Date: June 2021
Ref: NTIA–2021–0001

Please find our detailed feedback on your open questions:

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

Unfortunately, they are not sufficient. From our experience, every SBOM must include the respective license information for each software component in order to be accepted by automotive manufacturers. Law enforcement rules (detailed below) require clear identification of the license terms for each portion of software deployed inside an electronic device.

In Europe exists a process called homologation where a vehicle needs to be approved by governmental authorities. During homologation the software inside a vehicle is detailed and frozen. Each update needs to pass another homologation and for each homologation the software providers need to demonstrate that they hold the permissions/rights to use each piece of software, therefore the need to detail the respective license information.

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

Yes, please consider including Package URL (PURL) and a one-line description for the software component. Albeit not strictly necessary for a minimal-viable BOM it does help to remove confusion when the SBOM is reviewed by experts.

A recent example we experienced was "bootstrap". There exists a popular "bootstrap" component developed initially by Twitter for building webpages, and at the same existed another component with the same name called "bootstrap" that permits to boot a device inside the vehicle. Both cases had unique identifiers, however, the Supplier-Name field was the same since it was an external supplier modifying them. On such situation, both components were confused as being the same.

A PURL (and/or a one-line description) would help to remove confusion.

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.

One of the main needs for software identity is provenance. To know as much as possible the origin of the software, the people involved and permissions to deploy such software for example on embedded devices. Software identities today are weak regarding provenance. For example, in some parts of the world a person is given a personal name and a set of family names. That is what defines the identity of a person and permits to understand from which family it derives (and from that information understand the geography, the reputation behind the family members, and so forth).

In OSS we face a similar situation. If a well-known component such as GCC is supplied by Ubuntu and modified by them, it will generate an identifier competing for attention with the original GCC at the same level. There needs to be a recommendation for hierarchical identification. A modified GCC is effectively a child from the original GCC, such as you would find on a family member and from this information, we are able to map not just the origin of the software component, as well as map the related security vulnerabilities disclosed for one of the child or parent components (i.e. the canonical root of GCC).

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. For example, what about the supporting software to runs the SaaS solution? i.e. web server, firewall, etc.

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.

On such cases our audits are based on "best-effort possible". Alongside the SBOM we typically include a report including the investigation steps performed for identifying the possible third-party assets inside the binaries.

When it is not feasible on most cases to generate detailed software identification. For example, discovering the specific version of a library embedded inside a binary. For such cases it would help for the SBOM to permit flexibility and specify some fields as "NOT AVAILABLE" or “UNKNOWN”.

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.

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?

Some customers only want to see detailed SBOM for what goes inside a "target" (embedded device), whereas others demand to have complete information on all the components that have been used for building the software product.

On our side we solve this complex situation we adopted a data field called "tags" where multiple specific tags permit to output the level of detail required for the customer, and to identify if a component is used on production (delivered to customers) or used only during development.

The advantage of tags was the adaptability to new contexts not foreseen before, example of tags we use nowadays:
+ "target" (inside the embedded device in production mode)
+ "target-debug" (inside the device, only meant for debugging purposes)
+ "development"

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 14028.13 How can SBOM data be integrated with this additional data in a modular fashion?

Often we need to justify our choice/reason for adopting a specific software component. The main rationale is to identify OSS components under risk of no longer being developed by the respective maintainers in the medium-long term future (10 to 20 years).

One of the most critical factors from our perspective is the year of release. For example, a product using a software component/library released 10 years ago should be flagged as a security risk for the product owner and respective customers. Even if the data field would be included as optional, it would already help manufacturers to pressure for adoption of more recent of versions of the components to be adopted. Under the current system of just listing component versions, a reviewer needs an automated tool for uncovering the

component age and from our experience this does not happen often since the data is provided in different formats.

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.

From our experience the worst situations are formats that can only be produced and consumed by tools. In cases such as the SPDX RDF format where today exists a huge difficulty in producing a valid document, whereas the equivalent SPDX Tag/Value has become fairly popular exactly due to the human-readability characteristic.

The main problem is not even the access to SBOMs, it is the level of detail inside them. As example from the automotive industry in Europe, we are required to include the following data fields for each component:
+ identifiers (CVE, Maven, multiple ids supported)
+ name (human readable identifier for the component)
+ versions (versions found inside the software product)
+ description (one-line description of the component purpose)
+ license concluded (license applicable for component portions that go inside the product)
+ license references (licenses found inside the component package, and respective license choices)
+ license files (full text for the license terms)
+ special comments (rarely used)
+ copyright holders (who currently owns the component, such as the case of Java with Sun Microsystems references on the copyright headers, but with its copyright held today by Oracle)
+ copyright statements (data of all the copyright texts mentioned inside the package)
+ URL homepage
+ URL download (directly download link for the package(s))
+ tags (product, development, debug, etc)
+ paths (where the component-related files are located on the product)
+ dependencies

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.

We would say from experience that very few customers are able to verify components with a relatively complete graph. Albeit this is indeed a major problem, it has been even more difficult to accurately list everything that is used inside a software product, let alone map all the dependency graphs between these components.

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.

From our experience dealing with large manufacturers on the automotive industry we are required to report the vulnerabilities that are present on the components that are used.

Therefore, even at risk of being incomplete or inaccurate as time passes, it is important in our opinion to include the vulnerability information within the component information. Like software development, our work both on security and license compliance should be considered as a continuous process that is ever improving.

j. Risk Management:
Not all vulnerabilities in software 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.

We do this exact practice mostly for two reasons:
+ vulnerability applies only to a portion of the package that is not used
+ vulnerability cannot be exploited within the product (after review by your security experts)

This information is provided back to the customers.

In the future we would like to pinpoint the exact files inside a component that are impacted by a concrete vulnerability. This method would save time/effort for other parties to evaluate what are the security implications for their systems.

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?

Our experience has been to include the legal/security information as close as possible to the code itself when delivering it to customers. From that perspective we recommend the adoption of a folder placed on the root of the software product called "inventory" and inside this folder place the documentation artifacts inside intuitive names such as "legal"

(for license texts), "reports" (for the SBOMs and other testing reports) and other folders as required by other domains of software engineering.

Using this approach we would establish a human-readable process that would simultaneously be machine-readable.