Software Bills of Materials: Motivation, Formats, Tools, and Challenges

version: 1.0 (2024-07-01)

Jan Hensel
Bremen, Germany
ja_he@uni-bremen.de

***Abstract*— An increasing awareness of threats to the so-called "supply chain" of software has spurred rapid developments in the area of software bills of materials (SBOMs), including regulatory efforts to mandate them. This paper not only explains what SBOMs are, both abstractly and in the context of their role in increasing the security of the software supply chain, but also provides a brief survey of concrete SBOM formats, discusses the processes involving SBOMs, and explores some of the tools that facilitate SBOM use today.**

***Index terms*—SBOM, Supply Chain, Security, SPDX, CycloneDX**

I. Introduction

When a new vulnerability is discovered in a software product it will mostly be corrected quickly, ideally even before the vulnerability information itself is made public. But even a timely fix does not guarantee that all users of that product are also supplied with said fix; this can be dependent on intermediaries (e.g., software vendors) and the users' diligence to stay up to date. This issue is far greater still when the software product is a library or package that other products depend on, as now the list of intermediaries from whom action is required grows. Especially when the dependency is ubiquitous, as was the case with the Log4Shell vulnerability discovered in the Java logging framework Log4j, the issues' scale is exacerbated even further, as it now impacts an exponentially broader set of users. All of these users, be they administrators or maintainers, for example, now must discover whether they are impacted by the vulnerability and, if so, take the necessary steps to update; possibly, they then also have to notify their respective users (say, they offer a service) that there was a chance of a compromise to their service quality (perhaps their data's confidentiality was compromised), who now have to take steps of their own. Clearly, this is a recursive problem with both potentially severe real-world consequences and a large amount of human intervention required; accordingly, there are efforts to automate these processes by their respective parts.

One such part is the question of "What is in this thing?", asked to answer the larger question of "Am I affected by this vulnerability?" For example, an administrator may wonder which, if any, of the myriad services on their machines include the Log4j dependency, even if only transitively. They may then wonder about the version of Log4j that is included in the service binary, or perhaps some details as to how it was built to determine, whether they need to bring this service up to date. This example shows both the potential for automation in this process of vulnerability scanning as well the likely limitations of that potential: Clearly, knowing which components include which other components (recursively) would allow answering the question of "what is in this thing": if the complete data to generate a hierarchy of interrelated components and their version details existed, it would only need to be collated. But the larger question of affectedness remains yet unanswered, as there are often complex conditions for exploitability such as certain configuration details and also as – depending on the vulnerability – exposure to the vulnerability is often contingent on the specific use of the dependency (e.g., whether a certain function is ever used, or perhaps whether a certain function is not used in some case). These are both extremely relevant questions for mitigating vulnerabilities, but the first one is the much lower-hanging fruit.

One of the main developments intended to tug on that fruit is the concept of a *software bill of materials* (SBOM) **[1]**. An SBOM, abstractly, should capture included components and how they are related as well as some meta-information. The name derives from the "traditional" *bill of materials* (BOM) in manufacturing[1] where a component of a final product, which itself has subcomponents, could be delivered with an attached document detailing exactly which parts it contains (by part numbers), such that when a part is found to be defective in its design anything including it can be identified and, e.g., recalled. SBOMs are intended to serve this purpose of transparency in the supply chain as well, merely for software, and using that knowledge to mitigate any threats to the supply chain.

This paper will first explore the idea of this "supply chain" for software and discuss the threats and risks facing it in some

[1]Some may call this type of BOM a *manufacturing bill of materials* (M-BOM)

more detail, especially insofar as it motivates SBOMs and how SBOMs should help with these issues (Section II); It will then go through the relevant concrete SBOM formats (Section III) and explore the practices and processes around SBOMs (Section IV) before also evaluating some of the tooling that is available for these processes and for working with SBOMs of the different formats in general (Section V). Finally, conclusions to take away from this paper are summarized in Section VI.

II. Background

A major aspect of SBOMs value proposition is enhancing the security of the current, interconnected software infrastructure, especially (but not exclusively) with regard to open-source software in it. This section will motivate that idea but also highlight some of its limits.

A. A "Software Supply Chain"

In Section I, there has already been a brief mention of the so-called "software supply chain". An actual ("non-software") supply chain for physical goods is the (recursive) chain of everything that is necessary to produce the good, from subcomponents to raw materials as well as any auxiliary goods and services that are essential to the manufacturing. A *software* supply chain, analogously, can be thought of as the chain of all the things the software product depends on. In reality this type of list can become enormous, given enough pedantry in assembling it, and needs to be limited in granularity (for instance, the office supply of coffee is likely out of scope, despite what developers may say about how integral it is). A rather simplistic view would only consider the chain of dependencies but the most common **[2]**–**[4]** interpretation is that of intermediate artifacts being exchanged between actors (developers, machines) and composited to arrive at either an intermediate or a final artifact. Figure 1, presents a version of this interpretation, where there is a pipeline from developers through the various machines responsible in facilitating development and generating deliverable artifacts, which also may pull in external dependencies.

This concept, however, is best taken as a shallow metaphor. **[5]** A supply chain for a physical good produced by a company will note the dependency on other physical goods that

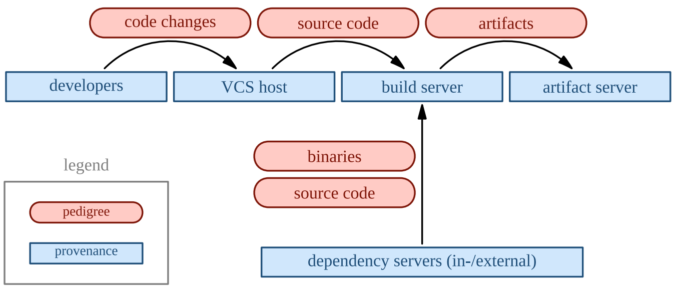

Figure 1: A simplified view of the "software supply chain".

are integral to the production of the main product. In contrast to physical supply chains, in the world of trivially copyable software a big part of software vendor's provided value is obviously not in shipping another copy of their library, but in promising a certain quality and in continually keeping their product up to date for their dependents to rely on. Further, for the production of physical products the sourcing of dependencies is supported by contracts, where the producing company purchases clearly specified subcomponents, raw materials, etc., from its suppliers. This is not the case in the modern software world, where almost any product significantly depends on open-source code, which is ubiquitous **[6]**, from apps all the way to the components driving modern cars, as anybody who takes a closer look at the relevant attribution and license reproductions by the manufacturer can also tell. Clearly, in this context, there is no contractual agreement for the delivery of the open-source code to the proprietary vendor, nor between interdependent open-source projects; their use is governed by whatever license terms apply, and the following excerpt from the MIT license **[7]** clearly sets the terms of that agreement:

> *THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT. IN NO EVENT SHALL THE AUTHORS OR COPYRIGHT HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER LIABILITY, WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING FROM, OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS IN THE SOFTWARE.*

Thus, while some widely-used projects may generally act as responsible software vendors, offering channels for reporting vulnerabilities, responsibly resolving those quickly, and disclosing them once fixed, this is not the case for all open-source projects; far from it! In any case, although the comparison to other supply chains may be tenuous, the "software supply chain" is certainly a useful descriptor for modern software products' dependence on externally developed components all the potential issues that come alongside the undisputed benefits.

Considering this supply chain, there are two main questions that arise regarding a software artifact: *what is in this thing*, and *where did this thing come from?* These questions correspond to the concepts of *provenance* and *pedigree*:

Provenance Information about the life cycle of an artifact, from its inception to its current state.

Pedigree Information regarding the constituent parts of an artifact.

In Figure 1, the presented view of the software supply chain is partitioned into provenance (blue) and pedigree (red). The developers and the various machines involved in storing and processing the software are part of the answer to the question of provenance. The intermediate artifacts, then, are the answer to the question of pedigree.

B. *Threats on the Supply Chain*

Considering the model of the supply chain in Figure 1, some threats are clearly identifiable. Any compromised machine in this chain represents a potential compromise to the integrity of the final product as well as its availability. For example, the VCS host being compromised would also threaten potential confidentiality of the code. Of course, developers can also be compromised and may, e.g., exfiltrate confidential code or inject of malicious code, threatening integrity. Thus, they also present an attack vector.

One major risk factor of a modern software product is, of course, the large set of external dependencies it will likely have and the potential for vulnerabilities they incur. Consider, for instance, the 2014 *Heartbleed* **[8]** vulnerability. This bug in OpenSSL's TLS implementation allowed attackers to read server memory, including all sorts of secrets. While it was introduced in 2012 and publicized in 2014, it took an unreasonable time to resolve and likely still affects some machines to this day. One worrying datapoint was, for example, a 2017 report by Shodan **[9]** which indicated that nearly 200,000 devices remained exposed by the vulnerability; two years later, over 90,000 devices reportedly **[10]** still remain affected. These numbers clearly indicate a slow response to a critical issue.

As another example, consider the widely-publicized 2021 *Log4shell* vulnerability **[11]** in the popular *Log4j* Java logging library. Here, if an attacker is able to influence the contents of a log message, they can use a specifically formatted[2] message to ultimately achieve a remote code execution, as long as certain, common, configuration conditions are met. **[12]** Log4shell affected a significant number of software products, as the deluge of scanners developed for it also attests. **[13]** And this sheer number of scanners also indicates that users and vendors did not have the desired level of insight into their software systems, having to rely on these external tools which, present an attack vector all of their own.

As US-president Biden's 2021 executive order **[14]** states it:

> *The development of commercial software often lacks transparency, sufficient focus on the ability of the software to resist attack, and adequate controls to prevent tampering by malicious actors. There is a pressing need to implement more rigorous and predictable mechanisms for ensuring that products function securely, and as intended.*

[2]containing a JNDI URL, to be exact

And the US government should be keenly aware of these threats. In late 2020, it became known publicly that US company *SolarWinds* was compromised by malicious actors, going back as early as October 2019, specifically in its *Orion* software, which had malicious code injected into it. **[15]** SolarWinds was supplying Orion to many customers, including parts of the US government, who used the software to monitor network infrastructure. In early 2021, the US government disclosed that 9 federal agencies were compromised, alongside 100 private sector companies. **[16]** In the conservative wording of the US deputy national security advisor, it took "an advanced persistent threat actor, likely of Russian origin" **[16]** who Microsoft hypothesize to employ "at Least 1,000 Engineers", **[17]** Mitigations employed and precautionary steps taken for such an attack must be equally as broad, a scale at which traceability is crucial.

The mode of attack against SolarWinds should also be noted: As they had compromised the build system, the attackers managed to inject code into the trusted software during the build step, the compiled software then being signed by the vendor and thus trusted by customers. **[18]** In his Turing Award lecture Ken Thompson walks through a possible injection of malicious code that mere analysis of source code could not detect, as it is the compilation step at which the code is injected. He poignantly titled this lecture "Reflections on Trusting Trust" **[19]** and his thoughts remain relevant: there is no way to be sure of the absence of a vulnerability and trusting a software supplier also implies trusting their trust, which one must be aware of.

Threats that may undermine that trust can be more obscure still: the *Rowhammer* bug can be used to flip bits in memory, *Spectre* and *Meltdown* are vulnerabilities in the speculative execution engine of many modern microprocessors that effectively allow unauthorized read-access to memory. These examples show the depth of information that may become necessary to trace the possible impacts of a compromised machine and underscore the extent of security related threats to the supply chain: machines or actors could be compromised, dependencies could have backdoors added to them, and simple human error while programming could cause vulnerabilities to crop up.

But there are further threats as well, such as end-of-life cases of crucial dependencies or compliance issues[3] and, the ever-present worries about license compliance that – though fairly static – can still pose a major threat to the supply chain.

Ultimately, it comes back to trust. Unavoidably, trust is given to the producers of microprocessors and DRAM units as well as the organizations and individuals who produce neces-

[3]Consider the case of a country being newly sanctioned, such that now it is no longer legal to use software components supplied by companies from that country. This could obviously force a rerouting of the supply chain to exclude any such components, which constitutes a "threat".

sary software. Trust must be given to employees who, in turn, put trust in their favored locksmith and the manufacturers of their IoT devices; trust is also given to open-source maintainers who cannot possibly verify completely, for instance, the originality of contributions they receive and instead must rely on trust to some extent. There is no way around some trust, and no surefire way to elide any possible threats that breaches of that trust might pose; but there are major gaps of verifiability and traceability in our current world of supplying software to each other and this is precisely where SBOMs should help.

C. *The Case for SBOM*

The use of SBOMs is directly motivated by the prevalence of these threats. Considering them each, there are two main areas of knowledge necessary to confidently tackle them: *what issues exist* and *do they affect the product?* For instance, to confidently eliminate concerns about dependency-caused vulnerabilities in a product, we would need to know both *which vulnerabilities do dependencies bring* and *which dependencies are included and how are included dependencies configured and used?* It is, of course, impossible to conclusively answer the first question, but new vulnerabilities are discovered daily and there are existing mechanisms[4] for automatically collating and leveraging this information[5]. However, the second question, whether a product is affected by a vulnerability, is sometimes much less straightforward to answer. The same seems true, e.g., for license compliance issues: license compatibility issues are explored at length and, while there is always the potential for a shift in legal interpretation, the much bigger issue is, yet again, the second question: *what is integrated, in which way is it integrated, and how is it licensed?* To summarize: there ought to be gains in threat mitigation to be made by helping answer questions of provenance and pedigree of software products.

As hinted at in Section I, SBOMs are intended to aid precisely with this: for a given software artifact they can convey which components it contains, which sources went into generating it, which tools were involved to assemble it, and which machines were involved. With this information, reacting to a newly disclosed vulnerability or a report of a compromised system becomes a less uncertain process, as the SBOM provides the majority[6] of information needed for seeking out potentially affected artifacts. How they may fit into the previously established "supply chain" view is shown by Figure 2.

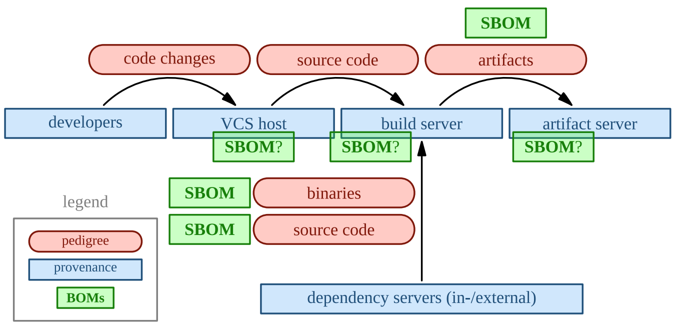

Figure 2: SBOMs in the "software supply chain", simplified.

[4]These are, necessarily, imperfect as well; more on them in Section IV.B.

[5]As will also be discussed, in detail, in Section IV.B.

Irrespective of their supposed merits, SBOM use may also simply become the status-quo, especially as there are regulatory efforts to mandate their use. For instance, the previously cited executive order **[14,** Section 4: Item e.vii**]** has SBOMs listed as one of the key approaches for improving security and sets the stage for mandating their use for government contracts, which follows up on a failed 2014 US-congressional bill **[20]**. On the European side, the *Cyber Resilience Act* (CRA) proposes "regulation on cybersecurity requirements for products with digital elements" and is currently brought forward by the European Commission. **[21]** The German BSI[7] issued a technical report **[22]** which outlines requirements of SBOMs and recommendations for their use, in order to support eventual compliance with the CRA. **[23]** The following section will detail what an SBOM could and should contain, taking this report into account.

D. *Contents of SBOMs*

In order to mitigate these threats as outlined an SBOM will need to hold certain information. At its core, an SBOM is a set of components which are related to each other, generally forming some sort of tree structure or directed acyclic[8] graph (DAG). It will typically be *about* something, e.g., a specific piece of software which anchors this graph.

SBOMs are a concept, though, rather than a specific format. In any context in which such documents are critical, there must be some basic set of mandatory information. This is also the case for the cybersecurity environment which both regulators (**[14]**, **[21]**)) and some industry players seem to be pushing for, and indeed there are numerous overlapping specifications and suggestions on minimum requirements for SBOM contents. **[22]**, **[24]**)

In Table 1 the similarities and differences between two requirements documents by governmental organizations, the US NTIA and the German BSI, are compared with each other. An obvious note is the omission of the license requirement on the American side, where a license field is only "recommended". **[24]** It is also important to note that while the NTIA states some additional fields (not listed in the table) as "recom-

[6]That there are always complex conditions even the most advanced SBOM format may not always be able to flexibly express is an issue also hit upon in Section IV.C.

[7]The *Bundesamt für Sicherheit in der Informationstechnik* is the federal office for information security in Germany.

[8]There is really no reason that it must be acyclic, but it would commonly be the case when the SBOM represents interdependent software packages, as dependency cycles usually are discouraged and often outright made impossible.

	NTIA	BSI[9]
SBOM	Author of SBOM Data	*Creator of SBOM*
	Timestamp	*Timestamp*
Component	Supplier Name	*Creator of the Component*
	Component Name	*Component Name*
	Version of the Component	*Version of the Component*
	Other Unique Identifiers	If unique identifiers exist, BSI requires them, listing them under "additional" fields
	Dependency Relationships	*Dependencies from other Components*
		License
		Hash-value of Executable Component

Table 1: The minimum elements of an SBOM, according to NTIA and BSI, and how they correspond to each other.

mended for consideration", while the BSI frames the minimum elements listed here as those that any SBOM will certainly have, but *requires* some additional fields (URIs of BOM, Code, Executable, additional Hashes, and unique IDs) *when they exist*, i.e., they must not be omitted. These requirements hint at the future shape of SBOMs, at least as they will be required by governments.

To summarize, an SBOM must have some meta-information about the document itself associated with it, as well as the information about the purpose of the document. And under this broad definition, an SBOM can describe a broad variety of things, not merely software components in the narrowest sense, which might called a *Source SBOM* or a *Build SBOM*, depending on whether it describes sources themselves or a package thereof. Besides these, they can describe later / higher stages in the deployment of a product, such as its deployment, which would be called a *Deployed SBOM*. Before deployment, sources may be analyzed, in the course of which an *Analyzed SBOM* might be created. This analysis, or the compilation, or many other steps, might be performed on other deployed systems, which may in turn be described by *Deployed SBOMs*, and so forth. These terms are descriptors of the lifecycle phases which an SBOM can describe, terminology also used, e.g., by the BSI **[22]**.

Much more important than these terms, though, is the intuition of this *chain of knowledge*, as it may be thought of: a crucial aspect of SBOMs' informational value is that this value compounds with the coverage it has over the real-world chain of components that are involved in creating the software products we all use. Having a description of the sources S of a deployed product, and then descriptions D_1 through D_n of every dependency with which S is compiled into binary executable C, is certainly desirable, e.g., from a security auditing perspective, but there is at least one glaring omission in the chain of production for C that should make it very hard to trust any description of C which may be given: there is no information about the system that compiled C from S and $D_1, \ldots, D_n$; who is to say whether a trustworthy, audited compiler was used, or a third party managed to inject their own malicious code into the build process? Having information on this machine would dramatically decrease uncertainty about C, understanding of course, that any level of detail in description could always have errors or omissions.

Understanding even these intricacies, and understanding the concept in general, there is an obvious question: how is this abstract concept expressed for real-world use, today? The answer, of course, comes via the various formats that are in use, which the next section will discuss.

III. Formats

SBOMs can come in many formats and encodings, some clearly specified in detailed documents, others born of necessity. Understanding their properties is important to discuss the use of SBOMs practically and to make an informed decision on these formats, as well as on associated tooling and processes. In this section, the most prominent formats will be discussed, both to understand how well they serve the requirements of SBOMs outlined previously but also to provide a background for the later sections.

First, the two most relevant formats, SPDX and CycloneDX, are discussed, in Section III.A and Section III.B, respectively. They are, for instance, the chief formats mandated[10] by the NTIA **[24]**; next to them, the NTIA also featured SWID, discussed here in Section III.C. Lastly, other formats are discussed, in Section III.D.

A. SPDX

The *Software Package Data Exchange* (SPDX) an open standard under the stewardship of the Linux Foundation, first published in 2011 and most recently revised (with version 2.3 **[25]**) in 2022.

Initially, SPDX was created in order to deal with license-compliance issues in open-source software and the project still presents itself as "first and foremost" dedicated to solving these issues. **[26]** The first association many may have with the project is the extensive and unambiguous license list[11] it maintains.

[9]The entries written in this column in *italics* are the author's translations of the German wording from [22].

[10]Although, this mandate may be broadened to include other formats, as [24] states.

[11]See https://spdx.org/licenses/.

But from this motivation, SPDX has been created as a general format to describe BOM data. An SPDX document always contains meta-information about itself, as well as blocks for package information, file information, snippet[12] information, and more. Elements listed in these blocks are put into relation with each other in separate relationship blocks, where blocks' unique identifiers (`SPDXID`s) are related to each other; this concept is taken so far that, e.g., the package that a document describes does not have to be the first block described in the document, but that anywhere in the document it is stated that the document (which has its own `SPDXID`) has a `DESCRIBES` relationship with the package block (given by its `SPDXID`).

An SPDX document can be expressed in various encodings. When expressed in the so-called "tag/value"-format, documents are blocks of tags, colon-delimited from their (potentially multiline) values, where each block corresponds to a type, such as the ones listed previously (e.g., package information). More familiar and perhaps interoperable encodings, such as JSON or YAML are supported as well, in which case, for the most part, the structure of independent blocks with unique identifiers separately related to each other stays the same. It is even possible to express SPDX documents as XLS spreadsheets.

After more than a decade of development SPDX continues to be a very capable and well-supported SBOM format, as its inclusion by BSI and NTIA **[22], [24]** testifies.

B. CycloneDX

The other major SBOM format that is commonly mentioned in the same breath as SPDX, is *CycloneDX* (sometimes, but rarely, abbreviated as CDX). It is an open specification as well, and backed by the OWASP foundation. CycloneDX was created in 2017 **[27]** and had its most recent version in 2023, with version 1.5. **[28]**

CycloneDX takes a different approach than SPDX, as the project notes: **[29]**

> *CycloneDX builds on top of the work SPDX has accomplished with license IDs, but varies greatly in its approach towards building a software bill of material specification.*

For instance, while relationships *can* also be specified separately from the component data itself, they can also be included directly, such that instead of expressing that X and Y have a "depends on" relationship, the information block for X can simply have its dependency Y listed. In CycloneDX as well, these relationships are usually expressible: dependency relationships are given by the top-level `dependencies`[13], constituent parts of the overall subject are expressible via `components`[14], recursively, other compositions being expressible in `compositions`[15], and relationships such as "built by" can be given in `formulation`[16]. Further, CycloneDX allows expressing the lifecycle phase of the SBOM, much as discussed in Section II.D, supporting even user-specified phases for organizations to fit their internal processes more closely. **[30, pg. 13]**

And there are more significant differences between SPDX and CycloneDX still. CycloneDX, in its current version 1.5, does not support snippets, which SPDX has supported for a long time. However, CycloneDX does plan to support snippets eventually **[31]** and the project also recently (in version 1.5) added support for another feature which was missing in comparison, annotations. The impression is that of a younger project that moves at a faster pace than SPDX.

As CycloneDX is the other specification both BSI and NTIA name **[22], [24]**, it is a fairly safe choice in the medium-term for complying with regulations.

C. SWID

Software Identification Tags (SWID tags) are an international (ISO) standard defined by ISO/IEC 19770-2:2015. SWID tags were created with the primary goal of aiding in software asset management (SAM), but they can also be used as a format for SBOMs, as their inclusion by the NTIA **[24]** underscores.

Syntactically, a SWID tag is an XML document that provides identifying information and metadata about a software product installed on a system. Semantically, its purpose is to be a clear identifier about a certain version of a product.

SWID is a sensible format, but when contrasted with SPDX and CycloneDX, it may not be as good a choice in the supply chain security context going forward. SWID tags were primarily designed for the *software asset management* (SAM) use case; while SPDX seems slower to evolve compared to CycloneDX, SWID is completely unchanged since 2015 **[32]**. SWID tags may not lend themselves as well to being consumer-generated (as opposed to producer-generated) and have a much stronger focus on SAM use cases such as tracking license information. And while the NTIA included the format in its recommendations **[24]**, it is absent from the guidance of the BSI, published two years later **[22]**. This omission seems likely to indicate SWID's diminished relevance in the SBOM space.

As a final example of a concrete specification, *Concise Software Identification* (CoSWID) tags are, in a nutshell, the binary,

[12]The term "snippet", in this context, refers to a piece of code. This could, for example, be the code for single function, a types definition and its method implementations. Especially in projects with a long history of contributors who may have copied over pieces of code from elsewhere that may be subject to their own licensing terms, this concept can be necessary to track.

[13]See https://cyclonedx.org/docs/1.5/json/#dependencies.
[14]See https://cyclonedx.org/docs/1.5/json/#components.
[15]See https://cyclonedx.org/docs/1.5/json/#compositions.
[16]See https://cyclonedx.org/docs/1.5/json/#formulation.

space-efficient counterpart to the plain-text and verbosely-XML encoded SWID tags. CoSWID tags are defined in the quite recent RFC 9393 **[33]** but will not factor into later discussions on tooling due to their recency and the resulting lack of support for it in mature tooling.

D. Miscellaneous Formats

But not only these standardized formats fit the idea of an SBOM, which is to describe software and perhaps hardware components and their relationships.

In a way, the data encoded in a Go `go.mod` file or a Rust `Cargo.toml` file is already a sort of SBOM, not for a concrete distributable version of the product, but perhaps for the product in general. The lockfiles (e.g., `go.sum`, `Cargo.lock`) then have the information for a specific state of code. Certainly even fairly basic tools using these data are then to be considered SBOMs, such as GitHub's dependency graph feature, which shows a repository's dependencies (as inferred by the content of such package manager meta-files). And many modern languages have a similar concept of package management either as enforced or as de-facto standards, so Node.js projects (`package.json`, `lock.json`), Dart projects (`pubspec.yaml`, `pubspec.lock`), and many more can be supported by such tooling.

Further, SBOM data has been useful to necessary for at least some people and organizations for a long time. Accordingly, long before any of the current SBOM standards and practices were formalized, spreadsheets were (and often still are) used for tracking the current state of dependencies that a certain product used or needed.

And even in this era of hopefully-sensible standardized formats, many tools still support their own formats as well. Oftentimes this is preferable, when the data is presented directly to a human, rather than the next machine in line; here, aligned, perhaps colored, or otherwise decorated formats may make the most sense.

IV. Practices and Processes

The previous sections clarified the concept of an SBOM and gave an overview of the concrete formats that allow expressing it. In this section, this foundational knowledge is prerequisite for understanding the processes which are supposed to ultimately leverage SBOMs in order to improve, for example, supply chain security. First, however, it is also important to understand what operations are common on SBOM data, not only to understand the discussion of these processes in Section IV.C, but also to understand the categorization of tools further on, in Section V.

A. Operations

The first question that must be answered, in order to arrive at any working SBOM-based processes or run SBOM-based analyses, is how to arrive at an SBOM. In other words, it is the issue of **SBOM generation**.

To some developers, the issue may seem trivial: for a Go project, simply look at the package-management files (`go.mod` and `go.sum`), start with those dependencies, and retrieve dependencies recursively[17] until complete. Unfortunately, however, the issue is not as straightforward in all cases. This compiled Go program might depend on non-Go dependencies such as OpenSSL; it will most likely depend on glibc[18] as well. And then, who is to say that it doesn't actually rely on an arbitrary binary file which it loads and executes? Clearly, there are cases in which a complete SBOM can not be generated by a generalized tool.

And then, there are languages and ecosystems in which even the initial assumptions about a package-manager do not hold. We may speak of **package-managed** and **package-unmanaged** environments, where Go or Rust (at least de-facto) belong to the former category, while C and C++ belong to the latter. A JetBrains survey **[34]** illustrates the (unsurprising) state of affairs in these languages: there is neither a de-facto standard method of package-/dependency-management in these languages, nor a de-facto standardized build system; take with that a resulting openness to "unorthodox" ways of including dependencies and the longer history that some of these projects have, and you arrive at SBOM generation for C and C++ based projects becoming a rather messy proposition.

Depending on downstream requirements, if it becomes necessary to deliver complete, self-contained documents, SBOMs for subcomponents, tools, machines, etc. have to be retrieved from their reference locations and integrated into one single document. From this alone, but not only this, at least two indispensable operations follow: **conversion** between SBOM formats and encodings as well as **merging** SBOM documents together.

When doing so, however, the organization or person taking responsibility for the final SBOM would do well to not blindly trust any tools offering these features, as silent omission of some information is a real danger, especially in cases where fields from one format do not quite map to fields of the other. And whether that omission will get noticed in megabyte-scale SBOM files depends on how well-equipped this organization or person is: At the very least, they would employ some tools of **quality assurance**, making sure no crucial information got lost in translation. Further, they will need to be able to inspect and modify SBOM contents by hand, at least in some cases, which might be called **editing** of SBOM data. Finally, for some organizations it may even make the most sense to **build specialized tools** internally for the exact verification of data or the exact conversion or merging functionality that is desired.

[17] In fact, the lock-file should already be complete.

[18] See https://www.gnu.org/software/libc/.

In any non-trivial organization, the **management** of SBOMs will also be a crucial operation. Work on SBOMs published by the organization may not only involve a mix of automatic generation and completion by humans familiar with the intricacies of the product, but also involve non-technical personnel, such as the legal department in regards to licensing matters. For many companies, internal processes require human sign-off to clarify who is responsible for a certain part of the information; it may be required for formats and tools to support a staged workflow where SBOMs are processed in stages and – perhaps even in their sections and subsections – can be moved along a workflow, such as "needs review", "under review", "signed off by …". Such a system should ideally also allow the publishing of SBOMs, to customers or in general, and facilitate their cryptographic signing and signature verification. Lastly, it seems likely that such a system would have a graphical user interface, for example as a web application; as such, it would make sense to integrate in it the results of any further analyses, to give a wider range of people, including management, insight into the state of a product.

The main kinds of these analyses, then, are those that address the threats outlined in Section II.B. One, perhaps rather simple, analysis is ensuring that no component comes from a banned supplier or originates in a sanctioned country, so as not to violate any internal policies, customer-mandated requirements, or regulatory obligations. Similarly, licenses of all included components can be analyzed for compatibility[19], including how they depend on each other[20].

B. Vulnerability Scanning

As pointed out in Section II.B, one of the main motivators for SBOMs is the automatic management of vulnerabilities in their impact on a given software component. This section will give an overview of the problem of vulnerability analysis and explain where SBOMs fit into that process, how they help, and , what the problem's requirements tell us about the SBOMs necessary informational value.

A complete SBOM should at least list all constituent parts of the component; keeping track of newly-discovered vulnerabilities for each of all subcomponents is a necessity to securing the overall software component.

One way to track vulnerability information is through the *National Vulnerability Database* (NVD). It is a repository of vulnerabilities that have been reported and analyzed by the security community and is maintained by NIST. For a long time the NVD has offered publicly accessible data feeds, and more recently APIs[21], of vulnerabilities found in software products. Through these data sources, vulnerability data can be retrieved for certain software components.

To query this database for information relevant to a certain product or component, one could filter by a *Common Platform Enumeration* (CPE) identifier. These are relatively brief strings of characters that allow to express a software platforms identity in as much detail, as is desired. For instance, here are strings representing OpenSSL in version `0.9.2b` and an older version of the SerenityOS desktop operating system, respectively:

```
cpe:2.3:a:openssl:openssl:0.9.2b:*:*:*:*:*:*:*
cpe:2.3:o:serenityos:serenityos:2019-12-30:*:*:*:*:*:*:*
```

The data returned are so-called *Common Vulnerabilities and Exposures* (CVEs), specifically, their identifiers; some of these were already cited in Section II. A CVE represents a specific vulnerability in a specific software product (as identified by CPE), describes what the vulnerability is, how it may be exploited, how it should be mitigated (crucially, which version fixes the vulnerability). CVEs are also assigned severity scores in the *Common Vulnerability Scoring System* (CVSS). These scores should help prioritize mitigation work, but have also been criticized **[35]** for incorrect or inflationary severity estimates.

The NVD certainly provides very useful information and should be a core data source for any vulnerability analysis performed, but it does have its downsides as well. Many components, for example, are not identified; the now commonly known Log4j has its own CPE identifier, but other similar packages in other languages (for example the *zerolog*[22] package for Go), are not identified. Another form of identifier for software components are *package URLs* (purls[23]); for example, *zerolog* could be identified by the following purl:

```
pkg:golang/github.com/rs/zerolog@v1.26.1
```

While for CVEs there is a central body assigning identifiers, purl instead relies on its specification reglementing how identifiers are formulated for identifier coherence.

The pace of the NVD has also been criticized, specifically that vulnerabilities were frequently published through other channels (such as social media platforms) before the NVD added them, sometimes with delays of several days. **[36]** Projects for which CVEs were filed have also found fault with the accuracy of the data provided, and, again, the slow reaction to

[19]This is, of course, only to the most current legal interpretation of these licenses, which requires lawyers to correctly understand judgements on these issues.

[20]For instance, it is commonly accepted that MIT-licensed components can be directly included in compiled executables, while GPLv3-licensed components may only be linked against, i.e., "referenced" by the compiled executable and not "part of it", as such.

[21]In fact, the feeds are being deprecated by the end of 2023 in favor of the APIs.

[22]See https://pkg.go.dev/github.com/rs/zerolog.

[23]The non-capitalization of "purl" is commonly used for purls, perhaps also serving to distinguish from *Persistent URLs* (PURLs).

corrections. **[35], [37]** For the vulnerability analysis of many projects it is therefore advisable to augment the NVD with other sources of data.

One such alternative source is the *GitHub Advisory Database*[24] which identifies vulnerabilities by separate identifiers, the *GitHub Security Advisory* (GHSA) identifiers, and allows scoring by CVSS. GitHub sources the data for this database from multiple databases besides the NVD, such as the *Go Vulnerability Database*[25], the *Rust Security Advisory Database*[26], or Google's list of *OSS-Fuzz*-found vulnerabilities[27]. More importantly though, GitHub also includes community contributions[28] to the data, allowing developers to declare vulnerabilities in their own packages. **[38]**

To many consumers of CVE data, the GitHub Advisory Database may be preferable to the NVD data. **[39]** For one, consuming from the NVD can be somewhat confusing as the data is actually, in a way, downstream from the CVE List[29] of the *MITRE Corporation*[30], but enriched with further data, such as scoring. **[40]** Despite this enrichment, as already noted above, it can be arduous to amend or correct CVE data in the NVD. GitHub, on the other hand, is usually more flexible in allowing reasonable, well-sourced changes to the CVE data they ultimately pass on.

Another major part of the value proposition of GitHub's advisory database is also its inclusion in the automated tool *Dependabot* on the GitHub platform, which notifies developers automatically of vulnerabilities which potentially affect their projects. While this inference, from experience, leaves room for false-positives, this automatic workflow of notifying developers with a possible mitigation ready to be applied is clearly very promising approach to lessen the friction for the necessarily involved developers (those humans! **[41]**). Unfortunately, though, these features are only available, and likely only feasible to implement, for the more modern, package-managed languages and ecosystems.

The GitHub Advisory Database and GHSAs seem to be a step in the right direction, to ease the friction of vulnerability analysis and mitigation as well as to allow sourcing vulnerability data directly from developers and other volunteers. These contributions are possible, in part, due to an open format specification by the *Open Source Security Foundation* (OSSF or OpenSSF): the *Open Source Vulnerability* (OSV) format[31] is intended to be the standard interchange format for vulnerabilities between different vulnerability databases. Evaluating this format in detail, however, is beyond the scope of this paper.

Further databases exist. Besides the name for the format, "OSV" is also frequently used to describe the *OSV Vulnerability Database*[32], which is perhaps the most immediately usable of these databases, due to the quality of the API documentation and the little introductory blurbs on the homepage. Another database leveraging OSV formatted vulnerabilities and aggregating them is the *GitLab Advisory Database*[33], which works similarly to GitHub's offering. *RedHat* also publishes[34] security advisories. Lastly, some tools, such as *cURL*[35], publish their own security advisories in more or less machine-friendly formats.

To summarize, a vulnerability analysis tool that ingests an SBOM should, for the contained components, leverage data not only from the NVD but also from other databases, at least the GitHub advisory database, likely specializing queries based on the language ecosystem in question. The retrieved data may not only be in different formats, but also differ in detail, quality, and trustworthiness; a tool should account for this in further processing. Depending on the tool in question, it may be a desirable feature to apply vulnerability mitigations to the product automatically. For instance, GitHub *Dependabot* can automatically create pull requests to resolve certain presumed vulnerabilities (usually by updating dependency versions). However, it is unlikely that such a tool could provide this feature without having to involve humans and, further, seems very unlikely from a current state of the information that impact analysis and mitigation generation could be generalized sufficiently to aid with most true vulnerabilities. In the following section (Section IV.C), we will also detail another approach being taken in this field, VEX.

C. *Beyond SBOM*

SBOMs are documents, nothing more, nothing less. As such, SBOMs alone can hardly "guarantee" any measure of security by themselves, no more than a shipping manifest can guarantee the contents of a shipping container. The document is simply a (syntactically and semantically constrained) medium for the information; around this, there must exist a process, just as there exists a process in the shipping industry due to which a commercial port may then trust a container to contain harmless rubber duckies rather than explosive chemicals.

This is why the NTIA encouraged **[24]** the adoption of "practices and processes" around SBOM actually leverage the value of these documents. The question then becomes, what could such a process look like? There are already a few descrip-

[24]See https://github.com/advisories.

[25]See https://pkg.go.dev/vuln/.

[26]See https://rustsec.org/.

[27]See https://github.com/google/oss-fuzz-vulns.

[28]See https://github.com/github/advisory-database/pulls.

[29]See https://www.cve.org/.

[30]MITRE is a not-for-profit organization funded, in this respect, by the US *Department of Homeland Security*, specifically, the *Cybersecurity & Infrastructure Security Agency* (CISA). They began work on the CVE List in the late 1990s whereas the NVD only came to be in the mid 2000s. [40]

[31]See https://ossf.github.io/osv-schema/.

[32]See https://osv.dev/.

[33]See https://advisories.gitlab.com/.

[34]See https://access.redhat.com/security/security-updates/.

[35]See https://curl.se/docs/security.html.

tions of such processes, usually referred to as "frameworks". One of these is the *Supply-chain Levels for Software Artifacts* (SLSA) framework, which will be described with some detail in the following as an example of such a framework, which focuses on the security of the software supply chain. Further examples are OWASP *Software Assurance Maturity Model* (SAMM), which is scoped much more broadly than SLSA, and the NIST *Secure Software Development Framework* (SSDF), which is likely most fitting for suppliers of government software products.

SLSA primarily specifies a set of guidelines for different levels of security, with the goal of enabling more trust and verifiability with respect to software artifacts. As of now, these all relate to the build step of an artifact, but should be expanded to sources and dependencies in the future. One of the most obvious insights about any process's adoption is that an incremental approach is much more achievable than a sudden, all-or-nothing overhaul. Accordingly, SLSA security levels are defined such that organizations can work their way up to the desired level from the so-called level 0, i.e., not delivering anything besides the binary. Table 2 shows a summary of the SLSA 1.0 **[42]** levels for the build track and their requirements.

Level	Requires
Build L0	nothing
Build L1	provenance information
Build L2	*signed* provenance information
Build L3	hardened build platform

Table 2: SLSA build-track levels.

SBOMs are a good choice to encode the provenance information and therefore very useful for SLSA. **[43]** However, they cannot directly help hardening the build platform, and while it should be possible to express the hardening measures taken, this is not the primary intended use case for the major SBOM formats. The SBOM may convey the checksums of the data it is to represent, as generated by the tool producing it, and then it itself may be signed by whoever is responsible for delivery, but their diligence in confirming the exact accuracy of all information may waiver.

Rather than trying to fit all possible information into an SBOM, it makes more sense to have higher-level documents that reference these SBOMs, as well as other artifacts, to then assert statements about those SBOMs and artifacts in a format that is specifically suited to those assertions. This is where the *in-toto* framework[36] should be useful: it allows, for instance, expressing which steps were run, which sources were used, the result of a code review, test results, or the configuration of the compiler that was used. **[44]** This is why SLSA recommends in-toto as the single suite of formats and conventions, choosing underlying predicates (such as SBOM for provenance) as necessary. **[45]**

Tools and concrete processes in this space are still being developed very actively. **[46], [47]** Due to this, and due to the fact that these processes extend beyond SBOM quite significantly, these tools are not discussed in Section V; however, an in-depth assessment of them would be of interest.

SLSA, as a quite abstract framework, recommends a set of practices for certain "levels" of security. As a concrete framework for expressing and generating the attestation information, SLSA recommend in-toto, which in turn supports referencing SBOM, which may encode the provenance information SLSA requires. In this constellation, SLSA hope to address some of the areas of improvement which they identify in the SBOM space: **[43]**

> “
> - *SBOMs currently don't include or require enough information to help users respond to build tampering and attacks […];*
> - *There's no well-established ecosystem to easily distribute and verify SBOM documents;*
> - *The most common method of generating SBOMs using only audit tools after the software's creation can result in less accurate SBOMs.*

But even having perfect and trustworthy provenance information, even knowing exactly which subcomponents it includes (i.e., pedigree information), it is often not clear whether a product is affected by a vulnerability in a certain subcomponent, as already hinted at multiple times in Section IV.B. This is because exploitability often depends on the *configuration* of a component, not merely its inclusion. For instance, the exploitability of the *Spring4Shell* vulnerability appearing in the *Spring Framework*[37] for Java depended on a certain pattern taken in the code, using the annotations the framework provides. In other cases, separate configuration files or even a dynamically set configuration may be relevant. Clearly, there can be no perfect format to convey any possible such condition (or combination thereof) from one machine to the next; while there probably still is much to gain by trying to automate these processes, ultimately, the human in the loop seems unlikely to become obsolete anytime soon.

To communicate vulnerability information to these humans and help them identify whether they are impacted by something and what mitigation measures to take, there exists another concept: *Vulnerability Exploitability Exchange* (VEX) is a document format proposed by the NTIA **[48]** to allow vendors to express vulnerability affectedness (including not being affected, looking into something, or already having resolved

[36]See https://in-toto.io/.

[37]See https://spring.io/projects/spring-framework.

the issue). VEX information is separate from an SBOM, as the SBOM is static due to its information (such as dependencies or licensing) being (hopefully) known at its creation time while VEX information is dependent on security research performed mainly after the product (and the SBOM) has been published. When a security researcher raises a vulnerability with a dependency, for instance, a vendor can then publish its status respective that vulnerability by publishing VEX documents. These documents may then say that an investigation is underway, and, later on, that the product is deemed "not affected" by the vendor (e.g., due to the use of the dependency).

V. Tooling

This final major section of the paper will give an overview and evaluation of the tools that are available in the current SBOM ecosystem to perform the operations discussed in Section IV.A, and support processes, such as the ones named in Section IV.C. The list of tools is necessarily incomplete and will become outdated quite quickly, as these tools continue to evolve, new tools are developed, and processes shift. Nonetheless, it should hopefully give readers new to the field a good idea of what tools may suite their use cases. The evaluation of these tools stems from personal experience since 2021 as well as dedicated research for this paper and the accompanying presentation. Unfortunately, it is impossible to make such a section both brief and complete, so brevity is favored and some tools are therefore omitted. For most of these tools it also is not possible to make an "apples to apples" comparison between them, as they often take different approaches, suit different use cases, and perhaps even complement each other rather than "competing" for the same exact niche.

Further, it is important to point out that, the practices and processes are still being developed, shaped, and consensus on in-practice standards has yet to be determined. Accordingly, the landscape of tools is shifting and incomplete and it is unreasonable to expect *the one tool* to handle any SBOM needs, as even the most capable developer making such a tool could only guess at the SBOM requirements, practices, and processes to come. Instead, here as well, the Unix philosophy, as quotedly stated by Doug McIlroy, applies: **[49]**

> *Write programs that **do one thing and do it well**. Write programs to **work together**. Write programs to handle text streams, because that is a universal interface.*
>
> (emphasis: author)

It seems clear that the first two points, as emphasized, are the truly crucial points for the SBOM use-case. The *universal interface* seems more important than the text streams that are supposed to provide it; perhaps the SBOM formats can stand in as this interface instead, as they are already meant to be the universal, interoperable interface between organizations, people, and tools. Users, at least for now, should look to be able to compose these tools to have the necessary flexibility in their SBOM use.

A. Generation

The first operation to tackle, as in Section IV.A, is the **generation** of an SBOM for a project. As noted already, it is *impossible* for a single tool to handle any, arbitrarily complex, project or system. That said, for straightforward projects in modern languages and ecosystems, generation of fairly complete SBOMs *is* a feasible problem to solve, and some tools already do it quite well. Users are nonetheless advised to ensure that output in content (list of components) and shape (relationship of components) matches their expectations.

A simple and straightforward example of such a generator is the *SPDX SBOM Generator*[38], which has no official affiliation with the SPDX project. It supports the simple, package-managed use-cases outlined previously (Go Modules, Cargo for Rust, Maven for Java, …) and generates a correct SBOM from them, containing all the components, properly identified, in their correct hierarchical order. However, this tool does not deal in license information inference; in a language like Go, where the package-management files do not contain license data, the concluded BOM will simply (and not incorrectly) state `NOASSERTION` for each component's license information.

An example of a tool which follows a slightly different idea for generation is *gh-sbom*[39]. This tool by GitHub's own *Advanced Security* organization[40], is in a few ways emblematic of the current SBOM tooling landscape: it relies heavily on external services (which is by no means a criticism), going so far as to piggyback on the GitHub CLI tool `gh` as an extension and subcommand thereof; it generates CycloneDX and SPDX in JSON encoding, but of very varying quality with no indication thereof; but when it works, it works quite nicely.

A user must have authenticated in the GitHub CLI and have installed this tool. They can then generate an SBOM to standard output (ready to be piped into `jq`, for instance) for the current repository, by default in SPDX-JSON format, but optionally (via `-c`) also instead as CycloneDX-JSON. This default is somewhat surprising, as, from limited testing at least, the generated CycloneDX BOM is more comprehensive than the SPDX BOM: the SPDX BOM simply includes no package identifiers, neither purl nor CPE, even though this information is included in the CycloneDX BOM (and is absolutely expressible in SPDX, using `ExternalRef`, for instance). Another symptom both of this difference in BOM quality as well as of surprising defaults is the crucial matter of license information: the tool only produces these when the `-l` flag is given, and only

[38]See https://github.com/opensbom-generator/spdx-sbom-generator.
[39]See https://github.com/advanced-security/gh-sbom.
[40]See https://github.com/advanced-security.

does so properly for CycloneDX, omitting all but the top-level package's license information in the SPDX case.

In its current state, at the very least, this tool is a convincing demonstration of what inference is possible, even for a very small tool, with rather quick generation times leveraging the APIs of a platform such as GitHub and using further external tools, such as ClearlyDefined[41], which it uses for license inference.

The perhaps most well-known and popular SBOM generation tool is *Syft*[42]. Being developed by *Anchore*, it is primarily portrayed as a SBOM generation tool for container images and file systems, but it is also capable of generating SBOMs for software projects, such as source code repositories.

For container images, Syft works well out of the box with no configuration. For other projects, however, some configuration may be required; here, Syft will usually be invoked to scan a directory (`dir:`) and the automatic inference of which information to focus on can lead to incomplete results, meaning it is sometimes advisable to constrain the used catalogers or to exclude certain files or directories from the scan. Information quality can also vary based on the output format selected; SPDX and CycloneDX each end up having their own shortcomings, but even the most informative format, Syft's own BOM format (JSON encoded), ends up compacting relationships of transitive into direct ones, flattening the entire dependency tree into one layer.

B. Conversion, Merging, and Editing

Especially considering that different tools output different formats and encodings, **conversion** and **merging** are indispensable tasks, likely for a long time to come. And while format-internal conversion between encodings is usually no problem, converting between even the major formats can lead to loss of information. There is a plethora of conversion tools out there; one of the tools mentioned previously, Syft, offers its own conversion between formats, for instance. The SPDX and CycloneDX projects each offer converter tools, both between the formats as well as to format-internally convert between the various encodings, such as JSON and XML. For merging, the most prominent tool is the *CycloneDX CLI*[43], but there are some requirements of the shape of the BOMs.

Unfortunately, it is impossible to give any authoritative recommendation on these tools at this time: the intricacies of format conversion and which not directly mappable fields get converted can be complicated to assess reliably in the first place, but as best practices are currently not even fully developed, it would be futile. Once the practices have been better established and thus requirements are clearer, once tools have matured, then a reliable analysis can be made.

[41]See https://clearlydefined.io.

[42]See https://github.com/anchore/syft.

[43]See https://github.com/CycloneDX/cyclonedx-cli.

Despite this, there are some tools, even now, that aim to assess SBOM quality. A simple example is eBay's *sbom-scorecard* tool, which generates a simple assessment of the proverbial boxes a certain SBOM ticks, e.g., how many of the listed components have associated license information. Another very similar tool is *sbomqs*[44], which is used to assemble the interesting site *SBOM Benchmark*[45]. Hopefully, this niche of SBOM tools continues to evolve and will nudge other tools and providers toward providing complete and useful documents.

At least for now, users will likely have to investigate the type of documents they receive, the type they are to provide, and inspect the tools they will use closely for matching those requirements. As pointed out in Section IV.A, at least for some organizations, it will likely be necessary to edit SBOMs directly. While SBOMs are ultimately just text files, there must be appropriate tools to work with them, much like most programs are simple (collections of) text files and yet there is a wealth of supportive tooling for programmers to edit them. In this vein, such tooling for the major SBOM standards could be very useful.[46]

Finally, it is not infeasible for organizations that have a real long-term need for a certain SBOM use case, to develop their own tools. The CycloneDX and SPDX projects both have good client libraries in major languages, such as Go, and many of the other promising projects in the SBOM space are open-source and permissively licensed, allowing tools to be built on top of them[47]. Hopefully, those that do, see reason to open-source these tools as well.

C. Vulnerability Analysis

The process of vulnerability analysis has already been discussed in Section IV.B, and has the potential (and difficulties) of an automated or semi-automated management of those vulnerabilities. In this section, a small selection of prominent tools in this space are presented in brief.

Likely the most prominent SBOM focused open-source vulnerability scanner out there at this time is *Grype*[48]. Created by Anchore, it is the partner tool for Syft, but also integrates some of its functionality.

Grype combines a lot of data sources in its own database; the code for generating such a database is also made available by Anchore, in *grype-db*[49]. Besides NVD data and GitHub's aggregated data, Grype pulls data from the feeds and trackers of

[44]See https://github.com/interlynk-io/sbomqs.

[45]See https://sbombenchmark.dev/.

[46]It is for this reason I have prototyped an SPDX language server implementation, which works quite well for simple quality-of-life things such as automatically filling out checksums or jumping to definitions / listing references, but it has not received any work in some time.

[47]In Go, for example, any package that is not located under `internal` is available for inclusion in other projects.

[48]See https://github.com/anchore/grype.

[49]See https://github.com/anchore/grype-db.

several Linux distributions as well, in keeping with its primary purpose as a container image vulnerability scanner.

Grype has rather good support for SBOM ingestion (with the scheme prefix `sbom:`) and can even add missing CPEs to the data it is given.

Another solid option in this category is *CVE-Binary-Tool*. This CLI tool developed by *Intel*, despite its name, aggregates a wide variety of sources for its vulnerability information: it was initially created to scan binaries, but is also capable of ingesting SBOM. While its output, by default, is suited to human consumption on the command line, it can be adjusted to a more machine-friendly format and ready-to-use CI solutions such as a GitHub action exist.

To execute `cve-bin-tool` with an SBOM, the file must be passed as an argument and the format should also be specified. It should be noted, that without clear identifiers (see Section IV.B) in the SBOM data, such a tool can not easily find components in the vulnerability data; unfortunately, the tool does not warn of this fact, though, highlighting the importance of being aware of SBOM-quality before use.

There are other projects in this vein, such as *OSV-Scanner*[50] by Google, which is a direct part of the OSV effort already discussed in Section IV.B.

D. Management

The tools listed so far are scoped fairly narrowly, some of them only performing a single main task. They can be categorized as the tools to arrive at a complete SBOM (which requires generation, merging, and conversion) and then analysis of such an SBOM based on external data (by the example of vulnerability scanning). Anybody planning to work in this fashion would have to compose these tools themselves; they might also have to manage lots of SBOMs somewhere or eventually deliver SBOMs, which, in a DIY approach would involve some work and friction as well. Therefore, there are tools that attempt to cover more of this workflow, which also try to take on the management responsibilities. These tools can be categorized as SBOM management tools, and they are an interesting indicator of the possible future direction of SBOM use.

It is difficult, from a current-day perspective, to envision and design "The Authoritative SBOM Management Suite". For one, the best practices such a tool should support and laud are not fully established and still shifting, as noted multiple times already. Whoever would be developing this tool would have to be tapped into multiple industries, know both software producers and consumers well, including, ideally, governmental entities, as well as the large space of open-source suppliers and consumers; besides all this, they would need the resources to develop an elaborate tool suitable for multiple use cases and have the foresight to design it to fit, or be adaptable to, the next decade of SBOM use. Instead, of course, several tools are being developed by several organizations, each with their own goals and incentives.

One upside to attempting to develop such a challenging project, however, is the influence it may give over the shape of things to come. It is unsurprising, then, that there are multiple companies developing paid products for the overall management of SBOMs. One company at the forefront of SBOM efforts is Anchore, which was already mentioned multiple times. Its offering, *Anchore Enterprise*[51], is categorized by Anchore as an "SBOM-powered supply chain management platform". Another platform is that of *Cybellum*[52], who focus on the manufacturing sector's various security and risk management needs. Finally, the perhaps most pure SBOM management in the commercial space is *Cybeats SBOM Studio*[53]. It is, however, difficult to assess commercial tools, even having had some of them demonstrated, without expensive hands-on experience; thus, this incomplete overview of the commercial tools will have to suffice.

There are, of course, also existing, commercially successful products which are slowly adding SBOM support. An example of such a tool is Synopsys' *BlackDuck*[54], a *software composition analysis* (SCA) tool touting features from advanced scanning, through vulnerability analysis, to managing all related data (including SBOM) for customers. For such actors with their own established market niches, the incentives to support SBOM may be lower, and the desire for continued customer lock-in could be a factor in SBOM adoption steps, such as adopting ingestion before data export.

But there are freely available tools as well! The predominant tool among them is *Dependency-Track*[55], another offering from under the OWASP umbrella and closely related[56] to the CycloneDX project. Dependency-Track, despite its name, offers a very broad range of SBOM management and inference features: it ingests SBOMs and compares against vulnerability data, checks customizable compliance policies, exposes a featureful API, and presents its information (and some simpler functionality) as a web application. None of this truly sets it apart from the other mentioned tools, but it is possibly the most complete and mature tool in the space, and gives a good idea of how SBOM management works right now.

[50]See https://github.com/google/osv-scanner.

[51]See https://anchore.com/platform/.

[52]See https://cybellum.com/platform/.

[53]See https://www.cybeats.com/sbom-studio.

[54]See https://www.synopsys.com/software-integrity/security-testing/software-composition-analysis.html.

[55]See https://github.com/DependencyTrack/dependency-track.

[56]e.g., in its contributors

VI. Conclusion

There are a number of separate takeaways readers should take with them from this paper:

1. **Those looking for guidance on compliance are best served by the NTIA's "Minimum Elements" [24] and the BSI's TR-03183-2 [22].**
   Which of these documents is more relevant will likely primarily depend on the intended market. However, the BSI requirements are slightly more demanding and seem to set up for the Cyber Resilience Act **[21]** and, as the more recent document, may set the tone for further legislation.
2. **For those having to choose a format today: SPDX and CycloneDX are the relevant formats, with no clear favorite between them.**
   Both of these formats serve the SBOM use case and are supported by a broad variety of tools. While SPDX has a longer history, CycloneDX iterates more quickly and may be faster to adopt additional features as they become useful. While conversion between the formats is possible and supported by tools, it can lead to loss of information and preferring one format over the other may be advisable. Considering that CycloneDX is the one that is directly supported by Dependency-Track, the premier free management platform, it can be speculated that CycloneDX may have a slight edge.
3. **Anybody needing to work with SBOMs today should explore a variety of tools.**
   The tooling ecosystem is still quite young and in flux. While this paper gives an overview over the currently common tools and some of their capabilities and limitations, any tools should be evaluated critically. Rather than looking for "The One Tool" for everything, users should look to be comfortable working with SBOMs and composing tools to fit their needs. As SBOMs are, ultimately, simply documents, organizations should not shy away from developing their own tools for working with them in their own way. The current tooling ecosystem, although very useful, requires further development and users and especially organizations are encouraged to contribute to it.
4. **Higher-order processes are needed for credibility of SBOM data and may also simply be required by customers or regulators.**
   One such process framework is SLSA, which, using the in-toto framework, integrates well with SBOM and is a good fit for the software supply chain security use case. More generally, the importance of human verification and accountability in the process should not be forgotten and, consequently, these humans must be supported with tooling in order to be able to fulfill these duties effectively.

Be it for security reasons, regulatory requirements, or to ensure license compliance, SBOMs will be playing a central part in securing the "software supply chain" in the near future. It is up not only to regulators and industry players but also to open-source contributors to ensure that the use of SBOMs in this future, rather than becoming a mere formality, meaningfully enhances supply chain security.

References

[1] NTIA, "Framing software component transparency: Establishing a common software bill of material (SBOM)," vol. 12, 2019, [Online]. Available: https://ntia.gov/files/ntia/publications/framingsbom_20191112.pdf

[2] Red Hat, "What is software supply chain security?." [Online]. Available: https://www.redhat.com/en/topics/security/what-is-software-supply-chain-security

[3] Synopsys, "What is software supply chain security?." [Online]. Available: https://www.synopsys.com/glossary/what-is-software-supply-chain-security.html

[4] M. Kaczorowski, "Secure at every step: What is software supply chain security and why does it matter?." [Online]. Available: https://github.blog/2020-09-02-secure-your-software-supply-chain-and-protect-against-supply-chain-threats-github-blog/

[5] iliana etaoin, "There is no 'software supply chain'." Accessed: Jul. 11, 2022. [Online]. Available: https://iliana.fyi/blog/software-supply-chain/

[6] Synopsys, "Open Source Security and Risk Analysis (8th edition)," 2023. [Online]. Available: https://www.synopsys.com/software-integrity/resources/analyst-reports/open-source-security-risk-analysis.html

[7] "The MIT License." [Online]. Available: https://opensource.org/license/mit/

[8] "CVE-2014-0160.." [Online]. Available: http://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2014-0160

[9] "Heartbleed Report (2017-01)." [Online]. Available: https://web.archive.org/web/20170123161742/https:/www.shodan.io/report/DCPO7BkV

[10] "Heartbleed Report." [Online]. Available: https://web.archive.org/web/20190711082042/https://www.shodan.io/report/0Wew7Zq7

[11] "CVE-2021-44228.." Accessed: Jul. 05, 2023. [Online]. Available: https://cve.mitre.org/cgi-bin/cvename.cgi?name=cve-2021-44228

[12] R. Hiesgen, M. Nawrocki, T. Schmidt, and M. Wählisch, "The Race to the Vulnerable: Measuring the Log4j Shell Incident." 2022.

[13] NCSC-NL, "Log4shell vulnerabilities." [Online]. Available: https://github.com/NCSC-NL/log4shell

[14] The White House, "Executive Order on Improving the Nation's Cybersecurity." [Online]. Available: https://www.whitehouse.gov/briefing-room/presidential-actions/2021/05/12/executive-order-on-improving-the-nations-cybersecurity/

[15] M. Willett, "Lessons of the SolarWinds hack," *Survival*, vol. 63, no. 2, pp. 7–26, 2021, [Online]. Available: https://www.tandfonline.com/doi/full/10.1080/00396338.2021.1906001

[16] J. Psaki and A. Neuberger, "Press Briefing by Press Secretary Jen Psaki and Deputy National Security Advisor for Cyber and Emerging Technology Anne Neuberger." [Online]. Available: https://www.whitehouse.gov/briefing-room/press-briefings/2021/02/17/press-briefing-by-press-secretary-jen-psaki-and-deputy-national-security-advisor-for-cyber-and-emerging-technology-anne-neuberger-february-17-2021/

[17] Kari Paul and agencies, "SolarWinds hack was work of 'at least 1,000 engineers', tech executives tell Senate." [Online]. Available: https://www.theguardian.com/technology/2021/feb/23/solarwinds-hack-senate-hearing-microsoft

[18] M. Fourné, D. Wermke, W. Enck, S. Fahl, and Y. Acar, "It's like flossing your teeth: On the Importance and Challenges of Reproducible Builds for Software Supply

Chain Security," 2023, [Online]. Available: https://teamusec.de/pdf/conf-oakland-fourne23.pdf

[19] K. Thompson, "Reflections on Trusting Trust," *Communications of the ACM*, vol. 27, no. 8, pp. 761–763, 1984.

[20] E. R. Royce, "Cyber Supply Chain Management and Transparency Act of 2014." Library of Congress, Dec. 04, 2014.

[21] European Commissoin, "Cyber Resilience Act."

[22] Bundesamt für Sicherheit in der Informationstechnik, "Technische Richtlinie TR-03183: Cyber-Resilienz-Anforderungen an Hersteller und Produkte." Jul. 12, 2023.

[23] Bundesamt für Sicherheit in der Informationstechnik, "SBOM-Anforderungen: TR-03183-2 stärkt Sicherheit in der Software-Lieferkette." [Online]. Available: https://www.bsi.bund.de/DE/Service-Navi/Presse/Alle-Meldungen-News/Meldungen/TR-03183-2-SBOM-Anforderungen.html

[24] NTIA, "The Minimum Elements For a Software Bill of Materials (SBOM)." [Online]. Available: https://www.ntia.doc.gov/files/ntia/publications/sbom_minimum_elements_report.pdf

[25] Linux Foundation, "The Software Package Data Exchange® (SPDX®) Specification Version 2.3." [Online]. Available: https://spdx.github.io/spdx-spec/v2.3/

[26] SPDX Project, "What is SPDX?." Accessed: Sep. 04, 2023. [Online]. Available: https://spdx.dev/resources/learn/

[27] CycloneDX Project, "History." Accessed: Jun. 08, 2023. [Online]. Available: https://cyclonedx.org/about/history/

[28] CycloneDX Project, "CycloneDX Specification Version 1.5." [Online]. Available: https://github.com/CycloneDX/specification/releases/tag/1.5

[29] CycloneDX Project, "CycloneDX Specification repository README." Accessed: Sep. 04, 2023. [Online]. Available: https://github.com/CycloneDX/specification/blob/master/README.md

[30] CycloneDX Core Working Group, "AuthoritativeGuide to SBOM." OWASP, Jun. 25, 2023.

[31] S. Springett, "CycloneDX vs. SPDX." [Online]. Available: https://www.youtube.com/watch?v=IQledp8WccU

[32] "Information technology — Software asset management — Part 2: Software identification tag," Oct. 2015.

[33] H. Birkholz, J. Fitzgerald-McKay, C. Schmidt, and D. Waltermire, "Concise Software Identification Tags." [Online]. Available: https://www.rfc-editor.org/info/rfc9393

[34] JetBrains, "The State of Developer Ecosystem 2021." [Online]. Available: https://www.jetbrains.com/lp/devecosystem-2021/cpp/

[35] D. Stenberg, "NVD makes up vulnerability severity levels." [Online]. Available: https://daniel.haxx.se/blog/2023/03/06/nvd-makes-up-vulnerability-severity-levels/

[36] L. G. A. Rodriguez, J. S. Trazzi, V. Fossaluza, R. Campiolo, and D. M. Batista, "Analysis of vulnerability disclosure delays from the national vulnerability database," 2018. Accessed: Sep. 10, 2023. [Online]. Available: https://sol.sbc.org.br/index.php/wscdc/article/download/2394/2358

[37] PostgreSQL Global Development Group, "CVE-2020-21469 is not a security vulnerability." Accessed: Sep. 13, 2023. [Online]. Available: https://www.postgresql.org/about/news/cve-2020-21469-is-not-a-security-vulnerability-2701/

[38] GitHub, "About the GitHub Advisory database." Accessed: Sep. 05, 2023. [Online]. Available: https://docs.github.com/en/code-security/security-advisories/working-with-global-security-advisories-from-the-github-advisory-database/about-the-github-advisory-database

[39] J. Bressers and K. Seifried, "Curl and the calamity of CVE." Open Source Security Podcast.

[40] MITRE Corporation, "Frequently Asked Questions." Accessed: Sep. 13, 2023. [Online]. Available: https://www.cve.org/ResourcesSupport/FAQs

[41] L. Williams, "Trusting Trust: Humans in the Software Supply Chain Loop," *IEEE Security & Privacy*, 2022.

[42] SLSA, "SLSA Specification Version 1.0." [Online]. Available: https://slsa.dev/spec/v1.0/

[43] B. Lum, I. Hepworth, and M. Kydyraliev, "SBOM + SLSA: Accelerating SBOM success with the help of SLSA." Accessed: Jul. 22, 2023. [Online]. Available: https://slsa.dev/blog/2022/05/slsa-sbom

[44] A. Sirish, "in-toto and SLSA." [Online]. Available: https://slsa.dev/blog/2023/05/in-toto-and-slsa

[45] SLSA, "Software attestations." Accessed: Jul. 11, 2023. [Online]. Available: https://slsa.dev/attestation-model

[46] A. Sirish, "In-Toto: Attestations and More for Software Supply Chain Security." [Online]. Available: https://youtu.be/ezV_oWBPqKw

[47] C. Kennedy and N. Schwartz, "Securing the Software Supply Chain with SBOM and Attestation." [Online]. Available: https://youtu.be/wX6aTZfpJv0

[48] NTIA, "Vulnerability Exploitability eXchange (VEX) — Use Cases."

[49] P. H. Salus, *A Quarter-Century of Unix*. Addison-Wesley, 1994.