new information, such as the discovery of a new security defect, is imperative.



In Summary

When it comes to using open source components to manufacture modern software, the bottom line is this – complete and precise intelligence is critical. Tools that lack visibility and precision cannot scale to the needs of modern software development. Inaccurate and/or incomplete data will leave organizations to deal with vulnerabilities, licensing, and other quality issues that lead directly to higher costs and reduced innovation.

Sonatype Nexus Intelligence and Advanced Binary Fingerprinting power a unique solution that enables organizations to:

- ▶ **Empower innovation** by equipping teams with the ability to precisely identify the highest quality open source components.
- ▶ **Scale fast** with component intelligence that is precise enough to enable automation at every phase of the software lifecycle.

- ▶ **Control component usage** with flexible policies that can promote granular decision support across varying teams, languages, and application profiles.

With precise identification on your side, you have the power to error-proof the software supply chain. This means eliminating, with certainty, the risks and inefficiencies that diminish innovation. This also means unlocking the full potential of talented developers so you can innovate faster and compete more effectively on a global playing field.

We welcome any questions that you might have and we encourage you to sample the incredible value of Nexus Intelligence and Advanced Binary Fingerprinting. **To take the next step, try our free tool, [Nexus Vulnerability Scanner](#), or schedule a demo today.**



Sonatype is the leader in software supply chain automation technology with more than 300 employees, over 1,000 enterprise customers, and is trusted by over 10 million software developers. Sonatype's Nexus platform enables DevOps teams and developers to automatically integrate security at every stage of the modern development pipeline by combining in-depth component intelligence with real-time remediation guidance.

For more information, please visit [Sonatype.com](https://www.sonatype.com), or connect with us on [Facebook](#), [Twitter](#), or [LinkedIn](#).

Headquarters

8161 Maple Lawn Blvd, Suite 250
Fulton, MD 20759
USA • 1.877.866.2836

European Office

168 Shoreditch High St, 5th Fl
London E1 6JE
United Kingdom

APAC Office

60 Martin Place, Level 1
Sydney 2000, NSW
Australia

Sonatype Inc.

www.sonatype.com
Copyright 2020
All Rights Reserved.

