DRAFT - This is a Work in Progress.

DRAFT - This is a Work in Progress.

Background & Problem statement

The NTIA Software Transparency working group on standards and formats was formed to assess current formats for software bills of materials against current requirements as well as forward-looking requirements identified by other working groups and communities of practice. 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 goals were to:

- Investigate the options available today
- Document workable and actionable machine-readable formats
- There was not a requirement to find a single solution/format (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.
- Consider International aspects of proposed solutions as this is not a US problem. There are parallel and analogous discussions in the EU to address these issues, and the software supply chain is global.

The working group identified two formats in widespread use: SPDX, an open source machine-readable format stewarded by the Linux Foundation, and SWID, an international standard used by 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 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 developer centric and geared for ease-of-ingestion within a developer workflow. The open source nature of the standard, as well as the open source tooling to generate it, supports broad adoption by a large and distributed population of developers who may not be associated with vendors or commercial software licensing. The accessibility of SPDX means that the sole developer of an experimental library can generate an SBOM with minimal effort at no cost. For this reason, SPDX is useful in the "long tail" of upstream open source software componentry.

DRAFT - This is a Work in Progress.

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. SWID tags have a compelling use case for the financial and legal management of commercial software that is procured by enterprises and must be accounted for on a financial and legal basis. Downstream consumers of SWID include security and patch management, financial, and legal risk managers who might not want or need to get into the granularity and volume of data provided by a developer-centric SPDX SBOM. Workflows matter, and if the business requirement is to make sure that licensed software is properly patched and paid-for, SWID has distinct advantages. SWID has a number of open source and proprietary implementations, as well as open source tooling to generate SWID tags.

DRAFT - This is a Work in Progress.

Overview of Key Formats

SWID Tag

Description

Software Identification (SWID) Tags, defined by the ISO/IEC 19770-2:2015 standard, provide a transparent way for organizations to track the software installed on their managed devices. 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) utilize SWID Tags in their standards.

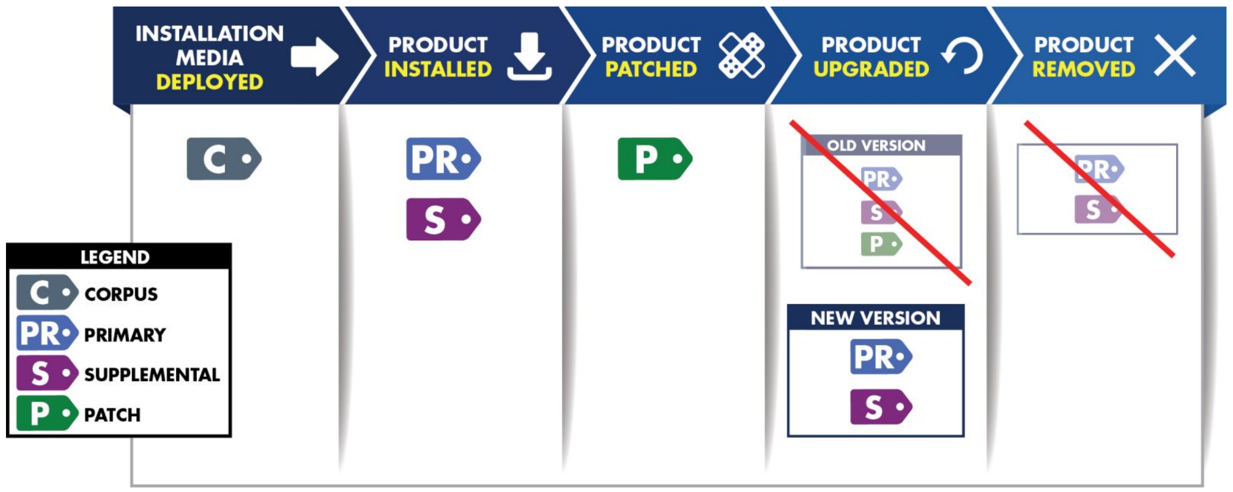
The SWID specification defines four types of SWID tags: primary, patch, corpus, and supplemental.

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. [1]**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 a 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. All four tag types come into play at various points in the software lifecycle, and support software management processes that depend on the ability to accurately determine where each software product is in its lifecycle.

[1] The following is an excerpt from NIST Internal Report (NISTIR) 8060: *Guidelines for the Creation of Interoperable Software Identification (SWID) Tags*.

DRAFT - This is a Work in Progress.



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. While not fully illustrated in the figure, supplemental tags can be associated with any corpus, primary, or patch tag to provide additional metadata about an installer, installed software, or installed patch respectively.

These software lifecycle events represent typical uses of tags within the software deployment lifecycle. While not depicted in the above, software discovery, configuration management, and vulnerability management processes generate other lifecycle events which may create and/or use corpus, primary, patch, and supplemental tags.

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. 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. CoSWID, a Concise Binary Object Representation (CBOR)-based binary representation of SWID tag information, is currently being standardized in the IETF. When published as an IETF RFC in the near future, CoSWID will provide an alternate SWID and SBOM format that is more suited to constrained IoT use cases. More information on CoSWID can be found later below.

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

DRAFT - This is a Work in Progress.

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.

The following is an example of a SWID tag.

```
<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>
```

Areas to Improve

TBD

DRAFT - This is a Work in Progress.

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.

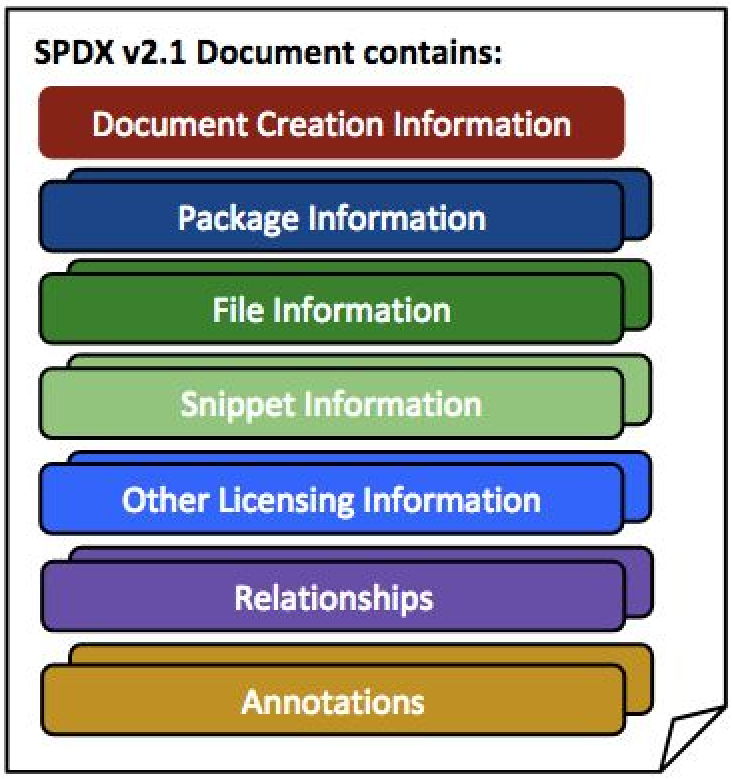
The specification is a living document, as as new use-cases are examined, it evolves.   Care is taken to provide backwards compatibility.   Development progresses through collaboration between technical, business and legal professionals from a range of organizations to create a standard that addresses 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. Its 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 its a matter of using the sections (and subset of the fields in each section) that describe the software and metadata information you’re planning too share.

DRAFT - This is a Work in Progress.


SPDX v2.1 Document contains:
Document Creation Information
Package Information
File Information
Snippet Information
Other Licensing Information
Relationships
Annotations

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. Its just 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 compents.
- 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.

Areas to Improve

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

DRAFT - This is a Work in Progress.

Translation and Linkage Guidance

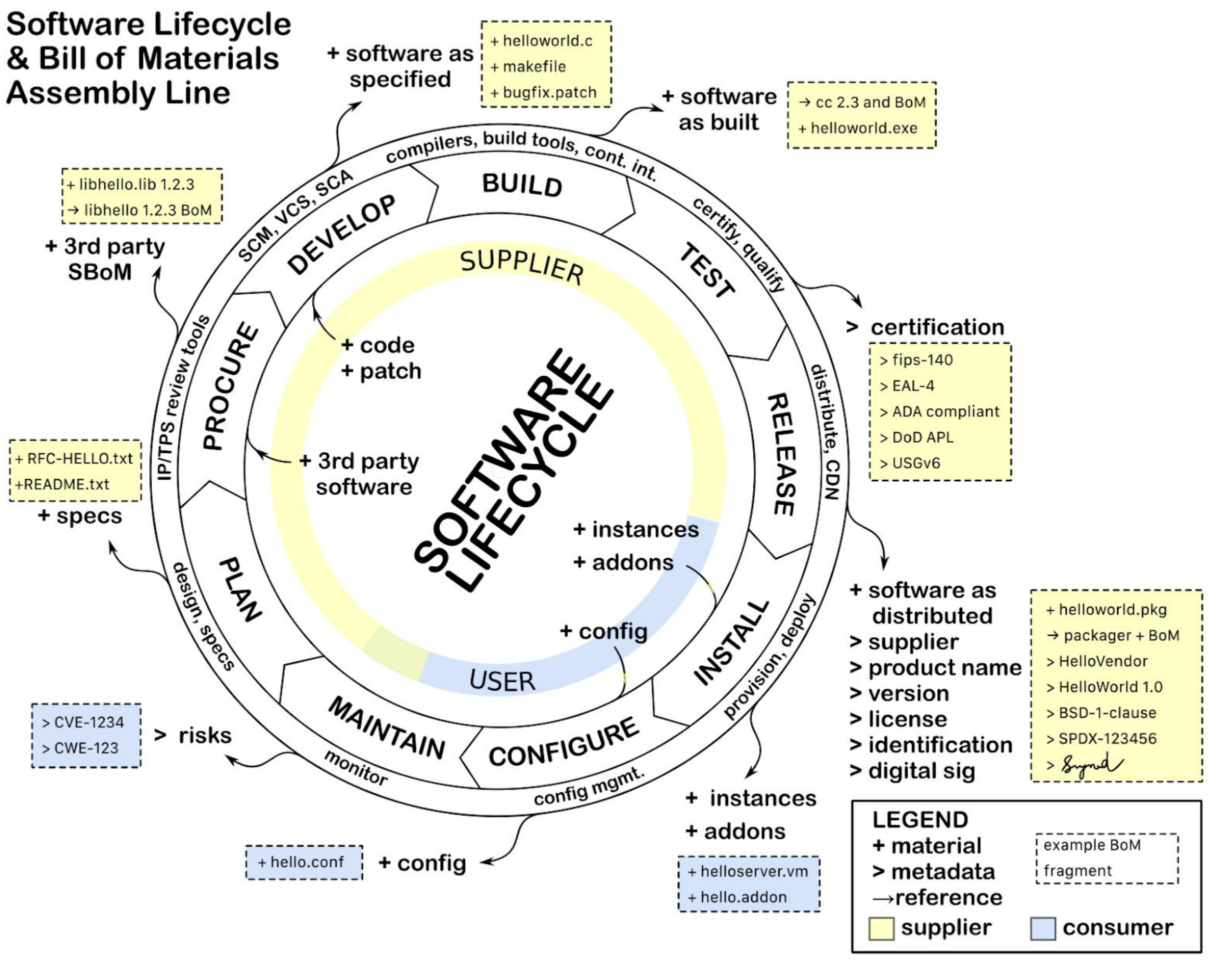
- Fields that correspond to each other, from minimum SBOM
- How to cross link between documents
-

DRAFT - This is a Work in Progress.

How to Apply

How to produce SBoM?

Software products start as ideas and concepts. Suppliers transform these ideas by adding or removing material until a final desired product is created. Consumers obtain these products and may further transform them to suit their needs. As a product is being used, users and suppliers find problems or get ideas for improvements. Products are then improved through the iterative cycle called software lifecycle.



Information that goes into SBoMs can be best obtained from the tools and processes used in each stage of the software lifecycle. One may leverage existing tools and processes to generate SBoMs. Such tools 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.

Many of the currently available off the shelf software lifecycle tools may not have the capability to generate SBoMs as a byproduct when they add material that goes into making a product. Suppliers should consider enhancing or retrofitting existing tools and processes to generate and maintain SBoMs. It may be impractical to retroactively generate complete SBoMs for older or existing products. SBoM can 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 it clear so that consumers can make informed use of SBoMs based on the available data.

How to deliver SBoM?

SBoMs can be bundled together with the software product itself as a compendium and stored with the installed software.

SBoMs can also be seen as a ledger to which information is added as software moves through the assembly line of various tools and processes, and through a chain of suppliers and consumers. A block chain inspired solution based on OpenChain could be one potential way to deliver SBoM [add SParts or OpenChain reference?].

How to update SBoM?

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. Errata may have to supplied for compendium type SBoMs.

How to consume SBoM?

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.

DRAFT - This is a Work in Progress.

Related Format Efforts Surveyed

TODO: Summarize, also include how can be linked to other formats.

Software Heritage Index (TBD)

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.

DRAFT - This is a Work in Progress.

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 components is:

- scheme: this is the URL scheme with the constant value of "pkg". One of the primary reason 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
```

DRAFT - This is a Work in Progress.

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.

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

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[2] 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

[2] 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.

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 following is an example of a CoSWID tag, in hex-based binary:

```
bf0f65656e2d5553207820636f6d2e61636d652e727264323031332d63652d7370312d76342d31
2d352d300cc2410101783041434d4520526f616472756e6e657220446574656374f722032303
13320436f796f74652045646974696f6e205350310d65342e312e350e2002bf181f745468652041
434d4520436f72706f726174696f6e18206861636d652e636f6d18219f0120ffff04bf18267823687
474703a2f2f7777772e676e752e6f72672f6c6963656e7365732f67706c2e7478741828676c6963
656e7365ff05bf182d6432303133182f66636f796f7465183473526f616472756e6e657220446574
6563746f72183663737031ff06bf11bf18186e7272646574656563746f722e657865141a200820e80
79f015820a314fc2dc663ae7a6b6bc6787594057396e6b3f569cd50fd5ddb4d1bbafd2b6affffffff
```

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.

The following is a more human-readable representation of the CBOR encoding of the example CoSWID:

```
BF                                       # map(*)
   0F                                    # unsigned(15)
   65                                    # text(5)
      656E2D5553                         # "en-US"
   20                                    # negative(0)
   78 20                                 # text(32)

636F6D2E61636D652E727264323031332D63652D7370312D76342D312D352D30 #
"com.acme.rrd2013-ce-sp1-v4-1-5-0"
   0C                                    # unsigned(12)
   C2                                    # tag(2)
      41                                 # bytes(1)
```

DRAFT - This is a Work in Progress.

```
         01                                # "\x01"
   01                                      # unsigned(1)
   78 30                                   # text(48)

41434D4520526F616472756E6E657220446574656374F72203230313320436F796F7
4652045646974696F6E20535031 # "ACME Roadrunner Detector 2013 Coyote
Edition SP1"
   0D                                      # unsigned(13)
   65                                      # text(5)
      342E312E35                           # "4.1.5"
   0E                                      # unsigned(14)
   20                                      # negative(0)
   02                                      # unsigned(2)
   BF                                      # map(*)
      18 1F                                # unsigned(31)
      74                                   # text(20)
         546865204143D4520436F72706F726174696F6E # "The ACME
Corporation"
      18 20                                # unsigned(32)
      68                                   # text(8)
         61636D652E636F6D                  # "acme.com"
      18 21                                # unsigned(33)
      9F                                   # array(*)
         01                                # unsigned(1)
         20                                # negative(0)
         FF                                # primitive(*)
      FF                                   # primitive(*)
   04                                      # unsigned(4)
   BF                                      # map(*)
      18 26                                # unsigned(38)
      78 23                                # text(35)

687474703A2F2F7777772E676E752E6F72672F6C6963656E7365732F67706C2E74787
4 # "http://www.gnu.org/licenses/gpl.txt"
      18 28                                # unsigned(40)
      67                                   # text(7)
         6C6963656E7365                    # "license"
      FF                                   # primitive(*)
   05                                      # unsigned(5)
   BF                                      # map(*)
      18 2D                                # unsigned(45)
      64                                   # text(4)
```

```
         32303133                       # "2013"
      18 2F                             # unsigned(47)
      66                                # text(6)
         636F796F7465                   # "coyote"
      18 34                             # unsigned(52)
      73                                # text(19)
         526F616472756E6E6572204465746563746F72 # "Roadrunner
Detector"
      18 36                             # unsigned(54)
      63                                # text(3)
         737031                         # "sp1"
      FF                                # primitive(*)
   06                                   # unsigned(6)
   BF                                   # map(*)
      11                                # unsigned(17)
      BF                                # map(*)
         18 18                          # unsigned(24)
         6E                             # text(14)
            72726465746563746F722E657865 # "rrdetector.exe"
         14                             # unsigned(20)
         1A 200820E8                    # unsigned(537403624)
         07                             # unsigned(7)
         9F                             # array(*)
            01                          # unsigned(1)
            58 20                       # bytes(32)

A314FC2DC663AE7A6B6BC6787594057396E6B3F569CD50FD5DDB4D1BBAFD2B6A #
"\xA3\x14\xFC-\xC6c\xAEzkk\xC6xu\x94\x05s\x96\xE6\xB3\xF5i\xCDP\xFD]\
xDBM\e\xBA\xFD+j"
            FF                          # primitive(*)
         FF                             # primitive(*)
      FF                                # primitive(*)
   FF                                   # primitive(*)
```

DRAFT - This is a Work in Progress.

Future Research areas (TBD)

- Globally unique and immutable component identifiers
- Archaeological digs, forensic, provenance