SBOM Elements & Considerations

Kate Stewart, Steve Winslow & David A. Wheeler, The Linux Foundation

As requested in NTIA's "Notice and Request for Comments on Software Bill of Materials Elements and Considerations" (NTIA-2021-0001), here are comments on the minimum elements for an SBOM, and a discussion on what other factors should be considered in the request, production, distribution, and consumption of SBOMs. We answered in order. No page count limit was given, so we have not tried to limit ourselves to one. This response was due June 17, 2021 to SBOM_RFC@ntia.gov.

1. Are the elements described above, including data fields, operational considerations, and support for automation, sufficient? What other elements should be considered and why?

The data fields described above are a good common starting point.

As always, what data fields are required depend on the goals:

- If some components may have been recompiled, then the components must be identified not only by a specific compilation (e.g., hash or URL of a package), but also by an identifier of its source code (e.g., URL pointing to its source code's source, which may also include transitive information on where that source came from). **Standardizing the connection from a built object to its respective sources is key for accurate vulnerability analysis.**
- Understanding what is known at a specific point in time is necessary to accurately track SBOMs for use-cases like Software as a Service and identifying the most recent version of an SBOM. **A timestamp of the SBOM should be added as a mandatory field** for any SBOM.

Other considerations:

- Must **work well for describing open source software** (OSS), since the majority of software inside codebases is OSS. Since OSS can be recompiled into components, systems must not assume that matching hashes of compiled programs are adequate to perform identity matches. A 2021 Synopsys report found that 98% of general codebases and of Android apps contained OSS, and that codebases averaged 528 OSS libraries per analyzed codebase. A previous 2020 Synopsys report found that on average 70% of the contents of analyzed codebases were OSS.
- **Automate matching between NVD and SBOM component identification** - It's vitally important that automated matching occur between all vulnerabilities listed in the US National Vulnerability Database (NVD) and SBOM component identification. This matching must support version ranges (since vulnerabilities often apply to many versions). Manual processes are

unable to cope in an accurate and timely way, as there is simply too much software and too many known vulnerabilities.

- **Enable linking and reference between SBOMs** - The depth of dependencies needs to be able to refer from one SBOM to another (and elements within), in order to manage the documents effectively and enable the knowledge to be shared effectively beyond the immediately named dependencies.
- **Enable linking and referencing to multiple global identifiers for the software** (CPE, PURLs, SWHIDs, etc.) As there is no one standard source for naming, an SBOM should be able to leverage other coordinate systems already established and also to refer to custom software that has no such global identifier.

2. Are there additional use cases that can further inform the elements of SBOM?

The executive order on cybersecurity focuses on cybersecurity, but SBOMs have many other uses. For example:

- ***Licensing*.** Historically some SBOM formats were created to help supply chain managers and lawyers identify licensing for the components. This use case informs supply chain transactions as well as merger and acquisition transactions, and enables companies to comply with the terms of software licenses for a component or system. This use case also helps in the identification of software received or distributed under an open source license that may have compliance requirements.
- ***Installation/update*.** Many software ecosystems have ecosystem-specific SBOM formats implemented by manifest files and package managers, whose original focus was primarily to ease initial installation and updating of transitive dependencies. These manifest and package management SBOMs enable introspection down to the source files, and even to specific sub-portions of the source files.

It is important that any SBOM requirements **do *not* interfere with the other potential uses of SBOMs**. It might appear that a solution could be requiring multiple, use-case-specific SBOM formats for a single component (e.g., one for cybersecurity, another for licensing, and another one for installation). However, that would likely lead to inefficiencies and overly complex software build toolchains, with multiple domain-specific formats potentially modeling software in inconsistent ways. In addition, there is a risk that the data in different formats could become inconsistent. Instead, allowing the use of additional SBOM data for other use case purposes, known today or in the future, is more likely to result in faster and more widespread adoption.

3. SBOM creation and use touches on a number of related areas in IT management, cybersecurity, and public policy. We seek comment on how these issues described below should be considered in defining SBOM elements today and in the future.

a. Software Identity: There is no single namespace to easily identify and name every software component. The challenge is not the lack of standards, but multiple standards and practices in different communities.

Software identity is a huge challenge today. The most viable solution today is to **allow SBOMs to record multiple ways to identify the same component**, do the same in databases such as the National Vulnerability Database, and provide data or services to help additional matching across multiple standards and practices. There are known limitations with using only one approach to identify all types and delivery methods for software. Closed source software packaged as binaries for distribution is often best identified using a strong cryptographic hash. Open source software that can be compiled by anyone is often best identified using the URL or ecosystem-specific coordinates for the source release, not just cryptographic hashes, since recompilation and reuse of version numbers for slightly different versions of software can occur.

A URL *can* identify an OSS project in many common cases (e.g., its source code repository and perhaps a specific version). Since projects are derived from other projects, an SBOM may need to include "project X which is derived from Y" (transitively) to increase the likelihood that a vulnerability in Y will trigger an examination of anyone using X. These can also identify source code repositories, which addresses the problem of correctly identifying software if recompiled.

In addition, many software packages are retrieved from well-known repositories such as Maven's Central Repository (for Java). In those cases, including a reference to that well-known repository using its identification scheme can often provide the necessary information.

Any type of software can be identified, and enabling SBOMs to appropriately incorporate multiple means to identify software is critical to adoption and suitability of an SBOM to a use case. If and when better systems for identifying software emerge, the SBOM format should be adaptable to any new, future system as well.

b. Software-as-a-Service and online services: While current, cloud-based software has the advantage of more modern tool chains, the use cases for SBOM may be different for software that is not running on customer premises or maintained by the customer.

Modern toolchains used by cloud service providers offer even greater transparency into the underlying software - in real time. These systems are regularly patched and managed through a dev-sec-ops configuration and deployment model such that the providers have deep insight into every component used in a service being provided. Cloud systems provide a potential path forward towards real time transparency and enabling an SBOM through new delivery methods (e.g. as an API function). It is important that a view in time of any particular service be able to be discovered. As a result, **we recommend that a timestamp** be added to the minimum elements for any SBOM.

c. Legacy and binary-only software: Older software often has greater risks, especially if it is not maintained. In some cases, the source may not even be obtainable, with only the object code available for SBOM generation.

An SBOM format should be flexible enough to be able to describe any form of software, including legacy and binary-only software where sources are not available. The metadata related to such software, such as an absence of known supplier or originator, may indicate to automated systems ingesting the SBOM that the software is less trustworthy.

Additionally, an SBOM format should preferably enable linking to, and updating or enhancing, previously-created SBOMs in an automated manner. In this way, if more information becomes available in the future due to further investigation, that new information can be published in a new SBOM that references back to the original one, with links to the original SBOM's elements.

d. Integrity and authenticity: An SBOM consumer may be concerned about verifying the source of the SBOM data and confirming that it was not tampered with. Some existing measures for integrity and authenticity of both software and metadata can be leveraged.

A cryptographic hash has proven effective in many cases for matching up SBOM data with the original source, and has been present in SPDX for 8 years.

We do need a freely available way to be able to sign an SBOM and attest to the fact it has not been tampered with. There are a number of open source projects emerging to support wrapping SBOMs with signatures, such as in-toto and sigstore (for example, sigstore notes that "later we will explore other formats (such as jars) and manifest signing, such as SBOM etc.").

e. Threat model: While many anticipated use cases may rely on the SBOM as an authoritative reference when evaluating external information (such as vulnerability reports), other use cases may rely on the SBOM as a foundation in detecting more sophisticated supply chain attacks. These attacks could include compromising the integrity of not only the systems used to build the software component,

but also the systems used to create the SBOM or even the SBOM itself. How can SBOM position itself to support the detection of internal compromise? How can these more advanced data collection and management efforts best be integrated into the basic SBOM structure? What further costs and complexities would this impose?

SBOMs should be available for the tooling that is used in a build system. It should also be possible to express in an SBOM the relationship between a built object and the tooling used to build it. This would enable attacks against the build system to become more transparent.

f. High assurance use cases: Some SBOM use cases require additional data about aspects of the software development and build environment, including those aspects that are enumerated in Executive Order 14028. How can SBOM data be integrated with this additional data in a modular fashion?

It should be possible that, when needed, all necessary information to reproduce a build should be discoverable from an SBOM.

g. Delivery. As noted above, multiple mechanisms exist to aid in SBOM discovery, as well as to enable access to SBOMs. Further mechanisms and standards may be needed, yet too many options may impose higher costs on either SBOM producers or consumers.

Different SBOM formats may evolve in varying ways, as communities improve upon prior formats in open and transparent ways. Ensuring that there is a version information for the SBOM format itself is going to be necessary, so that the ecosystem can evolve over time efficiently.

Ensuring a common set of minimum required fields, and enabling translation between multiple SBOM formats, can help to minimize siloing of data.

It's important to ensure that there are very easy-to-apply approaches for publicly sharing SBOM information.

Emerging projects such as Digital Bill of Materials (DBOM) will be useful to improve the delivery and provenance of the sharing of SBOM data, and automate risk analysis.

h. Depth. As noted above, while ideal SBOMs have the complete graph of the assembled software, not every software producer will be able or ready to share the entire graph.

SBOM formats such as SPDX allow anyone to require various depths in an SBOM. It is generally more of a policy decision to determine what depth is required for various use cases.

An SBOM format should be flexible enough to permit describing software at varying degrees of specificity, without being overly prescriptive about demanding a particular degree of detail. A format that enables more detail encourages software producers to improve the quality of the metadata about their own software.

A flexible format that also permits a lesser degree of detail can also lower the bar for newer or less-sophisticated software producers to begin generating SBOMs for their projects and products.

Providing less information may increase risks to recipients. Recipients can determine what level of risk they are comfortable with, and what level of specificity they will demand from their suppliers.

i. Vulnerabilities. Many of the use cases around SBOMs focus on known vulnerabilities. Some build on this by including vulnerability data in the SBOM itself. Others note that the existence and status of vulnerabilities can change over time, and there is no general guarantee or signal about whether the SBOM data is up-to-date relative to all relevant and applicable vulnerability data sources.

It should be possible to record what is known about vulnerabilities at a **specific point in time** in an SBOM, so that a coherent record can be used for assurance and insurance use cases. That said, vulnerability information is continually changing, so it must also be possible to link SBOM information with separate current information about vulnerabilities.

j. Risk Management. Not all vulnerabilities in software code put operators or users at real risk from software built using those vulnerable components, as the risk could be mitigated elsewhere or deemed to be negligible. One approach to managing this might be to communicate that software is "not affected" by a specific vulnerability through a Vulnerability Exploitability eXchange (or "VEX"),14 but other solutions may exist.

The primary purpose of an SBOM is to communicate objective, factual information about the contents of a software package. This data might include some aspects of subjective determinations about the included components, such as a published severity rating for a vulnerability. However, a particularized evaluation of "risk" when using a system may not always be suitable for inclusion in an SBOM. Risk should be evaluated differently depending on the software recipient's specific use case and method of using the received software.

That said, it should be possible to record (in an SBOM or by referring to it from an SBOM) that a given known vulnerability is not expected to be exploitable in a given larger system. In some systems, a component may have a vulnerability but it cannot be exploited in its larger context (e.g., that specific

vulnerability cannot be executed or the input to trigger it cannot be accepted). It is useful to have this kind of factual information.

4. Flexibility of implementation and potential requirements. If there are legitimate reasons why the above elements might be difficult to adopt or use for certain technologies, industries, or communities, how might the goals and use cases described above be fulfilled through alternate means? What accommodations and alternate approaches can deliver benefits while allowing for flexibility?

A key issue is ensuring that it is extremely easy for projects to produce SBOM information, and if necessary, encourage them to produce partial SBOM information instead of none. Projects have many other things to do, and it is better to enable incremental progress than to expect the perfect immediately.

As noted above, in some cases it may not be feasible to take a hash of a subcomponent that makes up a binary. Hashes should not be required below the individual file level (where a file could be a built object, a source file, a data file, etc.). Instead, there should be multiple ways to identify components and subcomponents to provide the flexibility necessary for adoption.

5. Other. NTIA invites comment on the full range of issues that may be presented in this Notice, including issues that are not specifically raised in the above questions.

The SBOM specification used should have certain properties:

- International standard. Many governments strongly prefer de jure standards. Choosing a format that is an international standard can increase the likelihood of widespread adoption.
- Publicly available standard. Some international standards are not publicly available, that is, fees must be paid to see the documentation. Standards that are not publicly available are far less likely to be adopted, and are more likely to be misapplied, because it is too expensive to buy all specifications relevant to the modern world.
- Support arbitrary ecosystems. Some SBOM formats are designed for a single ecosystem (e.g., npm JavaScript), making them insufficient for general SBOM purposes as systems are often created by combining a variety of subsystems from different ecosystems.
- Widespread implementation including OSS tooling support. Formats with little tooling to back it are unlikely to be widely used. Industry now widely uses and expects OSS tools; if there are few OSS tools to support a format, it is also unlikely to be widely used.
- Strong support for managing and describing OSS components. Modern codebases are mostly reused OSS components, even if the codebase as a whole is not. For example, the Synopsys "2021 Open Source Security and Risk Analysis Report" reports that codebases have 528 OSS

components on average (and that this average is growing). Their "2020 Open Source Security and Risk Analysis Report" reports that on average 70% of a codebase was OSS components.

- Support for multiple use cases. There are multiple uses for SBOMs, including identifying components with known vulnerabilities (a cybersecurity risk) and identifying potential licensing conflicts (a legal risk). A single specification that can cover multiple use cases is more likely to be adopted and reduces the risk of inconsistent data.