CycloneDX

Authoritative
Guide to SBOM

Implement and optimize use of Software Bill of Materials

OWASP

Second Edition

Table of Contents

About the Guide

CycloneDX is a modern standard for the software supply chain.

The content in this guide results from continuous community feedback and input from leading experts in the software supply chain security field. This guide would not be possible without valuable feedback from the CycloneDX Industry Working Group (IWG), the CycloneDX Core Working Group (CWG), the many CycloneDX Feature Working Groups (FWG), Ecma International Technical Committee 54, and a global network of contributors and supporters.

Copyright and License

Second Edition, 09 April 2024

Version	Changes	Updated On	Updated By
Second Edition	Updated for CycloneDX v1.6	2024-04-09	CycloneDX Core Working Group
First Edition	Initial Release	2023-06-25	CycloneDX Core Working Group

Preface

Welcome to the Authoritative Guide series by the OWASP Foundation and OWASP CycloneDX. In this series, we aim to provide comprehensive insights and practical guidance, ensuring that security professionals, developers, and organizations alike have access to the latest best practices and methodologies.

At the heart of the OWASP Foundation lies a commitment to inclusivity and openness. We firmly believe that everyone deserves a seat at the table when it comes to shaping the future of cybersecurity standards. Our collaborative model fosters an environment where diverse perspectives converge to drive innovation and excellence.

In line with this ethos, the OWASP Foundation has partnered with Ecma International to create an inclusive, community-driven ecosystem for security standards development. This collaboration empowers individuals to contribute their expertise and insights, ensuring that standards like CycloneDX reflect the collective wisdom of the global cybersecurity community.

One standout example of this model is OWASP CycloneDX, which is on track to becoming an Ecma International standard through Technical Committee 54 (TC54). By leveraging the strengths of both organizations, CycloneDX is poised to become a cornerstone of security best practices, providing organizations with a universal standard for software and system transparency.

As you embark on your journey through this Authoritative Guide, we encourage you to engage actively with the content and join us in shaping the future of cybersecurity standards. Together, we can build a safer and more resilient digital world for all.

Andrew van der Stock
Executive Director, OWASP Foundation

The Innovative History of OWASP CycloneDX

OWASP CycloneDX has carved a legacy steeped in innovation, collaboration, and a commitment to openness. OWASP continues to advance software and system transparency standards, prioritizing capabilities that facilitate risk reduction.

March 2018

OWASP CycloneDX v1.0

First general-purpose, security-focused Bill of Materials standard supporting software and hardware components. Introduced the world to Package URL for use with software security use cases.

March 2019

OWASP CycloneDX v1.1

First specification with complete pedigree support describing component lineage and the commits, patches, and diffs which make a forked version unique.

May 2020

OWASP CycloneDX v1.2

First specification to incorporate SWID (ISO/IEC 19770-2:2015) and services into inventory including data classifications, providers, and relationships between services and components.

May 2021

OWASP CycloneDX v1.3

First specification to incorporate support for composition completeness surpassing NTIA's framing of "known unknowns".

January 2022

OWASP CycloneDX v1.4

First specification to introduce vulnerability sharing and transparency, including Vulnerability Disclosure Reports (VDR) and Vulnerability Exploitability eXchange (VEX).

June 2023

OWASP CycloneDX v1.5

First specification to support AI Transparency, configuration and data components, and formulation describing how components were created, tested, trained, evaluated, and deployed.

December 2023

Ecma TC54 Established

First working group chartered with holistic supply chain goals of standardizing core data formats, APIs, and algorithms that advance software and system transparency.

April 2024

OWASP CycloneDX v1.6

First specification to support cryptographic assets for Post-Quantum Cryptography (PQC) readiness and first general-purpose attestation specification to digitally transform audit and attestation workflows.

Source: https://tc54.org/history

Introduction

CycloneDX is a modern standard for the software supply chain. At its core, CycloneDX is a general-purpose Bill of Materials (BOM) standard capable of representing software, hardware, services, and other types of inventory. CycloneDX is an OWASP flagship project, has a formal standardization process and governance model through Ecma Technical Committee 54, and is supported by the global information security community.

Design Philosophy and Guiding Principles

The simplicity of design is at the forefront of the CycloneDX philosophy. The format is easily understandable by a wide range of technical and non-technical roles. CycloneDX is a full-stack BOM format with many advanced capabilities that are achieved without sacrificing the design philosophy. Some guiding principles influencing its design include:

- Be easy to adopt and easy to contribute to
- Identify risk to as many adopters as possible, as quickly as possible
- Avoid blockers that prevent the identification of risk
- Continuous improvement - innovate quickly and improve over time
- Encourage innovation and competition through extensions
- Produce immutable and backward-compatible releases
- Focus on high degrees of automation
- Provide a smooth path to specification compliance through prescriptive design

Defining Software Bill of Materials

The U.S. National Telecommunications and Information Administration (NTIA) defines software bill as materials as "*a formal, machine-readable inventory of software components and dependencies, information about those components, and their hierarchical relationships*." OWASP CycloneDX implements this definition and extends it in many ways, including adding services as a foundational component in a Software Bill of Materials.

The Role of SBOM in Software Transparency

Software transparency involves providing clear and accurate information about the components used in an application, including their name, version, supplier, and any dependencies required by the component. This information helps identify and manage the risks associated with the software whilst also enabling compliance with relevant regulations and standards. With the growing importance of software in our daily lives, transparency is critical to building trust in software and ensuring that it is safe, secure, and reliable.

SBOMs are the vehicle through which software transparency can be achieved. With SBOMs, parties throughout the software supply chain can leverage the information within to enable various use cases that would not otherwise be easily achievable. SBOMs play a vital role in promoting software transparency, allowing users to make informed decisions about the software they use.

High-Level SBOM Use Cases

A complete and accurate inventory of all first-party and third-party components is essential for risk identification. SBOMs should ideally contain all direct and transitive components and the dependency relationships between them.

CycloneDX far exceeds the Minimum Elements for Software Bill of Materials as defined by the National Telecommunications and Information Administration (NTIA) in response to U.S. Executive Order 14028.

Adopting CycloneDX allows organizations to quickly meet these minimum requirements and mature into using more sophisticated use cases over time. CycloneDX is capable of achieving all SBOM requirements defined in the OWASP Software Component Verification Standard (SCVS).

A few high-level use cases for SBOM include:

- Product security, architectural, and license risk
- Procurement and M&A
- Software component transparency
- Supply chain transparency
- Vendor risk management

xBOM Capabilities

CycloneDX provides advanced supply chain capabilities for cyber risk reduction. Among these capabilities are:

- Software Bill of Materials (SBOM)
- Software-as-a-Service Bill of Materials (SaaSBOM)
- Hardware Bill of Materials (HBOM)
- Machine Learning Bill of Materials (ML-BOM)
- Cryptography Bill of Materials (CBOM)
- Operations Bill of Materials (OBOM)
- Manufacturing Bill of Materials (MBOM)
- Bill of Vulnerabilities (BOV)
- Vulnerability Disclosure Report (VDR)
- Vulnerability Exploitability eXchange (VEX)
- CycloneDX Attestations (CDXA)
- Common Release Notes Format

Software Bill of Materials (SBOM)

SBOMs describe the inventory of software components and services and the dependency relationships between them. A complete and accurate inventory of all first-party and third-party components is essential for risk identification. SBOMs should ideally contain all direct and transitive components and the dependency relationships between them.

Software-as-a-Service BOM (SaaSBOM)

SaaSBOMs provide an inventory of services, endpoints, and data flows and classifications that power cloud-native applications. CycloneDX is capable of describing any type of service, including microservices, Service Orientated Architecture (SOA), Function as a Service (FaaS), and System of Systems.

SaaSBOMs complement Infrastructure-as-Code (IaC) by providing a logical representation of a complex system, complete with an inventory of all services, their reliance on other services, endpoint URLs, data classifications, and the directional flow of data between services. Optionally, SaaSBOMs may also include the software components that make up each service.

Hardware Bill of Materials (HBOM)

CycloneDX supports many types of components, including hardware devices, making it ideal for use with consumer electronics, IoT, ICS, and other types of embedded devices. CycloneDX fills an important role in between traditional eBOM and mBOM use cases for hardware devices.

Machine Learning Bill of Materials (ML-BOM)

ML-BOMs provide transparency for machine learning models and datasets, which provide visibility into possible security, privacy, safety, and ethical considerations. CycloneDX standardizes model cards in a way where the inventory of models and datasets can be used independently or combined with the inventory of software and hardware components or services defined in HBOMs, SBOMs, and SaaSBOMs.

Cryptography Bill of Materials (CBOM)

A Cryptography Bill of Materials (CBOM) describes cryptographic assets and their dependencies. Discovering, managing, and reporting on cryptographic assets is necessary as the first step on the migration journey to quantum-safe systems and applications. Cryptography is typically buried deep within components used to compose and build systems and applications. As part of an agile cryptographic approach, organizations should seek to understand what cryptographic assets they are using and facilitate the assessment of the risk posture to provide a starting point for mitigation.

Operations Bill of Materials (OBOM)

OBOMs provide a full-stack inventory of runtime environments, configurations, and additional dependencies. CycloneDX is a full-stack bill of materials standard supporting entire runtime environments consisting of hardware, firmware, containers, operating systems, applications, and libraries. Coupled with the ability to specify configuration makes CycloneDX ideal for Operations Bill of Materials.

Manufacturing Bill of Materials (MBOM)

CycloneDX can describe declared and observed formulations for reproducibility throughout the product lifecycle of components and services. This advanced capability provides transparency into how components were made, how a model was trained, or how a service was created or deployed. In addition, every component and service in a CycloneDX BOM can optionally specify formulation and do so in existing BOMs or in dedicated MBOMs. By externalizing formulation into dedicated MBOMs, SBOMs can link to MBOMs for their components and services, and access control can be managed independently. This allows organizations to maintain tighter control over what parties gain access to inventory information in a BOM and what parties have access to MBOM information which may have higher sensitivity and data classification.

Bill of Vulnerabilities (BOV)

CycloneDX BOMs may consist solely of vulnerabilities and thus can be used to share vulnerability data between systems and sources of vulnerability intelligence. Complex vulnerability data can be represented, including the vulnerability source, references, multiple severities, risk ratings, details and recommendations, and the affected software and hardware, along with their versions.

Vulnerability Disclosure Report (VDR)

VDRs communicate known and unknown vulnerabilities affecting components and services. Known vulnerabilities inherited from the use of third-party and open-source software can be communicated with CycloneDX. Previously unknown vulnerabilities affecting both components and services may also be disclosed using CycloneDX, making it ideal for Vulnerability Disclosure Report (VDR) use cases. CycloneDX exceeds the data field requirements defined in ISO/IEC 29147:2018 for vulnerability disclosure information.

Vulnerability Exploitability eXchange (VEX)

VEX conveys the exploitability of vulnerable components in the context of the product in which they're used. VEX is a subset of VDR. Oftentimes, products are not affected by a vulnerability simply by including an otherwise vulnerable component. VEX allows software vendors and other parties to communicate the exploitability status of vulnerabilities, providing clarity on the vulnerabilities that pose a risk and the ones that do not.

CycloneDX Attestations (CDXA)

CycloneDX Attestations enable organizations to communicate security standards, claims, and evidence about security requirements, and attestations to the veracity and completeness of those claims. CycloneDX Attestations is a way to manage "compliance as code."

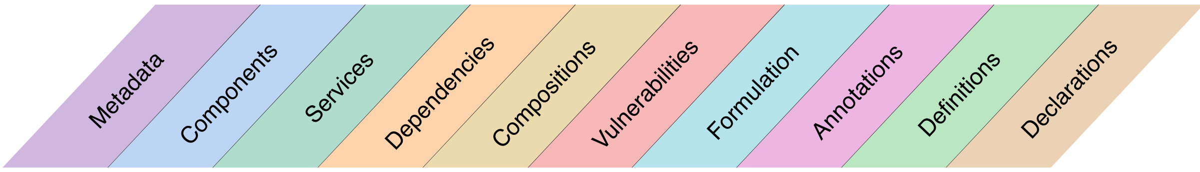
Common Release Notes Format

CycloneDX standardizes release notes into a common, machine-readable format. This capability unlocks new workflow potential for software publishers and consumers alike. This functionality works with or without the Bill of Materials capabilities of the specification.

CycloneDX Object Model

The CycloneDX object model is defined in JSON Schema, XML Schema, and Protocol Buffers and consists of metadata, components, services, dependencies, compositions, vulnerabilities, formulation, and annotations. CycloneDX is prescriptive, can easily describe complex relationships, and is extensible to support specialized and future use cases.

Within the root bom element, CycloneDX defines the following object types:



The object types are arranged in order and contain (but are not limited to) the following types of data:

Metadata	Supplier	Authors	Component		
	Manufacturer	Tools	Lifecycles		
Components	Supplier	Identity	Pedigree	Provenance	Evidence
	Component Type	Licenses	Hashes	Release Notes	Relationships
Services	Provider	Data Classification	Trust Zone		
	Endpoints	Data Flow	Relationships		
Dependencies	Components	Services			
Compositions	Completeness of:				
	Components	Services	Dependencies	Vulnerabilities	
Vulnerabilities	Details	Source	Exploitability (VEX)	Targets Affected	Proof of Concept
	Advisories	Risk Ratings	Evidence	Version Ranges	Recommendations
Formulation	Declared	Formulas	Tasks	Components	
	Observed	Workflows	Steps	Services	
Annotations	Per Person	Per Organization	Per Tool		
	Details	Timestamp	Signature		
Definitions	Standards	Requirements	Levels		
Declarations	Attestations	Evidence	Conformance	Mitigation Strategies	Assessors
	Claims	Counter Evidence	Confidence	Signatories	Signatures
Extensions	Properties	Per Organization	Per Team		
	Formal Taxonomy	Per Industry	...		

BOM Identity

The bom element has properties for serialNumber and version. Together these two properties form the identity of a BOM. A BOM's identity can be expressed using a BOM-Link, a formally registered URN capable of referencing a BOM or any component, service, or vulnerability in a BOM. Refer to the chapter on Relationships for more information.

Serial Number

Every BOM generated should have a unique serial number, even if the contents of the BOM have not changed over time. If specified, the serial number must conform to RFC-4122. The use of serial numbers is recommended.

Version

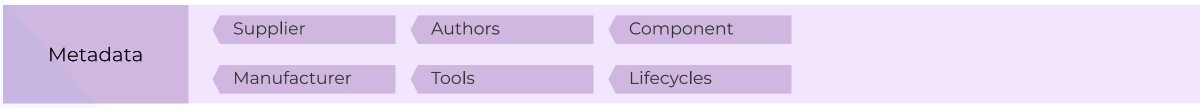
Whenever an existing BOM is modified, either manually or through automated processes, the version of the BOM should be incremented by 1. When a system is presented with multiple BOMs with identical serial numbers, the system should use the most recent version of the BOM. The default version is '1'.

The Anatomy of a CycloneDX BOM

The following are descriptions of the root-level elements of a CycloneDX BOM.

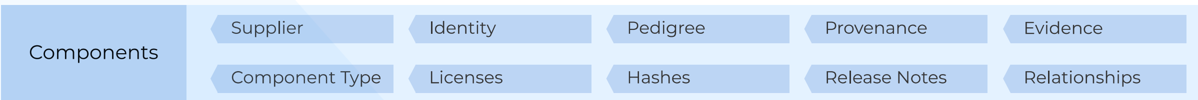
Metadata

BOM metadata includes the supplier, manufacturer, and target component for which the BOM describes. It also includes the tools used to create the BOM, and license information for the BOM document itself.



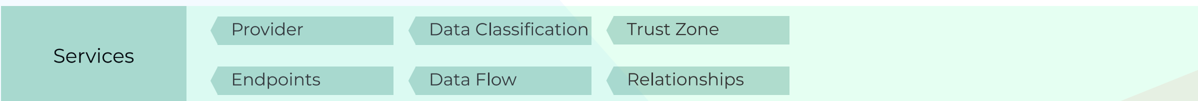
Components

Components describe the complete inventory of first-party and third-party components. The specification can represent software, hardware devices, machine learning models, source code, and configurations, along with the manufacturer information, license and copyright details, and complete pedigree and provenance for every component.



Services

Services represent external APIs that the software may call. They describe endpoint URIs, authentication requirements, and trust boundary traversals. The data flow between software and services can also be described, including the data classifications and the flow direction of each type.



Dependencies

CycloneDX provides the ability to describe components and their dependency on other components. The dependency graph is capable of representing both direct and transitive relationships. Components that depend on services can be represented in the dependency graph, and services that depend on other services can be represented as well.

Dependencies	Components	Services

Compositions

Compositions describe constituent parts (including components, services, and dependency relationships) and their completeness. The aggregate of each composition can be described as complete, incomplete, incomplete first-party only, incomplete third-party only, or unknown.

	Completeness of:			
Compositions	Components	Services	Dependencies	Vulnerabilities

Vulnerabilities

Known vulnerabilities inherited from the use of third-party and open-source software and the exploitability of the vulnerabilities can be communicated with CycloneDX. Previously unknown vulnerabilities affecting both components and services may also be disclosed using CycloneDX, making it ideal for both vulnerability disclosure and VEX use cases.

Vulnerabilities	Details	Source	Exploitability (VEX)	Targets Affected	Proof of Concept
	Advisories	Risk Ratings	Evidence	Version Ranges	Recommendations

Formulation

Formulation describes how something was manufactured or deployed. CycloneDX achieves this through the support of multiple formulas, workflows, tasks, and steps, which represent the declared formulation for reproduction along with the observed formula describing the actions which transpired in the manufacturing process.

Formulation	Declared	Formulas	Tasks	Components
	Observed	Workflows	Steps	Services

Annotations

Annotations contain comments, notes, explanations, or similar textual content which provide additional context to the object(s) being annotated. They are often automatically added to a BOM via a tool or as a result of manual review by individuals or organizations. Annotations can be independently signed and verified using digital signatures.

Annotations	Per Person	Per Organization	Per Tool
	Details	Timestamp	Signature

Definitions

Standards, requirements, levels, and all supporting documentation are defined here. CycloneDX provides a general-purpose, machine-readable way to define virtually any type of standard. Security standards such as OWASP ASVS, MASVS, SCVS, and SAMM are available in CycloneDX format. Standards from other bodies are available as well. Additionally, organizations can create internal standards and represent them in CycloneDX.

Definitions	Standards	Requirements	Levels

Declarations

Declarations describe the conformance to standards. Each declaration may include attestations, claims, counter-claims, evidence, counter-evidence, along with conformance and confidence. Signatories can also be declared and supports both digital and analog signatures. Declarations provide the basis for "compliance-as-code".

Declarations	Attestations	Evidence	Conformance	Mitigation Strategies	Assessors
	Claims	Counter Evidence	Confidence	Signatories	Signatures

Extensions

Multiple extension points exist throughout the CycloneDX object model, allowing fast prototyping of new capabilities and support for specialized and future use cases. The CycloneDX project maintains extensions that are beneficial to the larger community. The project encourages community participation and the development of extensions that target specialized or industry-specific use cases.

Extensions	Properties	Per Organization	Per Team
	Formal Taxonomy	Per Industry	...

Serialization Formats

CycloneDX can be represented in JSON, XML, and Protocol Buffers (protobuf) and has corresponding schemas for each.

Format	Resource	URL
JSON	Documentation	https://cyclonedx.org/docs/latest/json/
JSON	Schema	https://cyclonedx.org/schema/bom-1.6.schema.json
XML	Documentation	https://cyclonedx.org/docs/latest/xml/
XML	Schema	https://cyclonedx.org/schema/bom-1.6.xsd
Protobuf	Schema	https://cyclonedx.org/schema/bom-1.6.proto

CycloneDX relies exclusively on JSON Schema, XML Schema, and protobuf for validation. The entirety of the specification can be validated using officially supported CycloneDX tools or via hundreds of available validators that support JSON Schema, XML Schema, or protobuf.

Lifecycle Phases

The Software Development Life Cycle (SDLC) is a process that outlines the phases involved in software development from conception to deployment and maintenance. It typically includes planning, analysis, design, implementation, testing, deployment, and maintenance; each phase has its own activities and deliverables. The purpose of the SDLC is to provide a structured and systematic approach to software development that ensures the final product meets the customer's requirements, is of high quality, is delivered on time and within budget, and can be maintained and supported throughout its' lifecycle.

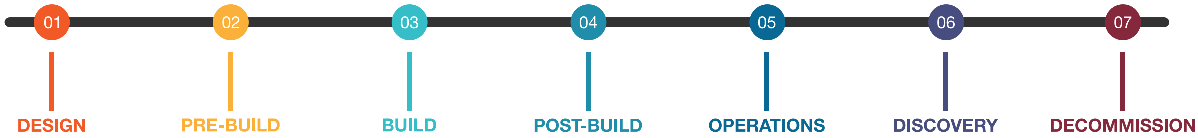
Lifecycle phases communicate the stage in which data in the BOM was captured. This support extends beyond software to capture hardware, IoT, and cloud-native use cases. Different types of data may be available at various phases of a lifecycle, and thus a BOM may include data specific to or only obtainable in a given lifecycle. Incorporating lifecycle phases in a CycloneDX BOM provides additional context of when and how the BOM was created. It becomes an additional datapoint that may be useful in the overall analysis of the BOM.

CycloneDX defines the following phases:

Phase	Description
Design	BOM produced early in the development lifecycle containing an inventory of components and services that are proposed or planned to be used. The inventory may need to be procured, retrieved, or resourced prior to use.
Pre-build	BOM consisting of information obtained prior to a build process and may contain source files, development artifacts, and manifests. The inventory may need to be resolved and retrieved prior to use.
Build	BOM consisting of information obtained during a build process where component inventory is available for use. The precise versions of resolved components are usually available at this time as well as the provenance of where the components were retrieved from.
Post-build	BOM consisting of information obtained after a build process has completed and the resulting components(s) are available for further analysis. Built components may exist as the result of a CI/CD process, may have been installed or deployed to a system or device, and may need to be retrieved or extracted from the system or device.
Operations	BOM produced that represents inventory that is running and operational. This may include staging or production environments and will generally encompass multiple SBOMs describing the applications and operating system, along with HBOMs describing the hardware that makes up the system. Operations Bill of Materials (OBOM) can provide a full-stack inventory of runtime environments, configurations, and additional dependencies.
Discovery	BOM consisting of information observed through network discovery providing point-in-time enumeration of embedded, on-premise, and cloud-native services such as server applications, connected devices, microservices, and serverless functions.
Decommission	BOM containing inventory that will be or has been retired from operations.

In addition, CycloneDX provides a mechanism to supply user-defined lifecycle phases as well.

Software Asset Management (SAM) is a set of processes, policies, and procedures that help organizations manage and optimize their software assets throughout their lifecycle. SAM involves the identification, acquisition, deployment, maintenance, utilization, and disposal of software assets to ensure compliance with licensing agreements, mitigate risks associated with software usage, and optimize costs. Likewise, IT Asset Management (ITAM) has a similar function, encompassing hardware, software, and other IT assets. Unlike the SDLC, which has widely accepted phases, SAM and ITAM lifecycles may vary. For example, the lifecycles defined in [ISO/IEC 19770-1:2017](), which specifies requirements for IT asset management systems, are different from the lifecycles defined in [NIST SP 1800-5](). The out-of-the-box lifecycles provided by enterprise ITAM solutions also vary by vendor and can further be customized by organizations adopting these products. Therefore, CycloneDX includes predefined lifecycles that apply to both SDLC and SAM/ITAM, while also providing the flexibility in defining custom lifecycles. This allows CycloneDX to be fully integrated with existing enterprise SAM/ITAM practices.



The following example illustrates a BOM that was produced in the build and post-build lifecycle phases. In addition, a custom phase (platform-integration-testing) was involved as well.

```
"metadata": {
  "lifecycles": [
    {
      "phase": "build"
    },
    {
      "phase": "post-build"
    },
    {
      "name": "platform-integration-testing",
      "description": "Integration testing specific to the runtime platform"
    }
  ]
}
```

Support for SAM and ITAM use cases is critical for enterprise adoption. An interesting distinction between SDLC and SAM use cases center around license compliance. Solutions supporting the SDLC typically involve open-source license compliance or intellectual property use cases. Whereas SAM is largely concerned with commercial license and procurement use cases. OWASP CycloneDX has extensive support for both. Refer to the "Use Cases" chapter for more information.

Use Cases

CycloneDX provides a comprehensive inventory of all software components, libraries, frameworks, and dependencies in a particular software application or system. It provides a detailed breakdown of the software supply chain, enabling transparency and accountability in software development. The benefits of BOMs are far-reaching and apply to various software, systems, and devices across different domains. Let's explore the types of software, systems, and devices that can significantly benefit from the transparency provided by Bills of Materials.

1. Operating Systems: Operating systems are the foundation for all software and devices, making them a critical component to benefit from software transparency. By having an SBOM for an operating system, developers, IT administrators, and end-users can understand the underlying software components, identify vulnerabilities, and apply patches when necessary. This allows them to make informed decisions regarding security, updates, and mitigating potential risks.

2. Software Applications: From productivity tools to enterprise applications, software applications of all types can benefit from an SBOM. It helps developers and users understand the software's building blocks, including open-source libraries, commercial components, and all other third-party dependencies. With an SBOM, developers can track vulnerabilities, identify license obligations, and facilitate timely updates to ensure the security and stability of their applications.

3. Internet of Things (IoT) Devices: IoT devices encompass a wide range of connected physical objects, such as smart home devices, industrial sensors, healthcare wearables, and more. Unfortunately, these devices often rely on software components that may introduce security risks. By implementing an SBOM, manufacturers and users can gain visibility into the software supply chain of IoT devices, identify vulnerabilities, and implement necessary security measures. This transparency can enhance the security and privacy of IoT ecosystems.

4. Medical Devices: In the healthcare sector, medical devices play a crucial role in patient care and safety. Transparency in the software components used in medical devices is paramount to ensure their reliability and security. An SBOM can help manufacturers, regulatory authorities, and healthcare providers understand the software components, identify potential vulnerabilities or risks, and establish appropriate maintenance and update protocols. This can enhance patient safety and regulatory compliance.

5. Automotive Systems: Modern vehicles heavily rely on software-driven systems for various functionalities, including infotainment, advanced driver assistance, and autonomous driving features. Transparency in the software components used in automotive systems is vital to ensure safety, security, and effective maintenance. An SBOM provides the transparency necessary to identify vulnerabilities, increase license compliance, and manage potential risks effectively.

6. Critical Infrastructure: Software systems that underpin critical infrastructure, such as power grids, transportation networks, and financial systems, demand utmost transparency and security. An SBOM can offer visibility into the software components used in these systems, helping stakeholders assess vulnerabilities, apply security patches, and mitigate potential risks. This transparency contributes to the resilience, reliability, and stability of critical infrastructure.

In the context of national security and military operations, the transparency provided by Software Bill of Materials is of utmost importance. Let's explore the specific types of software, systems, and devices in the national security and military domain that greatly benefit from software transparency:

1. Command and Control Systems: Command and control systems are crucial in military operations, facilitating real-time decision-making and coordination of forces. Transparency in the software components used in these systems allows military authorities to assess potential vulnerabilities and ensure the integrity and security of the systems. In addition, it enables the identification of potential backdoors, unauthorized access points, or malicious components, helping safeguard critical military operations and information.

2. Cybersecurity and Information Assurance Tools: In the realm of national security, robust cybersecurity and information assurance tools are vital to protect against cyber threats and ensure secure communication and data management. Software transparency in these tools enables military authorities to evaluate the software supply chain, identify vulnerabilities, and ensure the use of trusted and up-to-date components. This enhances the resilience and effectiveness of cybersecurity measures and helps counter potential attacks or data breaches.

3. Cryptographic Systems and Algorithms: Cryptographic systems and algorithms are critical in securing sensitive information, communications, and strategic operations. Transparency in the software components underpinning cryptographic systems allows military authorities to analyze the security properties of these components. In addition, it helps assess potential vulnerabilities, validate the use of approved cryptographic standards, and ensure the integrity of encryption algorithms employed in national security and military applications.

4. Intelligence Analysis and Data Processing Software: Intelligence analysis and data processing software are vital in gathering, analyzing, and interpreting vast amounts of information for national security purposes. Software transparency in these software systems provides military intelligence agencies with insights into the underlying components and dependencies. It helps identify potential vulnerabilities that could compromise the accuracy, confidentiality, or integrity of intelligence data. This transparency assists in maintaining the security and reliability of intelligence operations.

5. Unmanned Aerial Vehicles (UAVs) and Autonomous Systems: Unmanned Aerial Vehicles (UAVs) and autonomous systems are increasingly employed in national security and military operations. Transparency in the software components used in these systems enables military authorities to evaluate potential vulnerabilities and ensure the secure and reliable operation of UAVs. In addition, it helps identify potential risks associated with software-dependent functions, such as autonomous navigation, target acquisition, and mission execution, contributing to the overall effectiveness and safety of military operations.

6. Communication and Encryption Devices: Secure and reliable communication is critical for national security and military operations. Software transparency in communication and encryption devices, such as radios, cryptographic hardware, and secure communication protocols, ensures the evaluation of software components involved. It helps identify vulnerabilities, ensure compliance with encryption standards, and protect against potential interception, tampering, or unauthorized access, strengthening the confidentiality and integrity of sensitive communications.

The transparency provided by a Software Bill of Materials is vital to national security, benefiting a range of software, systems, and devices. The software transparency capabilities of CycloneDX enables military authorities to assess vulnerabilities, identify risks, and enhance the security and effectiveness of these critical assets. This transparency contributes to the protection of national security interests and the successful execution of military operations.

Let's explore some specific use cases that CycloneDX BOMs unlock.

Inventory

A complete and accurate inventory of all first-party and third-party components is essential for risk identification. BOMs should ideally contain all direct and transitive components and the dependency relationships between them.

CycloneDX is capable of describing the following types of components:

Type	Class	Description
Application	Component	A software application
Container	Component	A packaging and/or runtime format, not specific to any particular technology, which isolates software inside the container from software outside of a container through virtualization technology.
Cryptographic Asset	Component	A cryptographic asset including algorithms, protocols, certificates, keys, tokens, and secrets.
Data	Component	A collection of discrete values that convey information.
Device	Component	A hardware device such as a processor, or chip-set. A hardware device containing firmware SHOULD include a component for the physical hardware itself, and another component of type 'firmware' or 'operating-system' (whichever is relevant), describing information about the software running on the device.
Device Driver	Component	A special type of software that operates or controls a particular type of device.
File	Component	A computer file.
Firmware	Component	A special type of software that provides low-level control over a device's hardware.
Framework	Component	A software framework
Library	Component	A software library. Many third-party and open source reusable components are libraries. If the library also has key features of a framework, then it should be classified as a framework. If not, or is unknown, then specifying library is RECOMMENDED.
Machine Learning Model	Component	A model based on training data that can make predictions or decisions without being explicitly programmed to do so.
Operating System	Component	A software operating system without regard to deployment model (i.e. installed on physical hardware, virtual machine, image, etc)
Platform	Component	A runtime environment which interprets or executes software. This may include runtimes such as those that execute bytecode or low-code/no-code application platforms.

Type	Class	Description
Service	Service	A service including microservices, function-as-a-service, and other types of network or intra-process services.

The component type is a required property for every component. It is an abstract concept to aid development and security teams with separation of concerns. The types represent the highest level of abstraction in a modular system or design. They also aid Software Asset Management (SAM) and IT Asset Management (ITAM) systems in classifying the inventory of software and constituent parts.

CycloneDX supports multiple methods to assert identity including:

- Coordinates: The combination of the group, name, and version fields form the coordinates of a component.
- Package URL: Package URL (PURL) standardizes how software package metadata is represented so that packages can universally be identified and located.
- CPE: The Common Platform Enumeration (CPE) specification was designed for operating systems, applications, and hardware devices. CPE is maintained by the NVD.
- SWID: Software ID (SWID) as defined in ISO/IEC 19770-2:2015 is used primarily to identify installed software.
- OmniBOR: The OmniBOR Artifact ID is capable of identifying every source code file incorporated into each built artifact.
- SWHID: A Software Heritage ID is a unique identifier assigned to software artifacts to facilitate their identification, tracking, and preservation.

The following example illustrates component identity in CycloneDX.


```json
{
  "type": "library",
  "group": "com.example",
  "name": "awesome-library",
  "version": "1.0.0",
  "cpe": "cpe:2.3:a:acme:awesome:1.0.0:*:*:*:*:*:*:*",
  "purl": "pkg:maven/com.example/awesome-library@1.0.0",
  "omniborId": [ "gitoid:blob:sha1:261eeb9e9f8b2b4b0d119366dda99c6fd7d35c64" ],
  "swhid": [ "swh:1:cnt:94a9ed024d3859793618152ea559a168bbcbb5e2" ],
  "swid": {
    "tagId": "swidgen-242eb18a-503e-ca37-393b-cf156ef09691_1.0.0",
    "name": "Acme Awesome Library",
    "version": "1.0.0",
    "text": {
      "contentType": "text/xml",
      "encoding": "base64",
      "content": "U1dJRCBkb2N1bWVudCBkb2VzIGhlcmU="
    }
  }
}
```

CycloneDX also supports several identifiers specific to hardware devices. Refer to https://cyclonedx.org/capabilities/hbom/ for more information.

Assertion of identity can also be substantiated in the form of evidence, which includes the methods and techniques used during analysis, the confidence, and the tool(s) that performed the analysis. Refer to the "Evidence" chapter for more information.

Vulnerability Management

CycloneDX is ideal for vulnerability management and impact analysis through the support of comprehensive inventory and assertions of component identity. With this information, security teams can identify which components are affected by known vulnerabilities, estimate effort, and quickly prioritize remediation.

By leveraging CycloneDX in this way, organizations can enhance their software supply chain security and reduce the risks associated with software vulnerabilities.

> *Identifying known vulnerabilities in components can be achieved through the use of three fields: cpe, purl, and swid. Not all fields apply to all types of components. Components with a cpe, purl, or swid defined can be analyzed for known vulnerabilities.*

There are many tools and platforms that support vulnerability management use cases using CycloneDX, including OWASP Dependency-Track, often cited as a reference implementation for consuming and analyzing SBOMs. Using a platform such as Dependency-Track, organizations can quickly identify what is affected and where in their environment they are affected.

> *Not all sources of vulnerability intelligence support all three fields. The use of multiple sources may be required to obtain accurate and actionable results.*

Enterprise Configuration Management Database (CMDB)

A Configuration Management Database (CMDB) is a repository that stores information about an organization's assets, including hardware, software, and other components. Tracking assets in a CMDB involves collecting and maintaining accurate information about each asset's configuration, location, status, and relationships with other assets. This information helps organizations manage their assets more effectively, including monitoring their performance, identifying potential risks, and supporting incident management.

Software Asset Management (SAM) and IT Asset Management (ITAM) are typical applications that build upon CMDBs. There are tremendous benefits in capturing BOMs for assets tracked in a CMDB. Organizations gain a more comprehensive view of their assets, which can help them make more informed decisions about managing their IT and OT infrastructure. They also benefit from having the broadest array of possible use cases, including DevOps, vendor risk management, procurement, vulnerability response, and supply chain management.

CycloneDX complements and meets the requirements of ISO/IEC 19770-1:2017 which defines IT asset management systems, including license management, security management, and asset lifecycles, making it uniquely applicable for enterprise adoption.

Integrity Verification

Integrity verification is the process of ensuring that the software components have not been modified or tampered with since they were released. This helps to identify unauthorized modifications to software components that may introduce security vulnerabilities or cause the software to malfunction. Integrity verification uses a cryptographic hash function that is used to generate a unique digital fingerprint, or hash value, for each software component. The hash value can then be compared with the expected hash value for that component to ensure that it has not been altered.

CycloneDX can be used for integrity verification using cryptographic hashing algorithms. The specification allows for the inclusion of cryptographic hashes, such as SHA-256, SHA-384, or SHA-512, for each software component listed in the BOM. By calculating the hash of each file, package, or library and comparing it with the hash value listed in the BOM, organizations can verify the integrity of the software and detect unauthorized modifications.

The following example illustrates how to represent hashes on a component.

```
"components": [
    {
      "type": "library",
      "name": "acme-example",
      "version": "1.0.0",
      "hashes": [{
        "alg": "SHA-256",
        "content": "d88bc4e70bfb34d18b5542136639acbb26a8ae2429aa1e47489332fb389cc964"
      },{
        "alg": "BLAKE3",
        "content": "26cdc7fb3fd65fc3b621a4ef70bc7d2489d5c19e70c76cf7ec20e538df0047cf"
      }]
    }
  ]
}
```

In addition, external references (covered later in the "Relationships" chapter) also support hashes. The following example illustrates how CycloneDX can refer to an external BOM and include the hashes for that BOM. In doing so, the integrity of the external BOM can be evaluated prior to use.

```
"components": [
  {
    "type": "library",
    "group": "com.example",
    "name": "persistence",
    "version": "5.2.0",
    "externalReferences": [
      {
        "type": "bom",
        "url": "https://example.com/sbom.json",
        "hashes": [
          {
            "alg": "SHA-256",
            "content": "9048a24d72d3d4a1a0384f8f925566b44f133dd2a0194111a2daeb1cf9f7015b"
          }
        ]
      }
    ]
  }
]
```

CycloneDX supports SHA-1, SHA-2, and SHA-3 hashing algorithms along with BLAKE2b and BLAKE3.

By leveraging CycloneDX for integrity verification, organizations can enhance the security and reliability of their software applications and systems.

Authenticity

Authenticity refers to the assurance that a component, or the BOM itself, came from the expected source and has not been tampered with. Authenticity can be verified through the use of digital signatures and code-signing certificates, which are issued by trusted certificate authorities. These signatures allow users to verify the supplier's identity and ensure that the artifact has not been modified since it was signed.

When a BOM is signed, the authenticity and integrity of the BOM can be verified. This verification can ensure that the data in the BOM has not been altered. Using signed BOMs increases trust and confidence in a software product, particularly in cases where the product is used in sensitive or critical applications.

CycloneDX supports enveloped signing, including XML Signature (xmlsig) and JSON Signature Format (JSF). In addition, detached signatures are also supported.

The following example illustrates the use of enveloped signing using JSF.

```
"signature": {
  "algorithm": "RS512",
  "publicKey": {
    "kty": "RSA",
    "n": "qOSWbDOGS31lv3aUZVOgqZyLVrKXXRfmxFQxEylc...",
    "e": "AQAB"
  },
  "value": "HGIX_ccdlcqmaOpkxDzKH_j0ozSHUAUyBxGpXS..."
}
```

License Compliance

CycloneDX is ideal for both open-source and commercial license compliance. By leveraging the licensing capabilities of CycloneDX, organizations can identify any licenses that may be incompatible or require specific compliance obligations, such as attribution or sharing of source code. CycloneDX supports declared, observed, and concluded licenses.

CycloneDX can also help organizations manage their commercial software licenses by providing a clear understanding of what licenses are in use and which ones require renewal or additional purchases, which may impact the operational aspects of applications or systems. By leveraging CycloneDX for commercial license compliance, organizations can reduce the risks associated with license violations, enhance their license management practices, and align their SBOM practice with Software Asset Management (SAM) and IT Asset Management (ITAM) systems for enterprise visibility.

Solutions supporting the Software Development Life Cycle (SDLC) typically involve open-source license compliance or intellectual property use cases. Whereas Software Asset Management (SAM) is primarily concerned with commercial license and procurement use cases. OWASP CycloneDX has extensive support for both and can be applied to any component or service within a BOM.

Outdated Component Analysis

Relying on outdated components can have a significant impact on the security, stability, and performance of the software. Outdated components may have known vulnerabilities that can be exploited by attackers, leading to data breaches or other security issues. Additionally, newer versions of components may include bug fixes or performance improvements that can enhance the overall functionality of the software.

Updating components is not a one-time task but a continuous process. New vulnerabilities and bugs are constantly being discovered, and the latest updates are being released to fix them. Thus, it is crucial to regularly check for updates and keep components up to date. Ignoring updates and running software with outdated components can lead to increased time to mitigate vulnerabilities should a previously unknown vulnerability become known.

Identifying end-of-life components can be challenging as the data may be difficult to obtain. However, some sources of commercial vulnerability intelligence do provide this data, and also help identify up-to-date components that are otherwise no longer supported.

Provenance

Provenance refers to the history of the origin and ownership of a component. In the context of a software supply chain, provenance provides a way to trace the lineage of a component and ensure its authenticity is in alignment.

Provenance information can help software developers and users identify the source of a component, and helps to establish trust and accountability among different parties involved in the software supply chain, such as software vendors, distributors, and consumers.

By maintaining a record of provenance information throughout the software supply chain, organizations can improve their ability to detect and mitigate security risks, reduce the likelihood of supply chain attacks, and increase the overall reliability and quality of their software products.

Furthermore, regulatory compliance requirements (such as those related to data privacy, data protection, and intellectual property) often mandate the use of provenance tracking to ensure compliance with legal and ethical standards.

CycloneDX supports provenance via four distinct fields: author, publisher, supplier, and manufacturer. In addition, components that are modified from the original can be described along with the complete authorship, including commits and the person or account that authored and committed the modifications.

Pedigree

CycloneDX can represent component pedigree, including ancestors, descendants, and variants that describe component lineage from any viewpoint and the commits, patches, and diffs which make it unique. The addition of a digital signature applied to a component with detailed pedigree information serves as an affirmation of the accuracy of the pedigree.

Maintaining accurate pedigree information is especially important with open-source components whose source code is readily available, modifiable, and redistributable. Identifying changes to a component or a components coordinates along with information describing the original component, may be necessary for the analysis of various forms of risk.

Refer to the "Relationships" chapter for detailed information on pedigree.

Foreign Ownership, Control, or Influence (FOCI)

Foreign Ownership, Control, or Influence (FOCI) is a critical concern in the software supply chain that should be taken seriously by all organizations involved. FOCI refers to the degree to which foreign entities have control or influence over the operations or assets of companies in another government's jurisdiction. FOCI is a term specific to the U.S., but many world governments have similar concepts.

Indicators that may be relevant in identifying FOCI concerns can be derived from several fields, including author, publisher, manufacturer, and supplier but can also be extended to other fields such as the components group name. The CPE may also indicate the vendor and the PURL can identify a potentially foreign namespace or repository or download URL for the package. Many external references may also provide a clue, especially those pointing to the version control system (vcs) and commit history, issue tracker, distribution, and documentation websites.

Commercial sources of supply chain intelligence, including both physical and cyber, are available and can aid in identifying FOCI and other supply chain risk.

Export Compliance

CycloneDX can help organizations achieve export compliance in the software supply chain by providing a comprehensive inventory of all software components used in a product, including their origin, version, and licensing. This information can enable organizations to identify potential export control issues, such as using components developed in foreign countries or containing encryption technology, and take appropriate measures to ensure compliance.

Procurement

Purchasing of software and IT assets can be enhanced with bill of materials. Model contract language that would require BOMs for all new procurements and renewals of deployable software and any IT asset containing software should be considered. Sourcing may then strategically favor vendors who provide BOMs or further negotiate costs with vendors that don't. Procurement processes can be enhanced to request BOMs from vendors, which may then be consumed by the procurement system and shared with enterprise Software Asset Management (SAM) or IT Asset Management (ITAM) systems. Automating BOM requests, retrieval, consumption, and sharing across systems should be considered for organizations on a quest for digital transformation.

Vendor Risk Management

A Vendor Risk Assessment (VRA) is a process used to identify and evaluate potential risks or hazards associated with a vendor's operations and products and their potential impact on an organization. VRA is part of an overall Vendor Risk Management process. VRAs are often an integrated part of the procurement process for new vendors. VRAs may also be triggered periodically for existing vendors. VRA processes can be enhanced through the use of BOMs. With BOMs, not only can the supplier of the software or asset can be evaluated, but every supplier of the constituent components that make up the software or asset can be evaluated. Additionally, the report from a VRA can be specified in CycloneDX using the risk-assessment external reference type. The transparency that CycloneDX BOMs provide can result in more impactful assessments and significant risk reduction.

Supply Chain Management

Supply chain management is a strategic discipline that encompasses the coordinated planning, implementation, and control of the flow of goods, services, and information from the point of origin to the point of consumption. It involves a systematic approach to optimizing every aspect of the supply chain.

Dr. W. Edwards Deming, a renowned quality management expert, emphasized the importance of collaboration, data-driven decision-making, and a relentless pursuit of excellence throughout the entire supply chain. Deming believed that by focusing on quality and process improvement, organizations can achieve higher levels of customer satisfaction and long-term success.

Deming's supply chain management strategy included using fewer and better suppliers, utilizing the best quality components from those suppliers, and tracking component usage across the entire supply chain. By focusing on fewer suppliers, organizations can reduce variability and drive efficiency. Deming emphasized the importance of selecting suppliers who consistently deliver top-quality components, which improves the overall quality of products or services. Additionally, tracking component usage across the supply chain allows organizations to identify inefficiencies, optimize processes, and eliminate waste.

Supply chain management of physical goods shares several similarities with software supply chain management. Both disciplines involve sourcing, production, distribution, and inventory management to ensure the smooth flow of goods or software throughout the supply chain. Just as physical goods move from suppliers to manufacturers to end-users, software components are sourced, developed, and integrated to create a final software product. While there are differences in the nature of the products being managed, the core principles of efficient sourcing, production, and distribution are applicable to physical goods and software.

CycloneDX BOMs play a crucial role in supply chain management as they enhance collaboration and enable effective supply chain management and governance of software components from sourcing to deployment.

Composition Completeness and "Known Unknowns"

The inventory of components, services, and their relationships to one another can be described through the use of compositions. Compositions describe constituent parts (including components, services, and dependency relationships) and their completeness. The completeness of vulnerabilities expressed in a BOM may also be described. This allows BOM authors to describe how complete the BOM is or if there are components in the BOM where completeness is unknown.

Formulation Assurance and Verification

CycloneDX can describe declared and observed formulations for reproducibility throughout the product lifecycle of components and services. This advanced capability provides transparency into how components were made, how a model was trained, or how a service was created or deployed. Generally, the formulation is externalized from the SBOM into a dedicated Manufacturing Bill of Material (MBOM). The SBOM references the MBOM that describes the environment, configuration, tools, and all other considerations necessary to replicate a build with utmost precision. This capability allows other parties to independently verify inputs and outputs from a build which can increase the software's assurance.

Cryptography Asset Management

CycloneDX can describe a comprehensive inventory of cryptographic assets, encompassing keys, certificates, tokens, and more. This is a requirement of the OMB M-23-02, where such a system is characterized as a [...”software or hardware implementation of one or more cryptographic algorithms that provide one or more of the following services: (1) creation and exchange of encryption keys; (2) encrypted connections; or (3) creation and validation of digital signatures.”]

CycloneDX provides a structured framework for organizations to catalog and track their cryptographic resources, facilitating efficient management and ensuring security and compliance standards are met. By maintaining a detailed record of cryptographic assets, including their usage, expiration dates, and associated metadata, CycloneDX enables proactive monitoring and streamlined auditing processes. With CycloneDX, organizations can effectively safeguard their cryptographic infrastructure, mitigate risks associated with unauthorized access or misuse, and maintain the integrity and confidentiality of sensitive data across diverse digital environments.

Identifying Weak Cryptographic Algorithms

CycloneDX enables organizations to discover weak algorithms or flawed implementations that could compromise security. Through analysis of cryptographic data, including algorithms, key management practices, and usage patterns, organizations can pinpoint areas of concern and prioritize remediation efforts. CycloneDX facilitates proactive identification of weaknesses and vulnerabilities, allowing organizations to enhance the resilience of their cryptographic infrastructure and mitigate the risk of exploitation, thereby bolstering overall cybersecurity posture and safeguarding sensitive data against potential threats.

Post-Quantum Cryptography (PQC) Readiness

CycloneDX is crucial in preparing applications and systems for an impending post-quantum reality, aligning with guidance from the National Security Agency (NSA) and the National Institute of Standards and Technology (NIST). As quantum computing advancements threaten the security of current cryptographic standards, CycloneDX provides a structured approach to inventorying cryptographic assets and evaluating their resilience against quantum threats.

Most notably, public key algorithms like RSA, DH, ECDH, DSA or ECDSA are considered not quantum-safe. These algorithms occur in various components and may be hardcoded in applications but are more commonly and preferably used via dedicated cryptographic libraries or services. Developers often don't directly interact with cryptographic algorithms such as RSA or ECDH but use them via protocols like TLS 1.3 or IPsec, by using certificates, keys, or other tokens. With upcoming cryptographic agility it becomes less common to put in stone (or software) the algorithms that will be used. Instead, they are configured during deployment or negotiated in each network protocol session. CycloneDX is designed with these considerations in mind and to allow insight into the classical and quantum security level of cryptographic assets and their dependencies.

By cataloging cryptographic algorithms and their respective parameters, CycloneDX enables organizations to identify vulnerable or weak components that require mitigation or replacement with quantum-resistant alternatives recommended by NSA and NIST. Through comprehensive analysis and strategic planning facilitated by CycloneDX, organizations can proactively transition to post-quantum cryptographic primitives, ensuring the long-term security and integrity of their systems and applications.

Assess Cryptographic Policies and Advisories

A cryptographic inventory in machine-readable form brings benefits if one wants to check for compliance with cryptographic policies and advisories. An example of such an advisory is CNSA 2.0, which was announced by NSA in September of 2022. CNSA 2.0 states, among other things, that National Security Systems (NSS) for firmware and software signing needs to support and prefer CNSA 2.0 algorithms by 2025 and exclusively use them by 2030. The advised algorithms are the stateful hash-based signature schemes LMS and XMSS from NIST SP 800-208. With a cryptographic inventory that documents the use of LMS and XMSS by such systems, compliance with CNSA 2.0 can be assessed in an automated way.

Identify Expiring and Long-Term Cryptographic Material

CycloneDX significantly enhances the ability to identify and manage the risks associated with expiring and long-term cryptographic material. For instance, an RSA certificate set to expire in one week inherently presents a lower cryptographic risk compared to an identical certificate with a 20-year expiry period. This consideration is crucial, as an expired certificate can lead to significant service downtime, compounding the risk to operational security and reliability.

Ensure Cryptographic Certifications

Higher cryptographic assurance is provided by certifications such as FIPS 140-3 (levels 1 to 4) or Common Criteria (EAL1 to 7). To obtain these certifications, cryptographic modules need to undergo certification processes. For regulated environments such as FedRAMP, such certifications are important requirements. CycloneDX allows the capture of certification levels of cryptographic assets so that this property can be easily identified.

BOM Coverage, Maturity, and Quality

NTIA Minimum Elements

The U.S. National Telecommunications and Information Administration (NTIA) defines the following minimum elements of an SBOM. They are:

Field	CycloneDX Field	Description
Supplier	bom.metadata.supplier, bom.components[].supplier	The name of an entity that creates, defines, and identifies components.
Component Name	bom.components[].name	Designation assigned to a unit of software defined by the original supplier.
Component Version	bom.components[].version	Identifier used by the supplier to specify a change in software from a previously identified version.
Other Unique Identifiers	bom.components[].cpe,purl,swid	Other identifiers that are used to identify a component, or serve as a look-up key for relevant databases.
Dependency Relationship	bom.dependencies[]	Characterizing the relationship that an upstream component X is included in software Y.
Author of SBOM Data	bom.metadata.author	The name of the entity that creates the SBOM data for this component.
Timestamp	bom.metadata.timestamp	Record of the date and time of the SBOM data assembly.

CycloneDX highly encourages organizations to exceed the NTIA minimum elements whenever possible. Suggestions for other types of data will vary by use case but generally should include:

Field	CycloneDX Field	Description
BOM Lifecycles	bom.metadata.lifecycles[]	The stage in which data in the BOM was captured
BOM Generation Tools	bom.metadata.tools[]	The tool(s) used to create the BOM
Component Hash	bom.components[].hashes[]	The hash values of the file or package

Field	CycloneDX Field	Description
Component License	bom.components[].licenses[]	The license(s) in which the component is released under
Component Evidence	bom.components[].evidence[].identity.*	The evidence of identity including the methods, techniques, and confidence of how components were identified
Cryptographic Properties	bom.components[].cryptoProperties.*	The properties specific to cryptographic assets detailing the algorithms, keys, protocols, and other cryptographic material
External References	bom.components[].externalReferences[]	Locations to advisories, version control and build systems, etc
Services	bom.services[].*	A complete inventory of services including endpoint URLs, data classifications, etc which the product and/or individual components rely on
Known Unknowns	bom.compositions[].*	Assertions on the completeness of the inventory of components and services, along with the completeness of dependency relationships

SCVS BOM Maturity Model

The OWASP Software Component Verification Standard (SCVS) is a way for organizations to measure and improve their software supply chain assurance. SCVS is required in NIST SP 800-218 (SSDF v1.1) and similar frameworks.

In addition to the supply chain controls it recommends, SCVS also has a complementary BOM Maturity Model which allows bill of materials to be evaluated. The model consists of:

- a formal taxonomy of different types of data possible in a bill of materials, independent of BOM format
- a unique identifier, description, and other metadata about each item in the taxonomy
- the level of complexity or difficulty in supporting different types of data

The model can be used to evaluate:

- Incoming BOMs adherance to organizational policy by supporting the data required by various stakeholders
- BOM generation and consumption tools
- Current and future BOM formats against each other and their alignment with organizational requirements

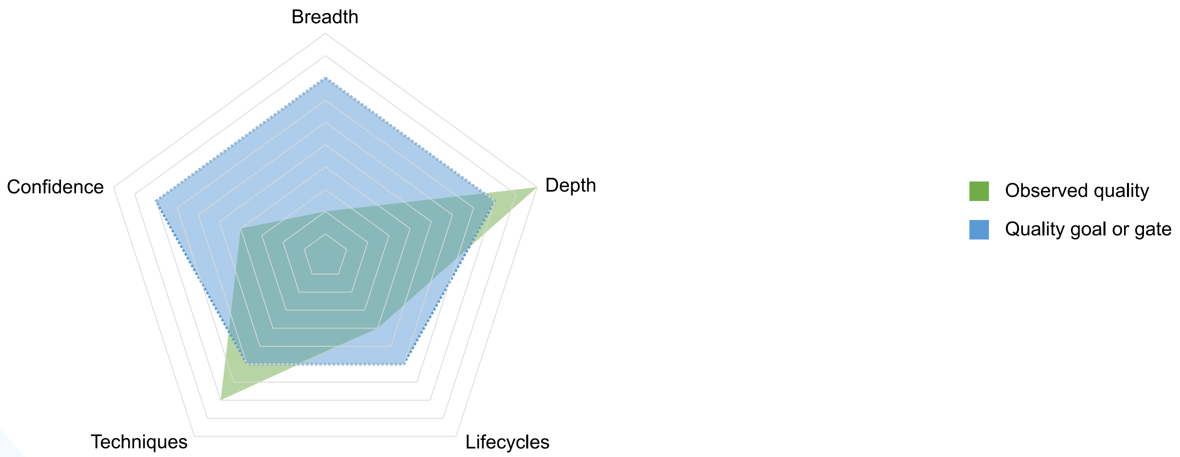
Combined with the ability to create profiles, SCVS will facilitate:

- The creation of a new breed of tools (SBOM Profilers) which evaluate BOMs against various profiles so that end users may know what types of analysis can be performed on them

- The adoption of organizational policy, defined in profiles, for what is acceptable and not acceptable for various use cases

SBOM Quality

SBOMs can be analyzed for their overall usefulness for given use cases. The "quality" of an SBOM may differ depending on the stakeholder role and type of analysis required for that role. Quality is a multidimensional construct and not a single characteristic. OWASP supports a holistic view of quality. The following illustrates an example of dimensions to consider in determining quality.



Dimension	Support	Description
Breadth	SCVS	The coverage in the types of data represented within a BOM.
Depth	SCVS	The amount of detail or difficulty needed to represent data within a BOM.
Lifecycles	CycloneDX	The number of lifecycles or the favorability of specific lifecycles in the creation of a BOM.
Techniques	CycloneDX	The approaches used to determine component identity.
Confidence	CycloneDX	The confidence of individual techniques, and the analysis of the sum of all techniques used to identity components.

The OWASP SCVS BOM Maturity Model is a formal taxonomy of different types of data possible in a Bill of Materials along with the level of complexity or difficulty in supporting different types of data. The BOM Maturity Model can be used as the basis for the Breadth and Depth dimensions.

Lifecycles are supported in CycloneDX. Refer to the "Lifecycle Phases" chapter for more information. Evidence is also a capability of CycloneDX. Identity evidence consists of:

- The field for which the evidence describes (name, version, purl, etc)

- The overall confidence derived from all supporting evidence

- The methods which include the techniques used to determine component identity and the confidence of each technique
- The tools used which performed the analysis

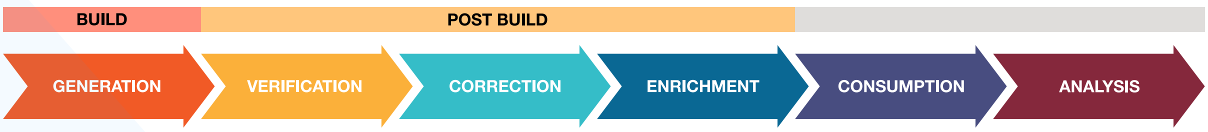
Together, the BOM Maturity Model and native features of CycloneDX can be leveraged to form a high-quality, high-confidence assessment of SBOM quality.

Generating CycloneDX BOMs

There are many ways to generate BOMs, each method having various trade-offs. CycloneDX recommends organizations establish a process around BOM generation that aligns with the needs of the business and that of the BOM consumer. In practice, BOM generation is a process, not a one-time event. As organizations mature their BOM efforts and consumers expect increased accuracy and expanded data, having an established process that can accommodate multiple generation methods and the ability to augment and correct BOM data throughout the generation process will provide strategic advantages.

The following process is the path most traveled by organizations that first adopt SBOMs. This process starts with SBOM generation, which is often performed during the build process, followed by consumption and analysis of the SBOM. Simultaneously, the SBOM is often published alongside the artifacts that result from the build process.

For some organizations, the process above is where their journey ends. However, for many other organizations, it's just the start. OWASP recommends that SBOM creation become an integrated and repeatable process aiming to achieve accurate and trustworthy results. The following is an example workflow that illustrates SBOM creation, verification, and enrichment using multiple tools and techniques.

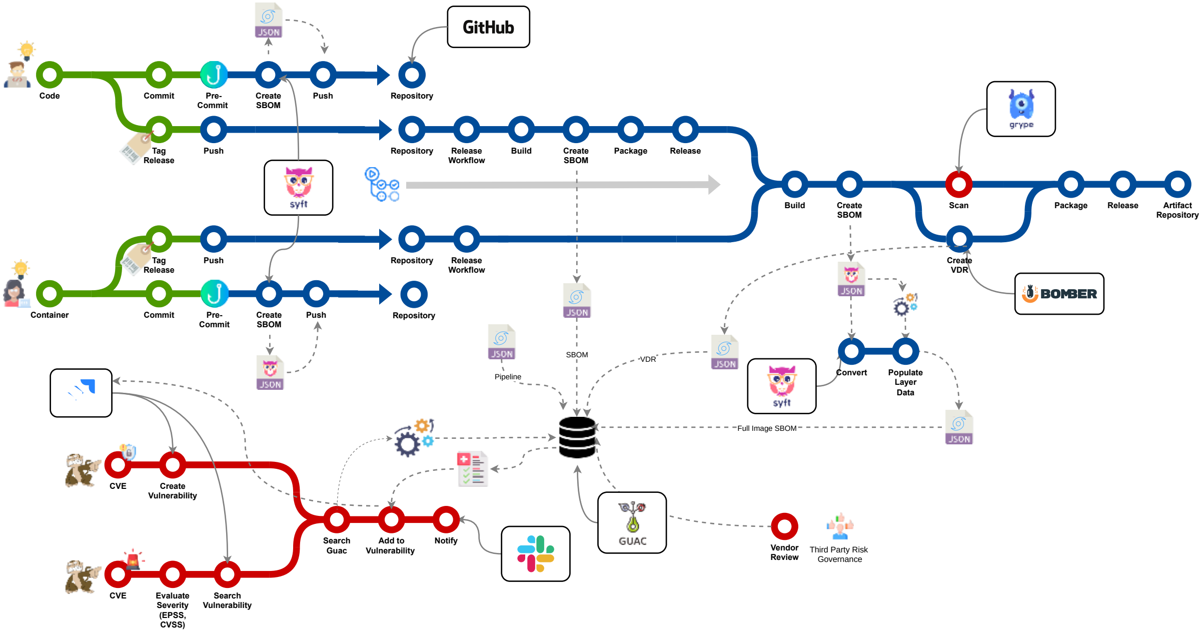


The benefits of this approach are numerous. It starts with SBOM generation in the build lifecycle. This typically involves a plugin specific to the build tool used, which often generates the most accurate and complete set of initial results. Build plugins often rely on manifests that can be manipulated or, in the case of unmanaged dependencies, may not include all dependent components.

The verification stage may involve specialized tools that perform different types of analysis against the build artifacts and compare the findings to the results in the SBOM. If there are deltas, then the resulting SBOM may need to be corrected.

One common scenario where correction often occurs is with modified or forked components. Manifest and binary analysis typically falls short in properly identifying modified components. Tools may identify the component as being modified or the upstream version but generally cannot distinguish what the modifications were, who made them, or for what purpose. Open source is the ultimate supply chain. Components can and will be modified. Often these modifications are to add new features or to backport security fixes. Describing these modifications in the SBOM greatly increases its accuracy and the perceived trustworthiness of the SBOM and the vendor who provided it. Tracking modifications is referred to as "pedigree" and is covered later in the "Relationships" chapter.

As the SBOM process evolves, it may become an integrated part of building software. One vision of this type of process comes from DJ Schleen who proposed the following reference architecture:



"SBOM Process" © 2023 DJ Schleen

The content in this architecture is beyond the scope of this guide, but is provided to illustrate what is possible using freely available open source tools.

Approaches to Generating CycloneDX SBOMs

There are many approaches to generating SBOMs. Each has its strengths, but all provide value in an SBOM process. Common approaches are listed below along with the lifecycles they could be executed in.

Approach	Lifecycles	Description
Build Plugin	Build	Specialized tool that integrates directly into native build systems
Software Factory	Pre Build, Build, Post Build	An approach whereby the system that orchestrates builds directly generates SBOMs
SCA	Pre Build, Build, Post Build	Software Composition Analysis, which may inspect manifests in version control pre-build, be integrated into builds, or perform analysis of built artifacts post-build
IAST/RASP	Post Build, Operations	Specialized tool that often involves instrumentation against running systems

Each approach may use multiple methods and techniques to identify components and other relevant data. The techniques used, the confidence, and call stack reachability can all be described granularly at the component level in CycloneDX. Refer to the "Evidence" chapter for more information.

Generating SBOMs for Source Files

SBOMs may describe individual source files and other digital assets in a directory or version control system. These types of SBOMs typically include file components, file hashes, and evidence of license and copyright statements. The primary purpose of this type of BOM is for license compliance and intellectual property use cases. They may also be used as an OpenChain Compliance Artifact. Oftentimes, license attribution reports can be derived from source SBOMs. Generating SBOMs from source files typically occur in the "pre build" lifecycle.

Integrating CycloneDX Into The Build Process

Integrating SBOM generation into the software's build system is the preferred starting point for producing SBOMs for cybersecurity use cases. Modern build systems rely on package manifests which describe the intent to use specific dependencies. Examples of manifests include pom.xml (Java/Maven), package-lock.json (Javascript/npm), and requirements.txt (Python).

There are three primary strategies for producing SBOMs during a build.

- Integration into build lifecycle
- Analyzing build artifacts external to lifecycle
- Software factory

Build Lifecycle vs. External to Lifecycle

Many build systems have a "lifecycle" that can affect dependency resolution. These lifecycles are often configurable by the developers and can profoundly affect component inventory and versions. For example, Maven resolves dependencies as it progresses through its lifecycle. A Maven build may also include optional profiles, which can alter what dependencies are included or excluded from the final deliverable. Analyzing pom.xml outside of Mavens' lifecycle will typically lead to erroneous results. On the Javascript front, many plugins to npm or web frameworks can dramatically affect component inventory. For example, many web frontends are optimized using a process called bundling which removes unused dependencies and/or functions through a process called "tree-shaking" and aggregates the Javascript into highly optimized bundles for efficient delivery to web and mobile browsers. In these scenarios, relying on package-lock.json as the source of truth would lead to an erroneous SBOM containing an inventory of components that are not distributed in the final artifact. In the case of software vendors, it is important only to include the components that are distributed with the final software. Not doing so may lead to increased and unnecessary support costs.

Software Factory

Integrating into individual builds, especially a build's lifecycle, has many advantages but generally takes more effort. Another approach is to target the generation at the software factories themselves. Software factories often comprise Continuous Integration and Continuous Delivery (CI/CD) systems. Organizations may customize their CI/CD environment to optimize software delivery and increase the efficiency of onboarding new software projects. A strategic option for many organizations is to reduce the effort necessary to create SBOMs by automating as much as possible. Once configured, generating SBOMs from software factories allows organizations to produce SBOMs for many software projects with little to no effort. GitHub Actions, GitLab Runners, Jenkins libraries, and Circle CI orbs are often used as the foundation for many software factories. While this approach can quickly scale across an organization, the accuracy of the SBOMs may be impacted as the software factories orchestrate the build tools; they are not directly part of the build systems lifecycle.

Generating BOMs at Runtime

Analyzing source files or build manifests has some limitations. They do not capture the environment in which the software is being run, the system dependencies that are used, which are not specified in the source files or manifests, and will be limited to the inventory of software components. Generating SBOMs at runtime is often achieved through observability or instrumentation. Examples of platforms capable of runtime generation include:

- Interactive Application Security Testing (IAST)
- Mobile Application Security Testing (MAST)

Generating SBOMs at runtime has many benefits including:

- Capturing the dependencies that are invoked and those which are not
- Capturing system dependencies of the underlying platform or operating system
- Capturing information and configuration about the runtime environment
- Capturing the use and reliance on external services such as those provided via HTTP and MQTT

The platforms capable of runtime generation are often used as part of the software's testing phase and orchestrated by CI systems. In addition, many IAST platforms also double as RASP (Runtime Application Security Protection) and can proactively mitigate specific types of attacks automatically.

Generating BOMs From Evidence (from binaries)

Oftentimes, especially for legacy software, the source or build files may not be available, and runtime instrumentation may not be possible. In these cases, analyzing the binary artifacts may be necessary. These same approaches may also be used by security firms specializing in firmware forensics associated with medical, IoT, and other types of devices.

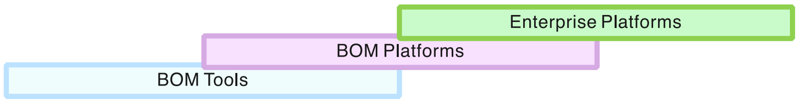
Refer to the "Evidence" chapter for more information.

Building CycloneDX BOMs Manually

CycloneDX evolved in the era of DevSecOps and has a strong focus on being highly automatable. Most CycloneDX tools are also focused on automation. However, some ecosystems such as C/C++ continue to mostly rely on unmanaged dependencies despite the availability of package managers. In these situations, manually managing dependencies often requires manual SBOM generation. Several tools exist to accomplish this task including OWASP Dependency-Track.

Consuming CycloneDX BOMs

Consuming CycloneDX BOMs can be done efficiently using various tools specifically designed to ingest and analyze BOMs. In general, there are three classifications of tools. They are:



1. **BOM Tools**: This classification of tool is generally small, purpose-built, and often a command-line utility. These types of tools generally focus on vulnerability scanning, license compliance, or dependency analysis. While there are many tools that provide this functionality, a few honorable open source mentions are Bomber, dep-scan, Grype, and Trivy. All these tools can accept CycloneDX BOMs as input and analyze them for known security risk.

2. **BOM Platforms**: These higher complexity tools offer robust and collaborative features and are generally purpose-built for BOM consumption. They typically consume BOMs from CI/CD pipelines or external systems, such as procurement. Notable open source projects in this category are GUAC, a supply chain intelligence platform, and OWASP Dependency-Track, a reference platform for BOM consumption and analysis.

3. **Enterprise Platforms**: Often times these are large CMDB's or similar systems that provide a wide-range of IT, procurement, and business applications. These platforms are typically more general-purpose, capable of a wide range of use cases, including SBOM consumption.

For a list of known tools that support the CycloneDX standard, visit the CycloneDX Tool Center.

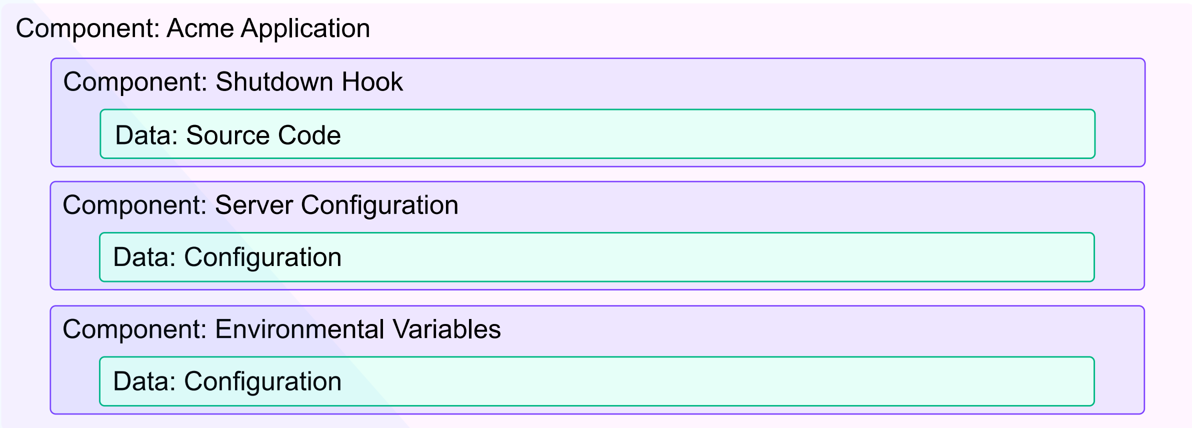
Leveraging Data Components

Data components provide the ability to inventory data as part of a bill of material. This specialized type of component benefits from all the other capabilities that CycloneDX provides, including tracking the provenance and pedigree of data.

A data "type" describes the general theme or subject matter of the data being specified. The following are supported types:

Type	Description
configuration	Parameters or settings that may be used by other components.
dataset	A collection of data.
definition	Data that can be used to create new instances of what the definition defines.
source-code	Any type of code, code snippet, or data-as-code.
other	Any other type of data that does not fit into existing definitions.

To help visualize a typical scenario, let's describe an application with a few different data components that represent custom source code and configurations bundled in an application.



Other possible scenarios include:

- Inclusion of all source code that makes up a component.
- Inclusion of inline datasets bundled with a component.
- Externalizing the data components using an External Reference of type 'bom'.
- Leveraging CycloneDX lifecycles and External References to create an Operations Bill of Materials (OBOM) linking the SBOM of the application, the HBOM of the hardware it's running on, and describing the runtime configuration of the system in the OBOM.

This example, similar to the previous illustration, involves Acme Application which includes the Javascript source code for a shutdown hook. In this case, both are from different suppliers.

```json
"components": [
  {
    "bom-ref": "acme-application",
    "type": "application",
    "name": "Acme Application",
    "version": "1.0.0",
    "supplier": { "name": "Acme Inc" },
    "components": [
      {
        "type": "data",
        "name": "Shutdown Hook",
        "supplier": { "name": "Example Company" },
        "data": [
          {
            "type": "source-code",
            "contents": {
              "attachment": {
                "contentType": "text/javascript",
                "encoding": "base64",
                "content": "Y29uc29sZS5sb2coJ0dvb2RCeWUnKQ=="
              }
            }
          }
        ]
      }
    ]
  }
]
```

CycloneDX does not attempt to normalize configurations into a common vocabulary. Systems and applications may have specialized ways of representing configurations that are specific to them. Rather, CycloneDX leverages existing support for name/value pairs (properties), attachments, and URLs to external resources. With this approach, common and specialized configuration mechanisms are supported. Consumers of BOMs with data components will need to understand the context and semantics of the data specified.

Introduction to Cryptographic Components

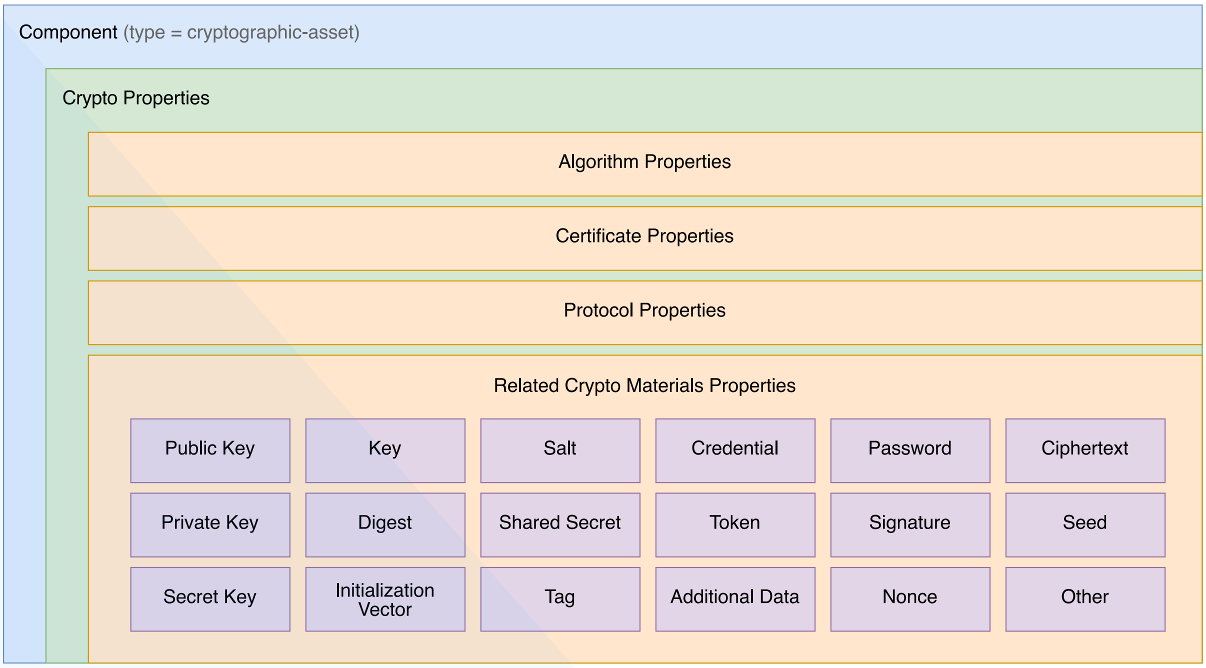
CycloneDX can describe cryptographic assets and their dependencies. Discovering, managing, and reporting on cryptographic assets is necessary as the first step on the migration journey to quantum-safe systems and applications. Cryptography is typically buried deep within components that are used to compose and build systems and applications.

Advances in quantum computing introduce the risk of previously-secure cryptographic algorithms becoming compromised faster than ever before. In May of 2022, the White House released a National Security Memorandum outlining the government's plan to secure critical systems against potential quantum threats. This memorandum contains two key takeaways for both agency and commercial software providers: document the potential impact of a breach, and have an alternative cryptography solution ready.

As cryptographic systems evolve from using classical primitives to quantum-safe primitives, there is expected to be more widespread use of cryptographic agility, or the ability to quickly switch between multiple cryptographic primitives. Cryptographic agility serves as a security measure or incident response mechanism when a system's cryptographic primitive is discovered to be vulnerable or no longer complies with policies and regulations.

As part of an agile cryptographic approach, organizations should seek to understand what cryptographic assets they are using and facilitate the assessment of the risk posture to provide a starting point for mitigation.

The following is a high-level architecture illustrating how cryptographic assets are implemented in CycloneDX.



Organizations should consider including cryptographic assets in their SBOMs and optionally producing a BOM specific for cryptographic material, otherwise known as a Cryptographic Bill of Material (CBOM).

Refer to the Authoritative Guide to CBOM for in-depth information about leveraging CycloneDX for cryptographic use cases.

Algorithm Example

A cryptographic algorithm is added in the components array of the BOM. The examples below list the algorithm AES-128-GCM.

```
"components": [
  {
    "type": "cryptographic-asset",
    "name": "AES-128-GCM",
    "cryptoProperties": {
      "assetType": "algorithm",
      "algorithmProperties": {
        "primitive": "ae",
        "parameterSetIdentifier": "128",
        "mode": "gcm",
        "executionEnvironment": "software-plain-ram",
        "implementationPlatform": "x86_64",
        "certificationLevel": [ "none" ],
        "cryptoFunctions": [ "keygen", "encrypt", "decrypt", "tag" ],
        "classicalSecurityLevel": 128,
        "nistQuantumSecurityLevel": 1
      },
      "oid": "2.16.840.1.101.3.4.1.6"
    }
  }
]
```

License Compliance

CycloneDX facilitates open-source and commercial license compliance. By leveraging the licensing capabilities of CycloneDX, organizations can identify any licenses that may be incompatible or require specific compliance obligations, such as attribution or sharing of source code.

Open Source Licensing

The following is an example of a components license. CycloneDX communicates this information using the SPDX license IDs along with optionally including a Base64 encoded representation of the full license text.

```
"licenses": [
  {
    "license": {
      "id": "Apache-2.0",
      "acknowledgement": "declared",
      "text": {
        "contentType": "text/plain",
        "encoding": "base64",
        "content": "RW5jb2RlZCBsaWNlbnNlIHRleHQgZ29lcyBoZXJlLg=="
      },
      "url": "https://www.apache.org/licenses/LICENSE-2.0.txt"
    }
  }
]
```

SPDX license expressions are also fully supported.

```
"licenses": [
  {
    "expression": "(LGPL-2.1 OR BSD-3-Clause AND MIT)",
    "acknowledgement": "declared"
  }
]
```

Declared and Concluded Licenses

Declared licenses and concluded licenses represent two different stages in the licensing process within software development. Declared licenses refer to the initial intention of the software authors regarding the licensing terms under which their code is released. On the other hand, concluded licenses are the result of a comprehensive analysis of the project's codebase to identify and confirm the actual licenses of the components used, which may differ from the initially declared licenses. While declared licenses provide an upfront indication of the licensing intentions, concluded licenses offer a more thorough understanding of the actual licensing within a project, facilitating proper compliance and risk management.

Acknowledgement	Description
declared	Declared licenses represent the initial intentions of authors regarding the licensing terms of their code.
concluded	Concluded licenses are verified and confirmed.

Using Evidence To Substantiate Concluded Licenses and Track Copyrights

In addition to asserting the declared or concluded license(s) of a component, CycloneDX also supports evidence of other licenses and copyrights found in a given component. These licenses are "observed" in the course of analyzing a software project and form the necessary evidence to substantiate a "concluded" license. For example:

```
"evidence": {
  "licenses": [
    { "license": { "id": "Apache-2.0" } },
    { "license": { "id": "LGPL-2.1-only" } }
  ],
  "copyright": [
    { "text": "Copyright 2012 Acme Inc. All Rights Reserved." },
    { "text": "Copyright (C) 2004,2005 University of Example" }
  ]
}
```

Refer to the "Evidence" chapter for more information.

Commercial Licensing

CycloneDX can also help organizations manage their commercial software licenses by providing a clear understanding of what licenses are in use and which ones require renewal or additional purchases, which may impact the operational aspects of applications or systems. By leveraging CycloneDX for commercial license compliance, organizations can reduce the risks associated with license violations, enhance their license management practices, and align their SBOM practice with Software Asset Management (SAM) and IT Asset Management (ITAM) systems for enterprise visibility.

The following example illustrates a commercial license for a given component.

```
"licenses": [
  {
    "license": {
      "name": "Acme Commercial License",
      "licensing": {
        "licensor": {
          "organization": { "name": "Acme Inc" }
        },
        "licensee": {
          "organization": { "name": "Example Co." }
        },
        "purchaser": {
          "individual": {
            "name": "Samantha Wright",
            "email": "samantha.wright@gmail.com",
            "phone": "800-555-1212"
          }
        },
        "purchaseOrder": "PO-12345",
        "licenseTypes": [ "appliance" ],
        "lastRenewal": "2022-04-13T20:20:39+00:00",
        "expiration": "2023-04-13T20:20:39+00:00"
      }
    }
  }
]
```

All commercial license fields are optional. The licensor, licensee, and purchaser may be an organization or individual. Multiple license types may be specified and include:

License Type	Description
academic	A license that grants use of software solely for the purpose of education or research.
appliance	A license covering use of software embedded in a specific piece of hardware.
client-access	A Client Access License (CAL) allows client computers to access services provided by server software.
concurrent-user	A Concurrent User license (aka floating license) limits the number of licenses for a software application and licenses are shared among a larger number of users.
core-points	A license where the core of a computer's processor is assigned a specific number of points.
custom-metric	A license for which consumption is measured by non-standard metrics.
device	A license that covers a defined number of installations on computers and other types of devices.
evaluation	A license that grants permission to install and use software for trial purposes.
named-user	A license that grants access to the software to one or more pre-defined users.
node-locked	A license that grants access to the software on one or more pre-defined computers or devices.
oem	An Original Equipment Manufacturer license that is delivered with hardware, cannot be transferred to other hardware, and is valid for the life of the hardware.
perpetual	A license where the software is sold on a one-time basis and the licensee can use a copy of the software indefinitely.
processor-points	A license where each installation consumes points per processor.
subscription	A license where the licensee pays a fee to use the software or service.
user	A license that grants access to the software or service by a specified number of users.
other	Another license type.

Solutions supporting the Software Development Life Cycle (SDLC) typically involve open-source license compliance or intellectual property use cases. Whereas Software Asset Management (SAM) is primarily concerned with commercial license and procurement use cases. OWASP CycloneDX has extensive support for both and can be applied to any component or service within a BOM.

Security and Quality Considerations for Tracking Commercial Licenses

When a commercial software license expires or the software has reached its licensed limits, a cascading series of events may occur, leading to security and quality issues, posing significant risks to users and systems. One of the immediate concerns is the denial of service, where the software ceases to function. If the software is part of a larger system, the entirety of the system may be at risk. Moreover, expired or exceeded licenses may trigger altered application behavior, as certain features or functionalities tied to the license may become disabled or restricted. This alteration can introduce instability, unexpected errors, or even malicious behavior, potentially compromising the integrity and confidentiality of data.

Expired licenses may lead to being abruptly cut off from essential updates, patches, and support channels provided by the software vendor. This leaves systems vulnerable to newly discovered vulnerabilities and exploits, as security patches may no longer be available. Without access to ongoing support, users are left without recourse in the event of technical glitches or critical issues, leading to prolonged downtime and increased susceptibility to cyberattacks. Thus, ensuring software licenses remain current is vital for maintaining both the security and quality of software systems.

Providing commercial license information in CycloneDX BOMs offers a comprehensive solution to these challenges. By including license details within BOMs, software consumers gain transparency into the licensing status of components used within a software application. This transparency facilitates effective license management, enabling organizations to track and monitor license expiration dates and renewal requirements. Consequently, software consumers can proactively address license expirations, ensuring uninterrupted access to critical support services, including software updates and security patches. Overall, integrating commercial license information into CycloneDX BOMs enhances security, quality, and compliance across the software supply chain.

Establishing Relationships in CycloneDX

CycloneDX has a rich set of relationships that provide additional context and information about the objects in the BOM's inventory. All relationships in CycloneDX are expressed explicitly. Some relationships are declared through the natural use of the CycloneDX format. These include assemblies, dependencies, and pedigree. Other relationships are formed via references to the object's identity in the BOM, referred to as bom-ref. The combination of these two approaches dramatically simplifies the specification, providing necessary guardrails to prevent deviation of its usage and providing an easy path to supporting enveloped signing and other advanced usages.

Component Assemblies

Components in a BOM can be nested to form an assembly. An assembly is a collection of components that are included in a parent component. As an analogy, an automotive dashboard contains an instrument panel component. And the instrument panel component contains a speedometer component. This nested relationship is called an assembly in CycloneDX.

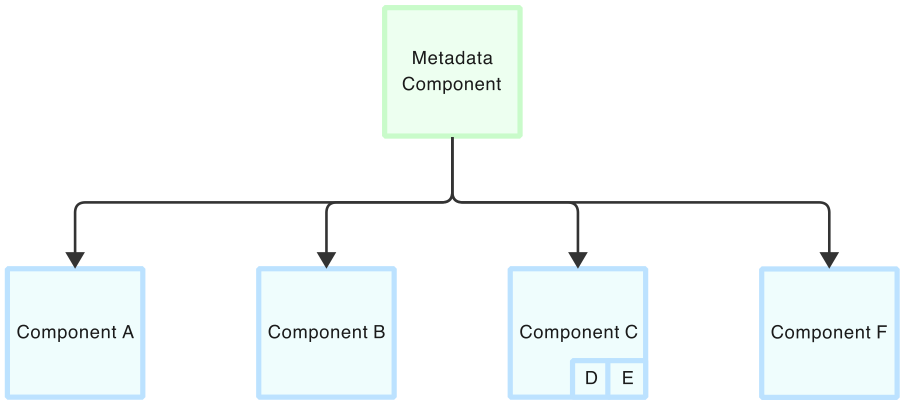
Software assemblies that can be represented in CycloneDX can range from large enterprise solutions comprising multiple systems, to cloud-native deployments containing extensive collections of related micro-services. Assemblies can also describe simpler inclusions, such as software packages that contain supporting files.

> *Assemblies, or leaves within an assembly, can independently be signed. BOMs comprising component assemblies from multiple suppliers can benefit from this capability. Each supplier can sign their respective assembly. The creator of final goods can then sign the BOM as a whole.*

The following example illustrates a simple component assembly. In this case, Acme Commerce Suite includes two other applications as part of its assembly.

```
"components": [
  {
    "type": "application",
    "name": "Acme Commerce Suite",
    "version": "2.0.0",
    "components": [
      {
        "type": "application",
        "name": "Acme Storefront Server",
        "version": "3.7.0",
      },
      {
        "type": "application",
        "name": "Acme Payment Processor",
        "version": "3.1.1",
      }
    ]
  }
]
```

In the following example, Components A-F are included in the metadata component, in this case, an application. Component C further includes an assembly of Components D and E which is how they were introduced as components of the application. An assembly is not an indication that Component C depends on Component D or E, rather Component C bundles Component D and E. If Component C depends on either D or E, dependency relationships should also be established.

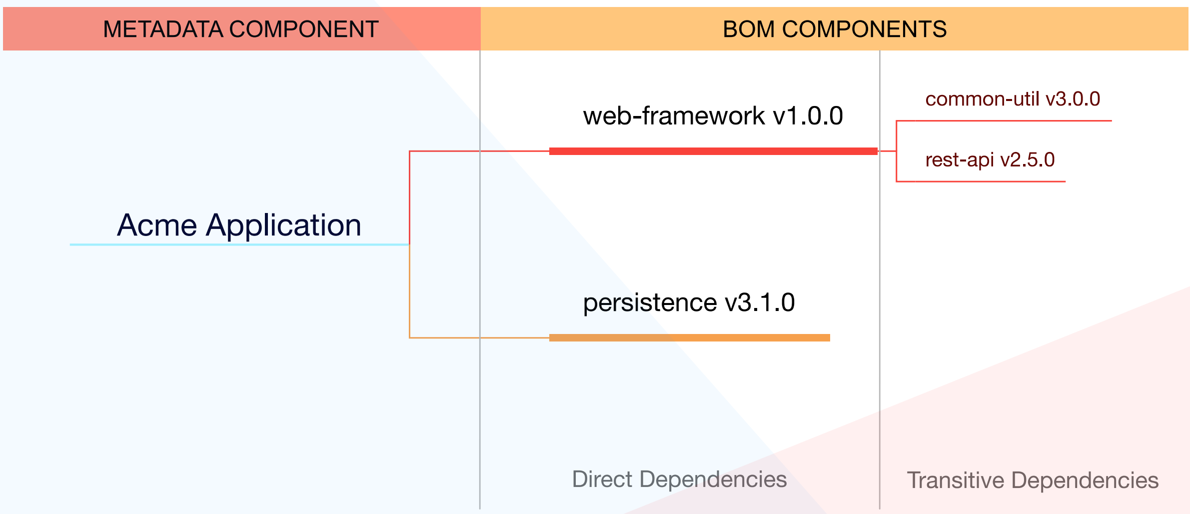


Service Assemblies

Services also have assemblies and work identically to those of components. While component assemblies describe a component that *includes* another component, service assemblies describe a service with other services *behind* it. A common cloud pattern is the use of API gateways which proxy and orchestrate connections to relevant microservices. The microservices themselves may not be directly accessible; rather, they are accessed exclusively through the API gateway. For this scenario, the API gateway service may contain an assembly of microservices behind it.

Dependencies

CycloneDX provides the ability to describe components and their dependency on other components. This relies on a component's bom-ref to associate the component with the dependency element in the graph. The only requirement for bom-ref is that it is unique within the BOM. Package URL (PURL) is an ideal choice for bom-ref as it will be both unique and readable. If PURL is not an option or not all components represented in the BOM contain a PURL, then UUID is recommended. A dependency graph is capable of representing both direct and transitive relationships. In CycloneDX representation dependencies, a dependency graph SHOULD be codified to be one node deep, meaning no nested child graphs. All relations are on the same level.



The dependency graph above can be codified with the following:

```
"dependencies": [
  {
    "ref": "acme-app",
    "dependsOn": [
      "pkg:maven/org.acme/web-framework@1.0.0",
      "pkg:maven/org.acme/persistence@3.1.0"
    ]
  },
  {
  "ref": "pkg:maven/org.acme/web-framework@1.0.0",
    "dependsOn": [
      "pkg:maven/org.acme/common-util@3.0.0",
      "pkg:maven/org.acme/rest-api@2.5.0"
    ]
  },
  {
    "ref": "pkg:maven/org.acme/common-util@3.0.0",
    "dependsOn": []
  },
  {
    "ref": "pkg:maven/org.acme/rest-api@2.5.0",
    "dependsOn": []
  }
]
```

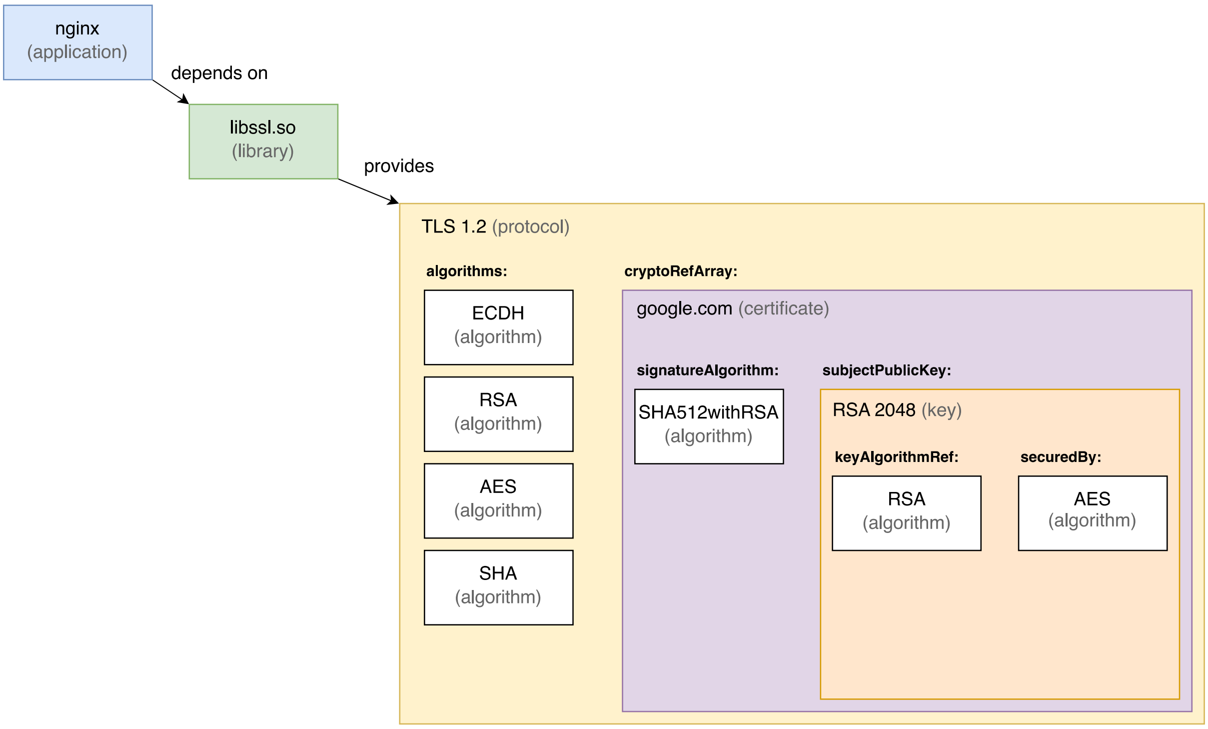
Components that do not have dependencies MUST be declared as empty elements within the graph. Components not represented in the dependency graph MAY have unknown dependencies. It is RECOMMENDED that implementations assume this to be opaque and not an indicator of a component being dependency-free.

As of CycloneDX v1.6, there are two types of dependencies: dependsOn and provides.

Dependency Type	Description
dependsOn	The bom-ref identifiers of the components or services that are dependencies of this dependency object.
provides	The bom-ref identifiers of the components or services that define a given specification or standard, which are provided or implemented by this dependency object. For example, a cryptographic library that implements a cryptographic algorithm. A component that implements another component does not imply that the implementation is in use.

The dependency type, dependsOn, is leveraged by classic SBOMs to define a complete graph of direct and transitive dependencies. However, for cryptographic and similar assets, "provides" allows for many additional use cases.

The following example shows an application (nginx) that uses the libssl cryptographic library. This library implements the TLSv1.2 protocol. The relationship between the application, the library and the protocol can be expressed by using the dependencies properties of the SBOM standard.



Refer to the [Authoritative Guide to CBOM](#) for in-depth information about leveraging CycloneDX for cryptographic use cases.

External References

External references provide a way to document systems, sites, and information that are relevant to a component, service, or the BOM itself. External references point to resources outside the object they're associated with and may be external to the BOM, or may refer to resources within the BOM.

External references are established through a URI (URL or URN) and, therefore, can accept any URL scheme, including https, mailto, tel, and dns. External references may also include formally registered URNs such as CycloneDX BOM-Link to reference CycloneDX BOMs or any object within a BOM. BOM-Link transforms applicable external references into relationships that can be expressed in a BOM or across BOMs.

External references provide an extensible and data-rich method of forming relationships.

Reference Type	Description
vcs	Version Control System
issue-tracker	Issue or defect tracking system, or an Application Lifecycle Management (ALM) system
website	Website
advisories	Security advisories
bom	Bill-of-materials (SBOM, OBOM, HBOM, SaaSBOM, etc)
mailing-list	Mailing list or discussion group
social	Social media account
chat	Real-time chat platform
documentation	Documentation, guides, or how-to instructions
support	Community or commercial support
source-distribution	The location where the source code distributable can be obtained. This is often an archive format such as zip or tgz. The source-distribution type complements use of the version control (vcs) type.
distribution	Direct or repository download location
distribution-intake	The location where a component was published to. This is often the same as "distribution" but may also include specialized publishing processes that act as an intermediary
license	The URL to the license file. If a license URL has been defined in the license node, it should also be defined as an external reference for completeness
build-meta	Build-system specific meta file (i.e. pom.xml, package.json, .nuspec, etc)

Reference Type	Description
build-system	URL to an automated build system
release-notes	URL to release notes
security-contact	Specifies a way to contact the maintainer, supplier, or provider in the event of a security incident. Common URIs include links to a disclosure procedure, a mailto (RFC-2368) that specifies an email address, a tel (RFC-3966) that specifies a phone number, or dns (RFC-4501) that specifies the records containing DNS Security TXT
model-card	A model card describes the intended uses of a machine learning model, potential limitations, biases, ethical considerations, training parameters, datasets
log	A record of events that occurred in a computer system or application, such as problems, errors, or information on current operations.
configuration	Parameters or settings that may be used by other components or services.
evidence	Information used to substantiate a claim.
formulation	Describes how a component or service was manufactured or deployed.
attestation	Human or machine-readable statements containing facts, evidence, or testimony
threat-model	An enumeration of identified weaknesses, threats, and countermeasures, dataflow diagram (DFD), attack tree, and other supporting documentation in human-readable or machine-readable format
adversary-model	The defined assumptions, goals, and capabilities of an adversary
risk-assessment	Identifies and analyzes the potential of future events that may negatively impact individuals, assets, and/or the environment. Risk assessments may also include judgments on the tolerability of each risk
vulnerability-assertion	A Vulnerability Disclosure Report (VDR) which asserts the known and previously unknown vulnerabilities that affect a component, service, or product including the analysis and findings describing the impact (or lack of impact) that the reported vulnerability has on a component, service, or product
exploitability-statement	A Vulnerability Exploitability eXchange (VEX) which asserts the known vulnerabilities that do not affect a product, product family, or organization, and optionally the ones that do. The VEX should include the analysis and findings describing the impact (or lack of impact) that the reported vulnerability has on the product, product family, or organization
pentest-report	Results from an authorized simulated cyberattack on a component or service, otherwise known as a penetration test

Reference Type	Description
static-analysis-report	SARIF or proprietary machine or human-readable report for which static analysis has identified code quality, security, and other potential issues with the source code
dynamic-analysis-report	Dynamic analysis report that has identified issues such as vulnerabilities and misconfigurations
runtime-analysis-report	Report generated by analyzing the call stack of a running application
component-analysis-report	Report generated by Software Composition Analysis (SCA), container analysis, or other forms of component analysis
maturity-report	Report containing a formal assessment of an organization, business unit, or team against a maturity model
certification-report	Industry, regulatory, or other certification from an accredited (if applicable) certification body
quality-metrics	Report or system in which quality metrics can be obtained
codified-infrastructure	Code or configuration that defines and provisions virtualized infrastructure, commonly referred to as Infrastructure as Code (IaC)
evidence	Data collected through various forms of extraction or analysis
formulation	The observed or declared formulas for how components or services were manufactured or deployed
poam	Plans of Action and Milestones (POAM) complement an "attestation" external reference. POAM is defined by NIST as a "document that identifies tasks needing to be accomplished. It details resources required to accomplish the elements of the plan, any milestones in meeting the tasks and scheduled completion dates for the milestones".
electronic-signature	An e-signature is commonly a scanned representation of a written signature or a stylized script of the persons name.
digital-signature	A signature that leverages cryptography, typically public/private key pairs, which provides strong authenticity verification.
rfc-9116	Document that complies with RFC-9116 (A File Format to Aid in Security Vulnerability Disclosure)
other	Use this if no other types accurately describe the purpose of the external reference

The following are example external references applied to a component:

```json
"components": [
  {
    "type": "application",
    "name": "portal-server",
    "version": "1.0.0",
    "externalReferences": [
      {
        "type": "advisories",
        "url": "https://example.org/security/feed/csaf"
      },
      {
        "type": "bom",
        "url": "https://example.org/support/sbom/portal-server/1.0.0",
        "hashes": [
          {
            "alg": "SHA-256",
            "content": "708f1f53b41f11f02d12a11b1a38d2905d47b099afc71a0f1124ef8582ec7313"
          }
        ]
      },
      {
        "type": "documentation",
        "url": "https://example.org/support/documentation/portal-server/1.0.0"
      }
    ]
  }
]
```

Establishing Relationships With BOM-Link

With CycloneDX, it is possible to reference a component, service, or vulnerability inside a BOM from other systems or other BOMs. This deep-linking capability is referred to as BOM-Link and is a formally registered URN, governed by IANA, and compliant with RFC-8141.

Syntax:

```
urn:cdx:serialNumber/version#bom-ref
```

Examples:

```
urn:cdx:f08a6ccd-4dce-4759-bd84-c626675d60a7/1
urn:cdx:f08a6ccd-4dce-4759-bd84-c626675d60a7/1#componentA
```

Field	Description
serialNumber	The unique serial number of the BOM. The serial number MUST conform to RFC-4122.
version	The version of the BOM. The default version is 1.
bom-ref	The unique identifier of the component, service, or vulnerability within the BOM.

There are many use cases that BOM-Link supports. Two common scenarios are:

- Reference one BOM from another BOM
- Reference a specific component or service in one BOM from another BOM

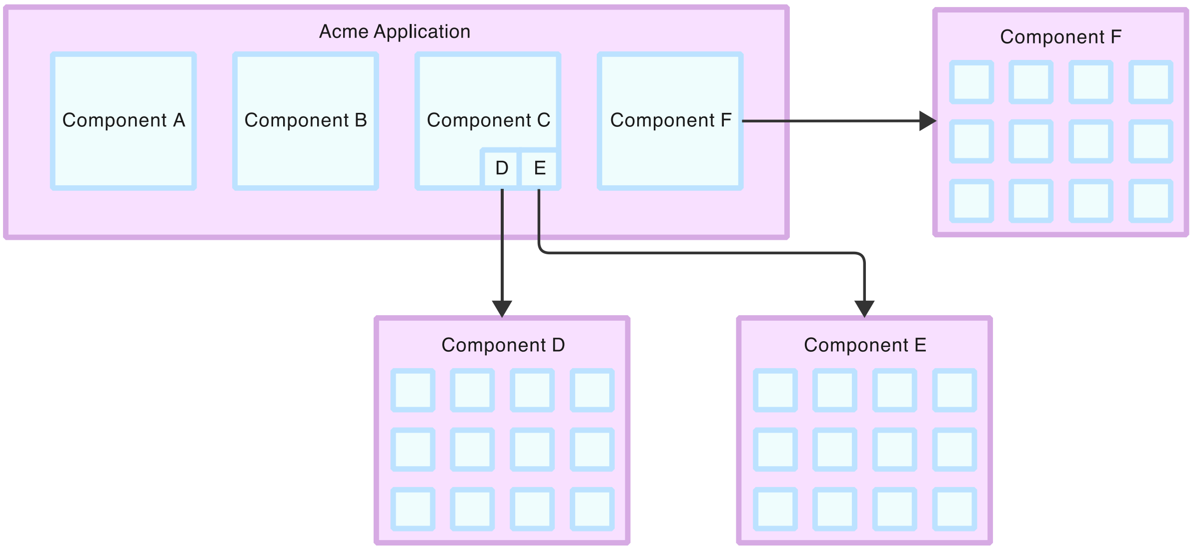
Linking to External BOMs

As mentioned earlier, external references point to resources outside the object they're associated with and may be external to the BOM, or may refer to resources within the BOM. External references can be applied to individual components, services, or to the BOM itself. For example, a component could specify an external reference pointing to the BOM describing that component.

```json
"externalReferences": [
  {
    "type": "bom",
    "url": "urn:cdx:bdd819e6-ee8f-42d7-a4d0-166ff44d51e8/5",
    "comment": "Refers to version 5 of a specific BOM.",
    "hashes": [
      {
        "alg": "SHA-256",
        "content": "c7be1ed902fb8dd4d48997c6452f5d7e509fbcdbe2808b16bcf4edce4c07d14e"
      }
    ]
  }
]
```

There are many common use cases where referencing external BOMs is desirable. One common case involves a component in a BOM, where the supplier of the component has published their own BOM specific to that component. The BOM for the application may simply list the component and refer to that component's externalized BOM for details of the inventory specific to that component. This is especially useful for proprietary components where the inventory may not be easily obtainable.

The following illustration provides an example of such a scenario. In this case, the supplier of the Acme Application includes Components A-F, Component C includes an assembly of D and E, and components D, E, and F are included in the BOM for Acme Application. The BOMs for D, E, and F are external and provided by other suppliers. The supplier of the Acme Application can leverage the BOMs provided by those suppliers by utilizing external references. Consumers should ensure they can resolve and process externally referencable BOMs when encountered.



The following example helps to illustrate what Component F may look like when represented in the BOM for Acme Application:


```json
"components": [
  {
    "bom-ref": "component-f",
    "type": "library",
    "name": "Component F",
    "version": "1.0.0",
    "externalReferences": [
      {
        "type": "bom",
        "url": "https://example.com/sbom/component-f-1.0.0.cdx.json",
        "hashes": [
          {
            "alg": "SHA-256",
            "content": "708f1f53b41f11f02d12a11b1a38d2905d47b099afc71a0f1124ef8582ec7313"
          }
        ]
      }
    ]
  }
]
```


Another common case involves individual BOMs, per layer, in a deployed stack. For example, a BOM may contain multiple components, each with external references to its own individual BOMs. A hardware component could link to the corresponding Hardware Bill of Material (HBOM), the operating system component could link to its corresponding SBOM, and an application component could do the same.

A third case involves a service defined in a BOM where the provider of the service has published a SaaSBOM containing the individual microservices that make up that consumer-facing service. They may also have published a corresponding SBOM defining the individual software components powering individual services.

Linking to Objects Within The Same BOM

With BOM-Link, relationships can also be established between objects in the same BOM. For example, let's establish a relationship where a component defines a threat model. In the example below, acme-application defines an external reference of type threat-model and uses BOM-Link to reference another component in the same BOM. The threat model components scope is excluded, indicating that it's omitted from inventory. The acme-threatmodel component in this example is a data component but could easily have been a file component. Using a data component allows for the inclusion of the threat model itself to be captured in the BOM. This approach may be ideal for audit use cases or for instances where access to external systems is prohibited, such as air-gapped environments.


```json
{
  "bomFormat": "CycloneDX",
  "specVersion": "1.6",
  "serialNumber": "urn:uuid:3e671687-395b-41f5-a30f-a58921a69b79",
  "version": 1,
  "components": [
    {
      "bom-ref": "acme-application",
      "type": "application",
      "name": "Acme Application",
      "version": "1.0.0",
      "externalReferences": [
        {
          "type": "threat-model",
          "url": "urn:cdx:3e671687-395b-41f5-a30f-a58921a69b79/1#acme-threatmodel"
        }
      ]
    },
    {
      "bom-ref": "acme-threatmodel",
      "type": "data",
      "name": "Acme Threat Model",
      "scope": "excluded",
      "data": [
        {
          "type": "other",
          "contents": {
            "attachment": {
              "encoding": "base64",
              "contentType": "application/pdf",
              "content": "VGhyZWF0IG1vZGVsIGdvZXMgaGVyZQ=="
            }
          }
        }
      ]
    }
  ]
}
```


Whether the goal is a separation of concerns or increased cost efficiency and quality, the modularity that CycloneDX provides is immensely powerful.

Linking External VEX to BOM Inventory

Vulnerability Exploitability eXchange (VEX) is a core capability of CycloneDX that can convey the exploitability of vulnerable components in the context of the product in which they're used. VEX information may be very dynamic and subject to change, while the product's SBOM will typically remain static until such time that the inventory changes. Therefore, it is recommended to decouple the VEX from the BOM. This allows VEX information to be updated without having to create and track additional BOMs.

In the following example, a vulnerability is identified in a component called Jackson Databind, and the VEX provides a direct link to the precise component within a BOM.

```json
"vulnerabilities": [
  {
    "id": "CVE-2018-7489",
    "source": {
      "name": "NVD",
      "url": "https://nvd.nist.gov/vuln/detail/CVE-2019-9997"
    },
    "analysis": {
      "state": "not_affected",
      "justification": "code_not_reachable",
      "response": ["will_not_fix", "update"],
      "detail": "An optional explanation of why the application is not affected by the vulnerable component."
    },
    "affects": [
      {
        "ref": "urn:cdx:3e671687-395b-41f5-a30f-a58921a69b79/1#jackson-databind-2.8.0"
      }
    ]
  }
]
```

Pedigree

CycloneDX can represent component pedigree including ancestors, descendants, and variants which describe component lineage from any viewpoint and the commits, patches, and diffs which make it unique. The addition of a digital signature applied to a component with detailed pedigree information serves as affirmation to the accuracy of the pedigree.

Pedigree	Description
ancestors	Describes zero or more components from which a component is derived. This is commonly used to describe forks from existing projects where the forked version contains a ancestor node containing the original component it was forked from.
descendants	Descendants are the exact opposite of ancestors. This provides a way to document all forks (and their forks) of an original or root component.
variants	Variants describe relations where the relationship between the components are not known. For example, if Component A contains nearly identical code to Component B. They are both related, but it is unclear if one is derived from the other, or if they share a common ancestor.

The following example illustrates two important aspects of pedigree, namely identity and provenance.

```
"components": [
{
    "type": "library",
    "group": "com.example",
    "name": "log4j-core",
    "version": "2.14.0",
    "purl": "pkg:maven/com.example/log4j-core@2.14.0?repository_url=registry.example.com",
    "pedigree": {
      "ancestors": [
        {
          "type": "library",
          "group": "org.apache.logging.log4j",
          "name": "log4j-core",
          "version": "2.14.0",
          "purl": "pkg:maven/org.apache.logging.log4j/log4j-core@2.14.0"
        }
      ]
    }
  }
]
```

The example above illustrates two important aspects of pedigree:

1) log4j-core from the Apache LOG4J 2™ project was modified. The modified version has an identity that is unique from its upstream source. Both the modified and original components are represented in the pedigree relationship.

2) According to the Package URL (purl), the original component was obtained from Maven Central (the default for Maven artifacts) while the modified component resides in a repository controlled by example.com. The provenance of the artifacts are maintained.

The pedigree capabilities in CycloneDX go much further than establishing relationships; the specification can also optionally provide transparency into the changes that were made and their purpose. For example, the precise commits made to the version control system can be represented.

```json
"pedigree": {
  "ancestors": [ ... ],
  "commits": [
    {
      "uid": "7638417db6d59f3c431d3e1f261cc637155684cd",
      "url": "https://location/to/7638417db6d59f3c431d3e1f261cc637155684cd",
      "committer": {
        "timestamp": "2022-02-13T20:20:39+00:00",
        "name": "Astra Snyder",
        "email": "astra.snyder@example.com"
      },
      "message": "Fixes security issue"
    }
  ]
}
```

Maintaining accurate pedigree information is especially important with open source components whos source code is readily available, modifiable, and redistributable. In the following example, a patch is described indicating that the purpose for the modification was to backport a security fix. In addition, the diff can be attached or referenced via a URL so that SBOM consumers can independently verify the validity and correctness of the patch.

```json
"pedigree": {
  "ancestors": [ ... ],
  "patches": [
    {
      "type": "backport",
      "diff": {
        "text": {
          "contentType": "text/plain",
          "encoding": "base64",
          "content": "ZXhhbXBsZSBkaWZmIGhlcmU="
        },
        "url": "https://example.com/path/to/changes.diff"
      },
      "resolves": [
        {
          "type": "security",
          "id": "CVE-2021-45105",
          "source": {
            "name": "NVD",
            "url": "https://nvd.nist.gov/vuln/detail/CVE-2021-45105"
          }
        }
      ]
    }
  ]
}
```

Composition Completeness and "Known Unknowns"

The inventory of components, services, and their relationships to one another can be described through the use of compositions. Compositions describe constituent parts (including components, services, and dependency relationships) and their completeness. The completeness of vulnerabilities expressed in a BOM may also be described. This allows BOM authors to describe how complete the BOM is or if there are components in the BOM where completeness is unknown or has been redacted.

Aggregate	Description
complete	The information is complete. No further relationships including constituent components, services, or dependencies are known to exist.
incomplete	The information is incomplete.
incomplete_first_party_only	The information is incomplete. Only relationships for first-party components, services, or their dependencies are represented.
incomplete_first_party_proprietary_only	The information is incomplete. Only relationships for third-party components, services, or their dependencies are represented, limited specifically to those that are proprietary.
incomplete_first_party_opensource_only	The information is incomplete. Only relationships for third-party components, services, or their dependencies are represented, limited specifically to those that are opensource.
incomplete_third_party_only	The information is incomplete. Only relationships for third-party components, services, or their dependencies are represented.
incomplete_third_party_proprietary_only	The information is incomplete. Only relationships for third-party components, services, or their dependencies are represented, limited specifically to those that are proprietary.
incomplete_third_party_opensource_only	The information is incomplete. Only relationships for third-party components, services, or their dependencies are represented, limited specifically to those that are opensource.
unknown	The information may be complete or incomplete. This usually signifies a 'best-effort' to obtain constituent components, services, or dependencies but the completeness is inconclusive.

The following illustrates how compositions can be used. In this example, there are three compositions.

1. In the first object, the component assembly and the dependencies of the application are both complete.
2. In the second object, the completeness of the component assembly is unknown.
3. In the third object, the component is listed in the BOM, but its information and completeness have been redacted.

```json
"compositions": [
  {
    "aggregate": "complete",
    "assemblies": [
      "pkg:maven/partner/shaded-library@1.0"
    ],
    "dependencies": [
      "acme-application-1.0"
    ]
  },
  {
    "aggregate": "unknown",
    "assemblies": [
      "pkg:maven/acme/library@3.0"
    ]
  },
  {
    "aggregate": "redacted",
    "assemblies": [
      "my-redacted-component"
    ]
  }
]
```

Formulation

CycloneDX can describe declared and observed formulations for reproducibility throughout the product lifecycle of components and services. This advanced capability provides transparency into how components were made, how a model was trained, or how a service was created or deployed. Generally, the formulation is externalized from the SBOM into a dedicated Manufacturing Bill of Materials (MBOM). The SBOM references the MBOM that describes the environment, configuration, tools, and all other considerations necessary to replicate a build with utmost precision. This capability allows other parties to independently verify inputs and outputs from a build which can increase the software's assurance.

Formulation establishes relationships with components and services, each of which can be referenced in a given formula through a series of workflows, tasks, and steps. As of this writing, the "Authoritative Guide to MBOM" is being drafted. When complete, it will serve as a reference for effectively using formulation for a wide variety of use cases.

The following example illustrates an SBOM where a component referenced the corresponding MBOM describing how the component was made. Independent access controls can be established by separating the SBOM inventory from potentially highly-sensitive MBOM data. For example, this allows organizations to provide SBOMs to a broader audience while keeping stricter control over who has access to the MBOM.

```
"externalReferences": [
  {
    "type": "formulation",
    "url": "https://example.com/mboms/acme-library-1.0.cdx.json",
    "hashes": [
      {
        "alg": "SHA-256",
        "content": "c7be1ed902fb8dd4d48997c6452f5d7e509fbcdbe2808b16bcf4edce4c07d14e"
      }
    ]
  }
]
```

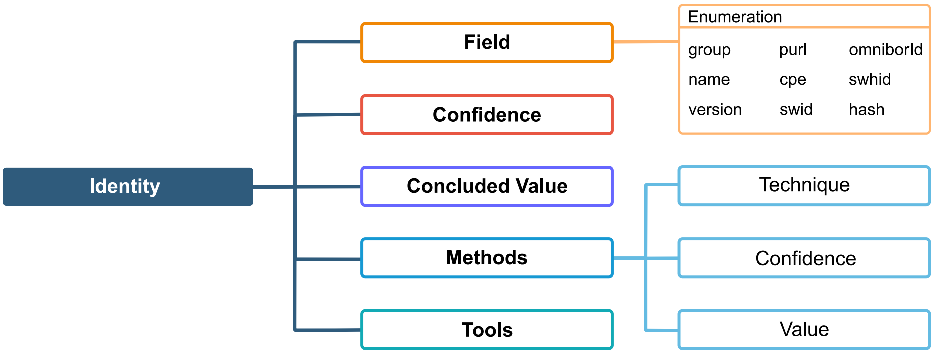
Evidence

As we've seen, a BOM is crucial for understanding the composition of the software and its associated risks. CycloneDX BOMs may include evidence substantiating the declared identity of components within the BOM. Additionally, the specification includes other observations about the component inventory such as multiple occurrences, call stack reachability, and evidence of licenses and copyrights.

Component Identity

CycloneDX includes evidence substantiating the declared identity of components within the BOM. This is vital for communicating the quality and general trustworthiness of the BOMs' contents. Evidence helps establish the accuracy of the BOM by validating that the declared components match the actual software components used.

Component identity evidence is made up of the following elements:



Field

The identity field of the component which the evidence describes.

Field	Description
group	The grouping name or identifier. This is often a shortened, single name of the company or project that produced the component, or its associated domain name.
name	The name of the component. This will often be a shortened, single name of the component.
version	The component version
purl	The Package URL (purl) specification
cpe	The Common Platform Enumeration (CPE) conforming to the CPE 2.2 or 2.3 specification
omniborId	The OmniBOR Artifact ID (gitoid)
swhid	The Software Heritage persistent identifier
swid	ISO-IEC 19770-2: Software Identification (SWID) Tags
hash	The cryptographic hash of the component

Confidence

Confidence is supported per-technique along with a cumulative of all methods used. The confidence is specified as a decimal, from 0 to 1, where 1 is 100% confidence.

Concluded Value

The value of the field (cpe, purl, etc) that has been concluded based on the aggregate of all methods (if available).

Methods

Multiple methods may be specified. Each method includes the specific technique used, the confidence of each technique, and the value of the evidence that the technique revealed.

Techniques

The technique used in this method of analysis.

Technique	Description
source-code-analysis	Examines the source code without executing it
binary-analysis	Examines a compiled binary through reverse engineering, typically via disassembly or bytecode reversal
manifest-analysis	Examines a package management system such as those used for building software or installing software
ast-fingerprint	Examines the Abstract Syntax Tree (AST) of source code or a compiled binary
hash-comparison	Evaluates the cryptographic hash of a component against a set of pre-computed hashes of identified software
instrumentation	Examines the call stack of running applications by intercepting and monitoring application logic without the need to modify the application
dynamic-analysis	Evaluates a running application
filename	Evaluates file name of a component against a set of known file names of identified software
attestation	A testimony to the accuracy of the identify of a component made by an individual or entity
other	Any other technique

Tools

The tools (components or services) which extracted the evidence, performed the analysis, or evaluated the results.

Example #1

The following example illustrates how different methods can be combined to substantiate a component's identity.


```
"components": [
  {
    "group": "com.google.code.findbugs",
    "name": "findbugs-project",
    "version": "3.0.0",
    "purl": "pkg:maven/com.google.code.findbugs/findbugs-project@3.0.0",
    "evidence": {
      "identity": [
        {
          "field": "purl",
          "confidence": 1,
          "concludedValue": "pkg:maven/com.google.code.findbugs/findbugs-project@3.0.0",
          "methods": [
            {
              "technique": "filename",
              "confidence": 0.1,
              "value": "findbugs-project-3.0.0.jar"
            },
            {
              "technique": "hash-comparison",
              "confidence": 0.8,
              "value": "7c547a9d67cc7bc315c93b6e2ff8e4b6b41ae5be454ac249655ecb5ca2a85abf"
            }
          ]
        }
      ]
    }
  }
]
```


Example #2

In the following example, two identity objects provide lower-confidence alternate CPEs. Vulnerability databases such as the National Vulnerability Database, which rely exclusively on CPE, often have erroneous or data fidelity issues that prevent precise reporting on affected products. CycloneDX solves this issue by allowing BOM authors to assert component identity, and optionally specify evidence of other possible identifiers to aid in vulnerability identification.


```
"evidence": {
  "identity": [
    {
      "field": "cpe",
      "confidence": 0.4,
      "concludedValue": "cpe:2.3:a:acme:acme-application:1.0.0:*:*:*:*:*:*:*"
    },
    {
      "field": "cpe",
      "confidence": 0.4,
      "concludedValue": "cpe:2.3:a:acme-systems:acme-application:1.0.0:*:*:*:*:*:*:*"
    }
  ]
}
```

Technique Confidence Recommendations

The following are recommendations for tool creators and BOM consumers. Each technique is a general category. Tools may employ general purpose or highly specialized rules and analysis, each with varying degrees of confidence.

Technique	Confidence	Guidance
source-code-analysis	0.3 - 1.0	Confidence will vary based on rules, type of analyzers used, or 1:1 matching of source with a known good dataset.
binary-analysis	0.2 - 0.7	The individual rules, analyzers, and dataset coverage will influence confidence.
manifest-analysis	0.4 - 0.6	Manifests have known limitations and abuse cases and have moderate confidence.
ast-fingerprint	0.3 - 1.0	Wide range of possible confidence due to source and binary variations, but it has the potential for precise results.
hash-comparison	0.7 - 1.0	Can successfully match components given a large dataset. Confidence may vary based on the cryptographic hash function used and its resistance to collisions.
instrumentation	0.3 - 0.8	Confidence similar to source-code-analysis with the added benefit of supporting call-stack evidence
dynamic-analysis	0.2 - 0.6	Low to moderate confidence due to the "black box" approach of many tools.
filename	0 - 0.1	Filename matching is low-confidence
attestation	0.7 - 1.0	The testimony of a supplier or trusted third-party, especially when legally binding, may have high confidence.

Occurrences

CycloneDX provides a mechanism to describe identical components spread across multiple locations. For example, a component may be used by a command-line tool and included as part of a user interface. As such, the component may be installed in multiple locations on the filesystem. CycloneDX provides a way to represent this using evidence.


```json
"components": [
  {
    "type": "library",
    "name": "acme-persistence",
    "version": "1.0.0",
    "evidence": {
      "occurrences": [
        {
          "bom-ref": "d6bf237e-4e11-4713-9f62-56d18d5e2079",
          "location": "/path/to/component"
        },
        {
          "bom-ref": "b574d5d1-e3cf-4dcd-9ba5-f3507eb1b175",
          "location": "/another/path/to/component"
        }
      ]
    }
  }
]
```


Reachability Using Call Stacks

Evidence of the components use through the call stack can be described using CycloneDX. This helps organizations understand the reachability and potential impact of a specific software component. By tracing the call stack, which describes how different components interact with each other, BOM producers and consumers have an elevated level of confidence that a component or vulnerable function within a component is invoked or not.


```json
"callstack": {
  "frames": [
    {
      "package": "com.apache.logging.log4j.core",
      "module": "Logger.class",
      "function": "logMessage",
      "parameters": [
        "com.acme.HelloWorld", "Level.INFO", "null", "Hello World"
      ],
      "line": 150,
      "column": 17,
      "fullFilename": "/path/to/log4j-core-2.14.0.jar!/org/apache/logging/log4j/core/Logger.class"
    },
    {
      "module": "HelloWorld.class",
      "function": "main",
      "line": 20,
      "column": 12,
      "fullFilename": "/path/to/HelloWorld.class"
    }
  ]
}
```

License and Copyright

CycloneDX incorporates SPDX license IDs and expressions to document stated licenses of open-source components and individual source files. Observed licenses and copyright statements are also fully supported in the form of evidence. In OpenChain terms, a CycloneDX BOM is classified as a compliance artifact.

Organizations seeking OpenChain conformance should review the specification and ensure all verification requirements are met, including fully documented processes for how the CycloneDX BOMs were created, distributed, and archived. The CycloneDX BOM Repository Server is a simple and effective way to automate the publishing, versioning, and archiving of BOMs.

```
"evidence": {
  "licenses": [
    {
      "license": {
        "id": "Apache-2.0",
        "url": "http://www.apache.org/licenses/LICENSE-2.0"
      }
    },
    {
      "license": {
        "id": "LGPL-2.1-only",
        "url": "https://opensource.org/licenses/LGPL-2.1"
      }
    }
  ],
  "copyright": [
    { "text": "Copyright 2012 Amce Inc. All Rights Reserved." },
    { "text": "Copyright (C) 2004,2005 Example Co" }
  ]
}
```

Scenarios and Recommendations

The following recommendations are for common scenarios that are frequently cited or inquired about by the CycloneDX community.

General Guidance

- The SBOM should have a single bom.metadata.component without subcomponents
- The SBOM should describe the software components and external services the application depends on in bom.components and bom.services, respectively
- The SBOM should include as much information about the components and services as possible
- The SBOM should describe the dependencies between software components and any services
- The SBOM should describe the lifecycles involved in the creation of the SBOM
- The SBOM should provide evidence of component identity, the methods and techniques used, and their associated confidence
- The SBOM should provide evidence of observed licenses and copyright statements

Microservice

- Each microservice should have an independent SBOM
- Optionally, a SaaSBOM can be leveraged to describe the inventory of all services that make up an application
    - Each service in the SaaSBOM can then reference the SBOM specific to that service

Single Application (monolith, mobile app, etc)

- Optionally, the runtime environment and configuration of the application may also be specified

Multi-Product Solution

- The SBOM should have a single bom.metadata.component and leverage subcomponents
- The "solution" is the bom.metadata.component. For each product included, ensure each is listed as a subcomponent of bom.metadata.component

Multi-Module Product

- The SBOM should have a single bom.metadata.component without subcomponents
- Each module should be its own component, specified under bom.components. Each module may then either:
    - Include subcomponents, thus creating a hierarchy, or
    - Use external references to link to each modules independent SBOM

Using Modified Open Source Software

- Include component pedigree for each modified open source component

SBOM as Resource Locator

- Use of external references transforms CycloneDX into a "table of contents" for all relevant information about a product or any component included in a product.
- Possibilities include referencing threat models, maturity models, and quality metrics

SBOM in Release Management

- For products defined in bom.metadata.component, include machine-readable release notes
- Create a publishing process for CycloneDX release notes which transforms them into PDF, Markdown, HTML, or plain text
- Leverage custom lifecycles and properties for release management and governance
- Sign SBOMs prior to distribution

Extensibility

Multiple extension points exist throughout the CycloneDX object model, allowing fast prototyping of new capabilities and support for specialized and future use cases. The CycloneDX project maintains extensions that are beneficial to the larger community. The project encourages community participation and the development of extensions that target specialized or industry-specific use cases.

There are three primary means of extending CycloneDX.

- CycloneDX properties
- CycloneDX properties using registered namespace
- XML extensions

Note on hardened schemas: The XML and JSON schemas are hardened by design. This prevents unexpected markup, object types, and values from being present in the SBOMs that have not been pre-defined in the schemas. Hardened schemas are required for many high-assurance use cases. The security protections inherent in hardened schemas benefit the entire CycloneDX community. While these protections are highly beneficial, they do restrict serialization formats that are not extensible by design, most notably JSON.

CycloneDX Properties

The CycloneDX standard is fully extensible, allowing for complex data to be represented in the BOM that is not provided by the core specification. In many cases, name-value pairs are a simple option. CycloneDX supports Properties which is a name-value store that can be used to describe additional data about the components, services, or the BOM that isn't native to the core specification. Unlike key-value stores, properties support duplicate names, each potentially having different values. CycloneDX properties are a core part of the specification and are supported in all serialization formats, including XML, JSON, and protocol buffers.

JSON Example

```json
"properties": [
  {
    "name": "Foo",
    "value": "Bar"
  }
]
```

XML Example

```xml
<properties>
    <property name="Foo">Bar</property>
</properties>
```

CycloneDX Properties and Registered Namespaces

The CycloneDX standard does not impose restrictions on the property names used. However, standardization can assist tool implementers and BOM consumers. CycloneDX achieves this through formally registered namespaces. These namespaces prefix the property name and are defined by the organization or project that registered the namespace.

Namespaces are hierarchical and delimited with a : and may optionally start with urn:. Examples include:

```
cdx:gomod:binary
cdx:npm:package:bundled
cdx:pipenv:package
```

Organizations and open source projects can register a dedicated namespace at the CycloneDX Property Taxonomy repository on GitHub. https://github.com/CycloneDX/cyclonedx-property-taxonomy

XML Extensions

XML is extensible by design. CycloneDX is a hardened schema, but it does allow for additional XML elements so long as they reside in a different namespace. This extensibility allows for representing more complex data structures in CycloneDX that would not otherwise be supported. One such extension commonly used is XML Signature, used for enveloped signing.

```xml
<bom xmlns="http://cyclonedx.org/schema/bom/1.6"
     serialNumber="urn:uuid:3e671687-395b-41f5-a30f-a58921a69b79"
     version="1">
    <components>
        ...
    </components>
    <ds:Signature xmlns:ds="http://www.w3.org/2000/09/xmldsig#">
        <ds:SignedInfo>
            <ds:CanonicalizationMethod Algorithm="http://www.w3.org/TR/2001/REC-xml-c14n-20010315"/>
            <ds:SignatureMethod Algorithm="http://www.w3.org/2000/09/xmldsig#rsa-sha1"/>
            <ds:Reference URI="">
                <ds:Transforms>
                    <ds:Transform Algorithm="http://www.w3.org/2000/09/xmldsig#enveloped-signature"/>
                </ds:Transforms>
                <ds:DigestMethod Algorithm="http://www.w3.org/2000/09/xmldsig#sha1"/>
                <ds:DigestValue>sZjV4XcMOuD6NA9bXEd2sGWQYEO=</ds:DigestValue>
            </ds:Reference>
        </ds:SignedInfo>
        <ds:SignatureValue>...</ds:SignatureValue>
        <ds:KeyInfo xmlns:ds="http://www.w3.org/2000/09/xmldsig#">
            <ds:X509Data>
                <ds:X509SubjectName>CN=bomsigner,OU=development,O=cyclonedx</ds:X509SubjectName>
                <ds:X509Certificate>...</ds:X509Certificate>
            </ds:X509Data>
            <ds:KeyValue>
                <ds:RSAKeyValue>
                    <ds:Modulus>...</ds:Modulus>
                    <ds:Exponent>AQAB</ds:Exponent>
                </ds:RSAKeyValue>
            </ds:KeyValue>
        </ds:KeyInfo>
    </ds:Signature>
</bom>
```

Appendix A: Glossary

- Chain of custody - Auditable documentation of point of origin as well as the method of transfer from point of origin to point of destination and the identity of the transfer agent.
- Component function - The purpose for which a software component exists. Examples of component functions include parsers, database persistence, and authentication providers.
- Component type - The general classification of a software components architecture. Examples of component types include libraries, frameworks, applications, containers, and operating systems.
- Direct dependency - A software component that is referenced by a program itself.
- Package manager - A distribution mechanism that makes software artifacts discoverable by requesters.
- Package URL (PURL) - An ecosystem-agnostic specification which standardizes the syntax and location information of software components.
- Pedigree - Data which describes the lineage and/or process for which software has been created or altered.
- Point of origin - The supplier and associated metadata from which a software component has been procured, transmitted, or received. Package repositories, release distribution platforms, and version control history are examples of various points of origin.
- Procurement – The process of agreeing to terms and acquiring software or services for later use.
- Provenance - The chain of custody and origin of a software component. Provenance incorporates the point of origin through distribution as well as derivatives in the case of software that has been modified.
- Software Identification (SWID) - An ISO standard that formalizes how software is tagged.
- Software Package Data Exchange (SPDX) - A Linux Foundation project which produces a software bill of materials specification and a standardized list of open source licenses.
- Third-party component – Any software component not directly created including open source, "source available", and commercial or proprietary software.
- Transitive dependency - A software component that is indirectly used by a program by means of being a dependency of a dependency.

Appendix B: References

The following resources may be useful to users and adopters of this standard:

- NTIA Multistakeholder Process on Software Component Transparency, Framing Working Group. (21 October 2021). *Framing Software Component Transparency: Establishing a Common Software Bill of Materials (SBOM), Second Edition*. https://www.ntia.gov/files/ntia/publications/ntia_sbom_framing_2nd_edition_20211021.pdf
- NTIA. (12 July 2021). *The Minimum Elements for Software Bill of Materials*. https://www.ntia.gov/files/ntia/publications/sbom_minimum_elements_report.pdf
- The White House. (12 May 2021). *Executive Order on Improving the Nation's Cybersecurity*. https://www.whitehouse.gov/briefing-room/presidential-actions/2021/05/12/executive-order-on-improving-the-nations-cybersecurity/
- SPDX License IDs
- SPDX License List
- OpenChain
- OWASP CycloneDX
- OWASP CycloneDX Tool Center
- OWASP CycloneDX BOM Repository Server
- OWASP Dependency-Track
- OWASP Software Component Verification Standard (SCVS)
- OWASP Software Component Verification Standard (SCVS) BOM Maturity Model

OWASP