Survey of Existing SBOM Formats and Standards

Credit: Photo by Patrick Tomasso on Unsplash

NTIA Multistakeholder Process on Software Component Transparency
Standards and Formats Working Group

Background & Problem Statement

Modern software systems involve increasingly complex and dynamic supply chains. Unfortunately, the composition and functionality of these systems lacks transparency; this contributes substantially to cybersecurity risks, alongside the cost of development, procurement, and maintenance. This has broad implications in our interconnected world; risk and cost affect collective goods, like public safety and national security, in addition to the products and services upon which businesses rely.

The NTIA Software Transparency Working Group on Standards and Formats was formed at the outset of the NTIA initiative in 2018 to assess available current formats for software bills of materials as well as forward-looking use-cases identified by other working groups or communities of practice.[1] The working group investigated existing standards, formats, and initiatives as they apply to identifying the external components and shared libraries (proprietary or open source) used in the construction of software products. The group analyzed efforts already underway by other groups related to communicating this information in a machine-readable manner. While proprietary formats that may meet these purposes exist, the group has not considered these proprietary formats.

The original survey was published in late 2019, after drafts were reviewed by the broader NTIA community. As the SBOM community grew, and implementation increased, the original working group expanded its focus to highlight the benefits of the SBOM tooling ecosystem, and the value of coordinating and harmonizing across the technical SBOM world. As part of this effort, the working group observed that, by 2021, parts of the document were out of date or less relevant. This 2021 revision offers several improvements on the original draft, and should be considered the stakeholder consensus until further updated. A key takeaway—that the baseline SBOM data can be conveyed in any of the formats described below, and the ecosystem can and should support interoperability between these data formats—holds for this revised document.

Goals of This Document

We propose that increased supply chain transparency can reduce cybersecurity risks and overall costs by:

- Enhancing the identification of vulnerable systems and the root cause of incidents
- Reducing unplanned and unproductive work
- Supporting more informed market differentiation and component selection
- Reducing duplication of effort by standardizing formats across multiple sectors
- Identifying suspicious or counterfeit software components

[1] This working group operated in parallel and coordination with three other efforts in the NTIA multistakeholder process on Software Component Transparency. More information about the process is available here: https://www.ntia.doc.gov/SoftwareTransparency. The related documents are available at https://ntia.gov/SBOM.

Collecting and communicating this information in such a manner can lower the cost, increase the reliability of, and increase our ability to trust our digital infrastructure.

The initial goals of this working group were to:

- Investigate the options available today.
- Document workable and actionable machine-readable formats.
- Acknowledge that no single solution/format will be required (i.e., we will not “proclaim a winner”).
- Determine how the solutions can work in harmony, since different formats were designed to address the requirements of different constituencies (e.g., developers, CFOs managing software entitlements), and mapping between well-documented formats is technically feasible.
- Support international feedback and buy-in to solutions as supply-chain security and software integrity is not just a U.S. problem, and participation in this process is global.

Three Key Formats

The working group identified three formats in widespread use: (1) Software Package Data Exchange (SPDX®), an open source machine-readable format with origins in Linux Foundation and recently approved as ISO/IEC standard; (2) CycloneDX (CDX), an open source machine-readable format with origins in the OWASP community; and (3) Software Identification (SWID), an ISO/IEC industry standard used by various commercial software publishers. Descriptions and use cases for each format, as well as a mapping between them, are detailed below.

It is important to note that although these three formats contain overlapping information, they have historically been used at different points in the software lifecycle, and are consumed by different types of users.

SPDX, a product of the open source software development community, is geared for ease-of-ingestion within a developer workflow and within corporations to support compliance and software transparency for open source and proprietary code. The open source nature of the format, as well as the availability of open source tooling to generate it, supports broad adoption by a large and distributed population of commercial international organizations, as well as developers who may not be associated with vendors. The accessibility of SPDX means that the sole developer of an experimental library can generate an SBOM with minimal effort at no cost. The ability to link artifacts to global reference systems via Common Platform Enumeration (CPE), Package URL (purl), Software Heritage persistent ID (SWHID), as well as other package build coordinates, enables flexibility to handle security use cases. The cost saving and ready availability of open source tools is attractive to commercial organizations as well. SPDX is useful in the “long tail” of upstream open source software componentry, for source and binary software artifacts.

CycloneDX is a lightweight open source standard with origins in the OWASP community. It supports a wide range of development ecosystems, a comprehensive set of use cases, and focuses on automation, ease of adoption, and progressive enhancement of SBOMs throughout build pipelines. The specification is in widespread use among organizations with security use cases and is equally capable of describing both open source and proprietary software. A large and growing collection of community and officially supported open source tools are available, and the project's website includes many examples for achieving various use cases. CycloneDX natively supports multiple standards for component identity including coordinates, Package URL, CPE, and SWID for both binary and source software artifacts.

SWID tags were designed with software inventory and entitlement management in mind. SWID tags support the inventory of commercial and open source software that is installed on a device through locating the SWID tag associated with the software. A developer can use freely available guidance on the creation of SWID tags to configure their build pipeline to produce SWID tags automatically during the software build and packaging process. With an orientation around deployed software, SWID tags follow the binary artifact and are updated as the compiled codebase changes. This lends itself to integration with automated scanning, and a host of risk-management use cases and tooling.

This document and this working group acknowledge that all three formats can be used to generate, exchange, and use SBOM data. While certain use cases may lend themselves to particular formats, this working group does not endorse any format specifically, and believes that each user should select that which meets their needs. This document offers an explicit guide to translate between the three for the “minimum viable” SBOM models to enable a more interoperable ecosystem.

Lifecycle of an SBOM

How to Produce SBOMs

Information that goes into SBOMs can be best obtained from the tools and processes used in each stage of the software lifecycle (See Figure 1, below). One may leverage existing tools and processes to generate SBOMs. Such tools and processes include intellectual property review, procurement review and license management workflow tools, software supply chain risk management, code scanners, pre-processors, code generators, source code management systems, binary code analysis tools, version control systems, compilers, build tools, continuous integration systems, packagers, compliance test suites, package distribution repositories, and app stores.

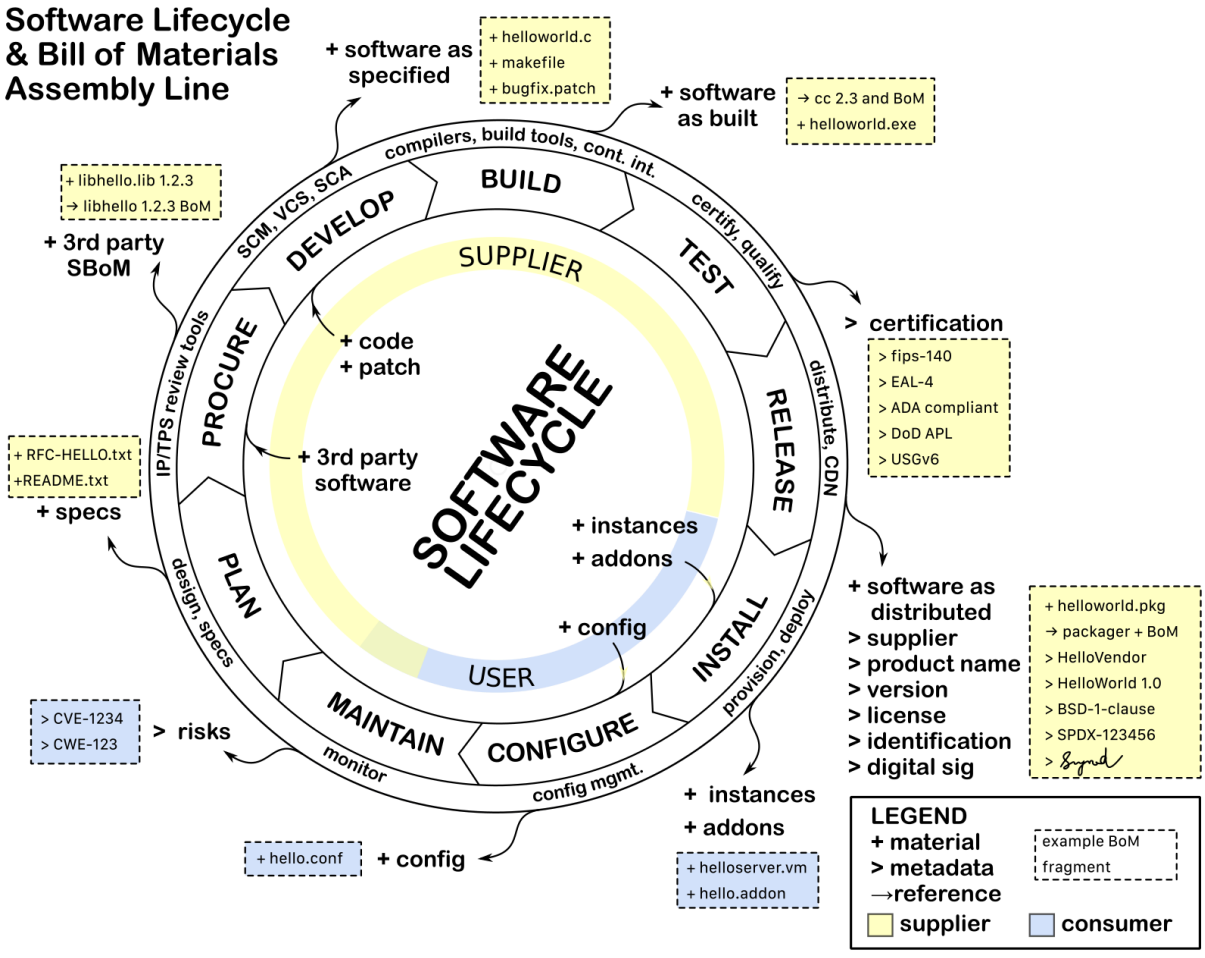

Figure 1: The Software lifecycle with multiple stages where underlying code might change, and thus the SBOM would be updated to reflect the changes.

Currently, not all off-the-shelf or open source software lifecycle tools have the capability to generate SBOMs. Analysis of software may happen after initial generation. Suppliers should consider enhancing or retrofitting existing tools and processes to generate and maintain SBOMs.

SBOMs may be considered incomplete by some users for specific use cases. If the SBOMs are incomplete, the consumer should reach out to the supplier to clarify in order to make informed use of SBOMs based on the available data.

The NTIA multistakeholder community as part of the Framing Working group has defined a baseline of SBOM data elements,[2] which is closely related to the subsequent Executive Order-mandated Minimum Elements document.[3] The discussion in this document is based on these references.

How to Deliver SBOMs

At the moment, there is no single set way to transmit SBOM downstream to the next user.[4] In open source products, the SBOM can be stored as metadata, with pointers to components. For compiled software, SBOMs can be bundled together with the software product itself as a compendium and stored with the installed software. The SBOM could also be made available in portals controlled by the supplier, stored in a pre-agreed location or with some other third party.

Stakeholders mentioned the potential value in accessing data from older SBOMs, so that users can understand the underlying components of software at a specific point in time. For example, a customer may want to know if a cloud-based service was potentially vulnerable at a certain point in the past as part of a forensic breach investigation. This document does not offer guidance on how to preserve past SBOMs. To learn more about how suppliers can deliver SBOMs, please read the Software Suppliers Playbook: SBOM Production and Provision.[5]

How to Update SBOMs

An SBOM should reflect the current state of a piece of software. If software or the software's underlying components are updated, then the list of underlying components should also be updated accordingly to ensure that SBOM data itself is up-to-date. Except for the information that is derived from the software artifact itself, other information in an SBOM can be declarative, or asserted by the author of the SBOM data. For example, the download location of the component names can be part of the SBOM.

Similarly, if the information known about the software changes, or an error was made in the original SBOM, a supplier may update the SBOM without updating the underlying code. Declared information may have to be corrected, changed, or added over time. Such changes

[2] https://www.ntia.gov/files/ntia/publication__s/ntia_sbom_framing_2nd_edition_20211021.pdf
[3] https://www.ntia.doc.gov/report/2021/minimum-elements-software-bill-materials-sbom
[4] https://www.ntia.gov/files/ntia/publications/ntia_sbom_sharing_exchanging_sboms-10feb2021.pdf
[5] https://www.ntia.gov/files/ntia/publications/software_suppliers_sbom_production_and_provision_-_final.pdf

can be appended to ledger-based SBOMs. Corrections or additions may be made to an SBOM, resulting in a new revised SBOM that should be auditable.

How to Consume SBOMs

For the most effective use of SBOM information, the data must be machine readable. Consumption must incorporate machine-to-machine automated processes. Each of the use cases discussed in the introduction (and further fleshed out in the Software Consumers Playbook: SBOM Acquisition, Management, and Use[6] document) can achieve maximum effectiveness only by integrating into automated processes. It is also important that the format can be translated into a human readable version.

Consumers may use SBOMs as input to their tools that support:

- asset management
- license and entitlement management
- intellectual property management
- regulatory and compliance management
- provisioning
- configuration management
- vulnerability management
- incident response
- software supply chain risk assessment and attestation

Usage of SBOMs for risk management may require additional risk data that may not be included with SBOMs, e.g., vendor-supplied data acquired during a procurement process. To learn more about how consumers can work with SBOMs, please read the Software Consumer Playbook.

[6] https://www.ntia.gov/files/ntia/publications/software_consumers_sbom_acquisition_management_and_use_-_final.pdf

Overview of Key Formats

SPDX

The Software Package Data Exchange (SPDX®) specification is an ISO/IEC standard[7] for communicating software bill of materials information. It facilitates the description of components, licenses, copyrights, and security information associated with software components in multiple file formats. The project has created and continues to evolve a set of data exchange standards that enables companies and organizations to share human-readable and machine-processable software metadata to facilitate software supply chain processes.

Software development teams across the globe use the same open source components, but in 2010, there was little infrastructure available to facilitate collaboration or analysis, or to share the results of analysis activities. As a result, many groups were performing the same work, leading to duplicated efforts and redundant information. To save time, and improve data accuracy, the SPDX project was formed to create a common metadata exchange format so that information about software packages and related content could be collected and shared, and tooling could be developed[8] to help automate these tasks.

SPDX information can be associated with a particular software product, component or set of components, an individual file, or even a snippet of code. The SPDX project focuses on creating and extending a “language” to describe the data that can be exchanged as part of a software bill of materials, and being able to express that language in multiple file formats (RDF/XML, XLSX, tag-value, JSON, YAML & XML) so that information about software packages and related content may be easily collected and shared with the goal of saving time and improving accuracy.

The specification is a living document. As new use-cases are examined, it evolves. Care is taken to provide backwards compatibility. Development progresses through collaboration between technical, business, legal, and now security professionals from a range of organizations to create a standard that addresses the needs of various participants in the software supply chain.

Companies and organizations are widely using and reusing open source and other software components. Accurate identification of the software from the executable to the source files that make up that executable is key to understanding if there may be a security vulnerability in it.

Description

The SPDX specification[9] describes the necessary sections and fields to produce a valid SPDX document. It is important to note that not all of these sections are required. The creation

[7] ISO/IEC 5962:2021 (https://www.iso.org/standard/81870.html)
[8] https://spdx.dev/resources/tools/
[9] https://spdx.github.io/spdx-spec/

information section is the only one that is mandatory. Then it is a matter of using the sections (and subset of the fields in each section) that describe the software and metadata information that the SPDX document creator is planning to share.

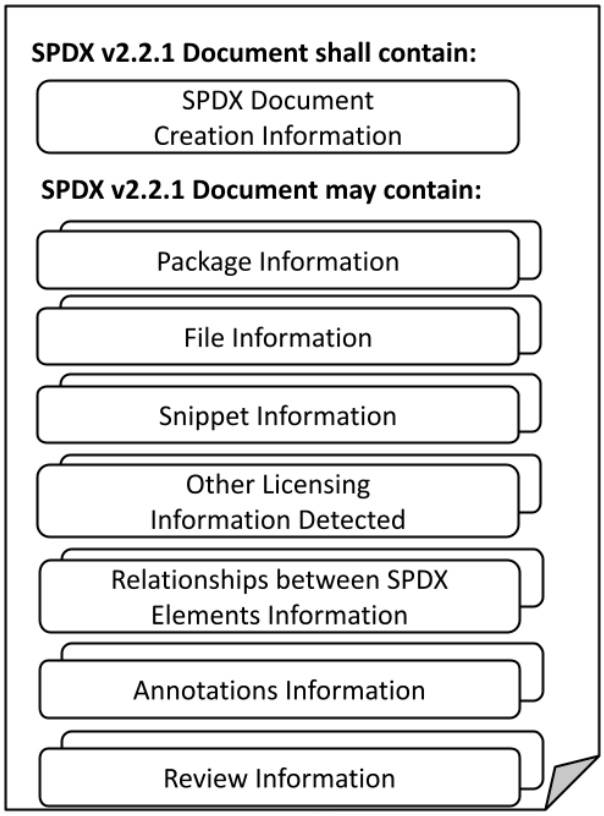

FIGURE 2: Overview of an SPDX document[10]

Each SPDX document can be composed from the following:

- **Creation Information:** One instance is required for each SPDX document produced. It provides the necessary information for forward and backward compatibility for processing tools (version numbers, license for data, authors, etc.).
- **Package Information:** A package in an SPDX document can be used to describe a product, container, component, packaged upstream project sources, contents of a tarball, etc. It is a way of grouping together items that share some common context. It is not necessary to have a package wrapping a set of files.
- **File Information:** A file's important metadata, including its name, checksum, licenses and copyright, is summarized here.
- **Snippet Information:** Snippets can optionally be used when a file is known to have some content that has been included from another original source. They are useful for denoting when part of a file may have been originally copied from a different file.
- **Other Licensing Information:** The SPDX License List[11] does not represent all licenses that can be found in packages, files, and snippets, so this section provides a way to

[10] This diagram has been reproduced from https://spdx.github.io/spdx-spec/composition-of-an-SPDX-document/
[11] https://spdx.org/licenses

summarize other license information that may be present in software being described, such as custom or proprietary licenses.
- **Relationships:** There are many different ways SPDX documents, packages, files and snippets can be related to each other; there are 43 relationships supported, with the ability to extend if needed, to enable effective system description.
- **Annotations:** Annotations are usually created when someone reviews the SPDX document and wants to pass on information from their review. However, if the SPDX document author wants to store extra information that doesn't fit into the other categories, an annotation can be used.

Each document is capable of being represented by a full data model implementation and identifier syntax. This permits exchange between data output formats (RDF/XML, tag-value, XLSX), and formal validation of the correctness of the SPDX document.[12] In the SPDX specification's version 2.2 release, the additional output formats of JSON, YAML, and XML have been added, as well as support for the "known unknowns" as identified in the original SBOM framing document. Further information on the data model underlying SPDX can be found in Appendix III of the SPDX Specification[13] and on the SPDX web site.[14]

Use Cases

- Describes system components and nuanced relationships between components
- High fidelity tracking of intellectual property (licensing, copyright) of software components
- Software supply chain risk assessments and component verification
- Listing contents of a software distribution
- Tracking executables back to individual source files and source snippets
- Container contents inventory
- Associating Common Platform Enumerations (CPEs), Software Heritage persistent IDs (SWHIDs), and Package URLs (purls) with specific packages to facilitate additional security analysis
- Identifying provenance of lines of code embedded in files

Key Features

- Documented artifacts can be checked using the provided hash values
- Rich facilities for intellectual property and licensing information
- Flexible model able to scale from snippets and files up to packages, containers, and even operating system distributions
- Ability to add mappings to other package reference systems and security systems
- Ability to logically partition and link documents associated with complex systems

[12] https://tools.spdx.org/app/validate/
[13] https://spdx.github.io/spdx-spec/appendix-III-RDF-data-model-implementation-and-identifier-syntax/
[14] https://spdx.dev/

SPDX and SBOM

SPDX can capture SBOM data because they are able to represent all of the components found in software development and deployment. SPDX is being used to represent distro .iso images, containers, software packages, binary files, source files, patches, and even snippets of code embedded in other files. SPDX provides a rich set of relationships to link the software elements together within documents, as well as between SBOM documents. An SPDX SBOM document is able to link out via External References to National Vulnerability Database and other packaging systems metadata.

Future Directions

- When desired, information to indicate vulnerabilities known at the time of the SPDX creation and also when/where/how these known vulnerabilities have been remediated in an update or patch or have been determined as being not exploitable in the software being delivered. This work is aligning with the emerging Vulnerability-Exploitability eXchange working group directions.[15]
- Enhancing the representation of pedigree and provenance information to support chain of custody discussions.
- Richer set of relationships and integrity checks between interactions.
- Identification of use cases currently not able to be represented by SPDX[16] and adding elements into the upcoming specification release to support these use cases.

CycloneDX

CycloneDX is a lightweight SBOM specification designed for use in software security contexts and supply chain component analysis. It can communicate inventory of software components, external services, and their relationships to one another. CycloneDX is an open source OWASP standard.

The project was created in 2017 with a goal of creating a fully automatable, security-focused SBOM standard. The core working group has produced immutable, backward compatible releases every year through a risk-based standards process. CycloneDX incorporates existing specifications including Package URL, CPE, SWID, and SPDX license IDs and expressions. CycloneDX SBOMs can be represented as XML, JSON, and Protocol Buffers (protobuf).

The dynamic nature of open source components whose source code is readily available, modifiable, and redistributable can be captured in CycloneDX. The specification can represent component pedigree including ancestors, descendants, and variants that describe component lineage from any viewpoint and the commits, patches, and diffs that make it unique.

[15] https://www.ntia.gov/files/ntia/publications/vex_one-page_summary.pdf
[16] https://github.com/spdx/spdx-spec/issues

The CycloneDX project maintains a community supported list of all known open source and proprietary tools[17] that support or interoperate with the standard.

Description

The CycloneDX specification[18] describes a prescriptive object model that creates consistency across implementations. The specification can be validated using XML Schema and JSON Schema, or by using the CycloneDX CLI[19]. Registered media types are provided for XML and JSON allowing for automated delivery and consumption of supported formats.

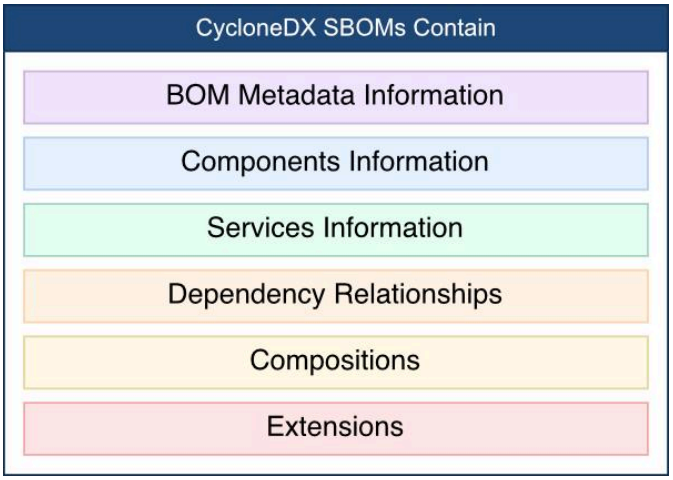

FIGURE 3: CycloneDX Overview[20]

Each CycloneDX SBOM may contain:

- **BOM Metadata**: Contains information including the tools used to produce the SBOM, the supplier, manufacturer, and the assembled software, component, firmware, or device that the SBOM describes.
- **Components**: Describes the complete inventory of first-party and third-party software components. This includes the type of component, its identity, license, copyright, hashes, and complete pedigree and provenance, including any alterations made to components. Components may be nested to form component assemblies, which may have their own supplier information. A digital signature may optionally be applied to components or component assemblies.
- **Services**: Describes external APIs that the software may call, which may include endpoint URI's, authentication requirements, and trust boundary traversals. The flow of data between software and services can also be described including the data

[17] https://cyclonedx.org/tool-center/
[18] https://cyclonedx.org/docs/latest
[19] https://github.com/CycloneDX/cyclonedx-cli
[20] This diagram has been reproduced from https://cyclonedx.org/specification/overview/

classifications, and the flow direction of each type. A digital signature may optionally be applied to services.
- **Dependencies**: Describes direct and transitive relationships. Components that depend on other components can be represented, as well as components that depend on services. Services that depend on other services can also be represented.
- **Compositions**: The completeness of inventory and relationships can be described using compositions. The aggregate of each composition can be described as complete, incomplete, incomplete first-party only, incomplete third-party only, or unknown. This allows SBOM authors to describe how complete the SBOM is, or if there are components in the SBOM where completeness is unknown.
- **Extensions**: Multiple extension points exist throughout the CycloneDX object model, allowing fast prototyping of new capabilities and support for specialized and future use cases.

Use Cases

- Description of inventory of software components and services
- Vulnerability analysis, remediation, and other security use cases
- Software supply chain risk assessments and attestations
- License compliance
- Pedigree and provenance including complete traceability of all modifications
- Integrity verification of signed components, component assemblies, and the SBOM
- Portable file-based formats useful for build-time creation and distribution, as well as a highly efficient machine-to-machine binary format

Key Features

- Prescriptive object model is simple to learn and adopt
- Open source standard - permissive, commercial friendly license
- Highly automatible with deep integration across many development ecosystems
- Supports multiple standards for component identity including Package URL, CPE, and SWID (tag id or complete SWID documents)
- Risk-based approach to standards development that expedites delivery of core functionality to the community
- Specification is extensible allowing rapid prototyping of new capabilities to meet organizational or industry-specific requirements

CycloneDX and SBOM

CycloneDX is a full-stack SBOM standard that can represent many different types of software applications, components, services, firmware, and devices. The specification is currently used across a wide-variety of industries to describe software packages, libraries, frameworks, applications, and container images. The project supports most major development ecosystems and provides implementations for software factories, such as GitHub actions, that further accelerate the ability for organizations to fully automate SBOM creation.

Future Directions

- Incorporate feedback from the community and the CycloneDX Industry Working Group to help guide the future direction of the standard
- Multiple extensions are currently being developed including auditing, formulation, and enhancements to the existing vulnerability extension[21]
- Continuous improvement of the core specification with yearly releases. Progress can be tracked using GitHub milestones[22].

SWID

Description

Software Identification (SWID) Tags were designed to provide a transparent way for organizations to track the software installed on their managed devices. It was defined by ISO in 2012 and updated as ISO/IEC 19770-2:2015[23] in 2015.[24] SWID Tag files contain descriptive information about a specific release of a software product.

The SWID standard defines a lifecycle: a SWID Tag is added to an endpoint as part of the software product's installation process, and deleted by the product's uninstall process. In this lifecycle, the presence of a given SWID Tag corresponds directly to the presence of the software product that the Tag describes. Multiple standards bodies, including the Trusted Computing Group (TCG) and the Internet Engineering Task Force (IETF), use SWID Tags in their standards.

To capture the lifecycle of a software component, the SWID specification defines four types of SWID tags: primary, patch, corpus, and supplemental. (See Figure 3)

1. **Primary Tag:** A SWID Tag that identifies and describes a software product is installed on a computing device.
2. **Patch Tag:** A SWID Tag that identifies and describes an installed patch that has made incremental changes to a software product installed on a computing device.
3. **Corpus Tag:** A SWID Tag that identifies and describes an installable software product in its pre-installation state. A corpus tag can be used to represent metadata about an installation package or installer for a software product, a software update, or a patch.
4. **Supplemental Tag:** A SWID Tag that allows additional information to be associated with any referenced SWID tag. This helps to ensure that SWID Primary and Patch Tags provided by a software provider are not modified by software management tools, while allowing these tools to provide their own software metadata.

---

[21] https://cyclonedx.org/ext/vulnerability/

[22] https://github.com/CycloneDX/specification/milestones

[23] https://webstore.ansi.org/Standards/ISO/ISOIEC197702015

[24] While ISO documents sit behind a paywall, anyone can freely use ISO-standardized specifications. See NIST Internal Report (NISTIR) 8060: *Guidelines for the Creation of Interoperable Software Identification (SWID) Tags* for a detailed explanation and guide of SWID tags.

Corpus, primary, and patch tags have similar functions in that they describe the existence and/or presence of different types of software (e.g., software installers, software installations, software patches), and, potentially, different states of software products. In contrast, supplemental tags furnish additional information not contained in corpus, primary, or patch tags.

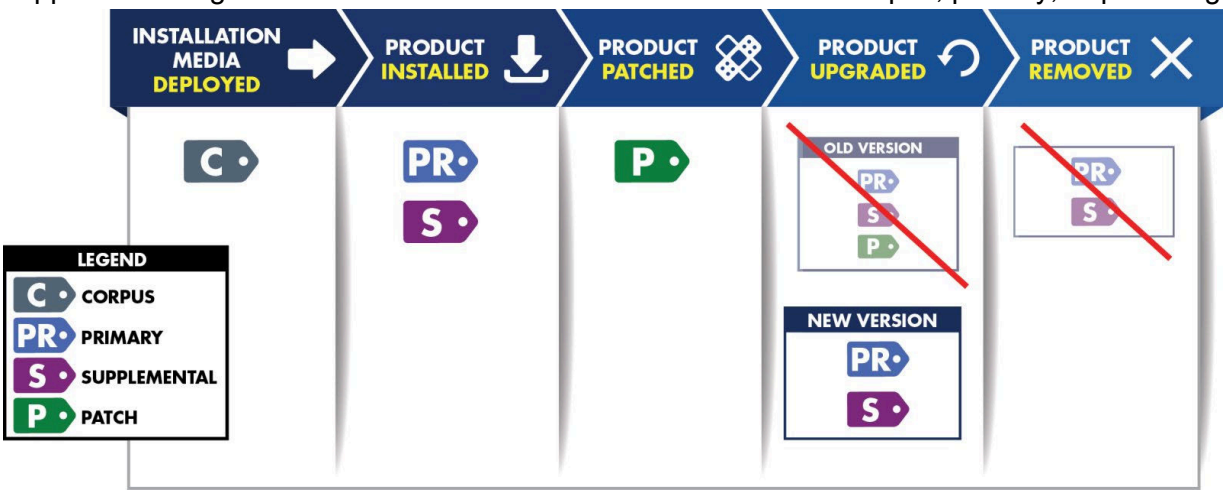

FIGURE 4: The Lifecycle of software on an endpoint documented by SWID tags. Souce: NISTIR 8060

The figure above illustrates the steps in the software lifecycle and the relationships among those lifecycle events supported by the four types of SWID tags. Supplemental tags can be associated with any other tag to provide additional metadata that might be of use. Taken as a body, SWID tags can support a wide range of functions, including software discovery, configuration management, and vulnerability management.

The following is an example of a primary SWID tag for a piece of compiled software by the ACME Corporation called Roadrunner Detector.[25] The tag defines the product name, version, and other details, as well as a hash for the binary.

```
<SoftwareIdentity
  xmlns="http://standards.iso.org/iso/19770/-2/2015/schema.xsd"
  xmlns:sha256="http://www.w3.org/2001/04/xmlenc#sha256"
  name="ACME Roadrunner Detector 2013 Coyote Edition SP1"
  tagId="com.acme.rrd2013-ce-sp1-v4-1-5-0"
  version="4.1.5">
  <Entity name="The ACME Corporation" regid="acme.com"
    role="tagCreator softwareCreator"/>
  <Link rel="license" href="www.gnu.org/licenses/gpl.txt"/>
  <Meta product="Roadrunner Detector" colloquialVersion="2013"
    edition="coyote" revision="sp1"/>
  <Payload>
    <File name="rrdetector.exe" size="532712"
```

[25] https://csrc.nist.gov/publications/detail/nist ir/8060/final

```
      SHA256:hash="a314fc2dc663ae7a6b6bc6787594057396e6b3f569cd50fd5ddb
      4d1bbafd2b6a"/>
  </Payload>
</SoftwareIdentity>
```

Use Cases

- SBOM for software components
- Continuous monitoring of installed software inventory
- Identifying vulnerable software on endpoints
- Ensuring that installed software is properly patched
- Preventing installation of unauthorized or corrupted software
- Preventing the execution of corrupted software
- Managing software entitlements

Key Features

- Provides stable software identifiers created at build time
- Standardizes software information that can be exchanged between software providers and consumers as part of the software installation process
- Enables the correlation of information related to software including related patches or updates, configuration settings, security policies, and vulnerability and threat advisories

SWID Tags and SBOM

SWID tags can be used as an SBOM, since they provide identifying information for a software component, a listing of files and cryptographic hashes for the constituent artifacts that make up a software component, and provenance information about the SBOM (tag) creator and software component creator. Tags can explicitly link to other tags, enabling a representation of a dependency tree.

The operational model for generating SWID tags allows the tags to be generated as part of the build and packaging process; this allows a SWID tag-based SBOM to be produced automatically when the related software component is packaged.

Future Directions

While SWID tags are an XML format, a more lightweight representation called CoSWID, a Concise Binary Object Representation (CBOR)-based binary representation of SWID tag information, is currently being standardized in the IETF to support the constrained IoT use case. More information on CoSWID can be found below.

Translation and Harmonization Guidance

Experts in SPDX, SWID, and CycloneDX engaged in a mapping exercise between the data fields in the three formats. Not all the fields are evenly mapped to each other, as the formats were originally designed for different purposes. However, the Working Group found the potential

for decent interoperability, as enough of the fields correspond with each other. This is particularly true for those fields related to the basic component data discussed in the Framing Group's Document[26].

Attribute	SPDX	CycloneDX	SWID
Author Name	`Creator`	`metadata/authors/author`	`<Entity> @role (tagCreator), @name`
Timestamp	`Created`	`metadata/timestamp`	`<Meta>`
Supplier Name	`PackageSupplier`	`Supplier publisher`	`<Entity> @role (softwareCreator/publisher), @name`
Component Name	`PackageName`	`name`	`<softwareIdentity> @name`
Version String	`PackageVersion`	`version`	`<softwareIdentity> @version`
Component Hash	`PackageChecksum Or VerificationCode`	`Hash "alg"`	`<Payload>/../<File> @[hash-algorithm]:hash`
Unique Identifier	`DocumentNamespace combined with SPDXID`	`bom/serialNumber component/bom-ref`	`<softwareIdentity> @tagID`
Relationship	`Relationship: DESCRIBES; CONTAINS`	`(Inherent in nested assembly/subassembly and/or dependency graphs)`	`<Link> @rel, @href`

Table 1: A mapping between SPDX, CycloneDX, and SWID to capture the core fields discussed in the "baseline component information" SBOM.

Below, we lay out those basic data fields that are similar to what is described as the "Baseline Component Information" as defined in the Framing Group's document. To illustrate this, we offer an example that captures many of the features of basic SBOM in SPDX, CycloneDX, and SWID.

Example Scenario

The example, illustrated in Figure 5, focuses on an imaginary piece of software called "Application" by an organization named Acme and demonstrates how a Software Bill of Materials (SBOM) can look in a fairly lightweight fashion. Acme's Application includes exactly two third-party components: Bob's Browser and Bingo Buffer. Bob's Browser, in turn, depends on third-party components. We know that the Browser includes Carol's CompressionEng, but we don't know if the Browser includes other components as well. Carol's CompressionEng, in

[26] https://www.ntia.gov/files/ntia/publications/ntia_sbom_framing_2nd_edition_20211021.pdf

turn, is written from scratch, and we know that it contains no third party components. Unfortunately, we don't know if Acme's Application's other dependency, Bingo Buffer, contains any third-party dependencies.

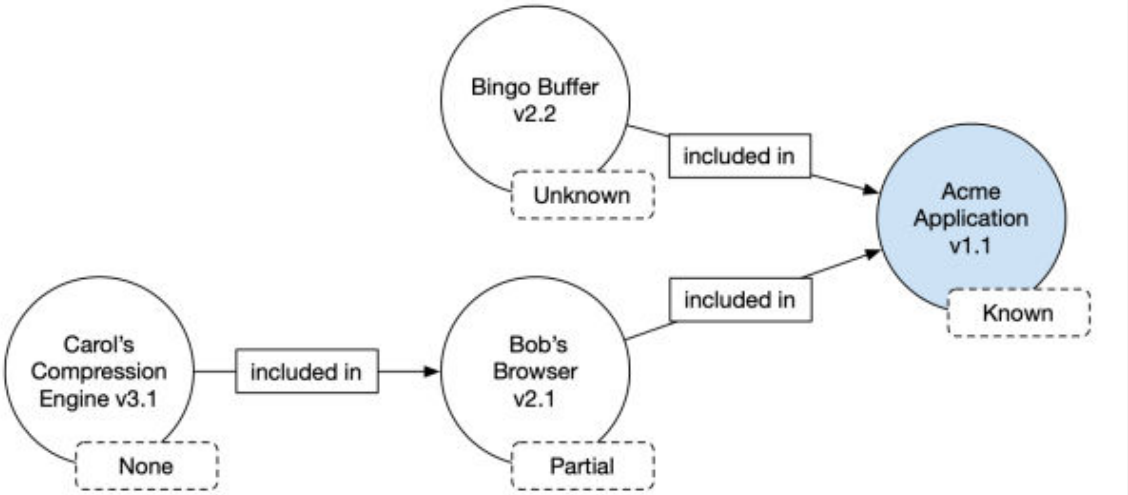

Figure 5: A toy example of software to illustrate how an SBOM can look.[27]

SPDX (tag-value[28])

```
SPDXVersion: SPDX-2.2
DataLicense: CC0-1.0
DocumentNamespace: http://www.spdx.org/spdxdocs/8f141b09-1138-4fc5-aefb-fc10d9ac1eed
DocumentName: SBOM toy example
SPDXID: SPDXRef-DOCUMENT
Creator: Organization: NTIA Standards and Formats Workgroup
Created: 2021-08-31T11:29:46Z
Relationship: SPDXRef-DOCUMENT DESCRIBES SPDXRef-Application-v1.1

PackageName: Application
SPDXID: SPDXRef-Application-v1.1
PackageVersion: 1.1
PackageSupplier: Organization: Acme
PackageDownloadLocation: NOASSERTION
FilesAnalyzed: false
PackageChecksum: SHA1: 75068c26abbed3ad3980685bae21d7202d288317
PackageLicenseConcluded: NOASSERTION
PackageLicenseDeclared: NOASSERTION
PackageCopyrightText: NOASSERTION
Relationship: SPDXRef-Application-v1.1 CONTAINS SPDXRef-Browser-v2.1
Relationship: SPDXRef-Application-v1.1 CONTAINS SPDXRef-Buffer-v2.2

PackageName: Browser
```

[27] More details on this example can be found in https://www.ntia.gov/files/ntia/publications/ntia_sbom_framing_2nd_edition_20211021.pdf

[28] This file format was selected for human readability in this document, submitting the example to https://tools.spdx.org/app/convert/ can convert it to XML, JSON, YAML, RDF/XML as well.

```
SPDXID: SPDXRef-Browser-v2.1
PackageVersion: 2.1
PackageSupplier: Person: Bob
PackageDownloadLocation: NOASSERTION
FilesAnalyzed: false
PackageChecksum: SHA1: 94568c26abbed3ad3980685deaf1d7202d268314
PackageLicenseConcluded: Apache-2.0
PackageLicenseDeclared: NOASSERTION
PackageCopyrightText: Copyright 2019 Bob
Relationship: SPDXRef-Browser-v2.1 CONTAINS SPDXRef-CompressionEng-v3.1
Relationship: SPDXRef-Browser-v2.1 CONTAINS NOASSERTION

PackageName: Buffer
SPDXID: SPDXRef-Buffer-v2.2
PackageVersion: 2.2
PackageSupplier: Organization: Bingo
PackageDownloadLocation: NOASSERTION
FilesAnalyzed: false
PackageChecksum: SHA1: 84568c26aabad3ad3980685beef1d7202d26831d
PackageLicenseConcluded: NOASSERTION
PackageLicenseDeclared: GPL-3.0-or-later
PackageCopyrightText: Copyright 2018 Bingo Inc.
Relationship: SPDXRef-Buffer-v2.2 CONTAINS NOASSERTION

PackageName: CompressionEng
SPDXID: SPDXRef-CompressionEng-v3.1
PackageVersion: 3.1
PackageSupplier: Person: Carol
PackageDownloadLocation: NOASSERTION
FilesAnalyzed: false
PackageChecksum: SHA1: 63568c26aebad3ad398bb85ce1f1d7202d27731a
PackageLicenseConcluded: NOASSERTION
PackageLicenseDeclared: NOASSERTION
PackageCopyrightText: NOASSERTION
Relationship: SPDXRef-CompressionEng-v3.1 CONTAINS NONE
```

CycloneDX (XML)

```
<?xml version="1.0" encoding="utf-8"?>
<bom xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xmlns:xsd="http://www.w3.org/2001/XMLSchema"
  serialNumber="urn:uuid:3e671687-395b-41f5-a30f-a58921a69b71" version="1"
  xmlns="http://cyclonedx.org/schema/bom/1.3">
  <metadata>
    <authors>
      <author>
        <name>Acme</name>
      </author>
    </authors>
    <component type="application">
      <name>Application</name>
      <version>1.1</version>
```

```
      <hashes>
        <hash alg="SHA-1">75068c26abbed3ad3980685bae21d7202d288317</hash>
      </hashes>
      <cpe>cpe:2.3:a:acme:application:1.1:*:*:*:*:*:*:*</cpe>
      <externalReferences />
      <components />
    </component>
    <manufacture>
      <name>Acme</name>
    </manufacture>
    <supplier>
      <name>Acme</name>
    </supplier>
  </metadata>
  <components>
    <component type="library" bom-ref="pkg:maven/org.bob/browser@2.1">
      <publisher>Bob</publisher>
      <group>org.bob</group>
      <name>browser</name>
      <version>2.1</version>
      <hashes>
        <hash alg="SHA-1">94568c26abbed3ad3980685deaf1d7202d268314</hash>
      </hashes>
      <cpe>cpe:2.3:a:bob:browser:2.1:*:*:*:*:*:*:*</cpe>
      <purl>pkg:maven/org.bob/browser@2.1</purl>
    </component>
    <component type="library" bom-ref="pkg:maven/org.carol/CompressionEng@3.1">
      <publisher>Carol</publisher>
      <group>org.carol</group>
      <name>CompressionEng</name>
      <version>3.1</version>
      <hashes>
        <hash alg="SHA-1">63568c26aebad3ad398bb85ce1f1d7202d27731a</hash>
      </hashes>
      <cpe>cpe:2.3:a:carol:compression_eng:3.1:*:*:*:*:*:*:*</cpe>
      <purl>pkg:maven/org.carol/CompressionEng@3.1</purl>
    </component>
    <component type="library" bom-ref="pkg:maven/org.bingo/buffer@2.2">
      <publisher>Bingo</publisher>
      <group>org.bingo</group>
      <name>Buffer</name>
      <version>2.2</version>
      <hashes>
        <hash alg="SHA-1">84568c26aabad3ad3980685beef1d7202d26831d</hash>
      </hashes>
      <cpe>cpe:2.3:a:bingo:buffer:2.2:*:*:*:*:*:*:*</cpe>
      <purl>pkg:maven/org.bingo/buffer@2.2</purl>
    </component>
  </components>
  <dependencies>
    <dependency ref="pkg:maven/org.bob/browser@2.1">
      <dependency ref="pkg:maven/org.carol/CompressionEng@3.1" />
    </dependency>
    <dependency ref="pkg:maven/org.bingo/buffer@2.2" />
  </dependencies>
```

```
  <compositions>
    <composition>
      <aggregate>complete</aggregate>
      <assemblies>
        <assembly ref="pkg:maven/org.carol/CompressionEng@3.1"/>
      </assemblies>
      <dependencies>
        <dependency ref="pkg:maven/org.carol/CompressionEng@3.1"/>
      </dependencies>
    </composition>
    <composition>
      <aggregate>unknown</aggregate>
      <assemblies>
        <assembly ref="pkg:maven/org.bingo/buffer@2.2"/>
        <assembly ref="pkg:maven/org.bob/browser@2.1"/>
      </assemblies>
    </composition>
  </compositions>
</bom>
```

SWID (XML)

```
<SoftwareIdentity
  xmlns="http://standards.iso.org/iso/19770/-2/2015/schema.xsd"
  xmlns:sha512="http://www.w3.org/2001/04/xmlenc#sha512"
  name="application"
  tagId="acme/application@1.1"
  version="1.1">
  <Entity name="acme" role="tagCreator softwareCreator" />
  <Link href="swid:bob/browser@2.1" rel="component" />
  <Link href="swid:bingo/buffer@2.2" rel="component" />
  <Payload >
    <File name="acme-application-1.1.exe"
       sha512:hash="BC55DEF84538898754536AE47CC907387B8F61D9ACD7D3FB8B8A624199682C8FBE
       6D1631088AE6A322CDDC4252D3564655CB234D3818962B0B75C35504D55689" />
  </Payload>
</SoftwareIdentity>

<SoftwareIdentity
  xmlns="http://standards.iso.org/iso/19770/-2/2015/schema.xsd"
  xmlns:sha512="http://www.w3.org/2001/04/xmlenc#sha512"
  name="browser"
  tagId="bob/browser@2.1"
  version="2.1">
  <Entity name="bob" role="tagCreator softwareCreator" />
  <Link href="swid:carol/compressionEng@2.2" rel="component" />
  <Payload >
    <File name="bob-browser-2.1.exe"
       sha512:hash="FF4893471E763B94165CC277A9FB01D7ED66256FDDD6467D91E35AFF8F445C6312
       832FD97DE1FD517606019BDC5F46E9E4E4814601E1FCB1010E90C2EBE54820" />
  </Payload>
</SoftwareIdentity>
```

```
<SoftwareIdentity
  xmlns="http://standards.iso.org/iso/19770/-2/2015/schema.xsd"
  xmlns:sha512="http://www.w3.org/2001/04/xmlenc#sha512"
  name="buffer"
  tagId="bingo/buffer@2.2"
  version="2.2">
  <Entity name="bingo" role="tagCreator softwareCreator" />
  <Payload >
    <File name="bingo-buffer-2.2.lib"
       sha512:hash="AEE705CEAFDBA5EE54462443E41A447FDA69BEDCB57FC4C284D41AD67C7499A8F1
       0C3B7D504A118986A3DF29564B3BD64B783C3B18BFA0F2AA4C779477A9D0D8" />
  </Payload>
</SoftwareIdentity>

<SoftwareIdentity
  xmlns="http://standards.iso.org/iso/19770/-2/2015/schema.xsd"
  xmlns:sha512="http://www.w3.org/2001/04/xmlenc#sha512"
  name="compressionEng"
  tagId="carol/compressionEng@3.1"
  version="3.1">
  <Entity name="carol" role="tagCreator softwareCreator" />
  <Payload >
    <File name="carol-compressionEng-3.1.lib"
       sha512:hash="BEB0E94E089B34DADA04A53A38AE268672CA69ABB34C79E14B446D0DD5F55BE034
       FC9F9D7DDF0655CDCDAB878604625805648FADA6E897541F483B2E92AE424C" />
  </Payload>
</SoftwareIdentity>
```

Software Identity Formats

When the NTIA working group started the discussion of software bill of materials, the following formats were also suggested to be considered for identifying software. The group worked with creators of these projects to identify the key elements, and the summaries are captured below, along with links for further information.

Concise SWID Tag (CoSWID)

Description

The Concise SWID (CoSWID) tag specification[29] is an alternate format for representing a SWID tag using the Concise Binary Object Representation (CBOR). A SWID tag, expressed in XML, can be fairly large. The size of a SWID tag can be larger than acceptable for use in constrained devices use cases (e.g., IoT). While containing the same information as a SWID tag, CoSWID tags reduce the size of a SWID by a significant amount. This size reduction is supported by using integer labels in CBOR in place of human-readable strings for data elements and commonly used values.

Use Cases

As an alternate representation of a SWID tag, CoSWID shares the same use cases as a SWID tag. Due to the reduced size, a CoSWID tag better supports implementation of these use cases for IoT and other constrained devices and networks.

Key Features

A CoSWID shares the same features of a SWID tag. This format reduces the footprint of a SWID tag, while expressing the same information. The following is an example of a CoSWID tag, in hex-based binary:

```
bf0f65656e2d5553207820636f6d2e61636d652e727264323031332d63652d7370312d76342d3
12d352d300cc2410101783041434d4520526f616472756e6e6572204465746563746f72203230
313320436f796f7465204564697469f6e205350310d65342e312e350e2002bf181f745468652
041434d4520436f72706f726174696f6e1820686163 6d652e636f6d18219f0120ffff04bf1826
7823687474703a2f2f7777772e676e752e6f72672f6c6963656e7365732f67706c2e747874182
8676c6963656e7365ff05bf182d6432303133182f66636f796f7465183473526f616472756e6e
6572204465746563746f72183663737031ff06bf11bf18186e727264657465636 3746f722e65786
5141a200820e8079f015820a314fc2dc663ae7a6b6bc6787594057396e6b3f569cd50fd5ddb4d
1bbafd2b6affffffff
```

The CoSWID in CBOR is 317 bytes in size, while the SWID tag in XML is 795 bytes in size. This represents a 60.1% reduction in size, while expressing the same information in both tags.

[29] The CoSWID format is described by https://datatracker.ietf.org/doc/draft-ietf-sacm-coswid/. This IETF draft is nearing publication as an IETF RFC.

Common Platform Enumeration (CPE)

Description

A related data format is the Common Platform Enumeration (CPE).[30] This format is "a standardized method of describing and identifying classes of applications, operating systems, and hardware devices present among an enterprise's computing assets." The CPE is used in various situations such as the National Vulnerability Database (NVD). CPE enables identification of specific applications, with or without version numbers, but does not by itself identify subcomponents.

Use Cases

CPEs are useful because they facilitate component to vulnerability matching via the NVD. CPE information contained within SBOMs can be queried against NVD search tools to discover potential related vulnerabilities.[31]

Key Features

A CPE identifier is composed of 11 elements along with a prefix of the cpe version (the latest cpe specification being cpe:2.3):

Element	**Definition**
part	Asset class: "a" for applications, "o" for operating systems, "h" for hardware devices
vendor	Organization that created or made the product
product	Commonly recognized name of product
version	Version of product
update	Update, service pack, or point release of the product
edition	Edition-related terms of the product (**Depreciated in cpe:2.3 but kept for backwards compatibility**)
sw_edition	Product tailoring for market or class of end users
target_sw	Computing environment product operates
target_hw	Instruction set architecture (x86) on which product operates
language	Language of product interface
other	General descriptive or identifying information that is vendor-product specific

[30] CPE is defined in https://csrc.nist.gov/projects/security-content-automation-protocol/specifications/cpe/
[31] NVD CPE search tool https://nvd.nist.gov/products/cpe/search

Elements put together to form a well-formed CPE identifier:

```
wfn:[part="o",vendor="microsoft",product="windows_vista",version="6\.0",
update="sp1",edition=NA,language=NA,sw_edition="home_premium",
target_sw=NA,target_hw="x64",other=NA]
```

Examples:

```
cpe:2.3:o:microsoft:windows_vista:6.0:sp1:-:-:home_premium:-:x64:-
cpe:2.3:o:linux:linux_kernel:2.6.35:rc3:*:*:*:*:*:*:*
cpe:2.3:a:google:chrome:5.0.375.82:*:*:*:*:*:*:*
cpe:2.3:o:cisco:ios:12.2(12)da4:*:*:*:*:*:*:*
```

Package-URL (purl)

Description

A package URL (purl) is an attempt to standardize existing approaches to reliably identify and locate software packages. A purl is a URL string used to identify and locate a software package in a mostly universal and uniform way across programming languages, package managers, packaging conventions, tools, APIs, and databases. Such a package URL is useful to reliably reference the same software package using a simple and expressive syntax and conventions based on familiar URLs.

A purl is a URL composed of seven components:

```
scheme:type/namespace/name@version?qualifiers#subpath
```

Components are separated by a specific character for unambiguous parsing.

Component	**Definition**	**Usage**
scheme	This is the URL scheme with the constant value of "pkg". One of the primary reasons for this single scheme is to facilitate the future official registration of the "pkg" scheme for package URLs.	Required
type	The package "type" or package "protocol" such as maven, npm, nuget, gem, pypi, etc.	Required
namespace	Some name prefix such as a Maven groupid, a Docker image owner, a GitHub user or organization.	Optional and type-specific
name	The name of the package.	Required
version	The version of the package.	Optional

qualifiers	Extra qualifying data for a package such as an OS, architecture, a distro, etc.	Optional and type-specific
subpath	Extra subpath within a package, relative to the package root.	Optional

Components are designed such that they can form a hierarchy, from the most significant component on the left, to the least significant components on the right.

A purl must NOT contain a URL Authority; i.e., there is no support for username, password, host and port components. A namespace segment may sometimes look like a host, but its interpretation is specific to a type.

Examples[32]

```
pkg:deb/debian/curl@7.50.3-1?arch=i386&distro=jessie
pkg:docker/gcr.io/customer/dockerimage@sha256:244fd47e07d1004f0aed9c
pkg:gem/ruby-advisory-db-check@0.12.4
pkg:golang/google.golang.org/genproto#googleapis/api/annotations
pkg:maven/org.apache.xmlgraphics/batik-anim@1.9.1?packaging=sources
```

Linkage
SPDX 2.2 recognizes PURLs as valid External References <type>.[33]
SWID: A PURL can be provided in a SWID tag using the “link” element.
CycloneDX: All versions of CycloneDX natively support PURL as part of a component’s identity.

SoftWare Heritage persistent IDentifiers (SWHID)

Description

Software Heritage[34] is a nonprofit initiative actively supported by a large number of organizations[35]—software, systems and tool vendors, IT users, academic and governmental institutions. It is building a universal archive of software source code, as a common infrastructure catering to a variety of use cases from industry to science and culture.

Use Cases

One of the use cases specifically listed on the SoftWare Heritage mission statement is source code tracking for industry[36]: “*Because industry cannot afford to lose track of any part of its source code, we track software origin, history, and evolution. Software Heritage will provide* ***unique software identifiers, intrinsically bound to software components****, ensuring persistent traceability across future development and organizational changes.*” These intrinsic

[32] Information was extracted from: https://github.com/package-url/purl-spec
[33] https://spdx.github.io/spdx-spec/3-package-information/#321-external-reference
[34] https://cacm.acm.org/magazines/2018/10/231366-building-the-universal-archive-of-source-code/fulltext
[35] https://www.softwareheritage.org/support/testimonials/
[36] https://www.softwareheritage.org/mission/industry/

identifiers are based on cryptographic signatures, have a precise formal definition[37], and are already available for the more than 10 billions of artefacts stored in the Software Heritage archive[38]. They are an essential building block for ensuring **integrity** of a source code base, and are currently being used by some major industry players to implement a part of their SBOM workflow, related to source code distribution obligations, as well as from the Wikidata community[39].

Linkage
SPDX 2.2: formally recognize SWH IDs as valid External References <type>[40].

[37] https://docs.softwareheritage.org/devel/swh-model/persistent-identifiers.html#persistent-identifiers
[38] https://archive.softwareheritage.org
[39] https://www.wikidata.org/wiki/Property:P6138
[40] https://spdx.github.io/spdx-spec/3-package-information/#321-external-reference

Changes

Significant changes between this and the previous edition include:

- Moved CycloneDX from being a related format to a main one with its own section
- Added section on Identity formats and updated with results of latest survey
- Updated overview of the lifecycle of an SBOM
- Added Timestamp to Baseline Attributes (aligning with Framing Document)
- Expanded harmonization and translation guidance between the three formats
- Removed the “Related Formats” section
- Updated diagrams, figures, and tables to align with Framing Document
- Made various editorial improvements and clarifications

About the Authors of This Document

This document was drafted by an open working group convened by the National Telecommunications and Information Administration in a multistakeholder process, including the following individuals and organizations: Tom Alrich (Tom Alrich LLC), Chris Clark (Synopsys), Patrick Dwyer (OWASP), Robin Gandhi (University of Nebraska at Omaha), Adam Weinrich(Checkmarx), Christopher Gates (Velentium), JC Herz (Ion Channel), Derek Kruszewski (aDolus), Art Manion (CERT Coordination Center), Bob Martin (MITRE), Chandan Nandakumaraiah (Juniper Networks), Brendan O’Connor (GitHub), Gary O’Neall (SPDX), Duncan Sparrell (sFractal), Steve Springett (OWASP), Kate Stewart (Linux Foundation), Tim Walsh (Mayo), David Waltermire (NIST), Steve Winslow (SPDX). Others participated, but do not wish to be named.