NISTIR 8060

Guidelines for the Creation of Interoperable Software Identification (SWID) Tags

David Waltermire
Brant A. Cheikes
Larry Feldman
Greg Witte

This publication is available free of charge from:
http://dx.doi.org/10.6028/NIST.IR.8060

National Institute of Standards and Technology
U.S. Department of Commerce

NISTIR 8060

Guidelines for the Creation of Interoperable Software Identification (SWID) Tags

David Waltermire
Computer Security Division
Information Technology Laboratory

Brant A. Cheikes
The MITRE Corporation
Bedford, Massachusetts

Larry Feldman
Greg Witte
G2, Inc.
Annapolis Junction, Maryland

This publication is available free of charge from:
http://dx.doi.org/10.6028/NIST.IR.8060

April 2016

U.S. Department of Commerce
Penny Pritzker, Secretary

National Institute of Standards and Technology
Willie May, Under Secretary of Commerce for Standards and Technology and Director

National Institute of Standards and Technology Internal Report 8060
86 pages (April 2016)

This publication is available free of charge from:
http://dx.doi.org/10.6028/NIST.IR.8060

Comments on this publication may be submitted to:

National Institute of Standards and Technology
Attn: Computer Security Division, Information Technology Laboratory
100 Bureau Drive (Mail Stop 8930) Gaithersburg, MD 20899-8930
Email: nistir8060-comments@nist.gov

Reports on Computer Systems Technology

The Information Technology Laboratory (ITL) at the National Institute of Standards and Technology (NIST) promotes the U.S. economy and public welfare by providing technical leadership for the Nation's measurement and standards infrastructure. ITL develops tests, test methods, reference data, proof of concept implementations, and technical analyses to advance the development and productive use of information technology. ITL's responsibilities include the development of management, administrative, technical, and physical standards and guidelines for the cost-effective security and privacy of other than national security-related information in federal information systems.

Abstract

This report provides an overview of the capabilities and usage of software identification (SWID) tags as part of a comprehensive software lifecycle. As instantiated in the International Organization for Standardization/International Electrotechnical Commission 19770-2 standard, SWID tags support numerous applications for software asset management and information security management. This report introduces SWID tags in an operational context, provides guidelines for the creation of interoperable SWID tags, and highlights key usage scenarios for which SWID tags are applicable.

Keywords

software; software asset management; software identification (SWID); software identification tag

Acknowledgments

The authors would like to thank Harold Booth, Bob Byers, Christopher Johnson, and Alex J. Nelson of the National Institute of Standards and Technology (NIST); Steve Klos of TagVault.org and 1E; Christine Deal and Charles Schmidt of The MITRE Corporation; Piotr Godowski and Brian Turner of IBM; Hopeton Smalling of OQI Cares, Inc.; Sharon Hope of NASA Jet Propulsion Laboratory; and John Richardson of Veritas for their reviews and contributions of feedback to this report. The authors would also like to thank Dr. Peter Fonash and Juan Gonzalez from the U.S. Department of Homeland Security (DHS) for their ongoing support for and contributions to this report. The authors would also like to thank Jessica Fitzgerald-McKay from the National Security Agency (NSA) for supporting the development of this report.

Document Conventions

This report provides both informative and normative guidance supporting the use of SWID tags. The key words “MUST”, “MUST NOT”, “REQUIRED”, “SHALL”, “SHALL NOT”, “SHOULD”, “SHOULD NOT”, “RECOMMENDED”, “MAY”, and “OPTIONAL” in this report are to be interpreted as described in Request for Comment (RFC) 2119 [RFC 2119]. When these words appear in regular case, such as “should” or “may”, they are not intended to be interpreted as RFC 2119 key words.

Some of the requirements and conventions used in this report reference Extensible Markup Language (XML) content. These references come in two forms, inline and indented. An example of an inline reference is: A patch tag is differentiated by the fact that the value of the `@patch` attribute within the <`SoftwareIdentity`> element is “`true`”.

In this example, the notation <`SoftwareIdentity`> can be replaced by the more verbose equivalent “the XML element whose qualified name is `SoftwareIdentity`”.

The general convention used when describing XML attributes within this report is to reference the attribute as well as its associated element, employing the general form “`@attributeName` for the <`prefix:localName`>”. Attribute values are indicated in quotations, such as the example “`true`” above.

In cases where any valid value may be provided for an XML attribute, this report specifies “`<any>`” as the attribute value.

This report defines a number of new XML attributes that are extensions to the SWID specification. These extension attributes are defined in a new XML namespace, http://csrc.nist.gov/ns/swid/2015-extensions/1.0. These new attributes will be assigned the prefix “n8060” mapped to this namespace. In guidelines and examples, extension attributes will be provided in the form “`@n8060:attributeName`”. The schema for these extensions can be downloaded at http://csrc.nist.gov/schema/swid/2015-extensions/swid-2015-extensions-1.0.xsd.

Indented references are intended to represent the form of actual XML content. Indented references represent literal content by the use of a fixed-length font, and parametric (freely replaceable) content by the use of an italic font. Square brackets “[]” are used to designate optional content.

Both inline and indented forms use qualified names to refer to specific XML elements. A qualified name associates a named element with a namespace. The namespace identifies the XML model, and the XML schema is a definition and implementation of that model. A qualified name declares this schema to element association using the format “*prefix*:*element-name*”. The association of prefix to namespace is defined in the metadata of an XML document and varies from document to document.

Many portions of this document include cross references to other sections. Such references are often indicated by the use of the “section symbol” (§), with two symbols indicating multiple sections (§§).

Table of Contents

List of Appendices

List of Figures

List of Tables

1 Introduction

The International Organization for Standardization (ISO) and the International Electrotechnical Commission (IEC) has published ISO/IEC 19770-2, an international standard for software identification tags, also referred to as SWID tags. A *SWID tag* is a structured set of data elements that identify and describe a software product. The first version of the standard, ISO/IEC 19770-2:2009 [ISO/IEC 19770-2:2009], was published in November 2009. A significantly revised version of the standard, ISO/IEC 19770-2:2015 [ISO/IEC 19770-2:2015], was published in October 2015, and is referenced herein as the *SWID specification*.

This report provides an overview of the capabilities and usage of SWID tags defined by the ISO/IEC 19770-2:2015 standard. Additionally, this report describes the use of SWID tags as part of comprehensive software asset management lifecycles and cybersecurity procedures. Section 1.1 discusses the software asset management and cybersecurity problems that motivated the development of SWID tags. Section 1.2 highlights the stakeholder benefits that can be gained as SWID tags become more widely produced and consumed within the software marketplace. Section 1.3 describes the purpose and target audiences of this report. Section 1.4 summarizes this section's key points, and Section 1.5 describes how the rest of this report is organized.

1.1 Problem Statement

Software is part of the critical infrastructure for the modern world. Enterprises and individuals routinely acquire software products and deploy them on the physical and/or virtual computing devices they own or operate. This report refers to an instance of such a computing device as a *device*. ISO/IEC 19770-5 [ISO/IEC 19770-5:2013], a companion standard to the SWID specification, defines *software asset management* (SAM) as "control and protection of software and related assets within an organization, and control and protection of information about related assets which are needed in order to control and protect software assets." A core SAM process is *software inventory management*—the process of building and maintaining an accurate and complete inventory of all software products deployed on all of the devices under an organization's or individual's operational control.

Accurate software inventories of managed devices are needed to support higher-level business, information technology, and cybersecurity functions. For example, enterprises need to know how many copies of a given product are installed in order to ensure compliance with software license agreements. To ensure they are not paying for unneeded licenses, enterprises also need to know where specific copies are installed. As another example, operations personnel need accurate and complete software inventories to ensure that all deployed software assets are authorized, appropriately patched, free of known exploitable weaknesses, and configured according to their organizations' security policies. Organizations may also use software inventory information to plan software investments and resources needed to support upgrades to and replacement of legacy systems.

Effective software inventory management depends on the ability to *discover*, *identify*, and *contextualize* software products installed on managed devices. Software discovery processes analyze observable states of a managed device to detect and enumerate discrete units of installed software. Discovery is technically challenging due to the enormous variation across the software

industry in what it means to be a unit of software. For example, a single unit of software may consist of a combination of executable files, data files, configuration files, library files, and so forth. A single unit of software may also include supporting software units which may be independently installed and executed, as well as changes to the underlying operating environment, such as the addition of device drivers and entries in operating-system maintained tables and databases. Discovery processes need to be able to handle all these sources of variation while avoiding "false positives" (i.e., erroneous reports that a unit of software is present) as well as "false negatives" (i.e., failures to report discovery of a unit of software that is, in fact, present).

Once the discrete units of software have been enumerated on a device, software identification processes assign identifying labels to those units. These labels are used in various contexts to refer to the products, report their presence, and correlate with other sources of information. A key requirement of the labeling process is that when the same unit of software is discovered on different devices, it must be assigned the same label. Identification is technically challenging because the identifying labels are not physically part of the software units, nor can they be discovered in the same manner. Instead, the labels are assigned using inferential techniques based on observable features which vary widely by software provider, operating environment, and device. These inferential techniques may be inaccurate, unreliable, and/or proprietary.

Assuming software units can be discovered and identified, software contextualization processes associate the identifying label with other sources of enriching information. For example, the label of a software unit may be used to collect key descriptive characteristics, such as the software unit's exact version, license keys, patch level, associated files in device storage areas, and associated configuration settings. As another example, the assigned software identifier may also be used to search for related patches, upgrades, vulnerabilities and remedies, and configuration checklists. Contextualization is technically challenging to the extent that it depends on widespread agreement on, and dissemination of, the identifying labels to be assigned to units of software.

The SWID tag standard was developed to help overcome the technical challenges associated with software discovery, identification, and contextualization, and thereby enhance the accuracy and reliability of software asset management processes. SWID tags aid discovery by furnishing a standardized indicator of a software product's presence on a device. Tags aid identification by including a consistent label for a product within its tag. Finally, tags aid contextualization by allowing a wide variety of related product details to be supplied, including the product's full name and version.

1.2 SWID Tag Benefits

SWID tags offer benefits to creators and publishers of software products, as well as those who acquire and use those software products. The SWID specification identifies these stakeholders as:

- **Tag producers:** Organizations and entities that create SWID tags for use by others in the market. Examples of tag-producing organizations include the organizations (or their authorized agents) involved in creating, licensing, packaging, and/or distributing software products. These organizations are best able to ensure that tags contain correct, complete,

and consistent data when describing software products. Tag producers are responsible for establishing a norm for data values in the tags they produce, with a goal of ensuring that the same data values are used, where possible and appropriate, across tags for similar products or product versions they produce. Tags may also be produced and distributed by third parties, as well as by automated discovery tools.

- **Tag consumers:** Organizations and entities that use information contained in SWID tags to support higher-level, software-related business and cybersecurity functions. Categories of tag consumers include software consumers, inventory/discovery tools, inventory-based cybersecurity tool providers (e.g., providers of software vulnerability management products, which rely on accurate inventory information to support accurate vulnerability assessment), and organizations that use these tools.

The implementation of SWID tags supports these stakeholders throughout the entire software lifecycle—from software creation and release through software installation, management, and de-installation. As more software creators also become tag producers and release their products with SWID tags, more consumers of software products are enabled to consume the associated tags. This gives rise to a "virtuous cycle" where all stakeholders gain a variety of benefits, including the abilities to:

- Consistently and accurately identify software products that need to be managed for purposes such as inventory, licensing, and cybersecurity, or for the management of software and software dependencies.
- Use stable software identifiers to report changes to a device's software load as products are installed, patched, upgraded, and removed.
- Exchange software information between software producers and consumers in a standardized format, regardless of software creator, platform, or management tool.
- Identify and manage software products equally well at any level of abstraction, regardless of whether a product consists of a single application or one or more groups or bundles.
- Correlate information about installed software with other information, including list(s) of authorized software, related patches, configuration settings, security policies, and threat advisories.
- Automatically track and manage software license compliance and usage by combining information within a SWID tag with independently-collected software entitlement data.
- Aggregate deployed software asset information across an enterprise, providing the organization with knowledge of what software is deployed on specific devices.
- Record details about the deployed footprint of installed products on devices, such as the list of supporting software components, executable and data files, system processes, and generic resources that may be included in the installation (e.g., device drivers, registry settings, accounts).
- Identify all organizational entities associated with the installation, licensing, maintenance, and management of a software product on an ongoing basis. This identification includes entities external to the software consumer (e.g., software creators, software licensors, packagers, and distributors), and those internal to the software consumer's organization.

- Use digital signatures to validate that information within a tag comes from a known source and has not been corrupted.

1.3 Purpose and Audience

This report has three purposes. First, it provides a high-level description of SWID tags in order to increase familiarity with the standard. Second, it provides tag implementation guidelines that supplement the SWID tag specification. Third, it presents a set of operational usage scenarios, which illustrate how SWID tags conforming to these guidelines can be used to achieve a variety of cybersecurity goals. By following the guidelines in this report, tag producers can have confidence they are providing all the necessary data, with the requisite data quality, to support the operational goals of each tag usage scenario. Additionally, tag consumers can have confidence that the tags they are using adequately support each tag usage scenario.

This report addresses four distinct audiences. The first audience is *software providers* – the individuals and organizations that develop, license, and/or distribute commercial, open source, and custom software products, to include software developed solely for in-house use. This report helps providers understand the problems addressed by SWID tags, why providers' participation is essential to solving those problems, and how providers may produce and distribute tags that support a wide range of usage scenarios.

The second audience is *providers of software build, packaging, and installation tools* – the individuals and organizations that develop tools used by *software providers* to build, package, release, and support installation of software. These tools can provide much of the information needed for SWID tag creation. If these tools provide the ability to generate SWID tags as part of their native functionality, SWID tags can be produced by *software providers* with minimal effort. Using this approach, SWID tags can be generated automatically for any software release managed by the tool, increasing the availability of SWID tags for related products. Support for such a SWID tag generation capability is critical for ensuring that SWID tags are provided as part of all commercial and open source software releases. Furthermore, it is important that installation tools support the consistent management of installed tags on devices during software installation, upgrade, patch, and removal processes. The guidance in this report identifies the SWID tag information needed to ensure that tags generated by these tools support the usage scenarios required by *software consumers*.

The third audience is *providers of inventory-based products and services* – the individuals and organizations that develop tools for discovering, monitoring, and managing software assets for any reason, including securing enterprise networks using standardized inventory information. This audience needs consistency in the content and interpretation of data in SWID tags collected from computing devices to make full use of this information. This report offers guidance to *software providers* on how to consistently implement tags to support SWID tag usage scenarios. The degree of consistency supported by this guidance helps *inventory-based product providers* to use tag data to materially enhance the quality and coverage of software information collected and utilized by their products.

The fourth audience is *software consumers* – the individuals and organizations that install and use commercial, open source, and/or in-house developed software products, and inventory-based products and services. This report helps *software consumers* understand the benefits of software

products that are delivered with SWID tags, and why they should encourage *software providers* to deliver products with SWID tags that meet their anticipated usage scenarios. The guidance provided in this report to *software providers* also promotes the production of tags that are useful in meeting the usage scenarios of *software consumers*.

Using a set of well-defined usage scenarios, this report identifies how the goals of these four audiences are interrelated. Consumers are trying to cope with software management and cybersecurity challenges that require accurate software inventory. Consumers also are seeking solutions that help to reduce the total cost of ownership for the software they manage. Consumers need to understand how SWID tags can help them, need providers to supply high-quality tags, and need implementers of inventory-based tools to collect and use tags.

Providers need to recognize that adding tags to their products will make their products more useful and more manageable, and also need this recognition to be reinforced by consumer demand for tag support. *Software build and installation tool providers* can assist *software providers* with producing tags for software releases and managing tags as part of software installation processes. Inventory-based tool implementers are uniquely positioned to recognize how tags can make their products more reliable and effective, and to work constructively with both consumers and providers to promote software tagging practices.

1.4 Section Summary

The following are the key points of this section:

- The ISO/IEC 19770-2:2015 international standard specifies the data format for SWID tags described by this report.
- SWID tags were developed to help enterprises meet the need for accurate and complete software inventories to support higher-level business and cybersecurity functions.
- SWID tags provide benefits to organizations that create and use tags.
- Four audiences have interrelated goals pertaining to SWID tags and tagging practices:
    - *Software providers* often want to increase the manageability of their products for their customers. To justify investing the resources necessary to become tag providers, they need consumers to send clear signals that they value product manageability as much as features, functions, and usability.
    - *Providers of software build, packaging, and installation tools* often want to support SWID tags in their tools. This support can assist software providers that use their tools with generating tags during product build and release processes. Additionally, these tools can support the management of installed tags on devices during software installation, upgrade, patch, and removal processes. Providers need clear guidance on what information needs to be included in a SWID tag to ensure that they generate useful tags for software consumers.
    - *Providers of inventory-based products and services* often want to use SWID tags as their primary method for identifying software. These providers need tags to become more widely available in order to make their specialized tools more

reliable and effective based on tag data. They act as software providers as well as software consumers, and thus have the needs and goals of both audiences.

- *Software consumers* often want software providers to supply tags along with their products as a common practice. Consumers need tags to be routinely provided in order to cope with the challenges of maintaining an accurate software inventory, and to address software-related management and cybersecurity issues.

- This report seeks to raise awareness of the SWID tag standard, promote understanding of the business and cybersecurity benefits that may be obtained through increased adoption of software tagging standards and practices, and provide detailed guidance to both producers and consumers of SWID tags to promote consistency and interoperability.

1.5 Report Structure

The remainder of this report is organized into the following sections and appendices:

- Section 2 presents general SWID tag concepts that are helpful for understanding the different types of tags, how tags are created, and how tags are made available for use.
- Section 3 presents the SWID tag data elements described in the SWID specification, and discusses specific data elements that support the authentication of tags.
- Section 4 provides implementation guidelines that address common issues related to tag deployment and processing on information systems.
- Section 5 provides implementation guidelines for specific types of tags.
- Section 6 presents usage scenarios for software asset management and software integrity management processes which are enhanced through the use of SWID tags.
- Appendix A presents a list of acronyms used in this report.
- Appendix B provides the references for the report.

2 SWID Tag Overview

A *SWID tag* is a standardized XML format for a set of data elements that identify and describe a software product. This report uses the term *tagged software product* (or, simply, *tagged product*) to refer to a software product that is installed on a device along with one or more discoverable tags describing that product.

This section provides a general overview of concepts related to SWID tags and is organized as follows. Section 2.1 describes the four types of SWID tags and the distinct role each type of tag plays at key points in the software lifecycle. Section 2.2 discusses common methods for creating tags. Section 2.3 discusses expectations regarding the deployment of SWID tags. Finally, Section 2.4 concludes with a summary of key points from this section.

2.1 SWID Tag Types and the Software Lifecycle

The SWID specification defines four types of SWID tags: *corpus*, *primary*, *patch*, and *supplemental*. Corpus, primary, and patch tags have similar functions in that they describe the existence and/or presence of different types of software, 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. Figure 1 illustrates the steps in the software lifecycle and the relationships among those lifecycle events supported by the four types of SWID tags, as follows:

- **Software Deployment.** Before the software product is installed (i.e., *pre-installation*), and while the product is being deployed, a corpus tag provides information about the installation files and distribution media (e.g., CD/DVD, distribution package).
- **Software Installation.** A primary tag will be installed with the software product (or subsequently created) to uniquely identify and describe the software product. Supplemental tags are created to augment primary tags with additional site-specific or extended information. Patch tags may also be installed during software installation to provide information about software fixes deployed along with the base software installation.
- **Software Patching.** When a new patch is applied to the software product, a new patch tag is provided, supplying details about the patch and its dependencies.
- **Software Upgrading.** As a software product is upgraded to a new version, new primary and supplemental tags replace existing tags, enabling timely and accurate tracking of updates to software inventory.
- **Software Removal.** Upon removal of the software product, relevant SWID tags are removed. This removal event can trigger timely updates to software inventory reflecting the product's removal.

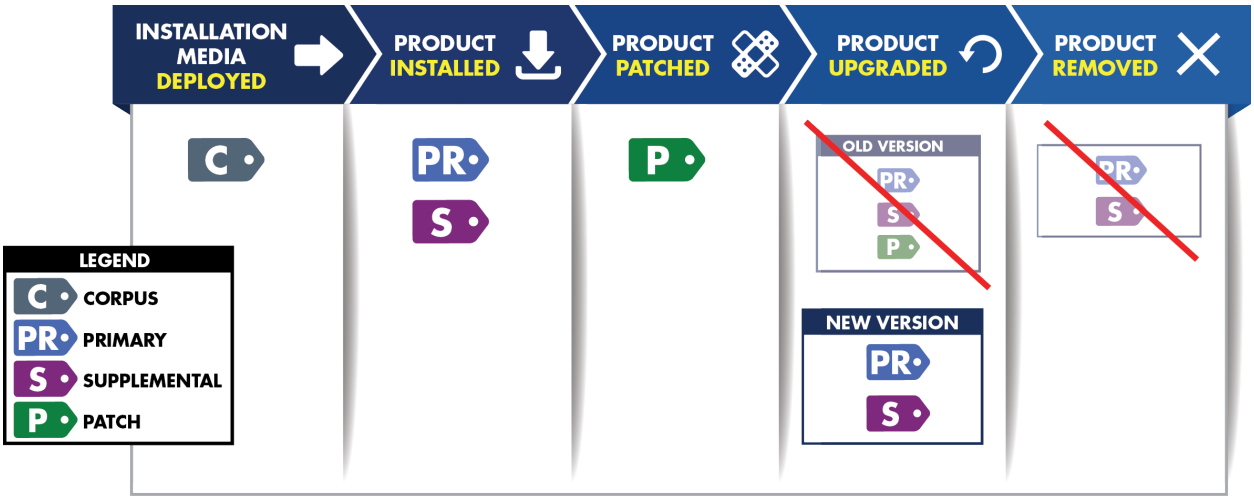

Figure 1: SWID Tags and the Software Lifecycle

The software lifecycle events described in Figure 1 represent typical uses of tags within the software deployment lifecycle. While not depicted in Figure 1, software discovery, configuration management, and vulnerability management processes generate other lifecycle events which may create and use primary, patch, and supplemental tags. The four tag types related to software lifecycle events are discussed in the following subsections.

2.1.1 Corpus Tags

Before software is installed, it is typically delivered or otherwise made available to a computing device in the form of a software installation package. The installation package contains the software in a pre-installation condition, often compressed in some manner. Common formats for installation packages include TAR and ZIP files, and "self-unpacking" executable files. In all cases, an installation procedure must be run to cause the software contained in an installation package to be unpacked and deployed on a target device. The SWID specification defines *corpus tags* for vendors and distributors to use to identify and describe products in such a pre-installation state. The availability of software identification and descriptive information for a software installation package enables verification of the software package and authentication of the organization releasing the package.

Corpus tags are intended to be used as inputs to software installation tools and processes; they are not installed on devices as part of an installation procedure. Corpus tags may be used by installation tools to verify the integrity of an installable product, and to authenticate the issuer of an installation package before executing the installation procedure (see §3.2). If a manifest of the installation files is included in the corpus tag (see §3.1.6 on the `<Payload>` element), installation package tampering can be detected and the installation procedure aborted if the payload data and the actual files do not match. When combined with other licensing data, corpus tags may aid consumers in confirming whether they have a valid license for a product before they install it. All of this information can be used as part of an automated policy decision to allow or prevent the software installation (see §6.2.1) on a device.

2.1.2 Primary Tags

As previously illustrated in Figure 1, primary tags are involved in different software lifecycle events. The SWID specification defines *primary tags* to identify and describe software products once they have been successfully installed on a computing device. The primary tag for each tagged product needs to furnish values for all data elements that are designated "mandatory" in the SWID specification. A minimal primary tag supplies the name of the product (as a string), a globally unique identifier for the tag, and basic information identifying the tag's creator.

Ideally, the software provider is also the creator of that product's primary tag; however, the SWID specification allows other parties (including automated tools) to create tags for products in cases where software providers have declined to do so or have delegated this responsibility to another party. The SWID specification recognizes a number of alternative roles for creators of tags, including organizations and individuals which aggregate, distribute, and license software products (see §3.1.2).

A globally unique tag identifier is essential information in many usage scenarios because it may be used as a globally unique proxy identifier for the software installation. The tag identifier of a primary tag can be considered a proxy identifier for the tagged product because there is a one-to-one relationship between the primary tag and the installed software it identifies. For increased processing efficiency, inventory and discovery tools may choose to identify and report tagged products using just their tag identifiers rather than their fully populated primary tags.

When a product is upgraded, the primary tag(s) associated with the old version are removed and replaced with primary tag(s) for the new version. When a product is removed from a device, its primary tag(s) are removed as well. By strictly maintaining the one-to-one association between installed software and associated tags, it is possible to continuously monitor installed software inventory and track software updates using SWID tag data (see §6.1.1).

Because software products may be furnished as suites or bundles or as add-on components for other products, the SWID specification defines a `<Link>` element (see §3.1.4), which may be used within a SWID tag to indicate relationships between the product described by the tag and other products or items that may be available. Two noteworthy relationships are:

- **Component.** Indicates that the product described by the primary tag has a separately installable software product as one of its components. In such situations, the product's primary tag points to the primary tag of the component product using a `<Link>` element where the `@rel` attribute is set to "`component`".
- **Requires.** Indicates that the product described by the primary tag depends on a separately installable software product. In such situations, the primary tag points to the primary tag of the required product using a `<Link>` element where the `@rel` attribute is set to "`requires`".

The relationships that may be expressed in primary tags are illustrated in Figure 2 below. This illustration shows how primary SWID tags may indicate the associations among several software components. The figure shows an overarching Productivity Suite that has a **component** relationship with each of its supporting applications. The figure also shows how the two

subsidiary products may **require** other component software (e.g., language pack, spell checking software).

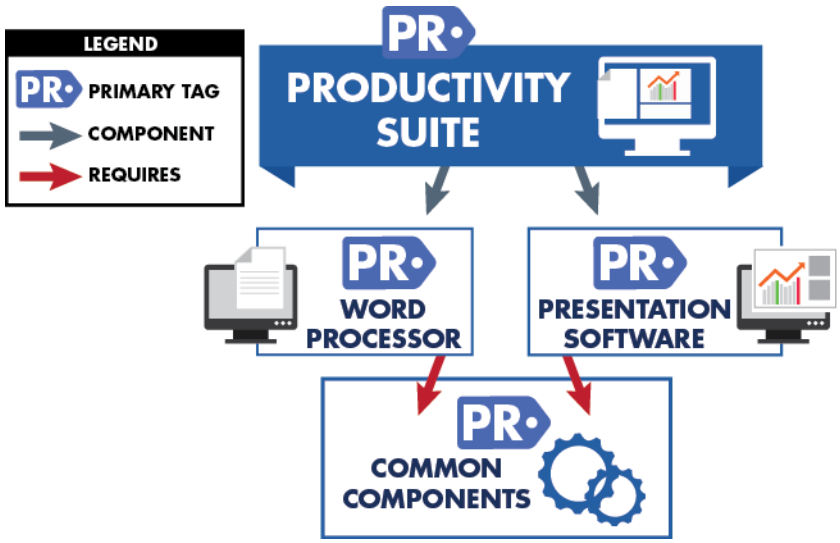

Figure 2: Primary Tag Relationships

2.1.3 Patch Tags

A software provider may release patches to correct errors in, or add new features to, a product. When a patch is installed on a device, changes are made to a product's installation footprint. Since a patch augments an existing installation, these changes need to be tracked separately. The SWID specification defines *patch tags* for software providers to identify and describe each patch. When a patch is installed, a patch tag is placed on the device in a location associated with the patched product. When a patch tag is not provided by a software provider, discovery tools can create a patch tag to indicate the previous application of a patch. In contrast with a patch, an *upgrade* is defined as a complete replacement of a product's installation footprint. An upgrade typically changes the product's version number and/or release details.

The data elements contained in a patch tag identify and describe the patch, rather than the product to which the patch is applied. For example, the product name and version recorded in a patch tag need not match the product name and version recorded in the patched product's primary tag. Instead, these attributes can be used to record the name and version of the patch assigned by the software provider.

A patch tag can indicate relationships between the patch described by the patch tag and other products or patches that may be available using the `<Link>` element (see §3.1.4). Three types of relationships are worth noting here:

- **Patches.** Indicates that a patch applies to a patched product. In such situations, the patch's patch tag points to the primary tag of the patched product using a `<Link>` element where the `@rel` attribute is set to "`patches`".

- **Requires.** Indicates that a patch requires the prior installation or co-existence of another patch. In such situations, the patch's patch tag points to the patch tag of the required patch using a `<Link>` element where the `@rel` attribute is set to "`requires`".
- **Supersedes.** Indicates that a patch can entirely replace another patch. In such situations, if the other patch is installed, it might be removed or overwritten when the new patch is installed. This is expressed using a `<Link>` element where the `@rel` attribute of the superseding patch tag is set to "`supersedes`" with a pointer to the patch tag of the superseded patch.

When used in this way, patch tags may assist in determining whether an installed product has all required patches applied (see §6.1.2). Figure 3 illustrates the tag relationships for four product patches that can be applied over time.

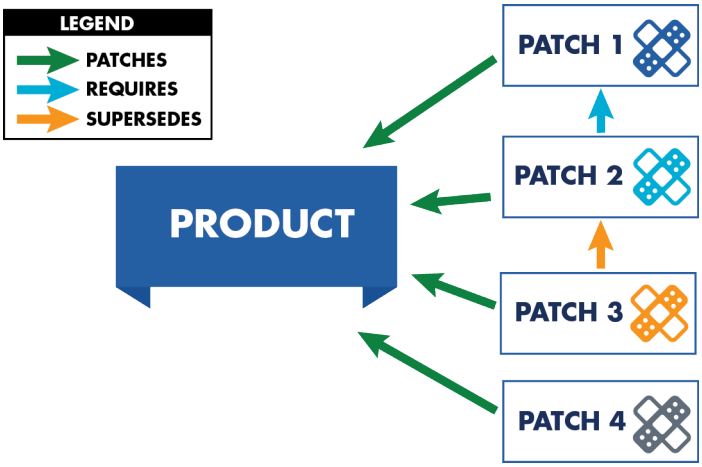

Figure 3: Patch Tag Relationships

All patches have `@rel=patches` Product, since they all patch the same product.

Patch 2 has `@rel=requires` Patch 1, since Patch 1 must be installed before Patch 2.

Patch 3 has `@rel=supersedes` Patch 2, since Patch 3 entirely replaces Patch 2. Although not depicted here, Patch 3 could also require or supersede Patch 1 depending on the specific scenario. Such additional relationships would be indicated by additional `<Link>` elements.

Patches 1 and 4 are completely independent of the other two patches, so their patch tags do not include any `<Link>` elements pointing to any of the other patch tags.

A patch can include a manifest of the new and/or changed files (see §3.1.6 for discussion on the `<Payload>` element), which can be used to verify that the actual patched files are present on the device. This allows for confirmation that the patch has been correctly installed and remains installed, preventing a malicious actor from deploying files that misrepresent the installation status of a patch.

2.1.4 Supplemental Tags

The SWID specification allows tags to be modified only by the tag creator. To provide a mechanism whereby consumers and software management tools may add arbitrary post-installation information of local utility, the SWID specification allows for any number of *supplemental tags* to be installed, either at the same time the primary tag is installed, or at any time thereafter.

Any entity may create a supplemental tag for any purpose. For example, supplemental tags may be created by automated tools in order to augment an existing primary tag with additional site-specific information, such as license keys and contact information for local responsible parties.

Each supplemental tag contains a pointer to the tagged product's primary tag using a `<Link>` element where the `@rel` attribute is set to "`supplemental`". When supplemental tags are present, a tag consumer may create a complete record of the information describing a product by combining the data elements in the product's primary tag with the data elements in any linked supplemental tags.

The relationships that may be expressed in supplemental tags are illustrated in Figure 4.

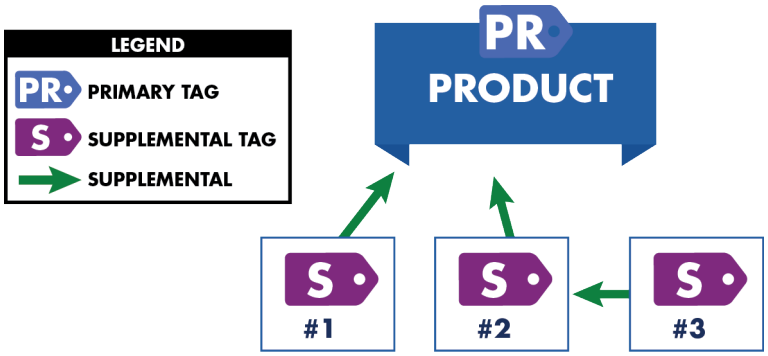

Figure 4: Supplemental Tag Relations

Supplemental tags may also be employed to augment non-primary tags. For example, a supplemental tag could add local information about a patch tag (e.g., to record a timestamp indicating when the patch was applied), or even about another supplemental tag (as illustrated in #3 above). In such situations, the supplemental tag also contains a `<Link>` element pointing to the tag that is having its information augmented.

A supplemental tag is intended to furnish data values that augment and do not conflict with data values provided by the primary tag and any of the product's other supplemental tags. If conflicts are detected, data in the primary tag, if provided by the software producer, is considered the most reliable, and tools can be expected to ignore conflicting data or to report all conflicting data as exceptions. For example, the mandatory product name recorded in a supplemental tag is expected to match the product name recorded in the product's primary tag; if they are different, the name recorded in the primary tag is considered the preferred name.

As previously illustrated in Figure 1, after a software product is upgraded, all primary, patch, and supplemental tags associated with the pre-upgrade version of the product will be removed. If

needed, new supplemental tags associated with the upgraded version may be deployed. When a software product is removed, all primary, patch, and supplemental tags associated with the product will be removed.

2.2 SWID Tag Creation

A SWID tag for a software package, product, or patch can be created in a number of ways. The following subsections describe the circumstances under which SWID tags are created and how the method of SWID tag creation impacts their use.

2.2.1 Corpus Tags

Corpus tags provide information about a software installation package. To ensure that the most complete and accurate information is provided, a corpus tag is ideally created through an automated process when the software installation package is created. Generation of a corpus tag is typically done as part of the software build and release process thereby ensuring that the tag is included with the software installation package and available for use during the installation processes described in Section 2.1.1. Additionally, the corpus tag can be signed to enable authentication of the tag provider as the software provider, verification of the tag's integrity, and verification of the software installation package footprint (see §3.2).

In cases where a corpus tag is not created by the software provider as part of the package creation process, a secondary party may generate a corpus tag using information gathered by analyzing an existing software installation package provided by another organization. This type of creation pattern is not ideal, since the secondary party may not have all the needed information, resulting in a corpus tag that contains incomplete or inaccurate information. Furthermore, while the tag can still be signed, the organization signing the tag will differ from the software installation package provider. This means that although the signed tag can still be used to authenticate the tag provider, to verify the tag's integrity, and to verify the software installation package footprint, the tag's signature cannot be used to authenticate that the tag provider is the software provider. This results in a lesser degree of assurance.

2.2.2 Primary and Patch Tags

To maximize the availability, accuracy, and completeness of information contained within a primary or patch tag, it is important that such a SWID tag be created during a product's build and/or release process by an authoritative source such as the software provider. As software is being prepared for release, build and packaging processes can automatically generate primary and patch tags using authoritative information made available during these processes. This is the ideal case for tag creation since the information available during these processes tends to be the most accurate and authoritative. Additionally, a primary or patch tag can be signed, enabling the authentication of the tag provider as the software provider, verification of the tag's integrity, and verification of the software's installed footprint (see §3.2).

If creation of a primary or patch tag at build or release time is not possible, a second approach is to create a SWID tag after the software is installed using authoritative information from a software package database. While this information may be equivalent to the information available during a build or packaging process, signing of these tags is not possible, since the software provider's signing key cannot be distributed. This limitation greatly reduces the

assurance of the SWID tag since the tag provider cannot be authenticated, and neither the tag's integrity nor the integrity of the software's installed footprint can be verified.

Lastly, a primary or patch tag can be created as the result of a technical analysis process. This type of creation can be performed by an entity that obtains a copy of a product after its release to market or by an automated software discovery tool that analyzes software installed on a device. In both cases, the resulting tag will likely contain incomplete or inaccurate information due to a lack of authoritative information. In the latter case, either the tag will be a matching tag retrieved from a store of previously analyzed tags, or the tag will be generated directly from the information available on the device. If a stored tag is used, it may be signed by the discovery tool vendor, allowing the signature to be used to authenticate the tag provider, verify the tag's integrity, and verify the installed footprint of the software discovered. However, the installed footprint might not be completely accurate, and the tag's signature cannot be used to authenticate that the tag provider is the software provider, resulting in a lesser degree of assurance.

2.2.3 Supplemental Tags

Supplemental tags provide a flexible and extensible means to augment the information provided in a corpus, primary, or patch tag to assist with the overall management of software, supporting related processes (e.g., configuration management, vulnerability management). The creators of supplemental tags can vary along with the uses of the supplemented information. The assurance of a supplemental tag can be improved through the application of a tag signature in the same ways as previous examples; however, in many cases, supplemental tags will be generated directly on the device, making signing impractical.

2.3 SWID Tag Deployment

This section describes various factors regarding the deployment of SWID tags. Section 2.3.1 describes how and where SWID tags should be deployed as the result of installing new software, applying a patch, or performing an update to existing software. Section 2.3.2 describes the deployment of SWID tags that are generated from existing package management data. Section 2.3.3 provides information about storing SWID tags within a repository separate from a software installation.

2.3.1 Deployment during Installation

The most common method of tag deployment is for a tag to be incorporated into a product's installation package, which then installs the tag on a device during the installation procedure. Such a procedure may be run during the installation of new software, when applying a patch, or when updating existing software. This method of tag deployment is available when the tag creator is able to ensure that the tag is included in the installation package. The SWID specification makes the following statements about SWID tag deployment in this situation:

> On devices with a file system, but no API [application programming interface] defined to retrieve SWID tags, the SWID tag data shall be stored in an XML file and shall be located on a device's file system in a sub-directory named "swidtag" (all lower case) that is located in the same file directory or sub-directory of the install location of the software component with which they are installed. It is recommended, but not required, that the swidtag directory is located at the top of the application installation directory tree. Any

> payload information provided must reference files using a relative path of the location where the SWID tag is stored. On devices that do not have a file system, the SWID tag data shall be stored in a data storage location defined and managed by the platform provider for that device. […] On devices that utilize both a file system for software installation as well as API access to the SWID tag files, it is recommended that the SWID tag data be stored in the API managed repository as well as stored as a file on the system. […] Finally, the SWID tag data may also be accessible via a URI, or other means […] [ISO/IEC 19770-2:2015, pp. 6-7].

The SWID specification provides the following rules and recommendations when creating names for SWID tag files:

> Filenames should be restricted to use only the characters listed in the Portable Filename Character Set defined in IEEE 1003.1:2013, 3.278 to maximize interoperability between platforms. If this limitation is too restrictive, the tag creator shall ensure that the characters used in the filename are valid characters for all platforms where their SWID tags may be stored on a file system. SWID tag base filenames (i.e., the filename without the .swidtag extension) shall be structured to be globally unique for the tag creator and product. SWID tag creators may use different approaches to defining the base portion of the SWID tag filename; however, if the filename aligns with the following structure, the filename will be unique for the product and recognizable by a system administrator <name of the tag creator> + <product name>.swidtag. The .swidtag file extension shall be used for all software identification tags. [ISO/IEC 19770-2:2015, p. 8]

Following this guidance, for example, the SWID tag referencing product name "ACME Roadrunner" from the software creator "acme.com" could be stored in a file named "`acme.com.acmeroadrunner.swidtag`".

As a general rule, after a tag has been installed on a device, software discovery processes are able to recover the tag from that device. This allows the product described by the tag to be listed in the software inventory associated with the device.

2.3.2 SWID Tag Generation from Existing Package Management Data

The second method of tag deployment is implicit and may be applicable in situations where software products are not accompanied by vendor-provided tags. Some operating environments furnish a standard package management system that automatically maintains a database of detailed product information as software products are installed or updated. In such cases, it may be possible to generate SWID tags dynamically by mapping fields in the package manager's database to tag data elements. Depending on the richness of the package manager's database, dynamically generated tags may contain highly reliable information, including all of the required data elements and many optional data elements.

When a tag is produced dynamically on the installation host in this way, digital signing of the tag is not possible. As a result, the integrity of the tag cannot be verified based on a digital signature unless an equivalent tag is also produced using the first method described above. Absent a verified tag signature, processes on the device and downstream consumers of the information

within a SWID tag may have less assurance of the tag's authenticity and integrity, limiting the use of the SWID tag information in usage scenarios that require a high degree of assurance.

2.3.3 Deployment of SWID Tags in a Repository

A third method of tag deployment is to store SWID tags in accessible repositories. By retrieving specific tags from an appropriate repository, software consumers can do the following:

- Confirm that a tag discovered on a device has not been modified; this can be done by comparing the tag on the device with the corresponding tag found in the repository.
- Restore a tag that has been inadvertently deleted.
- Repair a tag that has been improperly modified.
- Utilize the information in the tag to support various software-related management and analysis processes.

2.4 Summary

The SWID specification defines four different types of tags to support various portions of the software lifecycle, including software deployment, installation, patching, upgrading, and removal.

- Corpus tags provide information about a software distribution, such as information about the files that it includes. This information may be used to verify the integrity and/or authenticity of the software distribution during software deployment.
- Primary tags provide specific information (e.g., product naming information, the software creator, lists of files expected to be included) regarding a software product that has been installed or patched.
- Patch tags provide specific information about a software patch that has been provided to correct errors in, or add new features to, a product. A patch tag supplies information about the changes that a patch makes to the patched software product's installation footprint including files added, removed, or changed.
- Supplemental tags provide a mechanism to describe additional information related to other tags.

Any entity may create a supplemental tag for any purpose. For example, supplemental tags may be created by automated tools in order to augment an existing primary tag with additional site-specific information, such as license keys and contact information for local responsible parties.

By deploying tags in consistent locations, inventory processes and automated tools are able to reliably and accurately determine and maintain software inventory.

Tags may also be deployed in accessible repositories to make tag information available for use by other management and cybersecurity processes.

3 SWID Tag Structure

This section reviews the structure of SWID tags. Section 3.1 presents an overview of the basic tag data elements that are used to identify and describe software products. Section 3.2 discusses how SWID tags may be authenticated. Section 3.3 presents an example of the primary tag type, and Section 3.4 concludes with a summary of key points from this section.

To correctly implement tags, interested readers may obtain the SWID specification and the corresponding XML schema definition (XSD). When used with a validating XML parser, the XSD can be used to check that a SWID tag is conformant with the SWID specification. An up-to-date and extensively documented XSD may be downloaded at no cost from:

http://standards.iso.org/iso/19770/-2/2015-current/schema.xsd

3.1 SWID Tag Data Elements

This section discusses the basic data elements of a SWID tag, and explains how the four tag types described in Section 2.1 are distinguished from each other.

A SWID tag (whether corpus, primary, patch, or supplemental) is represented as an XML root element with several sub-elements. `<SoftwareIdentity>` is the root element, and it is described in Section 3.1.1. The following sub-elements are used to express distinct categories of product information: `<Entity>` (see §3.1.2), `<Evidence>` (see §3.1.3), `<Link>` (see §3.1.4), `<Meta>` (see §3.1.5), and `<Payload>` (see §3.1.6).

3.1.1 <SoftwareIdentity>: The Root of a SWID Tag

Besides serving as the container for all the sub-elements described in later subsections, the `<SoftwareIdentity>` element provides attributes to record the following descriptive properties of a software product:

- `@name`: the string name of the software product or component as it would normally be referenced, e.g., “ACME Roadrunner Management Suite”. A value for `@name` is **required** according to the SWID specification.
- `@version`: the detailed version of the product, e.g., “4.1.5”. In the SWID specification, a value for `@version` is **optional** and defaults to “0.0”. Sections 5.1.2 and 5.2.2 of this report provide guidelines that **require** a value for `@version` in corpus and primary tags.
- `@versionScheme`: a label describing how version information is encoded, e.g., “`semver`”. In the SWID specification, a value for `@versionScheme` is **optional** and defaults to “`multipartnumeric`”. Sections 5.1.2 and 5.2.2 of this report provide guidelines that **require** a value for `@versionScheme` in corpus and primary tags.
- `@tagId`: a globally unique identifier that may be used as a proxy identifier in other contexts to refer to the tagged product. A value for `@tagId` is **required** according to the SWID specification.

- `@tagVersion`: an integer that allows one tag for a software product to supersede another without suggesting any change to the underlying software product being described. Typical reasons for an increment in `@tagVersion` include correcting erroneous information in an older version of a tag, or adding new information that was not included in a previously deployed tag. A value for `@tagVersion` is **optional** and defaults to "`0`" according to the SWID specification.

  Under normal conditions, it would be unexpected to discover multiple tags in the same location on a device that all identify the same installed product by having the same `@tagId` attribute value, but different `@tagVersion` attribute values. Such a situation likely reflects a failure to properly maintain the device's inventory of SWID tags. Nevertheless, should such a situation be encountered, the tag with the highest `@tagVersion` is considered to be the most up-to-date tag, and the others may be ignored. When considering tags in this situation, it is important to verify tag signatures, if available, to ensure that the most up-to-date tag being considered contains a valid XML signature (see §3.2). Furthermore, it is important that this signature contains a valid certificate to avoid using a tag that might have been produced by an unauthorized party. For example, if a tag with a `@tagVersion` value of *x* is found to contain an invalid signature, a safer choice would be a valid tag with a `@tagVersion` value that is lower than but as close to *x* as possible (e.g., *x*-1), and that also has a valid signature.

- `@supplemental`: a boolean value that, if set to "`true`", indicates that the tag type is *supplemental* (see §2.1.4). A value for `@supplemental` is **optional** and defaults to "`false`" according to the SWID specification.

- `@patch`: a boolean value that, if set to "`true`", indicates that the tag type is *patch* (see §2.1.3). A value for `@patch` is **optional** and defaults to "`false`" according to the SWID specification.

- `@corpus`: a boolean value that, if set to "`true`", indicates that the tag type is *corpus* (see §2.1.1). A value for `@corpus` is **optional** and defaults to "`false`" according to the SWID specification.

Table 1 illustrates how the tag type may be determined by inspecting the values of `@corpus`, `@patch`, and `@supplemental`. If all these values are false, the tag type is *primary*. This report provides guidelines requiring that, at most, one of `@corpus`, `@patch`, or `@supplemental` be set to true (see §§ 5.1.1, 5.2.1, 5.3.1, and 5.4.1). The SWID specification requires that patch tags explicitly link to the primary tag(s) of the product(s) that are patched, using the "`patches`" relation in the `<Link>` `@rel` attribute. Section 5.4.2 of this document provides a guideline requiring supplemental tags to explicitly link to the supplemented tag(s) using the "`supplemental`" relation in the `<Link>` `@rel` attribute. Although they may link to other tags, there are no requirements that corpus or primary tags do so.

Table 1: How Tag Types Are Indicated

Tag Type	@supplemental	@patch	@corpus	<Link> required @rel
Corpus	false	false	true	N/A
Primary	false	false	false	N/A
Patch	false	true	false	patches
Supplemental	true	false	false	supplemental

3.1.1.1 Example 1—Primary Product Tag

This example illustrates a primary tag for version 4.1.5 of a product named "ACME Roadrunner Management Suite Coyote Edition." The globally unique tag identifier, or @tagId, is "com.acme.rms-ce-v4-1-5-0". The <Entity> element (see §3.1.2) is included so the example illustrates all data values required in a minimal tag that conforms to the SWID specification. Any additional identifying data (not shown) would appear in place of the ellipsis.

```
<SoftwareIdentity
  xmlns="http://standards.iso.org/iso/19770/-2/2015/schema.xsd"
  name="ACME Roadrunner Management Suite Coyote Edition"
  tagId="com.acme.rms-ce-v4-1-5-0"
  tagVersion="0"
  version="4.1.5">
  <Entity
    name="The ACME Corporation"
    regid="acme.com"
    role="tagCreator softwareCreator"/>
 …
</SoftwareIdentity>
```

3.1.1.2 Example 2—Supplemental Tag

This example illustrates a supplemental tag for an installed product. The globally unique identifier of the supplemental tag is "com.acme.rms-sensor-1". The <Entity> element (see §3.1.2) is included so the example illustrates all data values required in a minimal tag that conforms to the standard. The <Link> element (see §3.1.4) is included to illustrate how a supplemental tag may be associated with the primary tag shown above in Section 3.1.1.1. This supplemental tag may supply additional installation details that are not included in the product's primary tag (e.g., site-specific information such as contact information for the information steward.) These details would appear in place of the ellipsis.

```
<SoftwareIdentity
  xmlns="http://standards.iso.org/iso/19770/-2/2015/schema.xsd"
  name="ACME Roadrunner Management Suite Coyote Edition"
  tagId="com.acme.rms-sensor-1"
  supplemental="true">
  <Entity
```

```
    name="The ACME Corporation"
    regid="acme.com"
    role="tagCreator softwareCreator"/>
  <Link
    rel="related"
    href="swid:com.acme.rms-ce-v4-1-5-0">
 …
</SoftwareIdentity>
```

3.1.1.3 Example 3—Patch Tag

This example illustrates a patch tag for a previously installed product. The name of the patch is "ACME Roadrunner Service Pack 1", and its globally unique tag identifier is "`com.acme.rms-ce-sp1-v1-0-0`". `<Entity>` and `<Link>` elements are illustrated as before. Any additional identifying data (not shown) would appear in place of the ellipsis.

```
<SoftwareIdentity
  xmlns="http://standards.iso.org/iso/19770/-2/2015/schema.xsd"
  name="ACME Roadrunner Service Pack 1"
  tagId="com.acme.rms-ce-sp1-v1-0-0"
  patch="true"
  version="1.0.0">
  <Entity
    name="The ACME Corporation"
    regid="acme.com"
    role="tagCreator softwareCreator"/>
  <Link
    rel="patches"
    href="swid:com.acme.rms-ce-v4-1-5-0">
 …
</SoftwareIdentity>
```

3.1.2 <SoftwareIdentity> Sub-Element: <Entity>

Every SWID tag identifies, at minimum, the organization or individual that created the tag. Entities having other roles associated with the identified software product, such as its creator, licensor(s), or distributor(s), may optionally be identified using `<Entity>` elements contained within the `<SoftwareIdentity>` element. Each `<Entity>` element provides the following attributes:

- `@name`: the string name of the entity, e.g., "The ACME Corporation". A value for `@name` is **required** according to the SWID specification**.**
- `@regid`: the "registration identifier" of the entity. A value for `@regid` is **required** when the `<Entity>` element is used to identify the tag creator (e.g., `@role`="`tagCreator`"), otherwise `@regid` is **optional** and defaults to "`invalid.unavailable`" according to the SWID specification. Additional information about creation of `@regid` values is provided below.

- `@role`: the role of the entity with respect to the tag and/or the product identified by the tag. Every `<Entity>` element contains a value for `@role`, and additionally, every tag contains an `<Entity>` element identifying the tag creator. The `@role` attribute can list multiple roles with a space separating each role. Values for `@role` can be selected from an extensible set of allowed tokens, including:
    - **`aggregator:`** An organization or system that encapsulates software from its own and/or other organizations into a different distribution process (as in the case of virtualization), or as a completed bundle to accomplish a specific task (as in the case of a value-added reseller)
    - **`distributor:`** An entity that furthers the marketing, selling, and/or distribution of software from the original place of manufacture to the ultimate user without modifying the software, its packaging, or its labeling
    - **`licensor:`** A person or organization that owns or holds the rights to issue a software license for a specific software package
    - **`softwareCreator:`** A person or organization that creates a software product
    - **`tagCreator:`** The entity that creates a given SWID tag
- `@thumbprint`: a hexadecimal string containing a hash of the entity's signing certificate for SWID tags which are digitally signed (see §3.2). This allows the digital signature to be directly related to the entity specified.

Values for `@regid` are uniform resource identifiers (URIs) as described in RFC 3986 [RFC 3986]. Values for `@regid` are not required to be dereferenceable on the Internet. To ensure interoperability and to allow for open source project support, Section 6.1.5.2 of the SWID specification recommends that tag creators do the following when creating a value for `@regid`:

- Unless otherwise required, the URI should utilize the `http` scheme.
- If the `http` scheme is used, the "`http://`" may be left off the regid string (a string without a URI scheme specified is defined to use the "`http://`" scheme.)
- Unless otherwise required, the URI should use an absolute URI that includes an authority part, such as a domain name.
- To ensure consistency, the absolute URI should use the minimum string required (for example, `example.com` should be used instead of `www.example.com`).

For tag creators that do not have a domain name, the `mailto` scheme can be used in place of the `http` scheme to identify the tag creator by email address, e.g., `mailto:foo@bar.com`.

The example below illustrates a SWID tag containing two `<Entity>` elements. The first `<Entity>` element identifies the single organization that is both the software creator and the tag creator, and the second element identifies the organization that is the software's distributor:

```
<SoftwareIdentity …>
  …
```

```
  <Entity
    name="The ACME Corporation"
    regid="acme.com"
    role="tagCreator softwareCreator"/>
  <Entity
    name="Coyote Services, Inc."
    regid="mycoyote.com"
    role="distributor"/>
  …
</SoftwareIdentity>
```

3.1.3 <SoftwareIdentity> Sub-Element: <Evidence>

Not every software product installed on a device will be supplied with a tag. When a tag is not found for an installed product, third-party software inventory and discovery tools will continue to be used to discover untagged products residing on devices. In these situations, the inventory or discovery tool may generate a primary tag on the fly to record the newly discovered product. The optional `<Evidence>` element may then be used to store results from the scan that explain why the product is believed to be installed. To that end, the `<Evidence>` element provides two attributes and four sub-elements, all of which are optional according to the SWID specification:

- `@date`: the date the evidence was collected.
- `@deviceId`: the identifier of the device from which the evidence was collected.
- `<Directory>`: filesystem root and directory information for discovered files. If no absolute directory is provided, the directory is considered to be relative to the directory location of the SWID tag.
- `<File>`: files discovered and believed to be part of the product. If no absolute directory path is provided, the file location is assumed to be relative to the location of the SWID tag. If a parent `<Directory>` includes a nested `<File>`, the indicated file is relative to the parent location.
- `<Process>`: related processes discovered on the device.
- `<Resource>`: other general information that may be included as part of the product.

Note that `<Evidence>` is represented in a SWID tag in the same manner as `<Payload>` (see §3.1.6). There is a key difference, however, between `<Evidence>` and `<Payload>` data. The `<Evidence>` element is used by discovery tools that identify untagged software. Here, the discovery tool creates a SWID tag based on data discovered on a device. In this case, the `<Evidence>` element indicates only what was discovered on the device, but this data cannot be used to determine whether discovered files match what a software provider originally released or what was originally installed. In contrast, `<Payload>` data supplies information from an authoritative source (typically the software provider or a delegate), and thus may be used, for example, to determine if files in a directory match the files that were designated as being installed with a software component or software product.

The example below illustrates a SWID tag containing an <Evidence> element. The evidence consists of one file discovered in a folder named "rrdetector" within the device's standard program data area:

```
<SoftwareIdentity …>
  …
  <Evidence date="11-28-2014" deviceId="mm123-pc.acme.com">
    <Directory location=".." name="rrdetector">
      <File name="rrdetector.exe" size="532712"
            SHA256:hash="a314fc2dc663ae7a6b6bc6787594057396e
6b3f569cd50fd5ddb4d1bbafd2b6a"/>
    </Directory>
  </Evidence>
  …
</SoftwareIdentity>
```

In cases where the evidence is collected from a shared location (e.g., a Network Attached Storage [NAS] device), the provided @deviceId could reference that shared location, rather than the device where the discovery occurs. Using this approach helps to prevent a situation where software installed in a shared location is tagged multiple times with conflicting @deviceId values.

3.1.4 <SoftwareIdentity> Sub-Element: <Link>

Modeled on the [LINK] element from the Hypertext Markup Language (HTML), SWID tag <Link> elements are used to record a variety of relationships between tags and other items. A typical use of the <Link> element is to document a relation that exists between a product or patch described by a *source tag* (the tag containing the <Link> element) and a product or patch described by a *target tag* (the tag to which the <Link> element points). <Link> elements may also be used to associate a source tag with other arbitrary information elements.

A <Link> element is often used to associate a patch tag or supplemental tag to a primary tag (see §§2.1.3 and 2.1.4). Other uses include pointing to documents containing applicable licenses, vendor support pages, and installation media. The <Link> element has a number of attributes, two of which are required, as follows:

- @href: the value is a URI pointing to the item to be referenced. It can point to several different values, including:
  - a relative URI
  - a physical file location with any system-acceptable URI scheme (e.g., file://, http://, https://, ftp://)
  - a URI with "swid:…" as the scheme, which refers to another SWID tag by @tagId
  - a URI with "swidpath:..." as the scheme, which contains an XPATH query [XPATH 2.0]. This XPATH would need to be resolved in the context of the system

by software that can look up other SWID tags and select the appropriate tag based on the query.

- `@rel`: the value specifies the type of relationship between the SWID tag and the item referenced by `@href`.

Table 2 lists some of the pre-defined values of the `@rel` attribute used this report. A more complete list of values is presented in Section 8.6.7 of the SWID specification, and also in the schema. Note that this list may be extended to support future needs.

Table 2: <Link> Relations

Relation	Meaning
component	Defines a link to a component of the product. A component could be an independently functioning application that is part of a product suite or bundle, as well as a shared library, language pack, etc.
patches	Defines a link to the product to which the patch was applied.
requires	Defines a link to a required patch, or to any other software product that is required in order for the product described by the source tag to function properly.
supersedes	Defines a link to a superseded patch.
supplemental	Defines a link to a supplemental tag.
<any>	Additional relationships can be specified by referencing the Internet Assigned Numbers Authority (IANA) Link Relations registration library.[1]

The example below illustrates how a `<Link>` element may be used to associate a patch tag with the tag for the patched product:

```
<SoftwareIdentity
  …
  name="ACME Roadrunner Service Pack 1"
  tagId="com.acme.rms-ce-sp1-v1-0-0"
  patch="true"
  version="1.0.0">
  …
  <Link
    rel="patches"
    href="swid:com.acme.rms-ce-v4-1-5-0"/>
  …
</SoftwareIdentity>
```

The patch in this example is linked to the patched product's tag using that product's `@tagId`.

[1] See http://www.iana.org/assignments/link-relations/link-relations.xhtml for the current list of defined link relations.

3.1.5 <SoftwareIdentity> Sub-Element: <Meta>

<Meta> elements are used to record an array of optional metadata attributes related to the tag or the product. Several <Meta> attributes of interest are:

- @activationStatus: identifies the activation status of the product. The SWID specification provides several example values (e.g., "Trial", "Serialized", "Licensed", and "Unlicensed"), but any string value may be supplied. Valid values for @activationStatus are expected to be worked out over time by tag implementers.
- @colloquialVersion: the informal version of the product (e.g., 2013). The colloquial version may be the same through multiple releases of a software product where the @version specified in <SoftwareIdentity> is much more specific and will change for each software release.
- @edition: the variation of the product (e.g., "Home", "Enterprise", "Student").
- @product: the base name of the product, exclusive of vendor, colloquial version, edition, etc.
- @revision: the informal or colloquial representation of the sub-version of the product (e.g., "SP1", "R2", "Beta 2"). Whereas the <SoftwareIdentity> element's @version attribute will provide exact version details, the @revision attribute is intended for use in environments where reporting on the informal or colloquial representation of the software is important. For example, if, for a certain business process, an organization decides that it requires Service Pack 1 or later of a specific product installed on all devices, the organization can use the revision data value to quickly identify any devices that do not meet this requirement.

In the example below, a <Meta> element is used to record the fact that the product is installed on a trial basis, and to break out the full product name into its component parts:

```
<SoftwareIdentity …>
  …
  name="ACME Roadrunner Detector 2013 Coyote Edition SP1"
  tagId="com.acme.rd2013-ce-sp1-v4-1-5-0"
  version="4.1.5">
  …
  <Meta
    activationStatus="trial"
    product="Roadrunner Detector"
    colloquialVersion="2013"
    edition="coyote"
    revision="sp1"/>
  …
</SoftwareIdentity>
```

3.1.6 <SoftwareIdentity> Sub-Element: <Payload>

The optional <Payload> element is used to enumerate the items (e.g., files, folders, license keys) that may be installed on a device during software product installation. In general, <Payload> lists the files that may be installed with a software product, and could be a superset of those files (e.g., because some optional files might not be installed on the device).

The <Payload> element is a container for <Directory>, <File>, <Process>, and/or <Resource> elements, similar to the <Evidence> element (see §3.1.3). When the <Payload> element is used, information contained in the element is considered to be authoritative information about the software. This differs from the use of the <Evidence> element, which is used to store results from a scan that indicate why the product is believed to be installed. The following example illustrates a primary tag with a <Payload> element describing one file in a single directory:

```
<SoftwareIdentity …>
  …
  <Payload>
    <Directory root="%programdata%" name="rrdetector">
       <File name="EPV12.cab" size="1024000"
             SHA256:hash="a314fc2dc663ae7a6b6bc6787594057396e
6b3f569cd50fd5ddb4d1bbafd2b6a"/>
    </Directory>
  </Payload>
  …
</SoftwareIdentity>
```

3.2 Authenticating SWID Tags

Because SWID tags are XML documents discoverable on a device (see §2.3.1, §2.3.2) or by retrieving a tag from a repository (see §2.3.3), they are vulnerable to unauthorized or inadvertent modification like any other document. To recognize such tag modifications, it is necessary to validate that a SWID tag was produced by a known, trusted entity and has not been altered after creation. XML digital signatures embedded within a SWID tag can be used to prove the authenticity of the tag signer and to validate that changes have not been made to the original signed tag.

Applying an XML digital signature to a tag directly after it is created addresses the following risks:

- **Creation of a tag by an unauthorized creator.** Any entity can create a tag for a software product. By including a certificate acquired from a trusted certificate authority in the tag's digital signature, it is possible to validate the certificate, ensure that it has not been revoked, and determine that the signature was produced while the certificate was valid. By linking the signature to the <Entity> element having a @role value of "tagCreator" and using the @thumbprint attribute (see §3.1.2) to record the thumbprint of the certificate, it is possible to identify that the tag creator is also the tag signer. This provides a method for source authentication of a tag's creator.

- **Detecting unauthorized changes to a tag.** Changes made to a tag after it is created can be detected by validating the tag's signature. Such changes may occur due to accidental modification, incomplete copying, network errors, filesystem corruption, or an attacker wishing to misrepresent the software installation state or the information contained within a tag. The use of an XML digital signature in this way provides a means to validate the integrity of a tag. When coupled with authenticating the creator of a tag, this greatly increases the assurance of the tag data.
- **Detecting unauthorized changes to software installation media and packages.** If the creator of a corpus tag can be authenticated and the integrity of the tag can be verified, it is then possible to use the information in the tag's `<Payload>` element, if provided, to measure the integrity of software installation media or packages. This can be useful for detecting unauthorized changes to software installation media and packages, and can inform policy decisions pertaining to authorizing software installations (see §6.2.1).
- **Detecting unauthorized changes to installed software.** If the creators of primary and patch tags related to an installed software product can be authenticated and the integrity of the tags can be verified, it is then possible to use the information in each tag's `<Payload>` element, if provided, to measure the integrity of the related installed software product. This can be useful for detecting unauthorized changes to installed software and can inform policy decisions pertaining to authorizing software execution (see §6.2.2).

When considering the totality of these risks, SWID tags can enhance the assurance of software before and after it is installed in a standardized way, regardless of the target platform or software product installed.

Section 6.1.10 of the SWID specification states that:

> Signatures are not a mandatory part of the software identification tag standard, and can be used as required by any tag producer to ensure that sections of a tag are not modified and/or to provide authentication of the signer. If signatures are included in the software identification tag, they shall follow the W3C recommendation defining the XML signature syntax which provides message integrity authentication as well as signer authentication services for data of any type.

Digital signatures use the <`Signature`> element as described in the World Wide Web Consortium (W3C) XML Signature Syntax and Processing (Second Edition) specification [xmldsig-core] and the associated schema.[2] Users may also include a hexadecimal hash string (i.e., the "thumbprint") to document the relationship between the tag entity and the signature, using the <`Entity`> `@thumbprint` attribute.

Section 6.1.10 of the SWID specification also requires that a digitally-signed SWID tag enable tag consumers to:

---

[2] See http://www.w3.org/TR/xmldsig-core/#sec-Schema.

> Utilize the data encapsulated by the SWID tag to ensure that the digital signature was validated by a trusted certificate authority (CA), that the SWID tag was signed during the validity period for that signature, and that no signed data in the SWID tag has been modified. All of these validations shall be able to be accomplished without requiring access to an external network. If a SWID tag consumer needs to validate that the digital certificate has not been revoked, then it is expected that there be access to an external network or a data source that can provide [access to the necessary] revocation information.

Additional information on digital signatures, how they work, and the minimum requirements for digital signatures used for U.S. Federal Government processing can be found in the Federal Information Processing Standards (FIPS) Publication 186-4, *Digital Signature Standard (DSS)* [FIPS-186-4].

3.3 A Complete Primary Tag Example

A complete tag is illustrated below, combining examples from the preceding subsections. This example illustrates a primary tag that contains all mandatory data elements as well as a number of optional data elements. This example does not illustrate the use of digital signatures.

```
<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"/>
  <Entity
    name="Coyote Services, Inc."
    regid="mycoyote.com"
    role="distributor"/>
  <Link
    rel="license"
    href="www.gnu.org/licenses/gpl.txt"/>
  <Meta
    activationStatus="trial"
    product="Roadrunner Detector"
    colloquialVersion="2013"
    edition="coyote"
    revision="sp1"/>
  <Payload>
    <Directory root="%programdata%" name="rrdetector">
      <File name="rrdetector.exe" size="532712"
         SHA256:hash="a314fc2dc663ae7a6b6bc6787594057396e6b3f569c
d50fd5ddb4d1bbafd2b6a"/>
    </Directory>
```

```
  </Payload>
</SoftwareIdentity>
```

3.4 Summary

SWID tags are rich sources of information useful for identifying and describing software products, whether in a pre-installation state or installed on devices. A relatively small number of elements and attributes is required in order for a tag to be considered valid and conforming to the specification. The SWID specification defines other optional data elements and attributes that support a wide range of usage scenarios.

A minimal valid and conforming tag uses a `<SoftwareIdentity>` element to record a product's name and the tag's globally unique identifier, and contains an `<Entity>` element to record the name and registration identifier of the tag creator. While such a minimal tag is better than no tag at all in terms of enhancing the ability of SAM tools to discover and account for installed products, it falls short of satisfying many higher-level business and cybersecurity needs. To meet those needs, the SWID specification offers several additional elements, such as `<Evidence>` for use by scanning tools to record results of the discovery process; `<Link>` to associate tags with other items, including other tags; `<Meta>` to record a variety of metadata values; and `<Payload>` to enumerate files, etc., that comprise the installed product. Finally, digital signatures may optionally be used by any tag producer to ensure that the contents of a tag are not accidentally or deliberately modified after installation, and to provide authentication of the signer.

4 Implementation Guidance for All Tag Creators

Sections 4 and 5 provide implementation guidance for creators of SWID tags that is broadly applicable to common IT usage scenarios of interest to both public and private sector organizations. The primary purpose of this guidance is to help tag creators understand how to implement SWID tags in a consistent manner that will address common cybersecurity and operational IT usage scenarios, such as those defined in Section 6.

Each guideline in the next two sections is prefixed with a coded identifier for ease of reference. Such identifiers have the following format: *CAT-NUM*, where *CAT* is a three-letter symbol indicating the guideline category, and *NUM* is a number. Guidelines are grouped into the following categories:

- **GEN:** General guidelines applicable to all types of SWID tags (see §4.1 through §4.7),
- **COR:** Guidelines specific to corpus tags (see §5.1),
- **PRI:** Guidelines specific to primary tags (see §5.2),
- **PAT:** Guidelines specific to patch tags (see §5.3), and
- **SUP:** Guidelines specific to supplemental tags (see §5.4).

This section provides implementation guidelines that address issues common to all situations in which tags are deployed and processed. Section 5 provides guidelines that vary according to the type of tag being implemented.

4.1 Limits on Scope of Guidelines

This report assumes that tag creators are familiar with the SWID specification and ensure that implemented tags satisfy all requirements contained therein.

> **GEN-1.** When producing SWID tags, tag creators MUST produce SWID tags that conform to all requirements defined in the ISO/IEC 19770-2:2015 specification.

Guideline **GEN-1** establishes a baseline of interoperability that is needed by all adopters of SWID tags.

All guidelines in this report are intended solely to extend, and not to conflict with, guidelines provided by the SWID specification. Guidelines in this report either:

- Strengthen existing guidelines contained in the SWID specification by elevating "SHOULD" clauses contained in the SWID specification to "MUST" clauses, or
- Add guidelines to address implementation issues where the SWID specification is silent or ambiguous by adding new "SHOULD" or "MUST" clauses.

In no cases should this report's guidelines be construed as either weakening or eliminating existing guidelines in the SWID specification.

4.2 Authoritative and Non-Authoritative Tag Creators

SWID tags may be created by individuals, organizations, or automated tools under different conditions. The entity that creates a tag, as well as the conditions under which a tag is created, profoundly affect how a tag can be used. The quality, accuracy, completeness, and trustworthiness of the data in a tag will determine its suitability for specific usage scenarios.

Tags may be created by authoritative or non-authoritative entities. For the purposes of this report, an *authoritative tag creator* is a first- or second-party that creates a tag as part of the process of releasing software. A first-party authoritative tag creator is the software creator. A second-party authoritative tag creator aggregates, distributes, or licenses software on behalf of the software creator. Such parties typically possess accurate, complete, and detailed technical knowledge that is needed for creation of authoritative tags containing reliable information.

A *non-authoritative tag creator* is defined as an entity that has a third-party relation to the creation, maintenance, and distribution of the software. Non-authoritative tag creators typically create tags using product information that is gathered using forensic methods while discovering installed software on devices (e.g., through use of a discovery tool).

As a shorthand, this report uses the term “authoritative tag” to refer to tags created by authoritative entities, and “non-authoritative tag” to refer to tags created by non-authoritative entities. Unless otherwise specified, guidelines in this report are directed equally at both authoritative and non-authoritative tag creators. Guidelines prefixed with “[Auth]” are directed specifically at authoritative tag creators, and guidelines prefixed with “[Non-Auth]” are directed specifically at non-authoritative tag creators.

4.3 Implementing <SoftwareIdentity> Elements

This section provides guidelines to be observed by tag creators when implementing SWID tag `<SoftwareIdentity>` elements.

The SWID specification defines an international standard intended to be adopted and used worldwide. To support an international audience, it is necessary to permit tag creators to provide language-dependent attribute values in region-specific human languages. For example, a Japanese software provider may want to specify the value of a particular product’s `<SoftwareIdentity>` `@name` attribute as a string of Japanese characters.

The SWID tag XML schema provides multi-language support in two ways. First, the schema specifies UTF-8 as the allowed character encoding scheme for SWID tag files. Second, the schema allows the optional `@xml:lang` attribute to be included on all tag elements. By taking advantage of these features, a Japanese software provider could issue a SWID tag like this:

```
<SoftwareIdentity
  xmlns="http://standards.iso.org/iso/19770/-2/2015/schema.xsd"
  xml:lang="ja-jp"
  name="コンピュータ管理システム 2015"
  tagId="jp.largecomputerco.タグ番号 1"
  version="1.0">
```

```
  <Entity
    name="大型コンピュータ会社"
    regid="largecomputerco.jp"
    role="tagCreator softwareCreator"/>
  …
</SoftwareIdentity>
```

According to W3C documentation, *language tags* "are used to indicate the language of text or other items in HTML and XML documents" [W3C-langtags]. By supplying the language tag "`ja-jp`" as the value of the `<SoftwareIdentity>` `@xml:lang` attribute, the tag creator signals to tag consumers that various language-dependent attributes, such as the `<SoftwareIdentity>` `@name` attribute, are provided in Japanese. Additionally, the SWID schema is designed so that any value of `@xml:lang` specified on any tag element is inherited by all of that element's child elements, unless explicitly overridden. As a result, the value specified for the `<SoftwareIdentity>` `@xml:lang` attribute will, in effect, establish the *preferred language* that is used for all language-dependent attributes within the tag.

Knowing the preferred language of SWID tag attribute values can be very useful to tag consumers, and can relieve them of the need to attempt to perform language auto-detection. The following guideline requires that tag creators always specify the preferred language of a tag:

> **GEN-2.** The `<SoftwareIdentity>` element MUST specify an `@xml:lang` attribute with a non-blank value to indicate the default human language used for expressing all language-dependent attribute values.

Guideline **GEN-2** is directed towards both authoritative and non-authoritative tag creators. While authoritative tag creators can always be expected to know the default language of the tag, non-authoritative creators may need to use local knowledge to ascertain the most appropriate default language. This document recommends that non-authoritative tag creators supply the default language of the device where the tagged product resides.

> **GEN-3.** [Non-Auth] When specifying a value for the `<SoftwareIdentity>` `@xml:lang` attribute, non-authoritative tag creators SHOULD use the language tag corresponding to the default language of the device where the tagged product resides.

In some cases, tag creators may want to specify various SWID tag attribute values in more than one human language. This presents a number of challenges and potential interoperability issues; these are discussed further in Section 4.7.

4.4 Implementing <Entity> Elements

This section provides guidelines to be observed by tag creators when implementing SWID tag `<Entity>` elements. The guidelines in this section address four issues:

1. Providing detailed information about entities (see §4.4.1),
2. Preventing unnecessarily complex entity specifications (see §4.4.2),
3. Distinguishing between authoritative and non-authoritative tags (see §4.4.3), and
4. Furnishing information about the software creator (see §4.4.4).

4.4.1 Providing Detailed Information about Entities

The first issue to be addressed concerns the need for detailed information about the various entities associated with tags and/or tagged products.

Section 8.2 of the SWID specification requires that every SWID tag contain an `<Entity>` element where the `@role` attribute has the value "`tagCreator`", and the `@name` attribute is also provided. Although `<Entity>` elements may furnish `@regid` attribute values, the specification does not require this information to be provided. Instead, the specification defaults the `@regid` attribute value to "`http://invalid.unavailable`" when a value is not explicitly provided in the element. Because the `@regid` attribute serves as a unique reference to an organization, this report provides guidelines to ensure that explicit values are provided whenever possible.

The ability to provide `@regid` attribute values varies, depending on whether a tag creator is authoritative or non-authoritative. For authoritative tag creators, it is reasonable to require the `@regid` value on all `<Entity>` elements:

> **GEN-4.** [Auth] Every `<Entity>` element MUST provide an explicit (i.e., non-default) `@regid` attribute value.

Non-authoritative tag creators may be less able to provide reliable `@regid` information. While they are expected to provide a `@regid` value for the `<Entity>` element with the "`tagCreator`" `@role` that identifies their organization, they can only be encouraged to provide `@regid` information for entities in other roles. This leads to the following two guidelines:

> **GEN-5.** [Non-Auth] The `<Entity>` element containing the `@role` "`tagCreator`" MUST provide an explicit (i.e., non-default) `@regid` attribute value.

> **GEN-6.** [Non-Auth] Every `<Entity>` element SHOULD provide an explicit (i.e., non-default) `@regid` attribute value if such a value can be determined.

4.4.2 Preventing Complex Entity Specifications

The second issue to be addressed concerns the potential for unnecessarily complex specifications of entities and roles associated with the tag and/or the tagged product. The SWID specification allows a tag to furnish multiple `<Entity>` elements in order to support situations in which different organizations play different roles with respect to the tag and/or the tagged product. So if one organization were the tag creator, and a second organization were the software creator, this information could be specified as follows:

```
<Entity
  name="Organization 1 Corp"
  role="tagCreator"/>
<Entity
  name="Organization 2 Corp"
  role="softwareCreator"/>
```

This ability to furnish multiple <Entity> elements has an undesirable side effect, however. It now becomes possible for role information associated with a single organization to be spread across multiple <Entity> elements, as illustrated by this example:

```
<Entity
  name="Organization 1 Corp"
  role="tagCreator"/>
<Entity
  name="Organization 1 Corp"
  role="softwareCreator"/>
```

Such spreading of role information across <Entity> elements is discouraged. Furnishing entity information in this way increases the size of the tag unnecessarily, and creates additional processing complexity for tag consumers. To preclude this, the following guideline is provided:

> **GEN-7.** All <Entity> elements that provide the same @regid attribute value MUST provide the same @role attribute values.

Guideline **GEN-7** works in concert with guidelines **GEN-4** through **GEN-6** to achieve the desired effect. It should be clear that the following example satisfies these guidelines:

```
<Entity
  name="Organization 1 Corp"
  regid="org1.com"
  role="tagCreator softwareCreator"/>
<Entity
  name="Organization 2 Corp"
  regid="org2.com"
  role="licensor"/>
```

As will be seen later in Section 4.7, guidelines **GEN-4** through **GEN-7** also play an important role in addressing potential interoperability issues that could arise when tag creators specify attribute values in multiple human languages.

4.4.3 Distinguishing Between Authoritative and Non-Authoritative Tags

The third issue to be addressed here concerns the process by which a tag consumer may rapidly determine whether the tag creator is authoritative or non-authoritative.

When a tag contains an <Entity> element that specifies only a single @role of "tagCreator", tag consumers can safely assume that the tag creator is non-authoritative. To enable tag consumers to accurately determine that a tag is created by an authoritative source, authoritative tag creators are required to include one or more additional predefined role values, as follows:

> **GEN-8.** [Auth] Authoritative tag creators MUST provide an <Entity> element where the @role attribute contains the value "tagCreator" and at least one of these additional role values: "aggregator", "distributor", "licensor", or "softwareCreator".

If this guideline is observed, tag consumers may reliably distinguish between authoritative and non-authoritative tags according to this rule:

> If a tag contains an `<Entity>` element with a `@role` value that includes "`tagCreator`" as well as any of "`aggregator`", "`distributor`", "`licensor`", or "`softwareCreator`", then the tag is authoritative; otherwise, it is non-authoritative.

4.4.4 Furnishing Information about the Software Creator

The fourth issue to be addressed here concerns the furnishing of information about the software creator, if that information is known.

Explicit knowledge of the software creator is important for many software inventory scenarios, but the SWID specification does not require that this information be provided. To support software inventory scenarios, authoritative tag creators are expected to furnish information on the software creator's identity. If the tag creator is not the same as the software creator, authoritative tag creators are expected to know the appropriate `@name` and `@regid` attribute values for the software creator.

> **GEN-9.** [Auth] Authoritative tag creators MUST provide an `<Entity>` element where the `@role` attribute contains the value "`softwareCreator`".

Non-authoritative tag creators may be unable to accurately determine and identify the various entities associated with a software product, including the software creator. Nevertheless, because tag consumers may obtain substantial benefits from knowing a product's software creator, non-authoritative tag creators are encouraged to include this information in a tag whenever possible.

> **GEN-10.** [Non-Auth] Non-authoritative tag creators SHOULD provide an `<Entity>` element where the `@role` attribute contains the value "`softwareCreator`".

4.5 Implementing <Link> Elements

This section provides guidelines to be observed by tag creators when implementing SWID tag `<Link>` elements. As discussed in Section 3.1.4, `<Link>` elements are used to establish one or more relationships of various kinds between tags and other documents, including other tags. The guidelines in this section address two issues:

1. Linking a source tag to a known target tag, and
2. Linking a tag to a collection of tags.

A tag may contain multiple `<Link>` elements, and either method of linking may be used in each element. These methods are described in the following subsections.

4.5.1 Linking a Source Tag to a Known Target Tag

In many tag creation situations, there will be a need to embed a `<Link>` element in a *source tag* (i.e., a tag being created) which points to a preexisting *target tag*. For example, when a patch tag (see §2.1.3) is being created, it is important to link that patch tag to the tag that describes the product being patched. As another example, a supplemental tag (see §2.1.4) will typically be linked to another preexisting tag that is having its information supplemented. To establish these

relationships, tag creators use the `<Link>` `@href` attribute in the source tag to provide a pointer to the target tag.

In these situations, the tag creator will know the contents of the target tag, including its `@tagId`. Here, the target tag is known to the tag creator at the time that the source tag is being created. To link a source tag to a known target tag, tag creators are required to use the "`swid:`" scheme followed by the `@tagId` of the target tag.

> **GEN-11.** In order to link a source tag to a specific target tag with a `@tagId` that is known at the time the source tag is created, tag creators MUST set the value of the `<Link>` `@href` attribute in the source tag to a URI with "`swid:`" as its scheme, followed by the `@tagId` of the target tag.

This idea is illustrated below with two tag fragments.

Tag 1:

```
<SoftwareIdentity
  name="Application 1"
  tagId="com.largecomputerco.app1"
  …
</SoftwareIdentity>
```

Tag 2:

```
<SoftwareIdentity
  name="Application 2"
  tagId="com.largecomputerco.app2"
  …
  <Link rel="relation" href="swid:com.largecomputerco.app1"/>
</SoftwareIdentity>
```

In the above example, the source tag, Tag 2 (describing "Application 2"), is linked to the target tag, Tag 1 (describing "Application 1").

4.5.2 Linking a Tag to a Collection of Tags

In contrast to situations where the `@tagId` of the target tag is known, there also are many tag creation situations where there is a need to embed a `<Link>` element in a source tag which points either to a single target tag or to a *collection* of target tags with `@tagId` value(s) that cannot be known with certainty at the time the source tag is created. Consider the following scenarios:

- A primary tag (the source) is being created for a product that requires a commonly used shared library. This shared library is maintained by a third party, has its own primary tag, and is periodically upgraded in ways that maintain backwards compatibility. It is important that the source tag includes a link to the shared library's tag (the target), but it is not possible to specify a fixed `@tagId` for the target tag.

- A patch tag (the source) is being created for a patch that applies to a collection of products based on version. For example, imagine that a software provider has released a series of versions of a product, all in the "4.1.x" version series. A flaw is discovered in version 4.1.5 of the product, and it is determined that the flaw was introduced in version 4.1.0. A single patch is developed to correct this flaw, so its patch tag needs to link to all affected versions.

The SWID specification provides a mechanism that tag creators may use when linking a source tag to a collection of target tags. That mechanism is to set the `<Link>` `@href` attribute value to a URI with a scheme of "`swidpath:`", followed by an XPath 2.0 [XPath20] conformant query. Any such XPath query is expected to be used by a system to iterate over a set of SWID tags and identify matching tags by applying the XPath query to each tag and checking for a non-empty result. Because the SWID specification is not clear and specific about this usage, this document provides the following guidelines:

> **GEN-12.** When linking a source tag to one or more target tags with `@tagId` value(s) that cannot be determined at the time the source tag is created, tag creators MUST set the value of the `<Link>` `@href` attribute in the source tag to a URI with "`swidpath:`" as its scheme, followed by an XPath 2.0 [XPath20] conformant query. All characters contained in the XPath query which the URI specification [RFC3986] designates as *reserved* MUST be percent-encoded per the URI specification. All embedded SWID tag elements in the query MUST be prefixed with the "`swid:`" namespace.

The above guideline clarifies two points: (1) that URI reserved characters in the embedded XPath query must be percent-encoded, and (2) that the "`swid:`" namespace must be used for all SWID elements. Thus, in order to process such a query, the "`swidpath:`" scheme must be stripped off, any embedded percent encodings must be replaced with the decoded characters, and the XPath query processor must be supplied with the definition of the "`swid:`" namespace.

The following guideline advises query developers on how to prepare queries in a consistent and interoperable manner.

> **GEN-13.** Any XPath query used within a `<Link>` `@href` element MUST be designed in such a way that it can used by a system to iterate over a set of SWID tags and identify matching tags by applying the XPath query to each tag and checking for a non-empty result.

To illustrate the guidelines for linking tags, a series of examples is presented relative to this example tag:

```
<SoftwareIdentity
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xmlns="http://standards.iso.org/iso/19770/-2/2015/schema.xsd"
  xmlns:SHA256="http://www.w3.org/2001/04/xmlenc#sha256"
  xmlns:ext="http://example.org/ns/swid-example"
  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"/>
  <Entity
    name="Coyote Services, Inc."
    regid="mycoyote.com"
    role="distributor"/>
  <Meta
    activationStatus="trial"
    product="Roadrunner Detector"
    colloquialVersion="2013"
    edition="coyote"
    revision="sp1"
    ext:newattr="newvalue"/>
  <Payload>
    <Directory root="%programdata%" name="rrdetector">
      <File name="rrdetector.exe" size="532712"
SHA256:hash="a314fc2dc663ae7a6b6bc6787594057396e6b3f569cd50fd5dd
b4d1bbafd2b6a"/>
    </Directory>
  </Payload>
</SoftwareIdentity>
```

4.5.2.1 Example 1: Using an XPath Query to Refer to a Tag by its @tagId

Although the "`swid:`" scheme is intended to be used in cases where the target tag's `@tagId` value is known, the "`swidpath:`" scheme followed by an XPath query could also be used to achieve the same effect. The following XPath query will match the tag illustrated above in Section 4.5.2:

```
/swid:SoftwareIdentity[@tagId='com.acme.rrd2013-ce-sp1-v4-1-5-
0']
```

When incorporated into a `<Link>` element, the above query must be prefixed with the "`swidpath:`" scheme, and reserved characters must be percent-encoded, as follows:

```
<Link rel="relation"
href="swidpath:%2Fswid%3ASoftwareIdentity%5B%40tagId%3D%27com.ac
me.rrd2013-ce-sp1-v4-1-5-0%27%5D"/>
```

4.5.2.2 Example 2: Using an XPath Query to Refer to a Tag by Name and Tag Creator @regid

In the next example, an XPath 2.0 query is shown which matches any tag such that (1) the product name includes the string "`Roadrunner Detector`", and (2) the `<Entity>` element containing the "`tagCreator`" `@role` also contains the `@regid` value "`acme.com`":

```
/swid:SoftwareIdentity[contains(@name,'Roadrunner Detector')
  and swid:Entity[@regid='acme.com'
  and (count(index-of(@role,'tagCreator')) gt 0
    or contains(@role,'tagCreator')]]
```

Note: Depending on whether the XML parser is running in a schema-aware mode or not, XPath parsing behavior can vary. In schema-aware mode, the tag is validated against the SWID schema before the XPath query engine is run. The query above is designed with the assumption that the SWID tag parser can be either schema-aware, which causes the `@role` attribute to be parsed as a list of string values, or non-schema-aware, where the `@role` attribute is a single string value.

When incorporated into a `<Link>` element, the above query must be prefixed with the "`swidpath:`" scheme, and reserved characters must be percent-encoded, as follows:

```
<Link rel="relation"
href="%2Fswid%3ASoftwareIdentity%5Bcontains(%40name%2C%27Roadrun
ner%20Detector%27)%20and%20swid%3AEntity%5B%40regid%3D%27acme.co
m%27%20and%20(count(index-
of(%40role%2C%27tagCreator%27))%20gt%200%20or%20contains(%40role
%2C%27tagCreator%27)%5D%5D"/>
```

4.5.2.3 Example 3: Using an XPath Query to Refer to a Tag Containing a Known File

In the next example, an XPath 2.0 query is shown which matches any tag that includes a `<File>` element describing the "`rrdetector.exe`" file with a specific hash value:

```
/swid:SoftwareIdentity[descendant::swid:File[@name =
'rrdetector.exe'
  and @*[local-name() = 'hash'
  and namespace-uri() =
'http://www.w3.org/2001/04/xmlenc#sha256'
  and
.='a314fc2dc663ae7a6b6bc6787594057396e6b3f569cd50fd5ddb4d1bbafd2
b6a']]]
```

Such a query could be used in a patch tag that affects a specific file used by any number of other products, which may or may not be installed on a device at any given time. When incorporated into a `<Link>` element, the above query must be prefixed with the "`swidpath:`" scheme, and reserved characters must be percent-encoded, resulting in this `<Link>` element:

```
<Link rel="relation"
href="%2Fswid%3ASoftwareIdentity%5Bdescendant%3A%3Aswid%3AFile%5
```

```
B%40name%20%3D%20%27rrdetector.exe%27%20and%20%40*%5Blocal-
name()%20%3D%20%27hash%27%20and%20namespace-
uri()%20%3D%20%27http%3A%2F%2Fwww.w3.org%2F2001%2F04%2Fxmlenc%23
sha256%27%20and%20.%3D%27a314fc2dc663ae7a6b6bc6787594057396e6b3f
569cd50fd5ddb4d1bbafd2b6a%27%5D%5D%5D"/>
```

4.5.2.4 Example 4: Using an XPath Query to Refer to Tags with a Range of Versions

In the next example, an XPath 2.0 query is shown which matches any tag that describes a product with the name of "Roadrunner Detector" and with a version of 4.1.0 or greater within the 4.x release branch.

```
/swid:SoftwareIdentity[swid:Meta[@product = 'Roadrunner
Detector']
  and tokenize(@version,'\.')[1] cast as xs:unsignedInt = 4
  and tokenize(@version,'\.')[2] cast as xs:unsignedInt ge 1]
```

When incorporated into a `<Link>` element, the above query must be prefixed with the "`swidpath:`" scheme, and reserved characters must be percent-encoded, resulting in this `<Link>` element:

```
<Link rel="relation"
href="%2Fswid%3ASoftwareIdentity%5Bswid%3AMeta%5B%40product%20%3
D%20%27Roadrunner%20Detector%27%5D%20and%20tokenize(%40version%2
C%27%5C.%27)%5B1%5D%20cast%20as%20xs%3AunsignedInt%20%3D%204%20a
nd%20tokenize(%40version%2C%27%5C.%27)%5B2%5D%20cast%20as%20xs%3
AunsignedInt%20ge%201%5D%"/>
```

4.6 Implementing <Payload> and <Evidence> Elements

This section provides guidelines to be observed by tag creators when implementing SWID tag `<Payload>` and `<Evidence>` elements. The guidelines in this section address three issues:

1. Providing sufficient file information (see §4.6.1),
2. Selecting the hash function (see §4.6.2), and
3. Handling path separators and environment variables (see §4.6.3).

4.6.1 Providing Sufficient File Information

The first issue to be addressed here concerns the amount of payload/evidence information that needs to be provided.

Authoritative tag creators use the `<Payload>` element to enumerate the files and folders comprising a product or patch, whereas non-authoritative tag creators use the `<Evidence>` element for this purpose. Files are described using the `<File>` element, and folders are described using the <`Directory>` element.

The SWID specification requires only that the `<File>` element specify the name of the file using the `@name` attribute. This information is insufficient for most cybersecurity usage

scenarios. Additional information is needed to check whether files have been improperly modified since they were originally deployed. By including file size information within `<Payload>` and `<Evidence>` elements using the `@size` attribute, cybersecurity processes may efficiently test for changes that alter a file's size.

GEN-14. Every `<File>` element provided within a `<Payload>` or `<Evidence>` element MUST include a value for the `@size` attribute that specifies the size of the file in bytes.

Knowing a file's expected size is useful and enables a quick check to determine whether a file may have changed. Similarly, knowing a file's version as recorded in the file or filesystem can be useful when searching for installed products containing a file with a known version. This motivates the following guideline:

GEN-15. Every `<File>` element provided within a `<Payload>` or `<Evidence>` element MUST include a value for the `@version` attribute, if one exists for the file.

Because improper changes may also occur in ways that do not alter file sizes or versions, file hash values are also necessary. If there is a difference in the files' sizes, a change has occurred. If the size is the same, recomputing a hash will be necessary to determine if a change has occurred. Authoritative tag creators are expected to have sufficient knowledge of product details to be able to routinely provide hash values. Non-authoritative tag creators may not have the necessary knowledge of or access to files to provide hash information, but are encouraged to do so whenever possible.

GEN-16. [Auth] Every `<File>` element within a `<Payload>` element MUST include a hash value.

GEN-17. [Non-Auth] Every `<File>` element within an `<Evidence>` element SHOULD include a hash value.

4.6.2 Selecting the Hash Function

The second issue to be addressed here concerns selection of the hash function to be used when providing hash values.

Software products tend to be used long beyond the formal product support period. When selecting a hash function, it is important to consider the deployment lifecycle of the associated product. The hash value will likely be computed at the time of product release and will be used by tag consumers over the support lifecycle of the product and in some cases even longer. Stability in the hash functions used within SWID tags is desirable to maximize the interoperability of SWID-based tools while minimizing development and maintenance costs. Taking these considerations into account, it is desirable to choose a hash function that provides a minimum security strength of 128 bits to maximize the usage period.[3]

[3] According to NIST SP 800-57 Part 1 [SP800-57-part-1], when applying a hash function over a time period that extends beyond the year 2031, a minimum security strength of 128 bits is needed. Weak hash values are of little use and should be avoided.

GEN-18. Whenever `<Payload>` or `<Evidence>` elements are included in a tag, every `<File>` element SHOULD avoid the inclusion of hash values based on hash functions with insufficient security strength (< 128 bits).

According to [SP800-107], the selected hash function needs to provide the following security properties:

- **Collision Resistance:** "It is computationally infeasible to find two different inputs to the hash function that have the same hash value." This provides assurance that two different files will have different hash values.
- **Second Preimage Resistance:** "It is computationally infeasible to find a second input that has the same hash value as any other specified input." This provides assurance that a file cannot be engineered that will have the same hash value as the original file. This makes it extremely difficult for a malicious actor to add malware into stored executable code while maintaining the same hash value.

The SHA-256, SHA-384, SHA-512, and SHA-512/256 hash functions meet the 128-bit strength requirements for collision resistance and second preimage resistance.[4] This leads to the following guidelines:

GEN-19. [Auth] Whenever a `<Payload>` element is included in a tag, every `<File>` element contained therein MUST provide a hash value based on the SHA-256 hash function.

GEN-20. [Non-Auth] Whenever an `<Evidence>` element is included in a tag, every `<File>` element contained therein SHOULD provide a hash value based on the SHA-256 hash function.

GEN-21. Whenever a `<Payload>` or `<Evidence>` element is included in a tag, every `<File>` element contained therein MAY additionally provide hash values based on the SHA-384, SHA-512, and/or SHA-512/256 hash functions.

Due to the use of 64-bit word values in the algorithm, SHA-512 hash function implementations may perform better on 64-bit systems. For this reason, tag creators are encouraged to consider including a SHA-512 hash value.

4.6.3 Handling of Path Separators and Environment Variables

The third issue to be addressed here concerns interoperable handling of path separators and environment variables in `<File>` and `<Directory>` elements.

The SWID specification defines three attributes that may be used on `<File>` and `<Directory>` elements to fully specify a file or a folder:

- `location`: A string specifying the directory or location where a file was found or can

[4] See FIPS 180-4 [FIPS180-4].

be expected to be located;

- `name`: A string specifying the name of the filename or directory without any embedded path separators; and
- `root`: A string specifying a system-specific root folder that the "`location`" attribute is an offset from; if this is not specified, it is assumed that the "`root`" is the same folder as the location of the SWID tag, or is the directory specified by an enclosing `<Directory>` element.

While the "`name`" attribute is defined to explicitly exclude path separators, both "`location`" and "`root`" are permitted to include such separators. The problem is that path separators vary across operating environments. The most widely known difference is between Windows and UNIX/Linux systems; Windows uses the backslash "\" as the path separator, while UNIX/Linux systems use the forward slash "/".

It is important that tag consumers be able to reliably parse strings containing embedded path separators without having to guess the path separator. This document recommends that the path separator character be made explicit in `<Payload>` and `<Evidence>` elements. The following guideline achieves this by introducing a new `@n8060:pathSeparator` extension attribute:

> **GEN-22.** The `@n8060:pathSeparator` extension attribute SHOULD be used within `<Payload>` and `<Evidence>` elements to specify the path separator character used in embedded `<File>` and `<Directory>` elements.

In a related vein, the SWID specification also allows platform-specific environment variables to be used within any or all of the "`root`", "`location`", and "`name`" attributes. Once again, the format of such variables differs across platforms. On Windows machines, environment variables are enclosed in percent "%" characters, while UNIX/Linux systems prefix variables with a dollar "$" character. This document recommends that environment variable prefix and suffix characters be made explicit in `<Payload>` and `<Evidence>` elements. The following guidelines introduce the `@n8060:envVarPrefix` and `@n8060:envVarSuffix` extension attributes:

> **GEN-23.** The `@n8060:envVarPrefix` extension attribute SHOULD be used within `<Payload>` and `<Evidence>` elements to specify the character(s) used to prefix environment variables that may be embedded `<File>` and `<Directory>` elements.

> **GEN-24.** The `@n8060:envVarSuffix` extension attribute SHOULD be used within `<Payload>` and `<Evidence>` elements to specify the character(s) used to suffix environment variables that may be embedded `<File>` and `<Directory>` elements.

When guidelines **GEN-22** through **GEN-24** are observed, a `<Payload>` could be represented as follows:

```
<Payload n8060:pathSeparator="/" n8060:envVarPrefix="$"
    n8060:envVarSuffix="">
  <Directory root="$ETC/drivers" name="printers">
```

```
    <Directory name="printermodel">
      <File name="colordriver.shlib" size="234824"/>
      <File name="bwdriver.shlib" size="143854"/>
    </Directory>
  </Directory>
</Payload>
```

Given this information, a tag consumer should be able to straightforwardly construct these two fully-qualified filenames:

- `$ETC/drivers/printers/printermodel/colordriver.shlib`
- `$ETC/drivers/printers/printermodel/bwdriver.shlib`

To access these files on the device, device-specific information will still be required to determine the value of the "`$ETC`" environment variable.

4.7 Providing Attribute Values in Multiple Languages

Section 4.3 introduced a guideline to require that the `<SoftwareIdentity>` `@xml:lang` attribute be used to specify the preferred human language used within the tag to provide language-dependent attribute values. This is the most common scenario, i.e., that all language-dependent attribute values within a tag will be provided in the same language. Such tags will be termed *monolingual tags*. For example, a Japanese software provider wishing to create a monolingual tag in Japanese will set the preferred language of its tag to be Japanese (e.g., using the language tag "`ja-jp`" as the value for the `<SoftwareIdentity>` `@xml:lang` attribute), then specify all language-dependent attributes in Japanese.

Because the SWID specification permits the `@xml:lang` attribute to be used on *any* tag element, this enables tag creators to implement *multilingual tags*, tags that provide language-dependent attributes in more than one language. Suppose, for example, that a Japanese software provider wants to provide its organization's name in both Japanese and English. Technically, this is straightforward to do, as illustrated here:

```
<SoftwareIdentity
  xmlns="http://standards.iso.org/iso/19770/-2/2015/schema.xsd"
  xml:lang="ja-jp"
  name="コンピュータ管理システム 2015"
  tagId="jp.largecomputerco.タグ番号 1"
  version="1.0">
  <Entity
    name="大型コンピュータ会社"
    regid="largecomputerco.jp"
    role="tagCreator softwareCreator"/>
  <Entity xml:lang="en-us"
    name="Large Computer Company"
    regid="largecomputerco.jp"
    role="tagCreator softwareCreator"/>
  …
```

```
</SoftwareIdentity>
```

The first <Entity> element in the example inherits the value of @xml:lang specified by the <SoftwareIdentity> element. The second <Entity> element explicitly overrides the preferred language, signaling that information is being furnished in English (in this example).

Although the SWID specification provides the technical means to implement multilingual tags, in practice, implementing and processing such tags present a number of challenges and potential interoperability issues, so care must be taken.

This report does not offer guidelines to address these issues, since the marketplace requirements for multilingual tags are insufficiently clear to support development of robust guidelines to fully and effectively address all associated interoperability concerns. Instead, this report will discuss some of the most significant issues and suggest future directions for interoperability guidance.

4.7.1 Specifying Product Names in Multiple Languages

The <SoftwareIdentity> @name attribute specifies the preferred name of the software product, and is provided in the language indicated by the value of the @xml:lang attribute (which is now required per guideline **GEN-2**). The SWID specification makes no provisions for the specification of multiple product names, much less for the specification of multiple product names in different languages.

If the marketplace determines that there is a demand for recording a multilingual representation of a product name in a single SWID tag, two options (short of revising the SWID specification itself) exist:

1. Use the <Meta> element. Note, however, that because the <Meta> element described in the SWID specification does not offer a pre-defined attribute for alternate product names, the user community would need to agree on a new extension attribute to be used for this purpose.
2. Use supplemental tags. Here, the idea would be to create one or more supplemental tags, each of which specifies a different preferred language via the <SoftwareIdentity> @xml:lang attribute, that then provides the <SoftwareIdentity> @name attribute value in that preferred language. In other words, a U.S.-based software provider might issue a single primary tag providing the product name in English, along with four supplemental tags providing the product name in French, German, Japanese, and Spanish, respectively.

The first option is relatively compact, but depends on effective procedures for defining and managing extension attributes. The second option avoids the introduction of an extension attribute, but forces tag creators to generate arbitrary quantities of supplemental tags to address all human languages of interest.

4.7.2 Specifying <Entity> Elements in Multiple Languages

The example at the top of this section shows how a Japanese organization’s name could be provided in both Japanese and English in two separate <Entity> elements distinguished by

different @xml:lang attributes, one value inherited from <SoftwareIdentity> and the other an explicit value overriding the inherited value.

This multilingual capability also creates opportunities for entity information specified in one language to differ in unexpected and confusing ways from entity information specified in a second language. For example, there is nothing in either the SWID specification or the associated XML schema that would prevent the following scenario:

```
<SoftwareIdentity
  xmlns="http://standards.iso.org/iso/19770/-2/2015/schema.xsd"
  xml:lang="ja"
  name="コンピュータ管理システム 2015"
  tagId="jp.largecomputerco.タグ番号 1"
  version="1.0">
  <Entity xml:lang="en"
    name="The Large Computer Company"
    regid="largecomputerco.jp"
    role="tagCreator softwareCreator"/>
  <Entity xml:lang="fr"
    name="Le Grand Enterprise d'Informatique"
    regid="largecomputerco.jp"
    role="tagCreator"/>
  …
</SoftwareIdentity>
```

This example contains two problems. First, the tag's preferred language is specified as Japanese, but entity information is only provided in English and French. Given such a tag, it is not obvious how a tag consumer should decide which <Entity> element provides the preferred name of the tag creator. Second, the two <Entity> elements agree on the @regid, but disagree on the @role. Again, it is not obvious how to interpret such data. A number of equally odd scenarios can easily be envisaged.

Future guidelines may be needed to address issues like these. Such guidelines might include:

- A requirement that at least one <Entity> element be provided in the preferred language; and
- A requirement that all <Entity> elements which agree on the @regid value also agree on the @role value, regardless of the @xml:lang value.

4.7.3 Specifying <Payload> Elements in Multiple Languages

When software products are localized to a particular language region, files and folder names often change to match the localized language. To account for this, authoritative tag creators may want to create multilingual tags that provide <Payload> elements which enumerate alternate language-specific versions of files and folders. If this is done, it must be done in a way that enables tag consumers to simply and accurately determine the effective language-specific

payload. This information could be represented in multiple ways; for example, a tag creator could implement multiple alternative <Payload> elements, as illustrated here:

```
<SoftwareIdentity
  xmlns="http://standards.iso.org/iso/19770/-2/2015/schema.xsd"
  xml:lang="en"
  name="Joyful App 2015"
  tagId="com.largecomputerco.joyfulapp1"
  version="1.0">
  …
  <Payload
    <File name="joyfulapp.exe"/>
  </Payload>
  <Payload xml:lang="fr"
    <File name="appdejoie.exe"/>
  </Payload>
  …
</SoftwareIdentity>
```

In this example, the first <Payload> element is expressed in the tag's preferred language (English), inherited from <SoftwareIdentity>. The second <Payload> element is expressed in French. This is a simple and straightforward way to represent equivalent payloads in alternate languages. But because @xml:lang may be used on any element, there is nothing to prevent a tag creator from representing the payload in the following way:

```
  <Payload
    <File name="joyfulapp.exe"/>
    <File xml:lang="fr" name="appdejoie.exe"/>
  </Payload>
```

Arguably, this is a more compact representation, but raises questions of how someone might determine the effective language-specific payload for a given device. The above payload could potentially represent two files that are co-present on a device, or two different files, only one of which is present as determined by the local language. Future guidelines may be necessary to resolve potential interoperability issues like these.

For non-authoritative tag creators, two scenarios are pertinent. In the first scenario, when a tag is generated by a tool based on evidence collected on a device, the tool is unlikely to be able to determine and record that evidence in multiple languages. As a result, for such non-authoritative tag creators, including data in multiple languages in <Evidence> elements is not likely to be a concern. In contrast, when a tag is generated by a human analyst, and that tag is later deployed during discovery, it may be possible to provide data in multiple languages in <Evidence> elements. This topic requires further study and additional guidelines may be necessary as a result.

4.8 Updating Tags

The SWID specification prohibits modification of SWID tags by anyone other than the original tag creator. The primary reason for altering a tag after it has been installed on a device is to correct errors in the tag. In rare circumstances, it may be useful to update a tag to add data elements that logically belong in the tag and not in a separate supplemental tag. However, under normal conditions, tags should rarely be modified, and supplemental tags should be used to add identifying and descriptive product information.

When changes are made to a product's codebase that cause the product's version to change, those changes should be reflected by removing all original tags (primary, supplemental, and patch tags) and installing new tags as appropriate to identify and describe the new product version. Patches should be indicated by adding a patch tag to the installed collection of tags.

When an existing tag must be updated, it will rarely make sense to edit the tag in place, that is, to selectively modify portions of the tag as if using a text editor. Such editing actions would likely invalidate XML digital signatures stored in the tag. Thus it is expected that when a tag is updated, it is always fully replaced along with any embedded digital signatures.

When a tag must be updated to correct errors or add data elements, its `<SoftwareIdentity>` `@tagId` should not be changed. This is because tag identifiers may be used as proxy identifiers for pre-installation software packages, installed software products, or software patches. It is important that tag identifiers be usable as reliable persistent identifiers. This leads to the following guideline.

> **GEN-25.** When it is necessary to update a tag to correct errors in or add data elements to that tag, the tag's `<SoftwareIdentity>` `@tagId` SHOULD NOT be changed.

When tags are updated, however, it is important that the updates be implemented in a manner that supports easy change detection. Tag consumers should not be required or expected to fully compare all contents of discoverable tags to determine if any of the products have changed since the last time the tags were examined. To facilitate change detection by tag consumers, tag creators are expected to update the `<SoftwareIdentity>` `@tagVersion` attribute to indicate that a change has been made to the tag.

> **GEN-26.** When it is necessary to update a tag to correct errors in or add data elements to that tag, the tag's `<SoftwareIdentity>` `@tagVersion` attribute MUST be changed.

If this guideline is observed, tag consumers need only to maintain records of tag identifiers and tag versions discovered on a device. If a tag with a previously unseen tag identifier is found on a device, a tag consumer may conclude that a new product has been installed since the last time the device was inventoried. If a tag with a previously discovered tag identifier can no longer be discovered on a device, a tag consumer may conclude that a software product has been removed since the last time the device was inventoried. If, however, a tag is discovered on a device with a previously seen tag identifier but a new tag version, a tag consumer may conclude that identifying or descriptive metadata in that tag has been changed, and so the tag should be fully processed.

4.9 Summary

The primary purpose of all the guidelines in this report is to help tag creators understand how to implement SWID tags in a manner that will satisfy the tag handling requirements of IT organizations. The guidelines are intended to be broadly applicable to common IT usage scenarios that are relevant to all software providers and consumers.

This section provided implementation guidelines addressing issues common to all situations in which tags are deployed and processed. These are the key points from this section:

- Tags may be created by authoritative or non-authoritative entities. An *authoritative tag creator* is a first or second party that creates a tag as part of the process of releasing software. Authoritative tag creators typically possess accurate, complete, and detailed technical knowledge that is needed for creation of authoritative tags containing reliable information. A *non-authoritative tag creator* is an entity that has a third-party relation to the creation, maintenance, and distribution of the software. Discovery tools may also act as non-authoritative tag creators. Non-authoritative tag creators typically create tags using product information that is gathered using forensic methods while discovering installed software.
- The SWID specification supports an international audience, allowing tag creators to provide language-dependent attribute values in region-specific human languages. Guidelines in this section specify how tag creators should designate the default human language of language-dependent attribute values provided within a tag, and how such values may be provided in multiple languages.
- SWID tags provide detailed information about various entities associated with the software product described by the tag, as well as with the tag itself. Guidelines in this section address how complex entity specifications are to be avoided, how authoritative and non-authoritative tags are to be distinguished, and how information about the software creator is to be furnished.
- SWID tags may be explicitly linked to other tags and/or other resources in a variety of ways. Guidelines in this section address how source tags are to be linked to individual target tags, and/or to sets of target tags.
- Tag creators may provide detailed information about the files and folders comprising a software product. Guidelines in this section address how sufficient information may be provided, how cryptographic hashes may be provided, and how platform-specific path separators and environment variables may be incorporated in file or folder descriptions.

5 Implementation Guidance Specific to Tag Type

This section provides implementation guidelines that are specific to each of the four tag types defined in Section 2.1: corpus tags (see §5.1), primary tags (see §5.2), patch tags (see §5.3), and supplemental tags (see §5.4).

5.1 Implementing Corpus Tags

As noted in Section 2.1.1, corpus tags are used to identify and describe products in a pre-installation state. This section provides guidance addressing the following topics related to implementation of corpus tags: setting the `<SoftwareIdentity>` `@corpus` attribute (see §5.1.1), specifying `@version` and `@versionScheme` (see §5.1.2), and specifying `<Payload>` element information (see §5.1.3).

5.1.1 Setting the <SoftwareIdentity> @corpus Attribute

To indicate that a tag is a corpus tag, tag implementers set the value of the `<SoftwareIdentity>` `@corpus` attribute to "`true`". The SWID specification does not specifically prohibit tag implementers from also setting other tag type indicator attributes to "`true`" (e.g., `<SoftwareIdentity>` `@patch` and `<SoftwareIdentity>` `@supplemental`), but doing so would create confusion regarding how the information contained within the tag should be interpreted. This report provides guidelines to ensure that at most one tag type indicator attribute is set to "`true`".

> **COR-1.** If the value of the `<SoftwareIdentity>` `@corpus` attribute is set to "`true`", then the values of `@patch` and `@supplemental` MUST be set to "`false`".

5.1.2 Specifying the Version and Version Scheme in Corpus Tags

Corpus tags identify and describe software products in a pre-installation state. As part of the process of determining whether a given product is suitable for—or allowed to be installed on—a device, tag consumers often need to know the product's specific version. The SWID specification provides the `<SoftwareIdentity>` `@version` attribute for recording version information, but defines this attribute as optional with a default value of "`0.0`".

This report seeks to encourage software providers to assign product versions for their products, and to explicitly record software versions in tags released along with those products. In short, if a software product has an assigned version, that version must be specified in the tag.

> **COR-2.** If a software product has been assigned a version by the software provider, that version MUST be specified in the `<SoftwareIdentity>` `@version` attribute of the product's corpus tag, if any.

For many cybersecurity purposes, it is important to know not only a product's version, but also whether a given product version represents an "earlier" or "later" release of a product, compared to a known version. For example, security bulletins often warn that a newly discovered vulnerability was found in a particular version V of a product, but may also be present in "earlier versions." Thus, given two product versions V1 and V2, it is important to be able to tell whether V1 is "earlier" or "later" than V2.

In order to make such an ordering decision reliably, it is necessary to understand the structure of versions and how order is encoded in versions. This is no single agreed-upon practice within the software industry for versioning products in a manner that makes clear how one version of a product relates to another. The "Semantic Versioning Specification" [SEMVER] proposes a common interpretation of multi-part numeric versions, but it is not universally followed.

The SWID specification defines the `<SoftwareIdentity>` `@versionScheme` attribute to record a token that designates the "scheme" according to which the value of `<SoftwareIdentity>` `@version` can be parsed and interpreted. Like `@version`, the SWID specification defines `@versionScheme` as "optional" with a default value of `multipartnumeric`. Table 3 lists the allowed values of `@versionScheme` that are defined in the SWID specification.

Table 3: Allowed Values of @versionScheme

Value	Meaning
`multipartnumeric`	Numbers separated by dots, where the numbers are interpreted as integers (e.g., 1.2.3, 1.4.5, 1.2.3.4.5.6.7)
`multipartnumeric+suffix`	Numbers separated by dots, where the numbers are interpreted as integers with an additional string suffix (e.g., 1.2.3a)
`alphanumeric`	Strictly a string, sorting is done alphanumerically
`decimal`	A floating point number (e.g., 1.25 is less than 1.3)
`semver`	Follows the [SEMVER] specification
`unknown`	Other unknown version scheme; no attempt should be made to order versions of this type
`<any>`	Other version schemes that may be generally known in the market

The following guideline is provided in consideration of the fact that tag consumers have a critical interest in knowing not only a product's version, but also its versioning scheme and the semantics of that scheme.

> **COR-3.** If a corpus tag contains a value for the `<SoftwareIdentity>` `@version` attribute, it MUST also contain a value for the `<SoftwareIdentity>` `@versionScheme` attribute.

If a particular product's version does not conform to one of the pre-defined schemes listed in Table 3, whatever value a tag creator provides for the `<SoftwareIdentity>` `@versionScheme` attribute ought to be selected from a well-known public list of version scheme identifiers. Mechanisms for establishing, advertising, and curating such public lists are beyond the scope of this document. Ideally, such well-known public lists of version schemes will provide enough semantic definition of each scheme to enable tag consumers to determine whether a version V1 conforming to a particular scheme should be ordered "before" or "after" another version V2 conforming to that same scheme.

5.1.3 Specifying the Corpus Tag Payload

Corpus tags are used to document the installation media associated with a software product. This documentation enables the media to be checked for authenticity and integrity. At a minimum, corpus tags are required to provide `<Payload>` details that enumerate all the files on the installation media, including file sizes and hash values.

COR-4. A corpus tag MUST contain a `<Payload>` element that enumerates every file that is included in the tagged installation media.

5.2 Implementing Primary Tags

The primary tag for a software product contains descriptive metadata needed to support a variety of business processes. To ensure that tags contain the metadata needed to help automate IT and cybersecurity processes on information systems, additional requirements must be satisfied. This section provides guidance addressing the following topics: setting tag type indicator attributes to designate a tag as a primary tag (see §5.2.1), specifying version and version scheme information (see §5.2.2), specifying `<Payload>` or `<Evidence>` information (see §5.2.3), and specifying attributes needed support targeted search (see §5.2.4).

5.2.1 Setting the <SoftwareIdentity> Tag Type Indicator Attributes

To indicate that a tag is a primary tag, tag implementers ensure that the values of all three tag type indicators (the `<SoftwareIdentity>` `@corpus`, `@patch`, and `@supplemental` attributes) are set to "`false`". This is enforced by the following guideline.

PRI-1. To indicate that a tag is a primary tag, the `<SoftwareIdentity>` `@corpus`, `@patch`, and `@supplemental` attributes MUST be set to "`false`".

5.2.2 Specifying the Version and Version Scheme in Primary Tags

Primary tags identify and describe software products in a post-installation state. Like corpus tags, primary tag information about product versions and associated version schemes is important to enable tag consumers to conduct various cybersecurity operations. Unlike the case for corpus tags, however, guidelines for primary tags must distinguish between authoritative and non-authoritative primary tag creators.

PRI-2. [Auth] If a software product has been assigned a version by the software provider, that version MUST be specified in the `<SoftwareIdentity>` `@version` attribute of the product's primary tag.

PRI-3. [Auth] If a primary tag contains a value for the `<SoftwareIdentity>` `@version` attribute, it MUST also contain a value for the `<SoftwareIdentity>` `@versionScheme` attribute which accurately reflects the versioning scheme used in the `@version` attribute.

PRI-4. [Non-Auth] If a software product has been assigned a version by the software provider, and that version can be determined, the `<SoftwareIdentity>` `@version` attribute of the primary tag MUST contain that value.

PRI-5. [Non-Auth] If a primary tag contains a value for the `<SoftwareIdentity>` `@version` attribute, and the version scheme of that `@version` attribute value can be determined, the `<SoftwareIdentity>` `@versionScheme` attribute of the primary tag MUST contain that version scheme value.

As was true for corpus tags (see §5.1.2), it is important that the version schemes used in primary tags enable distinct versions of a product to be placed in a defined order, minimally so that consumers can determine whether one version of a product is "before" (earlier than) or "after" (later than) another version. Section 8.6.13 of the SWID specification provides a table of pre-defined values for the `@versionScheme` attribute with defined semantics (reproduced above in Table 3). If a value for the `@versionScheme` attribute is provided that is not listed among the pre-defined values, ideally that value ought to come from a well-known public list of version scheme identifiers. The public list would specify the meaning for each version scheme sufficiently to allow for comparing two versions and determining their relative order in a sequence of versions.

5.2.3 Specifying Primary Tag Payload and Evidence

Detailed information about the files comprising an installed software product is a critical need for cybersecurity operations. For example, such information enables software inventory and integrity tools to confirm that the product described by a discovered tag is, in fact, installed on a device. Authoritative tag creators are encouraged to provide a `<Payload>` element in the primary tag, and non-authoritative tag creators are encouraged to provide an `<Evidence>` element in the primary tag.

PRI-6. [Auth] A `<Payload>` element SHOULD be provided in a software product's primary tag.

PRI-7. [Non-Auth] An `<Evidence>` element SHOULD be provided in a software product's primary tag.

Note that guidelines **PRI-6** and **PRI-7** specify that payload and evidence information is supplied within the primary tag, and not within a supplemental tag. This is due to concerns about additional processing complexity and difficulties with assuring the reliability of such payload and evidence information when it is stored separately from the primary tag.

Ideally, `<Payload>` and `<Evidence>` elements should list every file that is found to be part of the product described by the tag. Such information aids in the detection of malicious software attempting to hide among legitimate product files. It also aids in reconciling authoritative and non-authoritative tags in cases where both kinds of tags exist on a device for the same product.

PRI-8. `<Payload>` and `<Evidence>` elements SHOULD list every file comprising the product described by the tag.

Although a full enumeration of product files is the ideal, at a minimum, only those files subject to execution, referred to here as *machine instruction files*, need to be listed. A machine instruction file is any file that contains machine instruction code subject to runtime execution, whether in the form of machine instructions that can be directly executed by computing hardware or hardware emulators; bytecode that can be executed by a bytecode interpreter; or scripts that

can be executed by scripting language interpreters. Library files that are dynamically loaded at runtime are also considered machine instruction files.

PRI-9. [Auth] If a `<Payload>` element is provided, it MUST list every machine instruction file comprising the product described by the tag.

PRI-10. [Non-Auth] If an `<Evidence>` element is provided, it MUST list every machine instruction file comprising the product described by the tag.

If a tag creator enumerates every file according to **PRI-8**, this can cause problems later for tag consumers. Recall that Section 4.6 of this document provides guidelines that require tag creators to supply file size, version, and hash information (see **GEN-11** through **GEN-18**). These guidelines are there to ensure that tag consumers can later use the provided information to confirm the integrity of files discovered on devices. The problem, however, is that particular files listed in a `<Payload>` or `<Evidence>` element might be changed for non-malicious reasons at arbitrary times after the product is installed. Data and configuration files are two examples.

If a tag consumer were to inspect a particular file listed in a tag's `<Payload>` or `<Evidence>` element, compare the file's hash value as listed in a tag to a new value computed from the actual file on a device, and discover a mismatch, that would be a "false positive" if that file was expected to be dynamic. If tag consumers were to generally find that performing such comparisons led to an unwieldy number of false positives, they might be inclined to stop using SWID tag payload and evidence information altogether, an undesirable outcome.

In the interest of minimizing the possibility of such false positives, this document provides guidelines for tag creators to explicitly mark all mutable files listed in a tag's `<Payload>` element with a special value. Specifically, this document introduces an `@n8060:mutable` extension attribute on `<File>` elements. The `@n8060:mutable` extension attribute takes a Boolean value with the default value of "`false`". Authoritative tag creators are required to set the `@n8060:mutable` attribute value to "`true`" for any `<File>` element that describes a non-static file. Non-authoritative tag creators are encouraged to do so whenever possible.

PRI-11. [Auth] If a `<File>` element included in a `<Payload>` element of a primary tag describes a file that can undergo authorized changes over time in ways that could alter its size, version, and/or hash value, the tag creator MUST set that file's `<File>` `@n8060:mutable` extension attribute to "`true`".

PRI-12. [Non-Auth] If it can be determined that a `<File>` element included in an `<Evidence>` element of a primary tag describes a file that can undergo authorized changes over time in ways that could alter its size, version, and/or hash value, the tag creator SHOULD set that file's `<File>` `@n8060:mutable` extension attribute to "`true`".

Observance of these guidelines by tag creators will help ensure that the resulting `<Payload>` and `<Evidence>` elements are useful to tag consumers attempting to verify the integrity of installed software products, while minimizing the potential number of false positives that such consumers may have to cope with.

5.2.4 Specifying Product Metadata Needed for Targeted Search

The SWID specification furnishes the <SoftwareIdentity> @name attribute to capture "the software component name as it would typically be referenced." This is also called the product's *market name*, i.e., the product name as used on websites and in advertising materials to support marketing, sales, and distribution. Market names for commercial software products often combine a variety of market-relevant descriptive elements, such as the provider's "brand name," and the product's market version, edition, revision, or update level. While any or all of these elements may appear in the <SoftwareIdentity> @name attribute of the product's primary tag, there is no consistency in whether or how those elements are included, making it difficult for a machine to reliably separate them out.

These metadata elements are often needed by local administrators, cybersecurity personnel, and supporting automated tools when performing targeted searches. To make this possible, there needs to be a way to individually refer to each descriptive element embedded within a product's market name. To this end, the SWID specification defines these <Meta> element attributes:

- @product: The base name of the product. The base name is expected to exclude substrings containing the software provider's name, as well as any indicators of the product's version, edition, or revision level.
- @colloquialVersion: The market version of the product. This version may remain the same through multiple releases of a software product, whereas the version specified in the <SoftwareIdentity> @version is more specific to the underlying software codebase and will change for each software release.
- @edition: The edition of the product.
- @revision: An informal designation for the revision of the product.

If these attributes are specified, targeted searches will be easier to define and execute, and it will also be possible to mechanically generate a valid CPE name from an input SWID tag.[5] The guideline is as follows:

> **PRI-13.** If appropriate values exist and can be determined, a <Meta> element MUST be provided and MUST furnish values for as many of the following attributes as possible: @product, @colloquialVersion, @revision, and @edition.

5.3 Implementing Patch Tags

As noted earlier in Section 2.1.3, a patch tag is used to describe localized changes applied to an installed product's codebase. This section provides guidance addressing the following topics related to implementation of patch tags: setting the <SoftwareIdentity> @patch attribute (see §5.3.1), and specifying <Payload> or <Evidence> information (see §5.3.2).

[5] See [NISTIR 8085] for an algorithm that may be used to generate such CPE names.

5.3.1 Setting the <SoftwareIdentity> @patch Attribute

To indicate that a tag is a patch tag, tag implementers set the value of the `<SoftwareIdentity>` `@patch` attribute to "`true`". The SWID specification does not specifically prohibit tag implementers from also setting other tag type indicator attributes to "`true`" (e.g., `<SoftwareIdentity>` `@corpus` and `<SoftwareIdentity>` `@supplemental`), but doing so would create confusion regarding how the information contained within the tag should be interpreted. This report provides guidelines to ensure that at most one tag type indicator attribute is set to true.

PAT-1. If the value of the `<SoftwareIdentity>` `@patch` attribute is set to "`true`", then the values of `@corpus` and `@supplemental` MUST be set to "`false`".

5.3.2 Specifying Patch Tag Payload and Evidence

Patches change files that comprise a software product, and may thereby eliminate known vulnerabilities. If patch tags clearly specify the files that are changed as a result of applying the patch, software inventory and integrity tools become able to confirm that the patch has actually been applied and that the individual files discovered on the device are the ones that should be there. Guidelines in this section propose that patch tags document three distinct types of changes:

1. **Update:** A file previously installed has been modified in place or replaced on the device with a modified version.
2. **Remove:** A file previously installed has been removed from the device.
3. **Add:** An entirely new file has been added to the device.

For files that are replaced or added, patch tags must include file sizes and hash values. As stated before in requirements **GEN-14** and **GEN-16**, authoritative tag creators are required to provide this information in the `<Payload>` element of the patch tag. Non-authoritative tag creators are encouraged to provide this information whenever possible in the `<Evidence>` element of the patch tag (see **GEN-14, GEN-17**).

PAT-2. [Auth] A patch tag SHOULD contain a `<Payload>` element that enumerates every file that is replaced, removed, or added by the patch.

PAT-3. [Auth] Each `<File>` element contained within the `<Payload>` element of a patch tag MUST include an extension attribute named `@n8060:patchEvent`, that has one of the following values:

- The string value "`update`" to indicate that the patch updates a pre-existing installed file by either modifying the file or replacing it with a new version of the file
- The string value "`remove`" to indicate that the patch removes a pre-existing installed file
- The string value "`add`" to indicate that the patch installs a new file that did not previously exist

PAT-4. [Non-Auth] A patch tag SHOULD contain an <Evidence> element that enumerates every file that was found to have changed as a result of the patch process.

5.4 Implementing Supplemental Tags

As noted in Section 2.1.4, supplemental tags are used to furnish identifying and descriptive information not contained in other tags. This section provides guidance addressing the following topics related to implementation of supplemental tags: setting the <SoftwareIdentity> @supplemental attribute (see §5.4.1), linking supplemental tags to other tags (see §5.4.2), and establishing the precedence of information contained in a supplemental tag (see §5.4.3).

5.4.1 Setting the <SoftwareIdentity> @supplemental Attribute

To indicate that a tag is a supplemental tag, tag implementers set the value of the <SoftwareIdentity> @supplemental attribute to "true". The SWID specification does not specifically prohibit tag implementers from also setting other tag type indicator attributes to "true" (e.g., <SoftwareIdentity> @corpus and <SoftwareIdentity> @patch), but doing so would create confusion regarding how the information contained within the tag should be interpreted. This report provides guidelines to ensure that at most one tag type indicator attribute is set to "true".

SUP-1. If the value of the <SoftwareIdentity> @supplemental attribute is set to "true", then the values of @corpus and @patch MUST be set to "false".

5.4.2 Linking Supplemental Tags to Other Tags

An individual supplemental tag may be used to furnish data elements that complement or extend data elements furnished in another individual tag. That is, a supplemental tag may not be used to supplement a collection of tags. A supplemental tag may supplement any type of tag, including other supplemental tags. Because the SWID specification does not clearly state how a supplemental tag should indicate its linkage to other tags, a clarifying guideline is provided here.

SUP-2. A supplemental tag MUST contain a <Link> element to associate itself with the individual tag that it supplements. The @rel attribute of this <Link> element MUST be set to "supplemental".

Note that the SWID specification also requires that every <Link> element provide a value for the @href attribute. Section 4.5 of this document provides pertinent guidelines for how tag creators should use the @href attribute to refer to other tags, in situations when the @tagId of the target is known (see **GEN-11**) and when it is not known (see **GEN-12** and **GEN-13**).

5.4.3 Establishing Precedence of Information

As noted earlier, a supplemental tag is intended to furnish data elements that complement or extend data elements furnished in another tag. This does not preclude situations in which a supplemental tag contains elements or attributes that potentially conflict with elements or attributes furnished in the tag being supplemented. For example, suppose a device contains a primary tag where the value of the <SoftwareIdentity> @name attribute is specified as

"Foo", and a supplemental tag is also present that is linked to the primary tag but specifies the value of the <SoftwareIdentity> @name attribute as "Bar".

This report takes the position that supplemental tags strictly extend, and never override, data elements found in the tags that are supplemented. In the example above, Foo is considered to be the correct value for @name, and the value of Bar furnished in the supplemental tag is ignored.

Because certain attribute values pertain to tags themselves—e.g., @tagId, @tagVersion, and <Entity> information about the tag creator—differences in those values between a supplemental tag and a supplemented tag are never construed as conflicts. In other cases, information in a supplemental tag may be combined with information in the supplemented tag to obtain a full description of the product. For example, a primary tag may provide an <Entity> element that specifies the tagCreator role, while a supplemental tag provides <Entity> elements specifying other roles such as softwareCreator and licensor. In this scenario, the primary and supplemental tag collectively furnish all Entity roles. If, however, both the primary and supplemental tags provide <Entity> elements specifying values for the same role (e.g., both tags specify different softwareCreator values), then the conflicting value in the supplemental tag is ignored. This leads to the following guideline.

> **SUP-3.** If a supplemental tag provides a data value that conflicts with corresponding data values in the tag being supplemented, the data value in the supplemented tag MUST be considered to be the correct value.

5.5 Summary

Key points presented in this section include:

- The <SoftwareIdentity> @corpus, @patch, and @supplemental attribute values are mutually exclusive, so that a tag type can be clearly distinguished as corpus, primary, patch, or supplemental.
- Corpus tags must provide <SoftwareIdentity> @version information, and also <SoftwareIdentity> @versionScheme information whenever possible. Corpus tags also must include <Payload> details.
- Primary tags must provide <SoftwareIdentity> @version information, and also <SoftwareIdentity> @versionScheme information whenever possible. Authoritative creators of primary tags are encouraged to provide <Payload> information, and also to include <Meta> attribute values needed to support metadata-based searching. Non-authoritative creators of primary tags are encouraged to provide <Evidence> information for any data used to detect the presence of the product.
- Patch tags should document all files updated, removed, or added by the patch.
- Supplemental tags may supplement any type of tag, but must be explicitly linked to the supplemented tag. Any data value supplied in a supplemental tag that conflicts with a corresponding data value in the supplemented tag is ignored.

6 SWID Tag Usage Scenarios

Enterprises are concerned with managing the security of *endpoints*, devices connected to their networks. Knowing what software products are installed on these endpoints, which vulnerabilities may be present in those products, and how to define policies around the management of software, are critical to achieving a variety of cybersecurity objectives. This section presents a set of usage scenarios (US) that illustrate how, based on the guidelines provided in Sections 4 and 5 of this document, security professionals can achieve three important cybersecurity objectives:

1. Minimize exposure to publicly disclosed software vulnerabilities (see §6.1),
2. Enforce organizational policies regarding authorized software (see §6.2), and
3. Control network resource access from potentially vulnerable endpoints (see §6.3).

By using SWID tags in accordance with the guidelines provided in this report, the security practitioner (e.g., Chief Information Security Officer (CISO), Information System Security Officer (ISSO)) can achieve these objectives quickly, accurately, and efficiently. Sections 6.1 through 6.3 each describe the cybersecurity objective to be achieved, followed by specific usage scenarios that contribute to achieving the objective. Section 6.4 describes how the guidelines presented in this report enable each scenario.

6.1 Minimizing Exposure to Publicly Disclosed Software Vulnerabilities

This section presents usage scenarios illustrating how SWID tags may be used by security practitioners to minimize risks from exploitation of endpoints with known vulnerabilities within enterprise networks. To minimize these risks, security practitioners need to maintain awareness of vulnerabilities related to installed software, especially those vulnerabilities for which a patch or other remediation has not been made available. Security practitioners also need to maintain awareness of changes to the software inventory on each endpoint, since each change could (intentionally or inadvertently) introduce new vulnerabilities. For example, a user might unintentionally roll back a patch that mitigates a critical vulnerability.

This section presents three usage scenarios related to this cybersecurity objective:

- US 1 – Continuously Monitoring Software Inventory (see §6.1.1),
- US 2 – Ensuring that Products are Properly Patched (see §6.1.2), and
- US 3 – Identifying Vulnerable Endpoints (see §6.1.3).

6.1.1 US 1: Continuously Monitoring Software Inventory

In this scenario, SWID tags are used to continuously monitor the inventory of software installed on endpoints within an enterprise network. Tags are key to the process of gathering and maintaining an up-to-date and accurate accounting of software inventory on each endpoint. SWID data may be aggregated, if needed, in regional and/or enterprise-wide repositories. Using this data, organizations are able to maintain an ongoing understanding of installed software inventory by continuously monitoring software change event notifications. Information provided by SWID tags contributes to an up-to-date and accurate understanding of the software on endpoints. As software changes are made, the endpoint's software inventory is updated to reflect

those changes. Modifications including installing, upgrading, patching, and removing software occur throughout the software lifecycle.

One or more software discovery or monitoring tools (referred to generically in this section as "discovery tools") can continuously monitor endpoints for software changes, either on an event-driven basis or through periodic assessment of installation locations. These tools discover changes, including modifications to existing SWID tags on the endpoint. This analysis should consider various sources for performing this discovery (see §2.3.1 for a discussion of SWID tag placement on devices), including:

- Local filesystems directly attached to the endpoint, such as files installed by traditional installation utilities and archived distributions (e.g., tar, zip);
- Temporary storage connected to the endpoint (e.g., external hard drives, Universal Serial Bus (USB) devices);
- Software contained in native package installers (e.g., RPM Package Manager (RPM)); and
- Shared filesystems (e.g., a mapped network drive or network-attached storage) that contain software that is executable from an endpoint.

SWID tags provide identification, metadata, and relationship information about an endpoint's installed software. Authoritative tags discovered on an endpoint can supply reliable and comprehensive information about installed software, whereas discovery tools can place non-authoritative SWID tags on the endpoint to leave a record of newly-discovered, untagged products. This is an important capability, since it is likely that some software will be untagged at the time of installation.

As the tools collect the data, SWID tags enable many reporting capabilities for enterprise system software inventories. SWID tags can be aggregated to one or more repositories (e.g., regional or enterprise) to enable accurate analysis and reporting of the software products installed on a set of organizational endpoints. This aggregation supports the exchange of normalized data pertaining to these products, an important component of effectively managing IT across an enterprise. SWID tags provide a vendor-neutral and platform-independent way to analyze the state of installed software (e.g., software installed, products missing, or software in need of patching) within the organization, and to monitor endpoints for the purpose of maintaining continual awareness of their security posture.

6.1.1.1 Initial Conditions

This usage scenario assumes the following conditions:

- A software discovery tool is installed on each enterprise-managed endpoint, and is configured to run on a defined schedule, on request, and/or in response to events generated on the endpoint, which may indicate there has been a change to the installed software inventory.
- The discovery tool records inventory data in a configuration management database (CMDB), which may or may not be co-resident on the endpoint.

- The CMDB retains information about products (and their associated tags, if any), which have been discovered in the past on each endpoint. The discovery tool is able to use the CMDB to compare current inventory state to the last known state in order to detect changes.
- At the time the discovery process is run on each endpoint, the discovery tool has sufficient access rights to the endpoint to discover each installed software instance and any associated metadata. This includes access rights to read SWID tags on the endpoint.
- Some installed software products might not have an associated SWID tag because an authoritative source did not furnish one.

6.1.1.2 Process

1. Upon detecting new or changed software in an installation location or in a filesystem mounted on the endpoint, the discovery tool will collect and process all SWID tags (primary, supplemental, and/or patch tags) present in that location. Changes to be detected may include:
   - New software products (or subcomponents) that were not previously recorded in the inventory,
   - Changes or updates to installed software products discovered previously, and
   - New or modified SWID tags, as indicated by a new `@tagId` or `@tagVersion` attribute value within the `<SoftwareIdentity>` element.
2. The discovery tool will update the CMDB with the data from newly discovered or changed SWID tags, creating new entries and/or modifying existing entries, as needed, for installed products and their related software components. Because the software version information is critical for understanding the related configuration and potential vulnerabilities of the software installed on the endpoint, if any primary tag contains such version information (using the `<SoftwareIdentity>` element's `@version` and `@versionScheme` attributes), then that information will be recorded (see **PRI-2, PRI-3, PRI-4, PRI-5**).

   The discovery tool will likely validate accessed tags using the SWID tag schema (see §3) to ensure that the tag conforms to the SWID specification. If any tags are identified as not being in compliance with the SWID specification (see **GEN-1**), those tags will not be recorded in the CMDB, since they may not be reliable for the purpose of software inventory.

   The discovery tool will consult the `<SoftwareIdentity>` `@xml:lang` attribute to determine the default human language used for expressing language-dependent values within the tag (see **GEN-2** and **GEN-3**). It will also collect any furnished metadata for analysis and reporting purposes (see **PRI-13**).
3. The tool will determine the type of tag discovered, using the `<SoftwareIdentity>` `@corpus`, `@patch`, and `@supplemental` attributes (see **PRI-1, COR-1, PAT-1, SUP-1**). The discovery tool will read any payload information provided within the tag, including the file names, sizes, and cryptographic hashes for each component of the software product. These values can later be used to perform file integrity verification (see **GEN-14, GEN-15, GEN-16, GEN-17, GEN-18, GEN-19, GEN-20, GEN-21, GEN-22, GEN-23, GEN-24, PRI-6, PRI-7, PRI-8, PRI-9, PRI-10, PRI-11, PRI-12, PAT-2, PAT-3, PAT-4)**.

4. The discovery tool will attempt to determine if the tag is authoritative by checking that the `@regid` of the `<Entity>` element containing the `@role` value "`tagCreator`" also contains the `@role` value of "`softwareCreator`", "`aggregator`", "`distributor`", or "`licensor`" (see **GEN-4**, **GEN-5**, **GEN-6**, **GEN-7**, **GEN-8, GEN-9, GEN-10**).

5. If a tag was not installed with the software, the discovery tool will create and deploy a non-authoritative tag to the endpoint for each instance of a discovered application. As an alternative, the discovery tool may be able to determine that a previously generated non-authoritative tag already exists in the CMDB which describes the newly-discovered product. This is possible if (1) the CMDB records all tags—authoritative and non-authoritative—discovered or deployed anywhere within the enterprise, and (2) the discovery tool is able to use information about the newly-discovered product to retrieve a matching tag from the CMDB. In this case, the discovery tool may simply deploy the matching tag from the CMDB (or an appropriate related source) to the endpoint, rather than generate and deploy a completely new non-authoritative tag.

   Information about the files discovered is important to support continuous monitoring for software vulnerabilities, so the deployed tag will list every machine instruction file comprising the software product discovered (see §5.2.3), using the `<Evidence>` element (see **PRI-7, PRI-8**). This information will include filenames, sizes, versions, and cryptographic hashes discovered (see **GEN-14, GEN-15, GEN-17, GEN-18, GEN-20, GEN-21, GEN-22, GEN-23, GEN-24**). It will also include any version information determined for the software product (see **PRI-4, PRI-5).**

   When SWID tags are discovered that do not conform to the 2015 release of the SWID specification, these tags are not stored in the CMDB, but their contents might still be useful to support the evidence collected above.

6. Many cybersecurity decisions will be based upon the authenticity and integrity of the SWID tags discovered. To validate the integrity of a discovered tag, the discovery tool can authenticate the certificate in the digital signature, validate the signature, and use the `@thumbprint` attribute of the `<Entity>` element (see §3.2) to ensure that the tag signer is also the tag creator.

7. The discovery tool will read the tag identifier (i.e., `@tagId`) and identify the tag location, along with the type of tag discovered or created: primary tags for installed software (see §2.1.2**)**, and patch tags for software patches (see §2.1.3). Supplemental tags can provide additional information and may be useful for inventory. If the tag identifier already exists in inventory, the discovery tool will determine if the tag version has changed by examining the value associated with the `<SoftwareIdentity>` element's `@tagVersion` attribute. If that tag version has been updated, the tool will register the updated values that were changed in the SWID tag (see **GEN-25, GEN-26**).

8. The CMDB will be updated, including sending notifications to applicable reporting systems in the enterprise. The CMDB will track the changes discovered to support SAM and security needs. This includes the location of discovered tags to enable subsequent extraction of the information contained in each tag when needed.

Periodically, the complete set of tags from each endpoint is either sent to the enterprise repository, or collected via a remote management interface by the discovery tool to create a baseline software inventory. Once this baseline inventory has been established, only software changes since the last exchange need to be provided and may be supplemented with a periodic full refresh. This provides for efficiencies in the velocity and volume of information that needs to be exchanged.

9. For a given endpoint, the discovery tool iterates through each tag in the repository, including non-authoritative SWID tags.

10. The endpoint-collected tags are added to the enterprise repository, recording relevant endpoint identification information (e.g., host name, IP addresses), the date and time of the data collection, and data about the discovery tool or remote management interface used.

11. The discovery tool will record relationships between tags, as indicated within the SWID tags discovered. For example, patch tags include a reference (using the `<Link>` element's `@href` and `@rel` attributes) to the software being modified (see **GEN-11, GEN-12, GEN-13**). Similarly, for supplemental tags recorded, the discovery tool will indicate the tag identifier for the primary tag of the software for which additional information is being provided (see **SUP-2**). If any data in the supplemental tag conflicts with the data in any tag it supplements, the data in the supplemented tag is considered the correct value (see **SUP-3**).

6.1.1.3 Outcomes

The process described above provides an accurate and automated method for collecting identifying and descriptive metadata about an endpoint's inventory of installed software. When used in this way, SWID tags enable the collection of a comprehensive inventory of installed software products by examining the system for SWID tags rather than attempting to infer inventory information by examining arbitrary indicators on the endpoint (e.g., registry keys, installed files).

SWID tags contribute to a reliable software inventory by supporting searching for specific product information or software characteristics (e.g., prohibited or required software, specific software versions or version ranges, software from a specific vendor). The SWID specification provides a rich set of data that may be used with specific query parameters to search for instances of installed software. In addition to the common name and version values, many SWID tags store extended information such as data identified through the `<Link>` and `<Meta>` elements. Details regarding attributes and values that can be useful for queries are described in Sections 3.1.4 and 3.1.5.

As an indirect result of maintaining a SWID tag-based inventory, the discovery tools can dynamically identify vulnerabilities and misconfigurations. For example, upon discovering a newly installed or changed software application, the discovery tool can check the configuration of that software using a pre-defined checklist. The discovery tool could also check for any known vulnerabilities for that new or updated product. If the tool identifies a misconfiguration or a software vulnerability, that condition may be reported for mitigation.

In many cases, the ability to consistently search for instances of installed software is important to achieving the organization's cybersecurity situational awareness goals. Query results may be used to trigger alerts based on pre-determined conditions (e.g., prohibited software detected) that may be useful in a continuous monitoring context. The practitioner is able to know what is installed and where it is installed, providing a critical foundation for other usage scenarios.

6.1.2 US 2: Ensuring that Products are Properly Patched

Enterprise security managers often need to quickly and easily generate reports about endpoints having installed software products that are missing one or more patches, as this may signal vulnerability to malicious activity. If a discovery tool also has a patch management capability, it will need to determine that all prerequisite patches are installed before installing any new patches. While this usage scenario focuses on an enterprise patch management approach, a local patch management capability that is executed on an individual endpoint can also directly read the inventory of patch tags from the local repository to enable localized patch verification and decision making.

6.1.2.1 Initial Conditions

This usage scenario assumes the existence of an enterprise repository, populated with SWID tags that are created and collected using the process described in US 1 (see §6.1.1). This includes application of guidelines **GEN-1** through **GEN-26, PRI-1** through **PRI-13**, and **SUP-1** through **SUP-3**.

6.1.2.2 Process

1. Through a dashboard or other internal process, the discovery tool determines that a given software product needs to be patched (e.g., for a functional update, due to a discovered vulnerability).

2. If the tag identifier of the required patch is known, the discovery tool searches through the patch tags recorded in the repository for records indicating that a patch tag with the designated identifier is installed on an endpoint. If the patch tag identifier is unknown, the discovery tool will search for patch tags with a name that matches the `<SoftwareIdentity>` `@name` of the desired patch.

3. The discovery tool then examines the patch tag to determine whether any other required predecessor patches are also present. This is done by inspecting embedded `<Link>` elements where the `@role` attribute value is "`requires`" (per §5.3.3 of the SWID specification), then confirming the presence of the target tag. If there is no such requirement, or if the required patches are also confirmed as installed on the endpoint, the endpoint is recorded as properly patched for this instance.

4. If desired, the discovery tool can validate each file expected to be added, modified, or removed by the given patch(es). Patch tags created in accordance with the guidelines in Section 5.3 (see **PAT-1**, **PAT-2, PAT-3, PAT-4**) require that the tag clearly specify the files that are modified as a result of applying the patch. The discovery tool enumerates each of the files shown as added or modified within the `<Payload>` element of a patch tag as indicated by the `@n8060:patchEvent` attribute. The tool compares the recorded filename and

cryptographic hash with the actual files that reside on the endpoint. The discovery tool can also confirm deletion of those files that the patch tag indicates should have been removed.

5. Where a patch is noted as missing, the discovery tool can take advantage of relationships to other patches, as described in Section 5.3.3 of the SWID specification, to see if that patch has been superseded by a newer patch. In this case, the discovery tool can examine known patch tags for any that are known to supersede the desired patch, noting that the former patch may no longer apply.

6. The search results are provided through the discovery tool's dashboard and/or reporting process.

6.1.2.3 Outcomes

The discovery tool user is able to accurately and quickly identify instances where a required patch or update is not installed on a given endpoint. If patched files are also assessed by taking advantage of `<Payload>` or `<Evidence>` elements contained in patch tags, the user is able to verify patch installations. The user is able to determine which endpoints meet (or do not meet) specific patch requirements, supporting security situational awareness and patch/vulnerability management as part of a continuous monitoring solution.

6.1.3 US 3: Identifying Vulnerable Endpoints

The remediation of known vulnerabilities through timely patching is considered a vulnerability management best practice. SWID tags improve vulnerability management by providing comprehensive, compact descriptions of installed software and patches, which may then be compared and correlated with vulnerability information.

Because SWID tags adhere to a consistent and standardized structure, they aid automated correlation of information published by vulnerability information sources (e.g., NIST's National Vulnerability Database, US-CERT [United States Computer Emergency Readiness Team] alerts, vulnerability advisories issued by vendors and independent security analysts) with the inventory information collected by discovery tools. Many vulnerability bulletins use the CPE specification to identify classes of products that are affected by a vulnerability [CPE23N]. [NISTIR 8085] describes a method to form CPE names automatically from input SWID tags. This capability can be used to translate a software inventory based on SWID tags to one based on CPE names. Given a vulnerability bulletin that references products using CPE names, this translation can then be used to identify potentially vulnerable endpoints.

If a tag creator uses the appropriate `<Meta>` attributes to specify additional detailed naming information in a product's primary tag (see §5.2.4), this information becomes readily available to publishers of vulnerability bulletins. By including appropriate references to those attribute values, bulletins make it easier for consumers to accurately search SWID-based inventory data for affected products. For example, if the presence or absence of a product vulnerability depends on software edition information, it is advantageous both for tag creators to specify the `<Meta>` `@edition` attribute, and for publishers of vulnerability bulletins to reference that value explicitly.

6.1.3.1 Initial Conditions

This usage scenario assumes the existence of an enterprise repository populated with SWID tags that are created and collected using the process described in US 1 (see §6.1.1). This includes application of guidelines **GEN-1** through **GEN-24, PRI-1** through **PRI-13**, and **SUP-1** through **SUP-3**.

6.1.3.2 Process

1. Using product advisories containing information about publicly disclosed software vulnerabilities, the discovery tool searches for endpoints on which the referenced software is installed. The search criteria may include SWID tag information such as the information provided in the primary tag `<SoftwareIdentity>` `@name` and `@version` (see **PRI-2**, **PRI-3, PRI-4, PRI-5**), and `<Meta>` `@revision` and `@edition` (see **PRI-13**). Additionally, by forming CPE names from SWID tags (see [NISTIR 8085]), the discovery tool can search for endpoints with software referenced by those CPE names included in the vulnerability bulletins.

2. If the bulletin references the tag identifier for the relevant tag for a software product or patch, the discovery tool will search for that identifier. SWID tags adhering to guidelines **PRI-1** through **PRI-5** enable the discovery tool to automatically and accurately correlate inventory and vulnerability data.

3. If the bulletin only references one or more known filename(s), but does not identify the software product itself, it will be necessary to search for software products and patches that include the file(s). Guidelines **PRI-6** through **PRI-12** recommend that filename information be captured in the `<Payload>` and/or `<Evidence>` elements of SWID tags to support this type of query. As a result, the discovery tool can search the `<Payload>` and/or `<Evidence>` portions of recorded tag information in the repository to look for software and patches of interest.

   For example, to identify instances of the "Heartbleed" bug,[6] the tool might search for any tags where the `<Payload>` and/or `<Evidence>` portions of recorded tags contain references to the vulnerable OpenSSL library. Products including this library can be identified and then those products can be searched for to identify vulnerable software installations.

4. Where a record exists that matches the query parameters, as described above, the associated endpoint is flagged as containing vulnerable software.

5. Where patch tag information is provided in the bulletin, the discovery tool queries the repository to determine whether the appropriate patch tag has been installed (see US 2, §6.1.2), including checks for predecessor or superseded patches.

[6] More information about the Heartbleed bug is available from www.heartbleed.com.

6. If the endpoint is found to contain vulnerable software but not the associated patch(es), the endpoint may be flagged as potentially in need of remediation activities.

Consider the case of a fictional vulnerability involving a known buffer overflow in the product named Acme Roadrunner, affecting versions between 11.1 and 11.7 and between 12.0 and 12.1 (inclusive). The issue is remediated in version 12.2 and later. There is also a patch KB123 that remediates the vulnerability. The discovery tool can review the collected SWID tags for the endpoint, searching for installed software instances that match:

`<SoftwareIdentity> @name="Acme Roadrunner"` and either:
whose major version is 11 and minor version is greater than or equal to 1; or
whose major version is 12 and minor version is less than 2.

And also the presence of the following in the software inventory:
`<SoftwareIdentity> @name="Acme_Roadrunner_KB123".`

Upon discovering a SWID tag that indicates the installation of a vulnerable version of the Acme Roadrunner product (e.g., Acme Roadrunner version 11.5), the discovery tool searches through the repository and discovers a patch tag named "`Acme_Roadrunner_KB123`" associated with that endpoint.

Given the above scenario, the discovery tool reports that the endpoint contains software with a known vulnerability, but the vulnerability appears to have been patched. This information can be reported for security situational awareness, also supporting security analysis.

6.1.3.3 Outcomes

Through the use of SWID tags for the description and discovery of vulnerable software, organizations are able to identify known vulnerabilities within their enterprise based on SWID-based software inventory data and published vulnerability bulletins.

6.2 Enforcing Organizational Software Policies

This cybersecurity objective area focuses on the use of SWID tags to help security practitioners minimize security risk by enforcing enterprise policies regarding authorized software. These policies may be implemented as blacklists (lists of prohibited products, with all unlisted products implicitly allowed) or whitelists (lists of allowed products, with all others prohibited). In addition, specific products may be designated as mandatory by the enterprise (e.g., antivirus and intrusion detection and prevention applications), possibly depending on the endpoint's role (e.g., end-user workstations versus Internet-facing web servers). Policies may be enforced at time of (attempted) product installation, and/or any time thereafter.

This section presents two usage scenarios:

- US 4 – Preventing Installation of Unauthorized or Corrupted Software (see §6.2.1), and
- US 5 – Discovering and Preventing Corrupted Software Execution (see §6.2.2).

6.2.1 US 4: Preventing Installation of Unauthorized or Corrupted Software

To strictly control what software may or may not be installed on enterprise endpoints, corpus tags may be used whenever a software installation is initiated to confirm that the software to be installed either is included in a product whitelist, or is not included in a product blacklist. There might be multiple lists of authorized or unauthorized software. For example, Windows clients might have a list, Mac OS X clients another, and Linux servers yet another. Similarly, endpoints that are dedicated to a particular role (e.g., "public-facing webserver") might have unique lists of allowed or prohibited software.

As described in Section 2.1.1, corpus tags may be used to authenticate the issuer of a software product installation package before carrying out the installation procedure. Identification information and other data in a corpus tag can be used to authorize or prohibit software installation during the installation procedure. Additionally, if a manifest of the installation files is included in the corpus tag (see §3.1.6 on the `<Payload>` element), the installation routine can confirm (from the whitelist) that the software's integrity has been preserved, preventing unauthorized modifications to software distributions.

6.2.1.1 Initial Conditions

This usage scenario assumes that the following conditions exist:

- A software product/package to be installed includes a corpus tag describing what will be installed.
- The software installation system is policy-aware, in that it is able to consult organization-specific whitelists/blacklists for the purpose of restricting software installations only to allowed products.
- If the issuer of the installation package is to be verified, the corpus tag must be digitally signed.

6.2.1.2 Process

1. Upon initiation of a software installation on an endpoint, the policy-aware installation system discovers a corpus tag included with the distribution package of the software product that was requested to be installed. If no corpus tag is found, some installation systems may be configured by enterprise policy to prevent the installation, ending this process.

2. If provided, the installation tool may examine the value of the `@thumbprint` attribute(s) of the `<Entity>` element of the signer, comparing it to a hexadecimal string that contains a hash of the signer's certificate. This allows each digital signature to be directly related to the entity specified. The installation tool validates the signer's certificate and the tag's signature if the corpus tag is signed. If the signature is found to be invalid, the installation may be prevented, ending this process.

3. The installation tool determines whether the `@tagId` or other data included in the tag (see **GEN-1**, **COR-1** through **COR-4)** matches the criteria specified in either a whitelist or a blacklist. If the evaluation of the tag is determined to be in violation of the whitelist or blacklist policy, then installation is prevented, ending this process.

4. The installation tool verifies the cryptographic hashes of the installation files using any `<Payload>` data included in the corpus tag (see **COR-4**, **GEN-14, GEN-15, GEN-16, GEN-18, GEN-19, GEN-21** through **GEN-24**). If any hash is found not to match, the installation is prevented, ending this process.

6.2.1.3 Outcomes

For the process described above, the application of SWID tags enables the organization to use automation to control installation of software and patches, and to verify the signer and integrity of each installation package prior to installation.

6.2.2 US 5: Preventing the Execution of Corrupted Software

Similar to US 4 described above, effective software asset management includes the ability to discover potentially compromised software that has already been installed on an endpoint. This is accomplished by comparing the known hash values of installed software/packages (as recorded in the local endpoint repository and/or the enterprise repository) to the files actually observed on the endpoint.

Detection of software tampering may be used for several purposes, including the following:

- To report potential compromise of an endpoint,
- To quarantine an endpoint pending further investigation, and
- To prevent execution of an application that shows signs of unauthorized alteration.

Organizations are encouraged to take advantage of this capability by using SWID tags to convey important information about the characteristics of installed software. Specifically, the ability to store and compare cryptographic hashes of installed executable software is a useful method to identify potential tampering or unauthorized changes.

This usage scenario provides an example of the benefit of a local repository that works in concert with an enterprise repository. The local endpoint is able to perform a comparison of the recorded cryptographic hash to the observed local file quickly enough to enable such a check on demand. Because some legacy cryptographic hash algorithms are easily spoofed, the use of a stronger methodology, as described in Section 4.6.2, will help provide confidence in the findings. Comparison of observed hash values with recorded values in the enterprise repository requires additional network and computing resources, and is more commonly performed as a periodic monitoring task.

6.2.2.1 Initial Conditions

This usage scenario assumes the existence of an enterprise repository populated with SWID tags that are created and collected using the process described in US 1 (see §6.1.1). This includes application of guidelines **GEN-1** through **GEN-26, PRI-1** through **PRI-12**, and **SUP-1** through **SUP-3**.

6.2.2.2 Process

1. For each endpoint, the discovery tool reads each primary and supplemental SWID tag (see **PRI-1**, **SUP-1**, **SUP-2**, **SUP-3**), examining the stored cryptographic hashes for each file

listed in the <Payload> or <Evidence> elements. This information may be collected and included in the SWID tag as described in guidelines **PRI-6** through **PRI-12** to provide detailed information about the files comprising an installed software product. Detailed information regarding the creation and comparison of cryptographic hashes with sufficient security strength is described in Section 4.6.2.

Similarly, the discovery tool examines each patch tag to gather the cryptographic evidence for files added or changed by that patch (see **PAT-1**, **PAT-2, PAT-3, PAT-4**).

2. The discovery tool calculates the current cryptographic hash of the actual files on those endpoints using the same algorithm as originally used in the SWID tags.

3. If any calculated file hash does not match the one provided in the SWID tag, the discovery tool reports the variance.

4. The tool may also, based upon the detection of potential tampering, prevent execution of that software product.

6.2.2.3 Outcomes

Identifying improperly altered executable files in an automated, accurate, and timely manner supports an organization's ability to prevent execution of files that have been infected by malware or otherwise modified without authorization.

6.3 US 6: Preventing Vulnerable Devices from Accessing Network Resources

A forward-looking approach to improving organizations' cybersecurity is to prevent potentially vulnerable devices from connecting to the network, or to move such devices to an isolated network segment for remediation or investigation. Currently, products are available that achieve this through proprietary methods, and groups are working on open standards to accomplish this goal. For example, the Trusted Computing Group's Trusted Network Connect Working Group has defined an open solution architecture that enables network operators to enforce policies regarding the security state of devices in order to determine whether to grant access to a requested network infrastructure. The use of SWID tags provides a technology-neutral way to verify a device's compliance with certain configuration policies (e.g., updated antivirus definitions, configuration compliance with baseline specifications) and safeguards against known software vulnerabilities (e.g., missing patches).

In cases where a discovery tool is able to detect that a device is providing inventory data that conflicts with known information, that condition may indicate that malicious software on that device is intentionally misrepresenting the device's condition. Such a conflict might provide an indicator of compromise that could trigger further investigation.

6.3.1 Initial Conditions

This usage scenario assumes the existence of an enterprise repository populated with SWID tags that are created and collected using the process described in US 1 (see §6.1.1). This includes application of guidelines **GEN-1** through **GEN-26, PRI-1** through **PRI-12**, and **SUP-1** through **SUP-3**.

6.3.2 Process

1. An endpoint attempts to access a given network resource, such as an enterprise wide area network (WAN) or a protected website.

2. Using information from the local SWID tag repository, a trusted client on the endpoint collects the information needed to support a network access decision. Examples of the information to be collected include:

   - The inventory of installed software and patches (see **PAT-1**)
   - File metadata, including names, hashes, sizes, and versions (see **PAT-2**, **PAT-3, PAT-4)**

3. The client securely transfers this information to a policy decision point.

4. The policy decision point renders a decision based on the provided data, and issues a network access recommendation to a policy enforcement point (e.g., a network switch). The policy enforcement point might allow the endpoint on the network, place it on a restricted network, or quarantine the endpoint in order to remedy a potential risk (e.g., by updating patches or antivirus definitions). If an endpoint is assessed to be in violation of policy, but not to present a significant risk, it may be allowed on the network, but subjected to detailed monitoring.

   SWID-related data provided to a policy decision point may be used to determine:

   - If any known software vulnerabilities exist on the endpoint (see §6.1.1),
   - If the endpoint is properly patched (see §6.1.2), and
   - If the endpoint is in compliance with whitelist or blacklist requirements (see §6.2.1).

6.3.3 Outcomes

Through the use of well-formed SWID tags, network access decision points are able to collect validation information quickly and accurately, enabling the organization to help prevent the connection of endpoints that represent a potential threat.

6.4 Association of Usage Scenarios with Guidelines

To aid in the association of usage scenarios to the guidelines described in Sections 4 and 5, the guidelines are organized into sets of families, described below, which help aggregate items into relevant groupings. These items follow the coded identifier format described in the introduction to Section 4 and are grouped into the following categories:

- **GEN:** General guidelines applicable to all types of SWID tags,
- **COR:** Guidelines specific to corpus tags,
- **PRI:** Guidelines specific to primary tags,
- **PAT:** Guidelines specific to patch tags, and
- **SUP:** Guidelines specific to supplemental tags.

The guideline families are:

- **Basic Compliance** – Tags must be well-formed as described in the ISO 19770-2:2015 specification, so that they are interoperable among discovery tools (see **GEN-1**).

- **Tag Type** – Discovery tools need to be able to determine the type of SWID tag that is discovered or recorded. Corpus, primary, patch, and supplemental tags are mutually exclusive tag types that are readily distinguishable (see **COR-1, PRI-1, PAT-1, SUP-1**).
- **Internationalization –** To support an international audience, tag creators are able to provide language-dependent attribute values in region-specific human languages (see **GEN-2**, **GEN-3**).
- **Entity Identification** – SWID tag creators must concisely and accurately record entities, which have an important role in addressing potential interoperability issues that could arise when tag creators specify attribute values in multiple human languages (see **GEN-4** through **GEN-7**). Additionally, discovery tools need to be able to inspect a SWID tag and rapidly determine the authority of the tag creator, since authoritative tags are more likely to be accurate than non-authoritative tags (see **GEN-8** through **GEN-10**).
- **Tag Relationships** – Discovery tools need consistent reference information to be able to link a source tag to one or more target tags (see **GEN-11** through **GEN-13**). Because the SWID specification does not clearly state how a supplemental tag should indicate its linkage to other tags, it is important to follow guideline **SUP-2** to establish a consistent relationship between the tags.
- **Payload and Evidence** – Detailed information about the files comprising an installed software product is critical to help confirm that the correct files are installed and to verify the integrity of those files. This information is derived from the `<Payload>` or `<Evidence>` information included in the SWID tag (see **GEN-14** through **GEN-24**, and **PRI-6** through **PRI-12**). Software installation products can also use payload information to prevent installation of software from media that might be corrupted (see **COR-4)**. Patch tags need to comply with guidelines **PAT-2** through **PAT-4** to document all changes to files resulting from application of the patch.
- **Tag Maintenance** – While SWID tags are rarely changed, when this occurs (e.g., to correct an error) the tag identifier stays the same but the tag version needs to be incremented to note that there is updated SWID tag information available (see **GEN-25**, **GEN-26**).
- **Software Version Identification** – Many cybersecurity objectives depend upon the ability to accurately identify the name and version of software products in both pre- and post-installation states. It is important that the version of a software product is provided along with the version scheme so that software installers (see **COR-2**, **COR-3**) and discovery tools (see **PRI-2** through **PRI-5**) can quickly and accurately determine a software product's version. Proper use of well-known version schemes helps ensure that software versions are able to be parsed by tools supporting automation.
- **Metadata** – Software products are often known by colloquial identifiers as well as by formal version references. Where metadata applies to the software identified by a SWID tag, it helps accurately identify products and components installed (see **PRI-13**).
- **Data Deconfliction** - Supplemental tags are secondary to the primary, corpus, and patch tags they support; guideline **SUP-3** helps ensure that the original tag (the one being supplemented) has the correct and primary information in case of any conflict.

Table 4 illustrates the association among the usage scenarios and the previously described guidelines. The usage scenarios demonstrate the rationale for each of the guidelines to help organizations consistently achieve important cybersecurity objectives through the appropriate creation and usage of well-formed SWID tags.

Table 4: Relationship of Guidelines to Usage Scenarios

Guideline Family	Guideline	Usage Scenario					
		1	2	3	4	5	6
Basic Compliance	GEN-1	●	●	●	●	●	●
Tag Type	COR-1	●	●		●		
	PRI-1	●	●	●		●	●
	PAT-1	●	●	●		●	●
	SUP-1	●	●	●		●	●
Internationalization	GEN-2	●	●	●		●	●
	GEN-3[N]	◒	◒	◒		◒	◒
Entity Identification	GEN-4[A]	●	●	●		●	●
	GEN-5[N]	●	●	●		●	●
	GEN-6[N]	◒	◒	◒		◒	◒
	GEN-7	●	●	●		●	●
	GEN-8[A]	●	●	●		●	●
	GEN-9[A]	●	●	●		●	●
	GEN-10[N]	◒	◒	◒		◒	◒
Tag Relationships	GEN-11	●	●	●		●	●
	GEN-12	●	●	●		●	●
	GEN-13	●	●	●		●	●
Payload and Evidence	GEN-14	●	●	●	●	●	●
	GEN-15	●	●	●	●	●	●
	GEN-16[A]	●	●	●	●	●	●
	GEN-17[N]	◒	◒	◒		◒	◒
	GEN-18	◒	◒	◒	◒	◒	◒
	GEN-19[A]	●	●	●	●	●	●
	GEN-20[N]	◒	◒	◒		◒	◒
	GEN-21	○	○	○	○	○	○
	GEN-22	◒	◒	◒	◒	◒	◒
	GEN-23	◒	◒	◒	◒	◒	◒
	GEN-24	◒	◒	◒	◒	◒	◒
Tag Maintenance	GEN-25	◒	◒			◒	◒
	GEN-26	●	●			●	●

Guideline Family	Guideline	Usage Scenario					
		1	2	3	4	5	6
Software Version Identification	COR-2				●		
	COR-3				●		
Payload and Evidence	COR-4				●		
Software Version Identification	PRI-2[A]	●	●	●			●
	PRI-3[A]	●	●	●			●
	PRI-4[N]	●	●	◒			●
	PRI-5[N]	●	●	◒			●
Payload and Evidence	PRI-6[A]	◒	◒	◒		◒	◒
	PRI-7[N]	◒	◒	◒		◒	◒
	PRI-8	◒	◒	◒		◒	◒
	PRI-9[A]	●	●	●		●	●
	PRI-10[N]	●	●	●		●	●
	PRI-11[A]	●	●	●		●	●
	PRI-12[N]	◒	◒	◒		◒	◒
Metadata	PRI-13	●	●	●			
Payload and Evidence	PAT-2[A]	◒	◒			◒	◒
	PAT-3[A]	●	●			●	●
	PAT-4[N]	◒	◒			◒	◒
Tag Relationships	SUP-2	●	●			●	●
Data Deconfliction	SUP-3	●	●			●	●

Symbol indicates: ● mandatory guideline, ◒ recommended guideline, ○ optional guideline

Appendix A—Acronyms

Selected acronyms and abbreviations used in this report are defined below.

API	Application Programming Interface
CA	Certificate Authority
CISO	Chief Information Security Officer
CMDB	Configuration Management Database
CPE	Common Platform Enumeration
DSS	Digital Signature Standard
FIPS	Federal Information Processing Standards
HTML	Hypertext Markup Language
IANA	Internet Assigned Numbers Authority
IEC	International Electrotechnical Commission
IEEE	Institute of Electrical and Electronics Engineers
IETF	Internet Engineering Task Force
IP	Internet Protocol
ISO	International Organization for Standardization
ISSO	Information System Security Officer
ITL	Information Technology Laboratory
NAS	Network Attached Storage
NIST	National Institute of Standards and Technology
NISTIR	National Institute of Standards and Technology Internal Report
RFC	Request for Comments
RPM	RPM Package Manager
SAM	Software Asset Management
SHA	Secure Hash Algorithm
SHS	Secure Hash Standard
SP	Special Publication
SWID	Software Identification
URI	Uniform Resource Identifier
U.S.	United States
US	Usage Scenario
USB	Universal Serial Bus
US-CERT	United States Computer Emergency Readiness Team
W3C	World Wide Web Consortium
WAN	Wide Area Network
XML	Extensible Markup Language
XPath	XML Path Language
XSD	XML Schema Definition

Appendix B—References

[CPE23N] Cheikes, B. A., Waltermire, D., and Scarfone, K. *Common Platform Enumeration: Naming Specification 2.3*. National Institute of Standards and Technology Interagency Report 7695, August 2011. http://csrc.nist.gov/publications/nistir/ir7695/NISTIR-7695-CPE-Naming.pdf [accessed 2/4/2016].

[FIPS180-4] U.S. Department of Commerce. *Secure Hash Standard (SHS)*, Federal Information Processing Standards (FIPS) Publication 180-4, August 2015. http://dx.doi.org/10.6028/NIST.FIPS.180-4.

[FIPS186-4] U.S. Department of Commerce. *Digital Signature Standard (DSS)*, Federal Information Processing Standards (FIPS) Publication 186-4, July 2013. http://dx.doi.org/10.6028/NIST.FIPS.186-4.

[ISO/IEC 19770-2:2009] International Organization for Standardization/International Electrotechnical Commission, *Information technology -- Software asset management -- Part 2: Software identification tag*, ISO/IEC 19770-2:2009, November 2009. http://www.iso.org/iso/catalogue_detail?csnumber=53670 [accessed 2/4/2016].

[ISO/IEC 19770-2:2015] International Organization for Standardization/International Electrotechnical Commission, *Information technology -- Software asset management -- Part 2: Software identification tag*, ISO/IEC 19770-2:2015, October 2015. http://www.iso.org/iso/catalogue_detail?csnumber=65666 [accessed 2/19/2016].

[ISO/IEC 19770-5:2013] International Organization for Standardization/International Electrotechnical Commission, *Information technology -- Software asset management -- Part 5: Overview and vocabulary*, ISO/IEC 19770-5:2013, 2013. https://www.iso.org/obp/ui/#iso:std:iso-iec:19770:-5:ed-1:v1:en [accessed 2/4/2016].

[NISTIR 7802] Booth, H., and Halbardier, A. (2011). *Trust Model for Security Automation Data 1.0*. National Institute of Standards and Technology Interagency Report 7802, 2011. http://csrc.nist.gov/publications/nistir/ir7802/NISTIR-7802.pdf [accessed 2/4/2016].

[NISTIR 8085] Cheikes, B. A., and Waltermire, D., *Forming Common Platform Enumeration (CPE) Names from Software Identification (SWID) Tags*. National Institute of Standards and Technology Interagency Report 8085 (Draft), December 2015. http://csrc.nist.gov/publications/PubsNISTIRs.html#NISTIR8085 [accessed 2/4/2016].

[RFC 2119] Bradner, S. *Key words for use in RFCs to Indicate Requirement Levels*. Internet Engineering Task Force (IETF) Network Working Group Request for Comments (RFC) 2119, March 1997. https://dx.doi.org/10.17487/rfc2119.

[RFC 3490] Faltstrom, P., Hoffman, P., and Costello, A. (2003). *Internationalizing Domain Names in Applications (IDNA).* Internet Engineering Task Force (IETF) Network Working Group Request for Comments (RFC) 3490, March 2003. https://dx.doi.org/10.17487/rfc5234.

[RFC 3986] Berners-Lee, T., Fielding, R., and Masinter, L. *Uniform Resource Identifier (URI): Generic Syntax*. Internet Engineering Task Force (IETF) Network Working Group Request for Comments (RFC) 3986, January 2005. https://dx.doi.org/10.17487/rfc3986.

[RFC 5234] Crocker, D., and Overell, P. *Augmented BNF for Syntax Specifications: ABNF*. Internet Engineering Task Force (IETF) Request for Comments (RFC) 5234, January 2008. https://dx.doi.org/10.17487/rfc5234.

[SEMVER] Preston-Werner, T. *Semantic Versioning 2.0.0.* http://semver.org/spec/v2.0.0.html [accessed 2/4/2016].

[SP800-107] Dang, Q. *Recommendation for Applications Using Approved Hash Algorithms*, NIST Special Publication (SP) 800-107 Revision 1, National Institute of Standards and Technology, August 2012. http://dx.doi.org/10.6028/NIST.SP.800-107r1.

[SP800-57-part-1] Barker, E. et al. *Recommendation for Key Management – Part 1: General*, NIST Special Publication (SP) 800-57 Part 1 Revision 4, National Institute of Standards and Technology, January 2016. http://dx.doi.org/10.6028/NIST.SP.800-57pt1r4.

[W3C-langtags] Ishida, R. *Language tags in HTML and XML*. W3C Article, September 2009. http://www.w3.org/International/articles/language-tags [accessed 2/4/2016].

[xmldisg-core] Bartel, M. et al. *XML Signature Syntax and Processing (Second Edition)*. World Wide Web Consortium (W3C) Recommendation, June 2008. http://www.w3.org/TR/xmldsig-core/ [accessed 2/4/2016].

[XPATH 2.0] World Wide Web Consortium (W3C) XML Path Language (XPath) 2.0 (Second Edition), December 2010. http://www.w3.org/TR/xpath20 [accessed 2/4/2016].