RUNSAFE SECURITY



RunSafe Identify:

Build-Time SBOM Generation and Vulnerability Management

Why observing a software build produces more accurate Software Bills of Materials and what that makes possible for vulnerability management.

RunSafeSecurity.com

Table of Contents

03 Executive Summary

04 Introduction: Why SBOM Accuracy Matters Now

05 SBOMs Designed for the Wild West of Embedded Software Development

08 Inside RunSafe's Build-Time Approach

10 What to Expect from a RunSafe SBOM

14 From Inventory to Vulnerability Management

15 Integrating RunSafe Identify into Your Software Development Lifecycle

16 Compliance Requirements Identify Supports

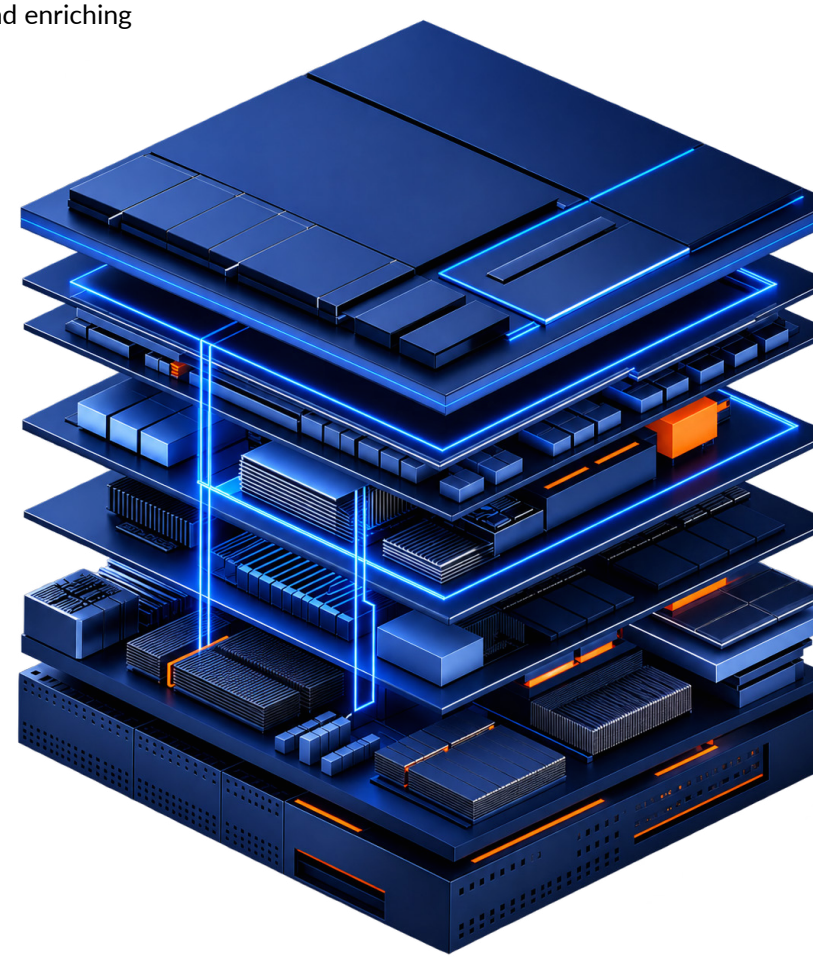
17 Summary

Executive Summary

Software Bills of Materials (SBOMs) have become foundational to vulnerability management and regulatory compliance, but their value depends on the accuracy of the data they collect. An SBOM that omits components can leave vulnerabilities undiscovered, while one that misidentifies software can create false positives and unnecessary remediation. These challenges are especially pronounced in embedded C and C++ software environments, where dependencies may be statically linked, vendored, copied directly into a project, installed at the system level, or acquired dynamically during the build.[1]

This white paper explains how RunSafe addresses these challenges by **generating SBOMs at build time,** starting with the files that actually participate in a specific build, and then using provenance, package, operating system, RTOS, repository, and project-specific context to resolve those files into accurate component identities.

A RunSafe SBOM combines detailed evidence of what was actually built, along with a practical package-level view for vulnerability and license management. More importantly, the SBOM becomes the foundation for the ongoing process of matching components to vulnerabilities, prioritizing exposure, enforcing policy, maintaining compliance evidence, and enriching component intelligence as products evolve.



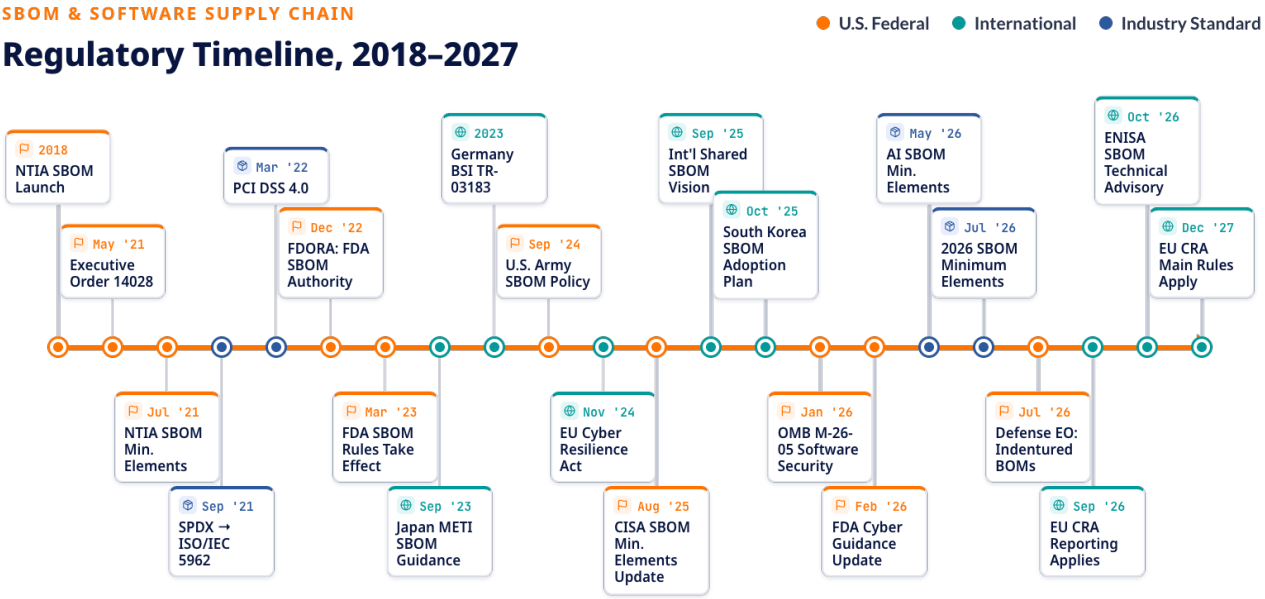
Introduction: Why SBOM Accuracy Matters Now

A Software Bill of Materials is a machine-readable inventory of the components, libraries, and files that make up a software product, along with information such as supplier, version, and license.

Governments and large buyers increasingly treat the SBOM as a baseline artifact of software transparency. The European Union Cyber Resilience Act (CRA), United States Food and Drug Administration (FDA) premarket cybersecurity requirements, and United States Army acquisition guidance all point in the same direction: **manufacturers are expected to know, document, and stand behind what is inside their software.**[2]

That makes SBOM accuracy the starting point for everything that comes after. Missing components can create vulnerability blind spots, while incorrectly identified components can lead to false positives and unnecessary remediation. And while these challenges are especially visible in complex embedded software environments, the same principle applies across software development: **the quality of vulnerability and compliance decisions is only as good as the software inventory behind them.**

RunSafe Security supports the full SBOM generation-to-management lifecycle within an organization's existing development ecosystem and is specifically designed to improve SBOM accuracy across environments. The remainder of this paper explains how RunSafe Identify discovers what went into a build, and how that **inventory supports the broader work of vulnerability management, license compliance, risk prioritization, policy enforcement, and reporting across the software lifecycle.**



SBOMs Designed for the Wild West of Embedded Software Development

While many SBOM generators are adequate for package-managed languages, most struggle with accuracy for complex embedded development environments.

A useful SBOM has to answer two questions accurately: what software actually went into this build (discovery), and is each component identified precisely enough to create an accurate map of vulnerabilities, licenses, and provenance (identity)?

For embedded software written in C and C++, both questions are unusually hard to answer. RunSafe Identify is designed to cover package-managed ecosystems and open-source use cases that other generators handle, and generate trustworthy SBOMs for C and C++ codebases that they do not.

A useful SBOM has to answer two questions accurately:

01 DISCOVERY
What software actually went into this build?

02 IDENTITY
Is each component identified precisely enough to create an accurate map of vulnerabilities, licenses, and provenance?

Why C and C++ Break Traditional SBOM Tools

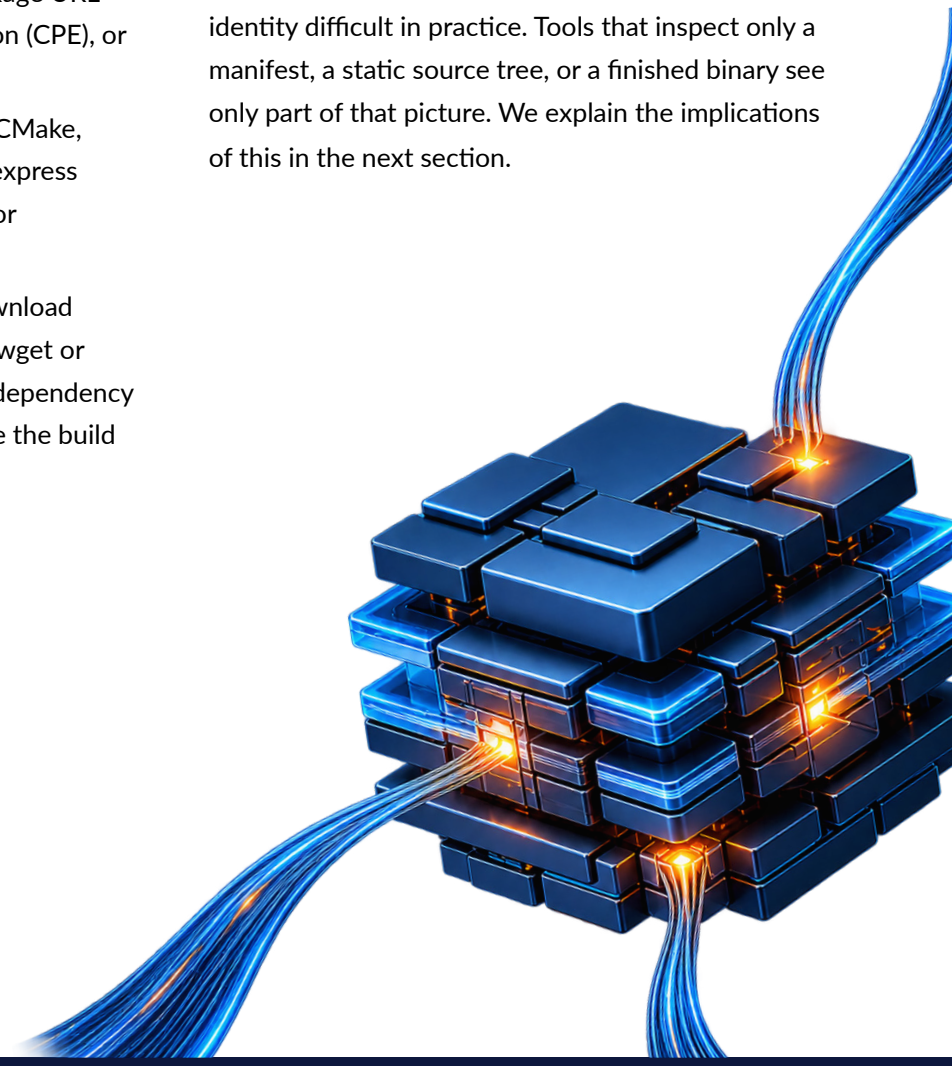
Modern software languages such as Python, JavaScript, and Rust have centralized registries and standard manifest files, including requirements.txt, package.json, and Cargo.toml. Those files provide a structured record of dependency names, versions, hashes, and licenses.

C and C++ have no universally adopted equivalent. Dependencies enter a project through several informal or project-specific mechanisms, each leaving a different metadata trail[1]:

- **Copy-paste integration**. Some libraries instruct developers to copy selected source and header files directly into a project. The Mongoose embedded web server, for example, can be integrated by copying mongoose.c and mongoose.h into the source tree, after which those files look indistinguishable from first-party code without in-depth analysis.
- **Vendored third-party directories.** A project may contain an entire third-party codebase with no manifest, package record, or link to the original repository. OpenCV's 3rdparty/ directory, for example, contains libraries such as libwebp that a manifest scanner may never see.

- **System-level dependencies.** A build may link against libraries installed on the operating system rather than stored in the project repository.
- **Static libraries.** A build may link a precompiled static library (a .a or .lib archive) whose object code is merged directly into the final executable. Unlike a shared library, it leaves no separate file on the target and no dynamic linking entry naming the dependency. Static libraries also frequently arrive prebuilt from chip vendors or SDKs with no accompanying source or version metadata.[3]
- **Git submodules.** Submodules identify a repository and commit, but they often pin to a commit hash rather than a semantic version, and that commit may not map cleanly to a Package URL (PURL), Common Platform Enumeration (CPE), or vulnerability record.
- **Build-system-defined dependencies.** CMake, Ninja, Bazel, and other build systems express dependencies differently, with behavior determined by the individual project.
- **Dynamic acquisition.** Some builds download source or libraries using tools such as wget or curl while the build is running, so the dependency may not exist in the source tree before the build begins.
- **Build configuration.** The same source tree can produce very different binaries depending on configure flags, CMake options, or preprocessor definitions. A repository may reference several open source components, but the configuration selected for a specific build determines which of them, if any, are compiled in. curl, for example, can be built against OpenSSL, wolfSSL, Mbed TLS, or GnuTLS as its TLS backend, and only the one chosen at configure time ends up in the product. A scan of the source tree reports every possibility, while the shipped firmware contains just one.

This fragmentation is what makes both discovery and identity difficult in practice. Tools that inspect only a manifest, a static source tree, or a finished binary see only part of that picture. We explain the implications of this in the next section.



Four Approaches to SBOM Generation

Most SBOM tools use one of a few common generation methods. Each can be useful, and are defined by where and when it looks at the software. A helpful way to compare them is by the kind of SBOM they produce, which reflects the vantage point they work from.[3]

METHOD	STAGE	HOW IT WORKS	LIMITATION
Source or manifest scanning (including package-manager records)	Source	Reads declared dependencies, package manifests, or installed-package records before the build	Depends on manifests that are weak, optional, or absent in embedded C and C++ and reports what the project declares rather than what the build includes
Binary or post-build scanning	Binary	Inspects the finished executable, library, firmware image, or container	Can miss statically linked or absorbed components and often infers identity from signatures
Manual creation	Varies	Engineers catalog components by hand	Does not scale reliably across products, configurations, and releases, and is prone to user error
RunSafe Identify build-time generation	Build-time	Records the files as the project compiles, links, and fetches dependencies	Requires access to the build

Every method makes some inferences. The difference is the point in time where those decisions are made. Source and manifest tools work before the build and describe what the project declares it will use. Binary scanners work after the build and reason backward from the finished artifact. Build-time generation runs during the build and records the files that actually participate as the software is assembled.

For the discovery question in particular, the timing and method are what allow a build-time tool to capture components that the other methods routinely miss or incorrectly declare.

Inside RunSafe's Build-Time Approach

RunSafe Identify generates an SBOM by observing the build rather than examining only its inputs or its final outputs. As the project compiles, links, and fetches dependencies, Identify records the files that participate, including application code, libraries, source and header files, object files, and binaries, and extracts the provenance and identity information available in each. Because the inventory is tied to the build as it happens, it reflects the specific software configuration being produced.

This matters in C and C++ because two builds from the same repository can contain different software dependencies. Compiler flags, target platforms, optional features, and build-system settings all change which source files, libraries, and dependencies are included. Build-time observation captures those differences instead of treating the repository as a fixed inventory.

This is also where the cases that defeat other tools become visible. Build-time SBOMs capture statically linked libraries, code vendored into the source tree, files copied directly into the project, dependencies downloaded during the build, and proprietary components that carry no public signature because they are recorded as they participate — even when they leave little or no distinguishable trace in a manifest or a finished binary.[4]

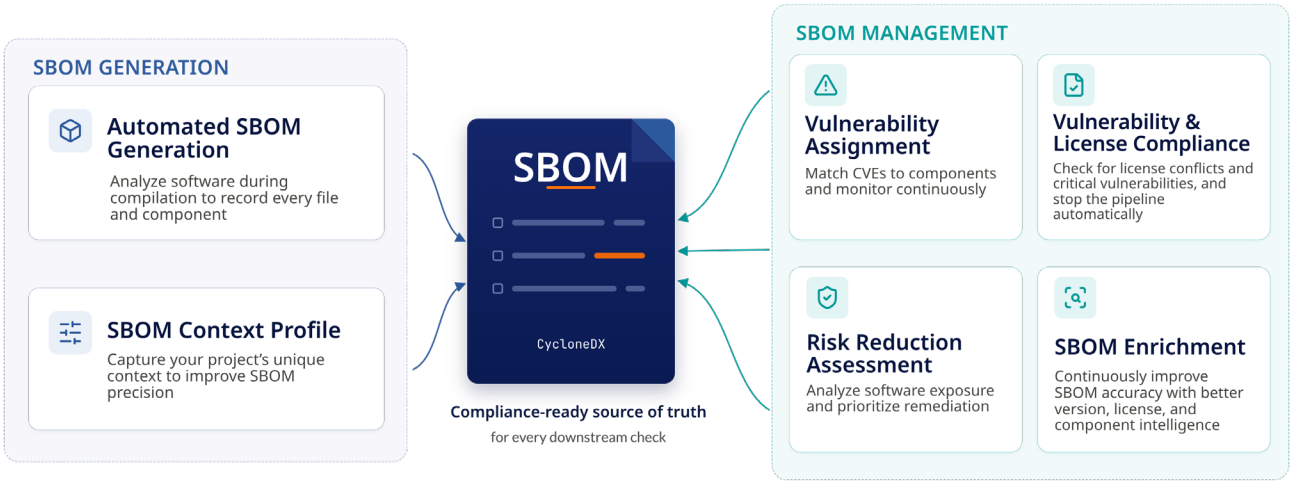


Figure 1. Automated build-time discovery and the SBOM Context Profile (the project-specific rules described later in this paper) produce a compliance-ready SBOM that feeds continuous management, including vulnerability assignment, license compliance, risk reduction, and ongoing enrichment.

What RunSafe Identify Captures During the Build

For the files and dependencies that participate in the build, Identify can capture and derive information that includes:

- File paths and relationships that show where a component appeared and how it was used
- Copyright, authorship, license, and other provenance information embedded in source files
- The individual source, header, object, library, binary, and supporting files used by the build
- Build-specific dependencies selected by the target platform and compile-time configuration
- System-installed libraries brought into the build from outside the repository
- Sub-packages supplied by a real-time operating system (RTOS), such as a version of GCC provided with a QNX installation5
- Files and components downloaded while the build is running
- Repository, package-manager, operating-system, RTOS, and other identity signals that help associate files with packages
- Component identities used to support PURL, CPE, vulnerability, and license matching

The result is an inventory based on what was observed in a specific build rather than everything present in the source tree or everything declared in a manifest. That improves both completeness and precision: **Identify captures components that other methods overlook while leaving out dependencies that exist in the repository but were never used.**



Figure 2. RunSafe's file-based SBOM provides greater precision, recall, and completeness than other methods and is further enhanced by our SBOM enrichment process.

What to Expect from a RunSafe SBOM

Because RunSafe's SBOM approach starts with files rather than packages, there are a few important distinctions to keep in mind.

Your SBOM Will Be Larger

With default settings, a RunSafe SBOM will be substantially larger than one produced by a package-manager-based or binary-scanning tool. That difference is a direct result of the amount and quality of evidence preserved through RunSafe's build-time approach.

Many SBOM tools produce a single entry for each package they can identify, whereas RunSafe Identify digs deeper into the individual files used by the package for that particular build.

Therefore, a third-party library containing hundreds of source files can produce hundreds of file components, each carrying available information such as its path, hash, license, copyright, authorship, provenance, and relationship to a parent package.

While this level of detail can be overwhelming to read directly, industry-recognized SBOMs are generated in machine-readable formats designed for processing by tooling. RunSafe generates a CycloneDX SBOM, and the RunSafe Security Platform presents the SBOM in a searchable, easier-to-understand view.

Importantly, this file-level detail makes the SBOM more actionable. By identifying the specific files included in a build, RunSafe can automatically triage out vulnerabilities that don't apply to the software actually present, reducing false positives and unnecessary investigation. Instead of treating every vulnerability associated with a package as equally relevant, teams can focus their remediation efforts on the vulnerabilities that warrant attention. **The result is an SBOM that not only documents software composition but also helps teams make faster, more informed security decisions.**[6]

Here is an example of where the extra detail pays off. A package-level SBOM records that a product contains libwebp, so every libwebp CVE that is published becomes a ticket someone has to investigate. A file-level SBOM records which libwebp source files the build actually compiled. When a vulnerability affects code that never entered the build, Identify can flag it as not applicable and auto-triage it instead of routing it to an engineer.

Your SBOM Will Capture More Dependencies

File-level evidence is most valuable when a dependency has no package manifest, has been copied or modified, or includes only a subset of the upstream project. A package-only inventory might identify the general library but obscure which parts of it were actually compiled.

The excerpt below shows a file component Identify recorded from a libwebp source file vendored inside OpenCV's 3rdparty/ directory, exactly the kind of dependency a manifest-based scanner may overlook.

SBOM COMPONENT predictor_enc.c

```
{
  "name": "predictor_enc.c",
  "authors": [
    { "name": "Google Inc" }
  ],
  "copyright": "Copyright 2016 Google Inc",
  "properties": [
    {
      "name": "filePath",
      "value":
        ".../opencv/3rdparty/libwebp/
         src/enc/predictor_enc.c"
    }
  ]
}
```

Another example comes from QNX 7.1, where RunSafe Identify detected GCC 8.3.0 bundled within the operating system's software distribution. While a package-level inventory might represent the QNX Software Development Platform as a single component, RunSafe identifies GCC separately, making a dependency that could otherwise go unnoticed visible for vulnerability analysis and management.[5]

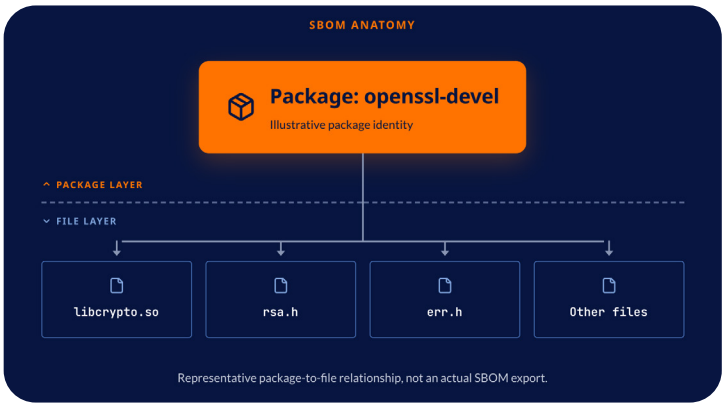
These examples demonstrate how RunSafe provides a more precise and defensible answer to the core SBOM question of what code went into a specific build. File-level detail also makes compile-time differences between releases visible: when settings cause one build to use files that another excludes, the SBOM shows not only that a package was present but also which portions of it were used in each build.

This additional visibility improves vulnerability detection and management. Organizations can identify dependencies that might otherwise be missed, distinguish which software is affected by a vulnerability, and prioritize remediation efforts based on what is actually present in their products.

Your SBOM Will Include a Package-Level View

A RunSafe SBOM represents software at two related levels. File components record the individual files observed during the build, while package components organize related files under recognizable software identities. Together, the two levels provide the detail needed to verify the build and the package-oriented structure that developers, security teams, and auditors expect when reviewing an SBOM.

RunSafe's automated package detection groups related file components into recognizable packages while preserving the underlying file-based evidence. For example, a developer looking for OpenSSL is more likely to search for the package name than for an individual OpenSSL file in the build. Identify presents OpenSSL as a parent package while retaining the observed files beneath it.[5]



This creates two complementary views: a package-level view that is easier to search, review, and use for vulnerability and license management, and a file-level view that shows what was actually compiled and supports the package identity assigned to those files.

Because C and C++ lack a universal package manager, Identify builds the package view by combining the identity and provenance evidence available in the build environment. Depending on the project and platform, this can include, but is not limited to:

- Git submodules[7]
- Operating-system package databases such as dpkg and apt on Debian and Ubuntu systems or RPM-based tooling on Red Hat systems
- Source-file metadata such as copyright and license headers
- Build-specific package information such as vcpkg on Windows
- RTOS metadata from QNX and VxWorks installations
- External identity mappings among repositories, product names, suppliers, PURLs, and CPEs
- Customer-supplied configuration for proprietary or unusually structured components[8]

Identify uses a hierarchy of evidence to resolve conflicts and select the most reliable identity. An operating-system package database, for example, may provide a more authoritative supplier and version than the name of the repository the source came from. The package view is therefore constructed from observed evidence rather than assigned from a filename or a binary signature, and the original file-level detail remains available for verification.

The resolved component identity also drives vulnerability matching. RunSafe creates packages, and those packages contain a CPE/PURL, the industry-recognized standard for package identification. Where a component has a CPE, Identify uses the resolved identity to determine it and match the component to Common Vulnerabilities and Exposures (CVEs) and their Common Vulnerability Scoring System (CVSS) scores. Identify also derives a PURL in scenarios where the build evidence supports one. This allows RunSafe's package view to map directly to vulnerability management and triaging through the RunSafe Security Platform.

The RunSafe Security Platform draws on more than 20 vulnerability data sources, including NIST's National Vulnerability Database, OSV, and GitHub Security Advisories, refreshed at least daily.

Your SBOM Can Be Fine-Tuned to Your Environment

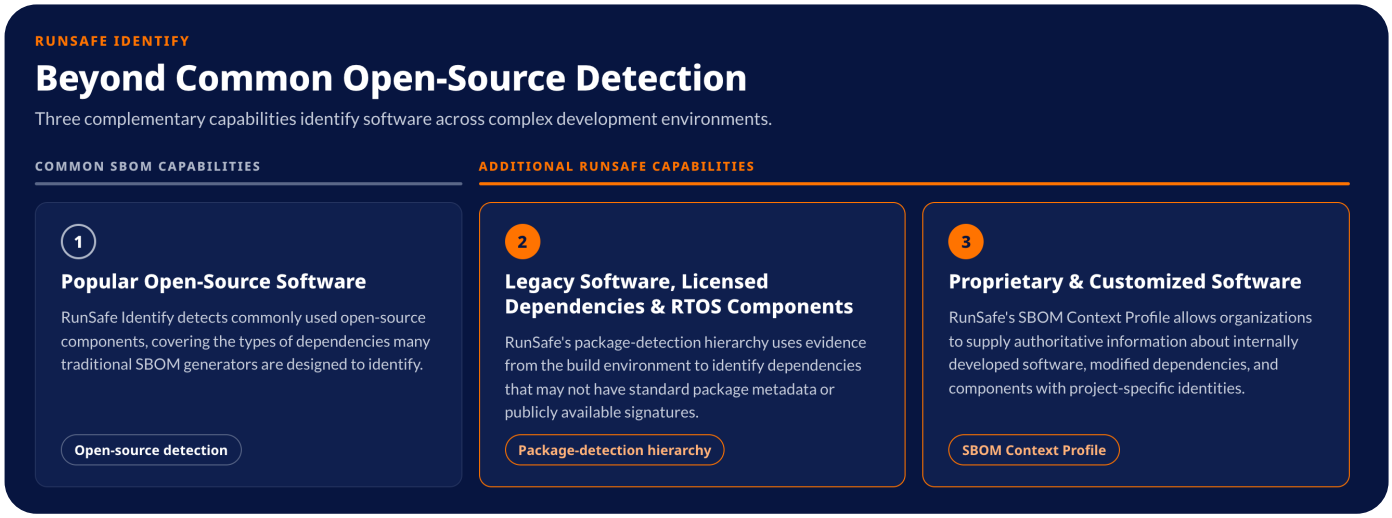
RunSafe automatically uses the evidence in the source tree, repository structure, operating system, RTOS, and build environment to identify components and generate an SBOM. However, for instances where a project has a unique context, like in the case of proprietary software, RunSafe's SBOM Context Profile captures this information and applies it to all future builds.[8]

For example, an organization that compiles an internal library into several products can establish that component's identity once, associating the source location with the correct name, supplier, version, and license, and subsequent SBOMs will apply the same information as new releases are generated.

RunSafe's approach goes beyond the open-source detection capabilities typically found in SBOM generators. It combines three complementary capabilities to identify software across more complex development environments:

1. **Popular open-source software:** RunSafe Identify detects commonly used open-source components, covering the types of dependencies many traditional SBOM generators are designed to identify.
2. **Legacy software, licensed dependencies, and RTOS components:** RunSafe's package-detection hierarchy uses evidence from the build environment to identify dependencies that may not have standard package metadata or publicly available signatures.
3. **Proprietary and customized software:** RunSafe's SBOM Context Profile allows organizations to supply authoritative information about internally developed software, modified dependencies, and components with project-specific identities.

Together, these capabilities give organizations visibility into both commonly used open-source software and the harder-to-identify components that traditional approaches overlook.



From Inventory to Vulnerability Management

An accurate inventory of the software components included in a build is the foundation for effective vulnerability management. Once a build produces an SBOM, the RunSafe Security Platform repeats the analysis on every build and release:

- **Enrich:** Component identity improves as better version, license, and component intelligence become available, and earlier SBOMs benefit from it rather than aging in place.
- **Assignment:** Components are matched to CVEs by PURL and CPE, and the match is re-evaluated as new vulnerabilities are published, so a release that shipped last quarter keeps reporting against current data.[9]
- **Triage:** File-level evidence removes vulnerabilities in code that the build never compiled, so the queue that reaches engineering is smaller and more relevant.[6]
- **Prioritize:** Exposure is assessed using severity, CVSS score, and Known Exploited Vulnerabilities (KEV) catalog status, so remediation effort follows demonstrated risk rather than raw CVE count.[5]
- **Enforce:** License conflicts and critical vulnerabilities are checked against policy, and the pipeline can be gated when policy requires it.[7]
- **Report:** Every build leaves a dated, CycloneDx artifact, which is what customers and regulators ask to see.
- **Monitor:** RunSafe Identify rescans SBOMs nightly, building a new report when any new vulnerabilities are identified for an SBOM. This allows customers to travel back in time to see what vulnerabilities they knew about when software was originally shipped all the way to the most recent information.[5]
- **Alert:** RunSafe Identify alerts users to newly discovered vulnerabilities so they can act quickly to resolve those vulnerabilities.[10]

These capabilities work together to improve the accuracy and usefulness of vulnerability reporting. Component identification and enrichment provide a stronger foundation for vulnerability assignment, while file-level evidence helps eliminate findings that don't apply to the software actually built. Prioritization, monitoring, and alerting build on that information to keep teams focused on relevant risks as new vulnerabilities emerge. The result is a smaller, more accurate set of findings for security and engineering teams to investigate and address.

Integrating RunSafe Identify into Your Software Development Lifecycle

RunSafe Identify runs where your builds run and generates an SBOM for every build and release. Because Identify runs on every build, SBOMs stay current release to release rather than aging between periodic scans, ensuring compliance is maintained across all software versions.[11]

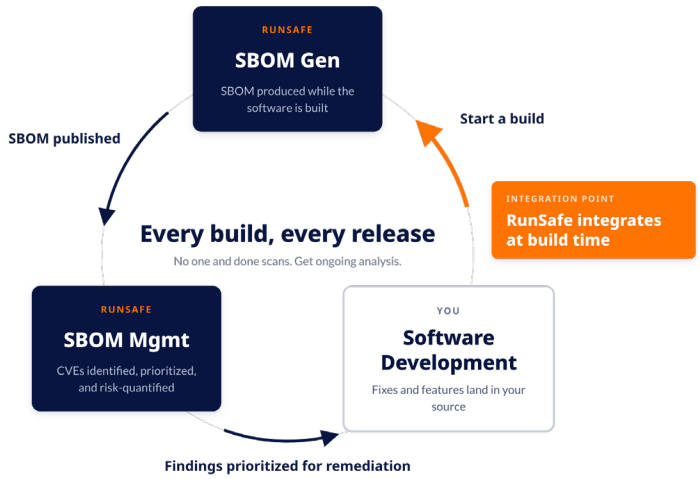


Figure 3. RunSafe integrates at build time, so an SBOM is produced for every build and release and feeds into ongoing analysis rather than a one-time scan.

RunSafe supports a software-as-a-service (SaaS) model on the RunSafe Security Platform and an on-premises model that points the SBOM server at a customer-hosted instance. SBOMs are generated for a variety of languages within your existing continuous integration and delivery (CI/CD) pipeline. For fully disconnected environments, including air-gapped defense and operational technology (OT) networks, Identify supports offline license files so that neither the build nor license validation requires an outbound connection.

RunSafe customers have integrated our Identify software in a variety of C/C++ environments, ranging from Yocto to Make/CMake and highly customized embedded build environments. For C/C++ builds, integration is a matter of prefixing your normal build command.[11]

RunSafe's SBOM management and vulnerability compliance also connect to RunSafe Protect, which hardens software against the exploitation of memory-based vulnerabilities. When an SBOM surfaces these vulnerabilities, Identify tells you what memory-safety vulnerabilities are present and how exposed you are, including severity and KEV status. Protect stops attackers from exploiting memory bugs while a fix is developed, tested, and shipped.[12]

Compliance Requirements Identify Supports

Because Identify produces CycloneDx, repeatable SBOMs with vulnerability and license data, it supports transparency and common reporting expectations.

The table below maps each requirement to the Identify output that supports it.

REQUIREMENT	WHAT IT ASKS FOR	SUPPORTING IDENTIFY OUTPUT
EU Cyber Resilience Act	Technical documentation of components and a vulnerability handling process. Reporting obligations applied from September 11, 2026; remaining obligations apply from December 11, 2027.	• A dated SBOM per build with continuous CVE matching and a record of what was known when • Produces a CRA compliant SBOM that has name/version/dependency information
FDA premarket cybersecurity (Section 524B)	An SBOM covering commercial, open source, and off-the-shelf components in a cyber device, plus a plan to monitor and address vulnerabilities	• CycloneDX SBOM per release, with ongoing monitoring through the RunSafe Security Platform
NTIA minimum elements	Supplier, component name, version, other unique identifiers, dependency relationship, author, and timestamp[13]	• Required elements are contained in the CycloneDX SBOM RunSafe generates • Fields populated from observed build evidence rather than declared manifests
US Army SBOM guidance	SBOM delivery alongside software acquisitions	• Repeatable generation inside the supplier's own pipeline, including air-gapped builds

Summary

RunSafe Identify generates SBOMs by observing the build, recording every file compiled, linked, or fetched, and extracting provenance from each. That build-time, file-based approach captures components that source and binary scanners miss, including statically linked libraries, vendored and copy-pasted code, proprietary components, and dependencies fetched during the build. It applies across many languages and is essential for C and C++, where no universally adopted package manager exists and integration is ad hoc.

The output is a CycloneDX SBOM detailed enough for technical verification and organized enough for practical use and deployable across SaaS, on-premises, and fully offline environments. Paired with the RunSafe Security Platform, the SBOM becomes a living foundation for vulnerability management, license compliance, and supply chain risk decisions, and with RunSafe Protect, it extends into runtime risk reduction.

When the next vulnerability in a widely-used C library is published, an accurate build-time inventory is what lets you answer whether your products are affected, which versions, and which files, in hours instead of weeks.



Additional Resources

[1] K. Schwalm, "The Wild West of C/C++ Development & What It Means for SBOM Generation," RunSafe Security, Sept. 23, 2025. https://runsafesecurity.com/blog/cpp-development-sbom-generation/

[2] RunSafe Security, "A Guide to SBOM Requirements & Standards Around the Globe," RunSafe Security, Feb. 11, 2025. https://runsafesecurity.com/blog/sbom-requirements-global-guide/

[3] K. Schwalm, "3 SBOM Generation Methods: Binary vs Build-Time vs Source Code," RunSafe Security, Feb. 23, 2026. https://runsafesecurity.com/blog/sbom-generation-methods/

[4] K. Schwalm, "How to Validate SBOM Accuracy for Embedded C/C++ Projects," RunSafe Security, Mar. 18, 2026. https://runsafesecurity.com/blog/validate-sbom-accuracy/

[5] RunSafe Security, "December 2025 Product Update: Platform Updates, New Reachability Analysis, and QNX Package Detection," Dec. 19, 2025. https://runsafesecurity.com/product-updates/december-2025-product-update/

[6] S. Blume, "Reducing Vulnerability Noise with Reachability Analysis," RunSafe Security, Apr. 9, 2026. https://runsafesecurity.com/blog/reachability-analysis/

[7] RunSafe Security, "October 2025 Product Updates: GitHub Support Now Available, License Compliance Features, and More," Oct. 14, 2025. https://runsafesecurity.com/product-updates/october-2025-product-update/

[8] RunSafe Security, "August 2025 Product Updates: Expanded VxWorks Coverage, Streamlined CLI, and Smarter SBOM Enrichment," Sept. 2, 2025. https://runsafesecurity.com/product-updates/august-2025-product-update/

[9] RunSafe Security, "From SBOM to Answer: Cutting Vulnerability Triage Time in Embedded Devices," Sept. 22, 2026. https://runsafesecurity.com/blog/sbom-vulnerability-management-embedded/

[10] RunSafe Security, "June 2026 Product Update: .NET SBOM Support, GitHub Enterprise Connections, and Vulnerability Alerting," July 9, 2026. https://runsafesecurity.com/product-updates/june-2026-product-update/

[11] RunSafe Security Documentation, "C/C++ SBOM Generation." https://docs.runsafesecurity.com/identify/compiled-sboms

[12] RunSafe Security, "Tech Deep Dive: RunSafe Security's Protect Solution." https://runsafesecurity.com/whitepaper/tech-deep-dive-runsafe-securitys-protect-solution/

[13] S. Fry, "Mapping CISA's 2026 SBOM Minimum Elements to CycloneDX and SPDX: An Implementation Guide," RunSafe Security, July 30, 2026. https://runsafesecurity.com/blog/sbom-minimum-elements-cyclonedx-spdx/

Additional reading: RunSafe Security, "SBOM Accuracy Checklist." https://runsafesecurity.com/sbom-checklist-for-cpp/

RUNSAFE SECURITY

ABOUT RUNSAFE SECURITY, INC.

RunSafe Security protects embedded software across critical infrastructure, delivering automated vulnerability identification and software hardening from build-time to runtime to defend the software supply chain and critical systems without compromising performance or requiring code rewrites. The RunSafe Security Platform includes the authoritative build-time SBOM generator fo embedded systems and C/C++ projects, automated vulnerability identification and risk quantification, license compliance, and patented memory relocation techniques to mitigate memory-based vulnerabilities.

Headquartered in McLean, Virginia, with an office in Huntsville, Alabama, RunSafe Security's customers span the aerospace and defense, energy, operational technology, industrial automation, transportation and automotive, medical device, and high-tech manufacturing verticals.

RunSafeSecurity.com

571.441.5076

Sales@RunSafeSecurity.com