Background & Problem Statement

Modern software systems involve increasingly complex and dynamic supply chains. Lack of systemic transparency into the composition and functionality of these systems contributes substantially to cybersecurity risk as well as the costs of development, procurement, and maintenance. In our increasingly interconnected world, risk and cost impact not only individuals and organizations directly but also collective goods like public safety and national security.

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

thus increasing trust and trustworthiness while lowering costs of our digital infrastructure.

The NTIA Software Transparency Working Group on Standards and Formats was formed to assess available current formats for software bills of materials as well as forward-looking use-cases identified by other working groups and communities of practice.[1] The working group investigated existing standards, formats and initiatives as they apply to identifying the external components and shared libraries, commercial or open source, used in the construction of software products. The group analyzed efforts underway in the community and industry related to assuring this transparency is readily available in a machine-readable manner.

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. “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 buyin to solutions as this is not just a US problem, and participation in this process is global.

Two key formats

The working group identified two formats in widespread use: Software Package Data eXchange (SPDX), an open source machine-readable format stewarded by the Linux Foundation, and Software Identification (SWID), an international standard used by commercial software

[1] <Insert language about the broader NTIA process with links to other docs>

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 two formats contain overlapping information, they are typically used at different points in the software life cycle 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. 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. These 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.

SWID tags were designed with software inventory and entitlements 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 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 both 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 either specifically, and believes that each user should select that which meets their needs. This document offers an explicit guide to translate between the two for the “minimum viable” SBoM models to enable a more interoperable ecosystem.

Lifecycle of an SBOM

How to produce SBoM?

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, code scanners, pre-processors, code generators, source code management systems, version control systems, compilers, build tools, continuous integration systems, packagers, compliance test suites, package distribution repositories and app stores.

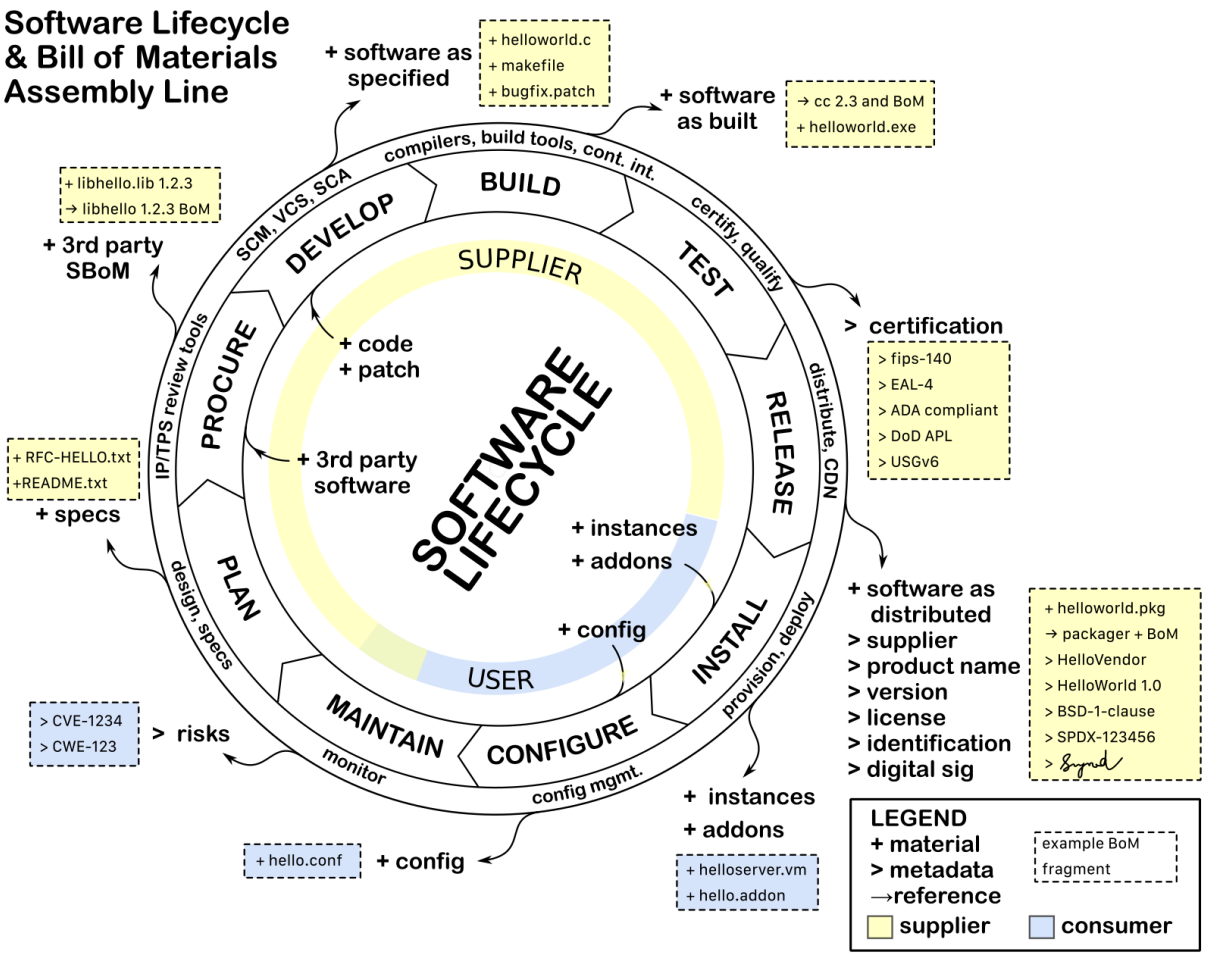

Figure 1: The Software lifecycle with multiple stages where underlying code might change, and thus the SBoM would be updated to reflect the changes.

Not all off-the-shelf or open source software lifecycle tools may have the capability to generate SBoMs as a byproduct when they add material that goes into a product today. Suppliers should consider enhancing or retrofitting existing tools and processes to generate and maintain SBoMs.

SBoM may be considered incomplete if some information about the materials added or removed through the stages of software lifecycle is missing or was never recorded. If the SBoMs are incomplete, suppliers should make that clear so that consumers can make informed use of SBoMs based on the available data.

How to deliver SBoM?

At the moment, there is no set single way to transmit SBoM data downstream to the next user. In open source products, the SBoM data 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 data could also be made available in portals controlled by the supplier or some other third party.

Stakeholders mentioned the potential value in accessing data older SBoMs, so that users can understand the underlying components of software at a specific point in time. .

How to update SBoM?

An SBoM should reflect the current state of a piece of software. If that software or 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 SBoM can be declarative.

Declared information may have to be corrected or changed over time. Such changes can be appended to ledger based SBoM(see Related Formats section). Errata may have to be supplied and carried forward with a specific SBoM.

How to consume SBoM?

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 Use Case document) can only achieve maximum effectiveness 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 asset management, license and entitlement management, regulatory and compliance management, provisioning, configuration management, vulnerability management and incident response tools. Usage of SBoMs for risk management may require additional risk data that may not be included with SBoMs.

Overview of Key Formats

SPDX

The Software Package Data Exchange (SPDX®) specification provides a standard language for communicating the components, licenses, copyrights, and security information associated with software components in multiple file formats.

Software development teams across the globe use the same open source components, but in 2010, there was little infrastructure exists to facilitate collaboration on the analysis or share the results of these analysis activities. As a result, many groups were performing the same work leading to duplicated efforts and redundant information. The SPDX project was formed to create a data exchange format so that information about software packages and related content may

be collected and shared in a common format with the goal of saving time and improving data accuracy.

An SPDX document can be associated with a particular software component or set of components, 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 be able to express that language in multiple file formats(RDFa, .xlsx, .spdx and soon .xml, .json, .yaml) 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 and legal 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 (collectively “Suppliers”) are widely using and reusing open source and other software components. Accurate identification of the software is key to understanding if there may be a security vulnerability in it.

Description

The SPDX specification 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 in every document. The only one that is mandatory is to have a “Document Creation Information” section for each document. Then it’s a matter of using the sections (and subset of the fields in each section) that describe the software and metadata information you’re planning to share.

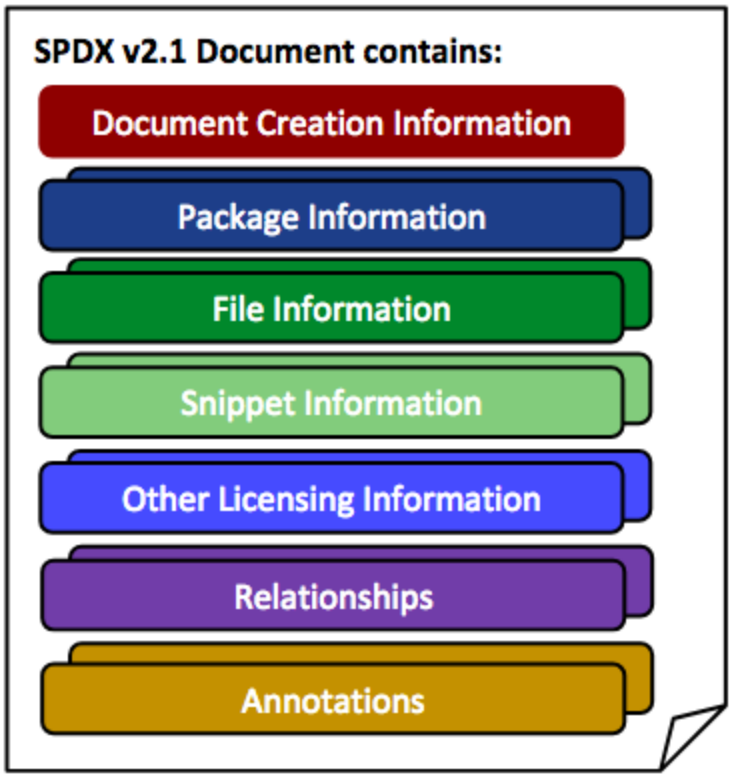

FIGURE 2 - Overview of an SPDX document. Souce:https://spdx.github.io/spdx-spec/

Each SPDX document can be composed from the following:

- **Document Creation Information:** One instance is required for each SPDX file 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 tar ball, etc. It's 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 meta information, 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 created under another license.
- **Other Licensing Information:** The SPDX license list does not represent all licenses that can be found in files, so this section provides a way to summarize other license that may be present in software being described.
- **Relationships:** Most of the different ways that SPDX documents, packages, files can be related to each other can be described with these relationships.
- **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, this mechanism 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 (RDFa, tag:value, spreadsheet), and formal validation of the correctness of the SPDX document. In the SPDX specification 2.2 release, the additional output formats of JSON, YAML and XML are planned to be supported. Further information on the data model can be found in Appendix III of the SPDX Specification and on the SPDX web site.

Use Cases

- Linux Distribution Package List [link]
  - See text/spreadsheet for details on individual components.
- SBOM for embedded software on IoT devices

Key Features

- documented artifacts can be checked (hash)
- Rich facilities IP and licensing information,
- flexible model able to scale from distros, containers, packages, files, snippets.
- Ability to add mappings to other package reference systems.
-

SPDX and SBoM

SPDX documents can easily capture SBoM data because…

Future Directions

- Information to indicate when/where/how known vulnerabilities have been remediated in an update or patch.
-

SWID Tag

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 in 2015[2]. SWID Tag files contain descriptive information about a specific release of a software product.

The SWID standard defines a lifecycle where 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. When this lifecycle is followed, 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 XXX)

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 which 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.

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.

[2] 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.

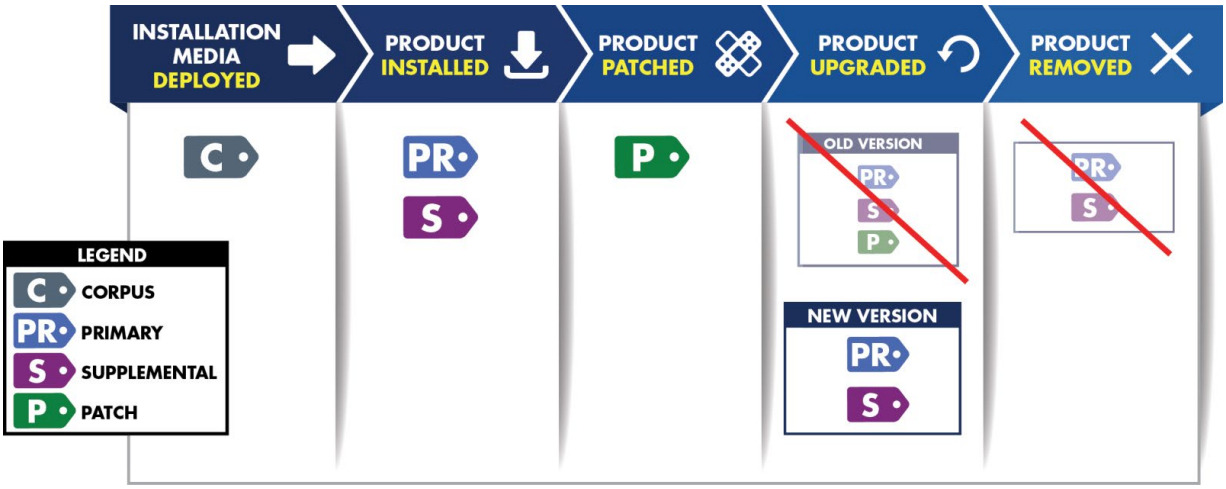

FIGURE 3 - The Lifecycle of software on an endpoint documented by SWID tags. Souce: NISTIR 8060

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 Roadrunnery Detector. 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"
      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"
                     SHA256:hash="a314fc2dc663ae7a6b6bc6787594057396e6b3f569cd50fd5d
                     db4d1bbafd2b6a"/>
         </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, provenance information about the SBOM (tag) creator and software component creator. Tags can explicitly link to other tags, enabling the representationof the dependency tree.

The operational model for generating SWID tags allow the tags to be generated as part of the build and packaging process, allowing 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 later below.

Translation and Harmonization Guidance

Experts in SPDX and SWID engaged in a mapping exercise between the data fields in the two formats. Not all the fields evenly mapped to each other--they were designed for different purposes, after all. However, the Working Group found the potential for decent interoperability. Enough of the fields correspond with each other, particularly for those around the basic component data discussed in the Framing Group's draft work.

Below, we lay out those basic data fields that are similar to what is described as the "Baseline Component Information." To illustrate this, we offer a toy example that captures many of the features of basic SBoM. Future drafts will include the example code in both SPDX and SWID.

Field	Represented in SPDX	Represented in SWID
Supplier	`(3.5) PackageSupplier:`	`<Entity> @role (softwareCreator/publisher), @name`
Component	`(3.1) PackageName:`	`<softwareIdentity> @name`
Unique Identifier	`(3.2) SPDXID`	`<softwareIdentity> @tagID`
Version	`(3.3) PackageVersion:`	`<softwareIdentity> @version`
Component Hash	`(3.10) PackageChecksum:`	`<Payload>/../<File> @[hash-algorithm]:hash`
Relationship	`(7.1) Relationship:`	`<Link>@rel, @href`
SBOM Author	`(2.8) Creator`	`<Entity> @role (tagCreator), @name`

TABLE 1: A mapping between SPDX and SWID to capture the core fields discussed in the "baseline component information" SBoM.

Example Scenario

The goal of a toy example is to demonstrate how a Software Bill of Materials (SBoM) can look in a fairly lightweight fashion. Our example, illustrated in Figure 4, is around a web application called Roadrunner Detector, developed by Coyote Software. This application has a number of software dependencies. The first dependency is "OpenTLS 10.1-1" produced by Acme Software Co., which provides TLS functionality for securing web requests, and cryptographic algorithm functions. It may or may not have further dependencies. The second dependency is for "Roadrunner Classifier Version 1.3.1", produced by Birds-R-Us. This, in turn, uses a simple component from GitHub called plumageMatcher that does not use any third party components.

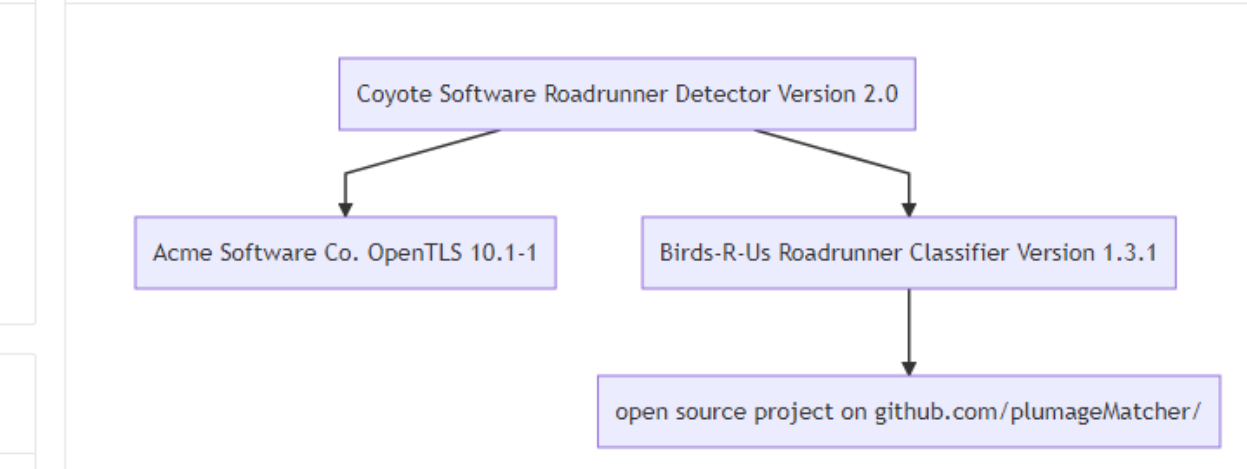

FIGURE 4: A toy example of software to illustrate how an SBoM can look.

<Forthcoming: documenting the above example in SWID and SPDX to show how dependencies are represented, and demonstrating that the formats are still human-readable as well as machine-readable.>

Related Format Efforts Surveyed

When the NTIA working group started the discussion of bill of materials, the following formats were also suggested to be considered for identifying software. The workgroup had presentations from the creators of these projects, and the summaries are captured below. Links where those who are interested, can find more information are provided.

Software Heritage Index

Description:

Use Cases:

SParts

Description
The Software Parts (SParts) project delivers a blockchain-based ledger that enables one to determine the chain of custody of all the software parts from which a product

(e.g., IoT device) is comprised of. The ledger provides both access to and accountability for software meta information of software parts exchanged among manufacturing supply chain participants. A software part is any software component that could be represented as one or more files. (e.g., source code, binary library, application, an operating system runtime, container, ...).

Use Cases:
This can be used to summarize pedigree of software through multiple hops in the supply chain. The envelope can contain SPDX, SWID and other SBOM information, as the participants

Package-URL (purl)

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 programing 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.

Description

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.

The definition for each component is:

- 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 for a hierarchy from the most significant 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

```
pkg:bitbucket/birkenfeld/pygments-main@244fd47e07d1014f0aed9c
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:github/package-url/purl-spec@244fd47e07d1004f0aed9c
pkg:golang/google.golang.org/genproto#googleapis/api/annotations
pkg:maven/org.apache.xmlgraphics/batik-anim@1.9.1?packaging=sources
```

Information above was extracted from: https://github.com/package-url/purl-spec

Linkage
SPDX: Next version of the specification (2.2) will formally recognize PURL's as valid External References <type>
SWID: ??

Package URL Comparison With CPE
The following is a comparison between Package URL and CPE using a single open source project as an example.

<table>
<tr><td>Reality</td><td><table>
<tr><td>Repository Type:</td><td>Maven</td></tr>
<tr><td>Group ID:</td><td>org.jboss.resteasy</td></tr>
<tr><td>Artifact ID:</td><td>resteasy-jaxrs</td></tr>
<tr><td>Version:</td><td>3.1.0-Final</td></tr>
</table></td></tr>
<tr><td>Package URL</td><td>pkg:maven/org.jboss.resteasy/resteasy-jaxrs@3.1.0-Final</td></tr>
<tr><td>CPE</td><td>cpe:2.3:a:redhat:resteasy:3.1.0:*:*:*:*:*:*:*</td></tr>
</table>

Due to the decentralized nature of Package URL, it is possible to accurately describe an open source project with identical values as the project itself identifies as. Of the three fields CPE requires (vendor, product, and version) the official CPE identifier isn't an exact match for any of

them. This makes automated vulnerability analysis, without human intervention, difficult and subject to error. Package URLs can be automatically generated from any modern dependency management system.

CoSWID Tag

The Concise SWID (CoSWID) tag specification[3] 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.

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.

Future Research for SboM-relevant Standards and Formats

- Globally unique and immutable component identifiers
- Archaeological digs, forensic, provenance
- Tool support, in creation, exchange, consumption of SBoM, either here or in another area? Framing will be mentioning need for tools for adoption
- How electronic delivery of an SboM is performed

[3] 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.