OWASP
Open Web Application Security Project

Standard

SCVS

Software Component Verification Standard

Version 1.0

Frontispiece

About the Standard

The Software Component Verification Standard is a grouping of controls, separated by control family, which can be used by architects, developers, security, legal, and compliance to define, build, and verify the integrity of their software supply chain.

Copyright and License

Version 1.0, 25 June 2020

Project Leads

- Steve Springett

Contributors and Reviewers

- Dave Russo
- Garret Fick
- JC Herz
- John Scott
- Mark Symons
- Pruthvi Nallapareddy
- Bryan Garcia

The Software Component Verification Standard is built upon the shoulders of those involved. The project is inspired by the OWASP Application Security Verification Standard and the work of their contributors.

If a credit is missing from the credit list above, please contact steve.springett@owasp.org or log a ticket at GitHub to be recognized in future updates.

Preface

Welcome to the Software Component Verification Standard (SCVS) version 1.0. The SCVS is a community-driven effort to establish a framework for identifying activities, controls, and best practices, which can help in identifying and reducing risk in a software supply chain.

Managing risk in the software supply chain is important to reduce the surface area of systems vulnerable to exploits, and to measure technical debt as a barrier to remediation.

Measuring and improving software supply chain assurance is crucial for success. Organizations with supply chain visibility are better equipped to protect their brand, increase trust, reduce time-to-market, and manage costs in the event of a supply chain incident.

Software supply chains involve:

- technology
- people
- processes
- institutions
- and additional variables

Raising the bar for supply chain assurance requires the active participation of risk managers, mission owners, and business units like legal and procurement, which have not traditionally been involved with technical implementation.

Determination of risk acceptance criteria is not a problem that can be solved by enterprise tooling: it is up to risk managers and business decision makers to evaluate the advantages and trade-offs of security measures based on system exposure, regulatory requirements, and constrained financial and human resources. Mandates that are internally unachievable, or that bring development or procurement to a standstill, constitute their own security and institutional risks.

SCVS is designed to be implemented incrementally, and to allow organizations to phase in controls at different levels over time.

Using the SCVS

SCVS has the following goals:

- Develop a common set of activities, controls, and best-practices that can reduce risk in a software supply chain
- Identify a baseline and path to mature software supply chain vigilance

Control Families

There are six control families that contain multiple controls that apply to different aspects of component verification or processes where component verification occurs. The control families are:

- V1: Inventory
- V2: Software Bill of Materials (SBOM)
- V3: Build Environment
- V4: Package Management
- V5: Component Analysis
- V6: Pedigree and Provenance

Software Component Verification Levels

The Software Component Verification Standard defines three verification levels. Higher levels include additional controls.

- SCVS Level 1 is for low-assurance requirements where basic forms of analysis would suffice.
- SCVS Level 2 is for moderately sensitive software where additional analysis or due diligence is required.
- SCVS Level 3 is for high-assurance requirements due to the sensitivity of data or use of the software.

Level 1

SCVS level 1 lays the groundwork from which to build upon. This level focuses on implementing best practices such as:

- creating software bill of materials with complete and accurate inventory
- utilizing continuous integration to produce repeatable builds
- performing analysis of third-party components with tools and intelligence that are publicly available

Adoption of level 1 can be achieved with modern software engineering practices.

Level 2

SCVS level 2 expands the breadth of level 1 capabilities. Level 2 is appropriate for software intensive enterprises and organizations with existing risk management frameworks and regulatory and/or contractual requirements. Level 2 also expands the number of stakeholders including those with non-technical roles. Adoption of level 2 may require additional resources and domain expertise to achieve.

Level 3

SCVS level 3 extends the depth of level 2 capabilities. Level 3 is applicable in critical infrastructure and systems with safety requirements. Auditability and end-to-end transparency in the supply chain is required to maintain a high security posture in these systems and the organizations that produce and maintain them.

How to Use This Standard

One of the best ways to use the Software Component Verification Standard is to use it as a method of assessment and a way to sequence incremental improvement. Tailoring the SCVS to your use cases will increase the focus on the security requirements that are most important to your organization. Organizations may choose to adopt certain controls from higher levels without implementing all higher-level controls.

One use case for the SCVS is to assess internal software development processes and capabilities. Another is to evaluate suppliers, which can include open source software maintainers, contract software developers, and commercial software vendors.

Assessing First Party Capabilities

SCVS provides a structured method to assess and qualify internal software capabilities and processes, and to identify areas where verifiability can be improved. Because of the tiered and topical structure of SCVS, the same organization can be at different levels within or across

categories of verification. Depending on required levels of assurance for a given system, different levels of SCVS controls may be required for system approval.

SCVS controls are formulated to be automatable, and to provide the basis for system approval on a continuous basis. Given the rate of change in software composition, especially for capabilities developed in a continuous integration pipeline, continuity of assurance requires continuous rather than one-time verification.

Evaluating Suppliers

SCVS provides a standardized way to assess supply chain transparency provided by contract or outsourced software suppliers, based on the documentation and metadata present in the supplier's software development workflow and/or provided with software deliverables. Specifically, the provision of a software bill of materials (SBOM) can be requested or required by software customers to establish supply chain visibility and differentiate suppliers.

Because of the tiered and topical structure of SCVS, it can be used for analysis of alternatives and/or used in whole or in part to evaluate proposals for procurement. SCVS controls at different levels can be used to qualify eligibility for procurement, or as an element of scoring proposals.

Supplier provision of software bills of materials and other verification data allows customers to monitor risks present in supplier software on an ongoing basis to achieve visibility regardless of supplier-provided notification or updates. The auditability of this data enables formulation and enforcement of contractual provisions to remediate known vulnerabilities within a specified time period.

Applying SCVS

The Software Component Verification Standard places emphasis on controls that can be implemented or verified through automation. The control families are not specific to a single development team. They represent stakeholders across an organization, including software developers, security and risk managers, and procurement departments. Active participation of all stakeholders is necessary to measure and improve cyber posture. Once an organization has determined the current maturity baseline, it can determine goals and timelines to improve maturity and devise methods for which the controls can be implemented or verified through automation.

Satisfying control requirements at a higher SCVS level may be more dependent on business process than available technical methods.

Assessment and Certification

OWASP's Stance on SCVS Certifications and Trust Marks

OWASP, as a vendor-neutral not-for-profit organization, does not certify any vendors, verifiers or software.

All such assurance assertions, trust marks, or certifications are not officially vetted, registered, or certified by OWASP, so an organization relying upon such a view needs to be cautious of the trust placed in any third-party or trust mark claiming SCVS certification.

This should not inhibit organizations from offering such assurance services, as long as they do not claim official OWASP certification.

Guidance for Certifying Software Component Supply Chains

The recommended way of verifying compliance of a software supply chain with SCVS is by performing an "open book" review, meaning that the auditors are granted access to key resources such as legal, procurement, build engineers, developers, repositories, documentation, and build environments with source code.

A certifying organization must include in any report the scope of the verification (particularly if any practice is out of scope), a summary of verification findings, with clear indications of how to resolve and improve results. In case of dispute, there should be sufficient supportive evidence to demonstrate that every verified practice has indeed been met.

Evaluators and software suppliers may describe assessments as having been performed using SCVS controls, as long as control levels are disclosed.

The Role of Automated Verification

Whenever possible, automation should be used to verify the controls detailed in SCVS in order to increase efficiency and accuracy. Some controls cannot be verified through automation. However, for the controls that can, automation is encouraged if the results can be validated through other means.

For higher levels of assurance controls may be independently validated using automated methods.

V1: Inventory Requirements

Control Objective

An accurate inventory of all components used in the creation of software is a foundational requirement for further analysis. The following controls incorporate single application inventories, organizational inventories, and approaches to enable software transparency when procuring new software or systems.

Component identification varies based on the ecosystem the component is part of. Therefore, for all inventory purposes, the use of identifiers, such as Package URL, may be used to standardize and normalize naming conventions for managed dependencies.

An organization-wide inventory of all first-party and third-party (including open source) components allows for greater transparency within the organization, promotes software standardization and reuse, and allows for rapid impact analysis.

Verification Requirements

#	Description	L1	L2	L3
1.1	All direct and transitive components and their versions are known at completion of a build	✓	✓	✓
1.2	Package managers are used to manage all third-party binary components	✓	✓	✓
1.3	An accurate inventory of all third-party components is available in a machine-readable format	✓	✓	✓
1.4	Software bill of materials are generated for publicly or commercially available applications	✓	✓	✓
1.5	Software bill of materials are required for new procurements		✓	✓
1.6	Software bill of materials continuously maintained and current for all systems			✓
1.7	Components are uniquely identified in a consistent, machine-readable format	✓	✓	✓
1.8	The component type is known throughout inventory			✓

1.9	The component function is known throughout inventory			✓
1.10	Point of origin is known for all components			✓

V2: Software Bill of Materials (SBOM) Requirements

Control Objective

Automatically creating accurate Software Bill of Materials (SBOM) in the build pipeline is one indicator of mature development processes. SBOMs should be a machine readable format. Each format has different capabilities and use-cases they excel in. Part of SBOM adoption is identifying the use-cases and capabilities best suited to specific purposes. While SBOM format standardization across an organization may be desirable, it may be necessary to adopt more than one to meet functional, contractual, compliance, or regulatory requirements.

Verification Requirements

#	Description	L1	L2	L3
2.1	A structured, machine readable software bill of materials (SBOM) format is present	✓	✓	✓
2.2	SBOM creation is automated and reproducible		✓	✓
2.3	Each SBOM has a unique identifier	✓	✓	✓
2.4	SBOM has been signed by publisher, supplier, or certifying authority		✓	✓
2.5	SBOM signature verification exists		✓	✓
2.6	SBOM signature verification is performed			✓
2.7	SBOM is timestamped	✓	✓	✓
2.8	SBOM is analyzed for risk	✓	✓	✓
2.9	SBOM contains a complete and accurate inventory of all components the SBOM describes	✓	✓	✓

2.10	SBOM contains an accurate inventory of all test components for the asset or application it describes		✓	✓
2.11	SBOM contains metadata about the asset or software the SBOM describes		✓	✓
2.12	Component identifiers are derived from their native ecosystems (if applicable)	✓	✓	✓
2.13	Component point of origin is identified in a consistent, machine readable format (e.g. PURL)			✓
2.14	Components defined in SBOM have accurate license information	✓	✓	✓
2.15	Components defined in SBOM have valid SPDX license ID's or expressions (if applicable)		✓	✓
2.16	Components defined in SBOM have valid copyright statements			✓
2.17	Components defined in SBOM which have been modified from the original have detailed provenance and pedigree information			✓
2.18	Components defined in SBOM have one or more file hashes (SHA-256, SHA-512, etc)			✓

V3: Build Environment Requirements

Control Objective

Software build pipelines may consist of source code repositories, package repositories, continuous integration and delivery processes, and test procedures, along with the network infrastructure and services that enable these capabilities. Every system in the pipeline may provide an entry-point in which flaws, failures, and misconfigurations can compromise the software supply chain. Hardening the systems involved and implementing best practices reduces the likelihood of compromise.

Verification Requirements

#	Description	L1	L2	L3
3.1	Application uses a repeatable build	✓	✓	✓

3.2	Documentation exists on how the application is built and instructions for repeating the build	✓	✓	✓
3.3	Application uses a continuous integration build pipeline	✓	✓	✓
3.4	Application build pipeline prohibits alteration of build outside of the job performing the build		✓	✓
3.5	Application build pipeline prohibits alteration of package management settings		✓	✓
3.6	Application build pipeline prohibits the execution of arbitrary code outside of the context of a jobs build script		✓	✓
3.7	Application build pipeline may only perform builds of source code maintained in version control systems	✓	✓	✓
3.8	Application build pipeline prohibits alteration of DNS and network settings during build			✓
3.9	Application build pipeline prohibits alteration of certificate trust stores			✓
3.10	Application build pipeline enforces authentication and defaults to deny		✓	✓
3.11	Application build pipeline enforces authorization and defaults to deny		✓	✓
3.12	Application build pipeline requires separation of concerns for the modification of system settings			✓
3.13	Application build pipeline maintains a verifiable audit log of all system changes			✓
3.14	Application build pipeline maintains a verifiable audit log of all build job changes			✓
3.15	Application build pipeline has required maintenance cadence where the entire stack is updated, patched, and re-certified for use		✓	✓
3.16	Compilers, version control clients, development utilities, and software development kits are analyzed and monitored for tampering, trojans, or			✓

	malicious code			
3.17	All build-time manipulations to source or binaries are known and well defined	✓	✓	✓
3.18	Checksums of all first-party and third-party components are documented for every build	✓	✓	✓
3.19	Checksums of all components are accessible and delivered out-of-band whenever those components are packaged or distributed		✓	✓
3.20	Unused direct and transitive components have been identified			✓
3.21	Unused direct and transitive components have been removed from the application			✓

V4: Package Management Requirements

Control Objective

Open source components intended for reuse are often published to ecosystem-specific package repositories. Centralized repositories exist for many build systems including Maven, .NET, NPM and Python. Repositories internal to an organization may additionally provide reuse of first-party components as well as access to trusted third-party components.

Package managers are often invoked during the build process and are responsible for resolving component versions and retrieving components from repositories.

While there are tremendous business, technical, and security benefits to using package managers and centralized repositories, they are often targets for adversaries. Implementing best practices can dramatically reduce risk of compromise in the software supply chain.

Verification Requirements

#	Description	L1	L2	L3
4.1	Binary components are retrieved from a package repository	✓	✓	✓
4.2	Package repository contents are congruent to an authoritative point of origin	✓	✓	✓

	for open source components			
4.3	Package repository requires strong authentication		✓	✓
4.4	Package repository supports multi-factor authentication component publishing		✓	✓
4.5	Package repository components have been published with multi-factor authentication			✓
4.6	Package repository supports security incident reporting		✓	✓
4.7	Package repository automates security incident reporting			✓
4.8	Package repository notifies publishers of security issues		✓	✓
4.9	Package repository notifies users of security issues			✓
4.10	Package repository provides a verifiable way of correlating component versions to specific source codes in version control		✓	✓
4.11	Package repository provides auditability when components are updated	✓	✓	✓
4.12	Package repository requires code signing to publish packages to production repositories		✓	✓
4.13	Package manager verifies the integrity of packages when they are retrieved from remote repository	✓	✓	✓
4.14	Package manager verifies the integrity of packages when they are retrieved from file system	✓	✓	✓
4.15	Package repository enforces use of TLS for all interactions	✓	✓	✓
4.16	Package manager validates TLS certificate chain to repository and fails securely when validation fails	✓	✓	✓
4.17	Package repository requires and/or performs static code analysis prior to publishing a component and makes results available for others to consume			✓

4.18	Package manager does not execute component code	✓	✓	✓
4.19	Package manager documents package installation in machine-readable form	✓	✓	✓

V5: Component Analysis Requirements

Control Objective

Component Analysis is the process of identifying potential areas of risk from the use of third-party and open-source software components. Every component, direct or transitive, is a candidate for analysis. Risk inherited through the use of third-party software may directly affect the application or systems that rely on them.

Known Vulnerabilities

There are multiple public and commercial sources of vulnerability intelligence. Vulnerabilities become known when they are published to services such as the National Vulnerability Database (NVD), or are otherwise documented in public defect trackers, commit logs, or other public source.

Component Version Currency

If component versions are specified, then determining if a component is out-of-date or end-of-life is possible. Outdated and end-of-life components are more likely to be vulnerable and less likely to be supported as first-class entities. Out-of-date components can slow down system remediation due to interoperability issues. Using up-to-date components reduces exposure time and may include remediations of non-disclosed vulnerabilities.

Component Type

Frameworks and libraries have unique upgrade challenges and associated risk. Abstractions, coupling, and architectural design patterns may affect the risk of using a given component type. Libraries, frameworks, applications, containers, and operating systems are common component types.

Component Function

Identifying and analyzing the purpose of each component may reveal the existence of components with duplicate or similar functionality. Potential risk can be reduced by minimizing the number of components for each function and by choosing the highest quality components for each function.

Component Quantity

The operational and maintenance cost of using open source may increase with the adoption of every new component. Decreased ability to maintain growing sets of components over time can be expected. This is especially true for teams with time-boxed constraints.

License

Third-party and open-source software is typically released under one or more licenses. The chosen license may or may not allow certain types of usage, contain distribution requirements or limitations, or require specific actions if the component is modified. Utilizing components with licenses which conflict with an organizations objectives or ability can create risk to the business.

Verification Requirements

#	Description	L1	L2	L3
5.1	Component can be analyzed with linters and/or static analysis tools	✓	✓	✓
5.2	Component is analyzed using linters and/or static analysis tools prior to use		✓	✓
5.3	Linting and/or static analysis is performed with every upgrade of a component		✓	✓
5.4	An automated process of identifying all publicly disclosed vulnerabilities in third-party and open source components is used	✓	✓	✓
5.5	An automated process of identifying confirmed dataflow exploitability is used			✓
5.6	An automated process of identifying non-specified component versions is used	✓	✓	✓
5.7	An automated process of identifying out-of-date components is used	✓	✓	✓
5.8	An automated process of identifying end-of-life / end-of-support components is used			✓
5.9	An automated process of identifying component type is used		✓	✓
5.10	An automated process of identifying component function is used			✓
5.11	An automated process of identifying component quantity is used	✓	✓	✓

5.12	An automated process of identifying component license is used	✓	✓	✓

V6: Pedigree and Provenance Requirements

Control Objective

Identify point of origin and chain of custody in order to manage system risk if either point of origin or chain of custody is compromised. For internal package managers and repositories it is important to maintain pedigree and provenance data for imported components.

Verification Requirements

#	Description	L1	L2	L3
6.1	Point of origin is verifiable for source code and binary components		✓	✓
6.2	Chain of custody if auditable for source code and binary components			✓
6.3	Provenance of modified components is known and documented	✓	✓	✓
6.4	Pedigree of component modification is documented and verifiable		✓	✓
6.5	Modified components are uniquely identified and distinct from origin component		✓	✓
6.6	Modified components are analyzed with the same level of precision as unmodified components	✓	✓	✓
6.7	Risk unique to modified components can be analyzed and associated specifically to modified variant	✓	✓	✓

Guidance: Open Source Policy

The following points should be viewed as suggestions based on the success and best practices of organizations employing them. They are not part of SCVS.

- All organizations that use open source software should have an open source policy
- The open source policy is supported and enforced by cross-functional stakeholders
- The open source policy should address:
  - The age of a component based on its release or published date
  - How many major or minor revisions old are acceptable
  - Guidance for keeping components continuously updated via automation
  - Exclusion criteria for components with known vulnerabilities
  - Mean-time-to-remediate criteria for updating at-risk components
  - Restrictions on using components that are end-of-life or end-of-support
  - Criteria for supplier selection or exclusion
  - Usage-based list of acceptable licenses
  - Prohibited components list
  - Mechanisms and permissions for providing modifications back to the community producing the component

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.
- **CycloneDX** - A software bill of materials specification designed to be lightweight and security-focused.
- **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 bill of materials (SBOM)** – A complete, formally structured, and machine-readable inventory of all software components and associated metadata, used by or delivered with a given piece of software.
- **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:

OWASP Projects

- OWASP Packman
- OWASP Software Assurance Maturity Model (SAMM)

Community Projects

- Open Source Security Coalition - Threats, Risks, and Mitigations in the Open Source Ecosystem

Others

- InnerSource
- Cybersecurity Maturity Model Certification (CMMC)
- NIST 800-53 Security and Privacy Controls for Federal Information Systems and Organizations
- NIST 800-161 Supply Chain Risk Management Practices for Federal Information Systems and Organizations
- NIST 800-171 Protecting Controlled Unclassified Information in Nonfederal Systems and Organizations
- NTIA Documents on Software Bill of Materials
- Model Procurement Contract Language Addressing Cybersecurity Supply Chain Risk
- Guide on Cybersecurity Procurement Language in Task Order Requests for Proposals for Federal Facilities
- Energy Sector Control Systems Working Group (ESCSWG)

SBOM Formats

- CycloneDX
- SPDX

- SPDX XML
- ISO/IEC 19770-2:2015 (SWID)